Laboratorium 10: „Gniazda sieciowe Netlink i Genetlink” (jedne zajęcia)

dr inż. Arkadiusz Chrobot

25 lutego 2019

Spis treści

Wprowadzenie

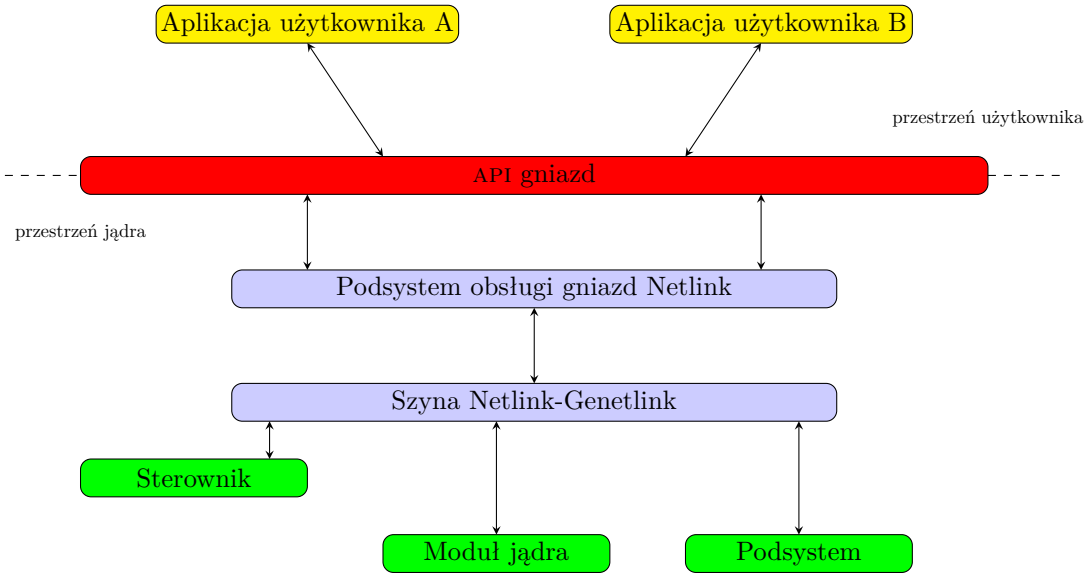
Gniazda BSD mogą być użyte nie tylko do komunikacji między komputerami połączonymi w sieć, ale także do komunikacji lokalnej, między procesami użytkownika, a nawet między jądrem a procesami użytkownika. Niniejsza instrukcja dotyczy tego trzeciego przypadku, z wykorzystaniem gniazd Netlink i Genetlink. Rozdział 1 traktuje o szczegółach komunikacji lokalnej z użyciem wspomnianych gniazd. Rozdział 2 zawiera opis elementów API przestrzeni użytkownika i przestrzeni jądra, które używane jest komunikacji za pomocą Netlink i Genetlink. W rozdziale 3 zamieszczone i opisane są kody źródłowe przykładowych modułów i programów użytkowych, które komunikują się z wykorzystaniem gniazd Netlink i Genetlink.

1. Komunikacja lokalna przy użyciu gniazd sieciowych

Istnieje wiele mechanizmów, dzięki którym procesy użytkownika i jądro systemu Linux mogą się komunikować. Ich obszerny spis zamieszczono w artykule znajdującym się na stronie WWW o adresie `http://wiki.tldp.org/kernel_user_space_howto`. Wśród nich wymieniono także komunikację lokalną za pomocą gniazd. Do jej realizacji można wykorzystać protokół internetowy UDP, surowe gniazda (ang. *raw socket*) i zdefiniowany własny protokół lub gniazda Netlink. To ostatnie rozwiązanie ma kilka godnych uwagi zalet:

- nawiązanie połączenia z przestrzenią jądra jest proste i bezpieczne, bo korzysta z gotowych i sprawdzonych mechanizmów (gniazda sieciowe),
- komunikacja z użyciem gniazd Netlink jest asynchroniczna,
- gniazda Netlink umożliwiają nadawanie komunikatów do jednego odbiorcy (ang. *unicast*), jak i do wielu odbiorców (ang. *multicast*),
- komunikacja procesów użytkownika z jądrem może być dwukierunkowa,
- ponieważ w komunikacji z użyciem gniazd Netlink używane są proste protokoły, to przetwarzanie komunikatów nimi przesyłanych jest mniej kosztowne niż przesyłanie komunikatów UDP,
- gniazda Netlink mogą być używane w modułach jądra, co nie jest cechą wszystkich mechanizmów komunikacji jądro - procesy użytkownika.

Netlink jest rodziną gniazd, która wykorzystuje protokoły datagramowe do przesyłania danych między przestrzenią użytkownika i przestrzenią jądra. Ta komunikacja może być dwukierunkowa dwukierunkowa. Dodatkowo, może być unikastowa (jeden nadawca - jeden odbiorca) i mulitkastowa (jeden nadawca - wielu odbiorców). Połączenia w przypadku gniazd Netlink nazywają się szynami (ang. *bus*). Typowo z jednym podsystemem jądra korzystającym z mechanizmu Netlink jest związana jedna szyna, choć możliwe jest też rozwiązanie, w którym kilka podsystemów korzysta z tej samej szyny. Maksymalna liczba szyn jest ograniczona i wynosi 32. Aby poradzić sobie z tym ograniczeniem twórcy jądra Linuksa wprowadzili mechanizm Generic Nelink nazywany w skrócie Genetlink, który jest multiplekserem osadzonym na szynie Netlink i pozwala zarejestrować na niej do 65520 *rodzin*. Rodzina jest odpowiednikiem szyny wirtualnej. Innymi słowy, pojedyncza szyna wirtualna może być współdzielona przez wiele podsystemów jądra. Ogólną architekturę tego rozwiązania przedstawia rysunek 1. Procesy użytkownika mogą obsługi-



Rysunek 1: Architektura podsystemu gniazd Netlink i Genetlink (na podstawie `https://wiki.linuxfoundation.org/networking/generic_netlink_howto`)

wać gniazda Netlink i Genetlink przy użyciu tych samych funkcji, co gniazda internetowe, ale wymaga to dodatkowego nakładu pracy, który związany jest z przygotowywaniem struktury komunikatów. To zdanie znacznie upraszcza biblioteka `libnl`. W dystrybucjach Linuksa wywodzących się od dystrybucji Debian, w tym w Ubuntu, można ją zainstalować wraz z dodatkiem umożliwiającym obsługę komunikacji Genetlink przy pomocy następujących poleceń:

```
apt-get install libnl-3-200
apt-get install libnl-3-200-dbg
apt-get install libnl-3-dev
apt-get install libnl-genl-3-dev
apt-get install libnl-genl-3-200
apt-get install pkg-config
```

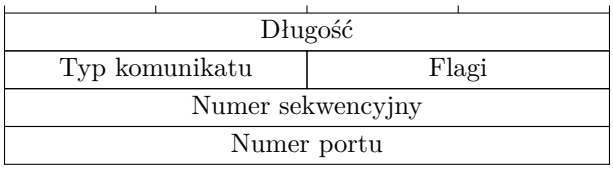
Ostatnie z podanych poleceń instaluje polecenie, które zwraca informacje na temat zainstalowanych w systemie bibliotek i jest pomocne między innymi w kompilacji programów używających `libnl`.

1.1. Gniazda Netlink i Genetlink

Gniazda Netlink i Genetlink umożliwiają komunikację między jądrem systemu operacyjnego, a procesami użytkownika przy użyciu prostego, datagramowego protokołu. Komunikaty Genetlink przed wysłaniem są opakowywane w komunikaty Netlink, dlatego ich format zostanie opisany w podrozdziale 1.2 jako pierwszy. Protokół dla gniazd Genetlink jest objaśniony w podrozdziale 1.3.

1.2. Gniazda Netlink

Komunikat przesyłany przez gniazdo Netlink musi zawierać nagłówek, oraz może składać się ze zbioru atrybutów (również zagnieżdżonych), które zapisane są w formacie TLV (ang. *Type, Length, Value*), czyli opisującym ich typ, wielkość, wyrażoną w bajtach oraz wartość. Strukturę nagłówka protokołu gniazda Netlink ilustruje rysunek 2. Pole „Długość“ (32 bity) zawiera rozmiar komunikatu, wraz z nagłówkiem.



Rysunek 2: Format nagłówka komunikatu netlink (na podstawie `ftp://ftp.rfc-editor.org/in-notes/rfc3549.txt`)

Pole „Typ” (16 bitów) zawiera identyfikator typu komunikatu. Możliwe są dwie kategorie wiadomości: komunikaty kontrolne i komunikaty z danymi. Te ostatnie są specyficzne dla podsystemu jądra, do którego i z którego są wysyłane. Te pierwsze dzielą się na 16 typów (stała `NLMSG_MIN_TYPE`), choć używanych jest tylko cztery:

`NLMSG_NOOP` - brak operacji, ten typ jest wykorzystywany do sprawdzenia, czy określona szyna jest dostępna;

`NLMSG_ERROR` - komunikat zawiera kod błędu;

`NLMSG_DONE` - komunikat końcowy wieloczęściowej wiadomości, w której każdy komunikat ma ustawioną flagę `NLM_F_MULTI`;

`NLMSG_OVERRUN` - komunikat informujący o utracie danych, czyli zagubieniu innych komunikatów.

Pole „Flagi” jest także 16 bitowe i każdy bit tego pola opisuje stan pojedynczej flagi. Aby ułatwić operowanie na tych bitach zdefiniowano następujące stałe (maski):

`NLM_F_REQUEST` - komunikat zawiera żądanie, każdy komunikat wysyłany z przestrzeni użytkownika do przestrzeni jądra musi mieć tę flagę ustawioną, inaczej jądra zwróci kod błędu `EINVAL`;

`NLM_F_CREATE` - proces użytkownika zamierza wydać polecenie lub dodać informacje konfiguracyjne dla podsystemu jądra;

`NLM_F_EXCL` - używana z flagą `NLM_F_CREATE`, by jądro zgłosiło błąd, jeśli dodawane informacje konfiguracyjne już są w jądrze systemu obecne;

`NLM_F_REPLACE` - aplikacja użytkownika chce zastąpić istniejące informacje konfiguracyjne nowymi,

`NLM_F_APPEND` - nakazuje dołączenie nowych informacji konfiguracyjnych do już istniejących,

`NLM_F_DUMP` - proces użytkownika żąda całości informacji konfiguracyjnych, co skutkuje przesłaniem wiadomości wieloczęściowej przez jądro;

`NLM_F_MULTI` - komunikat jest częścią wieloczęściowej wiadomości,

`NLM_F_ACK` - proces z przestrzeni użytkownika żąda potwierdzenia przez jądro wykonania zażądanej operacji,

`NLM_F_ECHO` - proces użytkownika żąda potwierdzenia wykonania operacji przy pomocy komunikacji unkastowej, jeśli proces otrzymuje również powiadomienia o zdarzeniach, to potwierdzenie nie będzie mu nadmiarowo wysłane przy pomocy komunikacji multikastowej.

Pole „Numer sekwencyjny” jest 32 bitowe i zawiera wartość numeryczną (liczbę). Jest ono użyteczne, jeśli proces użytkownika ustawił w nagłówku komunikatu flagę `NLM_F_ACK`, wówczas jądro wysyła komunikat potwierdzający z tym samym numerem sekwencyjnym. Jeśli komunikat jest nadawany przez jądro i informuje o wystąpieniu określonego zdarzenia (transmisja multikastowa), to w tym polu jest zawsze wartość `0`. Pole „Numer portu” (32 bity) zawarty jest identyfikator numeryczny. Temu polu podsystem Netlink nadaje wartość PID procesu użytkownika, który wysyła wiadomość i ma otwarte tylko jedno gniazdo Netlink. Jeśli ten proces otworzy kolejne gniazdo tego typu, to otrzyma ono inną wartość portu. Wiadomości wysyłane przez jądro systemu zawsze mają port równy `0`. Za nagłówkiem komunikatu dołączane są atrybutu. Pojedynczy atrybut ma format określony strukturą typu `struct nlattr` (listing 1)

Listing 1: Typ strukturalny `struct nlattr`

```
struct nlattr {
        __u16           nla_len;
        __u16           nla_type;
};
```

Dwa najstarsze bity pola `nla_type` sygnalizują czy atrybut posiada atrybuty zagnieżdżone i czy wartość atrybutu jest zapisana w porządku bajtów, jaki przewidziany jest dla sieci. Istnieje również specjalny typ komunikatów Netlink, które służą do sygnalizowania przestrzeni użytkownika, że wystąpił błąd w przetwarzaniu otrzymanego od niej komunikatu. Ich format jest zdefiniowany strukturą `struct nlmsgerr` (listing 2).

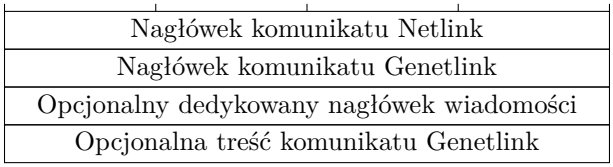
Listing 2: Typ strukturalny `struct nlmsgerr`

```
struct nlmsgerr {
        int             error;
        struct nlmsghdr msg;
};
```

Pole `error` w tej strukturze jest 32 bitowe i zawiera kod błędu, a pole `msg` komunikat, który spowodował wyjątek.

1.3. Gniazda Genetlink

Komunikaty Genetlink są umieszczone wewnątrz komunikatów Netlink. Ich ogólny format przedstawia rysunek 3. W tego typu komunikacie za nagłówkiem Netlink umieszczany jest nagłówek Genetlink,



Rysunek 3: Format komunikatu Genetlink (na podstawie `https://wiki.linuxfoundation.org/networking/generic_netlink_howto`)

po którym z kolei może występować nagłówek protokołu specyficznego dla podsystemu jądra, do którego komunikat jest wysłany. Treść komunikatu (ang. *payload*), czyli zbiór atrybutów, też jest opcjonalna. Rysunek 4 przedstawia strukturę nagłówka protokołu Genetlink. Pole „Polecenie” jest ma wielkość 8 bi-

Rysunek 4: Nagłówek komunikatu Genetlink (na podstawie `people.netfilter.org/pablo/netlink/netlink.pdf`)

tów i zawiera polecenie, które mechanizm Genetlink powinien wykonać. Możliwe są następujące wartości tego pola:

`CTRL_CMD_GETFAMILY` - to polecenie pozwala ustalić identyfikator numeryczny rodziny na podstawie jej nazwy, która jest wyrażona za pomocą łańcucha znaków, a także otrzymać zapisane w postaci atrybutów informacje o operacjach i grupach multikastowych związanych z daną rodziną;

`CTRL_CMD_GETOPS` - pozwala uzyskać zbiór operacji, które związane są z daną rodziną, numer tej rodziny należy przesłać jako atrybut wiadomości;

`CTRL_CMD_GETMCAST_GRP` - polecenie pozwala uzyskać informacje o grupach multikastowych związanych z daną rodziną.

Identyfikatory rodzin i grup multikastowych są nadawane w trakcie wykonania, więc należy uzyskać o nich informacje w początkowej fazie komunikacji przy pomocy gniazd Genetlink, posługując się nazwą określonej rodziny. Pole „Wersja" jest również 8 bitowe i zawiera numer wersji, dzięki czemu można wprowadzić zmiany w formacie, bez niebezpieczeństwa braku zachowania kompatybilności z kodem obsługującym wcześniejsze wersje formatu. Pole „Zarezerwowane" jest 16 bitowe i obecnie nie jest wykorzystywane.

2. Opis API

Opis API dla komunikacji przez gniazda Netlink i Genetlink został podzielony na dwie części. Część 2.1 opisuje API dla komunikacji Netlink, z podziałem na przestrzeń użytkownika i przestrzeń jądra. Część 2.2 przedstawia API Genetlink, również z podziałem na części dotyczącą jądra i procesów użytkownika. W instrukcji opisane są tylko wybrane elementy obu API. Więcej szczegółów można znaleźć w dokumentacji biblioteki `libnl` dostępnej pod adresem `https://www.infradead.org/~tgr/libnl/` oraz w plikach nagłówkowych jądra Linuksa `linux/netlink.h`, `net/netlink.h` i `linux/genetlink.h`

2.1. API Netlink

Ten podrozdział jest poświęcony API do obsługi komunikacji za pomocą gniazd Netlink. Jego pierwsza część opisuje API po stronie jądra systemu, a druga po stronie przestrzeni użytkownika.

2.1.1. API jądra

Nagłówek komunikatu Netlink jest opisany typem strukturalnym, który przedstawia listing 3.

Listing 3: Typ strukturalny `struct nlmsghdr`

```
1 struct nlmsghdr {
2         __u32           nlmsg_len;      /* Length of message including header */
3         __u16           nlmsg_type;     /* Message content */
4         __u16           nlmsg_flags;    /* Additional flags */
5         __u32           nlmsg_seq;      /* Sequence number */
6         __u32           nlmsg_pid;      /* Sending process port ID */
7 };
```

Znaczenie poszczególnych pól jest wyjaśnione w zamieszczonych obok nich komentarzach, a ponadto można te listing porównać z rysunkiem 2.

Listing 4 przedstawia budowę typu strukturalnego `struct nla_policy`. Struktury tego typu opisują zawartości atrybutów komunikatu i pomagają w jego interpretacji.

Listing 4: Typ strukturalny `struct nla_policy`

```
1 struct nla_policy {
2         u16             type;
3         u16             len;
4 };
```

Wartość pola `type` określa typ atrybutu. W pliku nagłówkowych `linux/netlink.h` jest zdefiniowany typ wyliczeniowy, którego elementy mogą być użyte do nadania wartości temu polu. Oto niektóre z nich:

`NLA_U8` - ośmiobitowa liczba naturalna,

`NLA_U16` - szesnastobitowa liczba naturalna,

`NLA_U32` - trzydziestodwubitowa liczba naturalna, eitem[`NLA_U64`]- sześćdziesięcioczterobitowa liczba naturalna,

`NLA_S8` - ośmiobitowa liczba całkowita,

`NLA_S16` - szesnastobitowa liczba całkowita,

`NLA_S32` - trzydziestodwubitowa liczba całkowita,

`NLA_S64` - sześćdziesięcioczterobitowa liczba całkowita,

`NLA_STRING` - ciąg znaków,

`NLA_NUL_STRING` - ciąg znaków zakończony znakiem o kodzie `ascii` równym zero.

Pole `len` zawiera rozmiar atrybutu. Jeśli typ atrybutu wyraźnie wskazuje na jego rozmiar, np. `NLA_U8`, to w strukturze go opisującej nie trzeba nadawać polu `len` wartości.

Informacje z przestrzeni użytkownika są odbierane przez moduły jądra za pomocą zdefiniowanych w nich funkcji wywoływanych zwrotnie (ang. `callback functions`). Listing 5 przedstawia typ strukturalny `netlink_kernel_cfg`. Zawiera on pole `input`, które jest wskaźnikiem na taką funkcję.

Listing 5: Typ strukturalny `struct netlink_kernel_cfg`

```
truct netlink_kernel_cfg {
        unsigned int    groups;
        unsigned int    flags;
        void            (*input)(struct sk_buff *skb);
        struct mutex    *cb_mutex;
        int             (*bind)(struct net *net, int group);
        void            (*unbind)(struct net *net, int group);
        bool            (*compare)(struct net *net, struct sock *sk);
};
```

Zatem funkcja wywoływana zwrotnie, która obsługuje odbiór komunikatów powinna mieć następujący prototyp:

```
void input_header(struct sk_buff *skb);
```

,gdzie `struct sk_buf` jest typem określającym bufor na pakiety sieciowe. Funkcji jest przekazywany wskaźnik na taki bufor, który zawiera odebraną wiadomość Netlink.

Pole `groups` w strukturze typu `struct netlink_kernel_cfg` zawiera liczbę grup multikastowych używanych przez moduł jądra w komunikacji Netlink.

Pozostałe czynności obsługi komunikacji Netlink, takie jak tworzenie i usuwanie gniazd Netlink oraz obsługa otrzymanych i wysyłanie nowych wiadomości za pomocą tych gniazd można przeprowadzić przy użyciu następujących podprogramów:

`struct sock * netlink_kernel_create(struct net *net, int unit, struct netlink_kernel_cfg *cfg)` - funkcja `inline`, która pozwala utworzyć gniazdo Netlink. Zwraca ona adres struktury typu `struct sock`, opisującej nowo utworzone gniazdo, lub `NULL` w przypadku niepowodzenia. Jako parametry wywołania przyjmuje adres zmiennej `init_net`, która jest strukturą związaną z listą urządzeń sieciowych, numer szyny oraz adres zainicjowanej zmiennej typu `struct netlink_kernel_cfg`.

`void netlink_kernel_release(struct sock *sk)` - funkcja, która usuwa gniazdo Netlink. Nie zwraca ona żadnej wartości, a jako argument wywołania przyjmuje adres struktury opisującej gniazdo, które ma być usunięte.

struct nlmsghdr *nlmsg_hdr(const struct sk_buff *skb) - funkcja **inline**, która zwraca adres nagłówka komunikatu Netlink z bufora pakietu, którego adres został jej przekazany jako argument wywołania.

NLMSG_ALIGN(len) - makro, które pozwala odczytać z pola **nlmsg_len** nagłówka całkowitą wielkość komunikatu.

NLMSG_DATA(nlh) - makro, które zwraca, jako wartość typu **void *** adres treści komunikatu. Jego argumentem jest adres nagłówka komunikatu.

int netlink_rcv_skb(struct sk_buff *skb, int (*cb)(struct sk_buff *, struct nlmsghdr *)) - funkcja ta dokonuje wstępnego sprawdzenia poprawności otrzymanego komunikatu Netlink i wywołuje funkcję wskazywaną przez jej parametr **cb** celem jego przetworzenia, o ile jest poprawny. Po zakończeniu tej funkcji **netlink_rcv_skb()** usuwa komunikat z bufora. Z analizy jej kodu źródłowego wynika, że zawsze zwraca ona zero, ale wysyła do przestrzeni użytkownika komunikat zawierający kod błędy wykonania funkcji wskazywanej przez **cb**. Ta ostania funkcja powinna mieć następujący prototyp:

```
int cb(struct sk_buff *buffer, struct nlmsghder *header)
```

i zwracać wartość **0** w przypadku powodzenia przetwarzania komunikatu lub wartość ujemną np. **-EINTR** świadczącą o wystąpieniu wyjątku.

int nlmsg_len(const struct nlmsghdr *nlh) - funkcja **inline**, która zwraca rozmiar treści komunikatu, tj. części zawierającej atrybuty.

int nla_parse(struct nlattr **tb, int maxtype, const struct nlattr *head, int len, const struct nla_policy *policy) - funkcja ta interpretuje (parsuje) dane komunikatu, zamieniając. Wynik jej działania jest zapisywany w tablicy wskaźników na struktury typu **nlattr**, która jest przekazywana jako pierwszy argument jej wywołania. Budowa takiej struktury jest przedstawiona na listingu 1. Drugim argumentem wywołania opisywanej funkcji jest liczba atrybutów jakie przewiduje format komunikatu. Trzecim jest adres pierwszego atrybutu w komunikacie i jest to zazwyczaj ten sam adres, który zwracany jest przez makro **NLMSG_DATA**. Czwarty argument, to rozmiar treści komunikatu, który można uzyskać przy pomocy **nlmsg_len()**. Piąty to tablica struktur typu **struct nla_policy**, której zawartość odzwierciedla kolejność i budowę atrybutów komunikatu. Budowę typu strukturalnego **nla_policy** przedstawia listing 4. Jeśli komunikat ma tylko jeden atrybut, to wystarczy, aby jako piąty argument opisywanej funkcji został przekazany adres pojedynczej struktury opisującej ten argument. Opisywana funkcja zwraca **0**, jeśli interpretacja atrybutów się powiedzie, lub wartość różna od zera w przeciwnym przypadku.

u8 nla_get_u8(const struct nlattr *nla) - funkcja **inline** pozwalająca uzyskać wartość atrybutu, która jest ośmiobitową liczbą naturalną, z określonego elementu tablicy przekazanej wcześniej do wywołania funkcji **nla_parse()** jako jej pierwszy argument wywołania.

u16 nla_get_u16(const struct nlattr *nla) - funkcja działa podobnie jak **nla_get_u8()**, ale zwraca liczbę szesnastobitową.

u32 nla_get_u32(const struct nlattr *nla) - funkcja działa podobnie jak **nla_get_u8()**, ale zwraca liczbę trzydziestodwubitową.

u64 nla_get_u64(const struct nlattr *nla) - funkcja działa podobnie jak **nla_get_u8()**, ale zwraca liczbę sześćdziesięcioczterobitową.

s8 nla_get_s8(const struct nlattr *nla) - funkcja **inline** pozwalająca uzyskać wartość atrybutu, która jest ośmiobitową liczbą całkowitą, z określonego elementu tablicy przekazanej wcześniej do wywołania funkcji **nla_parse()** jako jej pierwszy argument wywołania.

s16 nla_get_s16(const struct nlattr *nla) - funkcja działa podobnie jak **nla_get_u8()**, ale zwraca liczbę szesnastobitową.

s32 nla_get_s32(const struct nlattr *nla) - funkcja działa podobnie jak **nla_get_u8()**, ale zwraca liczbę trzydziestodwubitową.

s64 nla_get_s64(const struct nlattr *nla) - funkcja działa podobnie jak **nla_get_u8()**, ale zwraca liczbę sześćdziesięcio czterobitową.

size_t nla_strlcpy(char *dst, const struct nlattr *nla, size_t dstsize) - funkcja kopiuje ciąg znaków z atrybutu opisanego strukturą typu **struct nlattr** do wskazanego bufora. Jak pierwszy argument wywołania przyjmuje adres tego bufora, jako drugi wspomnianą strukturę, która jest elementem tablicy przekazywanej jako pierwszy argument wywołania **nla_parse()**. Trzecim argumentem funkcji jest rozmiar bufora. Zwraca rozmiar przekopiowanego ciąg znaków.

W buforze docelowym przekopiowany ciąg znaków zawsze zakończony jest znakiem o kodzie ASCII równym zero.

struct sk_buff *nlmsg_new(size_t payload, gfp_t flags) -funkcja `inline`, która tworzy bufor pakietu na komunikat Netlink. Jako pierwszy argument pobiera ona rozmiar komunikatu. Jeśli nie jest on z góry znany, to jako tego argumentu można użyć stałej `NLMSG_DEFAULT_SIZE`. Drugim argumentem jest znacznik typu przydziału, np. `GFP_KERNEL`. Funkcja zwraca adres utworzonego bufora.

void nlmsg_free(struct sk_buff *skb) - usuwa z pamięci bufor utworzony przez `nlmsg_new()`.

struct nlmsghdr *nlmsg_put(struct sk_buff *skb, u32 portid, u32 seq, int type, int payload, int flags) - funkcja `inline`, która wpisuje do nagłówka komunikatu znajdującego się w buforze, którego adres został jej przekazany jako pierwszy argument wywołania, odpowiednie wartości. Drugim argumentem wywołania tej funkcji jest numer portu, do którego ma być wysłany komunikat. Trzecim argumentem jest numer sekwencyjny, a czwartym typ komunikatu (patrz rysunek 2). Piąty argument to rozmiar treści komunikatu, a szóstym flagi (patrz rysunek 2). Funkcja zwraca adres struktury nagłówka komunikatu, lub `NULL` jeśli w buforze jest za mało miejsca na komunikat.

int nla_put(struct sk_buff *skb, int attrtype, int attrlen, const void *data) - - funkcja ta pozwala dodać atrybut do komunikatu Netlink. Jako pierwszy argument przyjmuje ona adres bufora na komunikat, jako drugi stałą oznaczającą typ komunikatu, jako trzeci rozmiar tego komunikatu, a jako czwarty adres zmiennej zawierającej wartość komunikatu. Funkcja zwraca `0` w przypadku powodzenia lub wartość wyrażenia `-EMSGSIZE` w przypadku, gdyby rozmiar bufora był zbyt mały na dodanie do niego atrybutu.

void nlmsg_end(struct sk_buff *skb, struct nlmsghdr *nlh) - funkcja finalizuje tworzenie komunikatu Netlik. Jej wywołanie jest konieczne tylko wtedy, gdy do tego komunikatu były dodawane atrybuty. Przyjmuje ona dwa argumenty wywołania, które są, odpowiednio, adresem bufora na komunikat oraz adresem struktury nagłówka komunikatu. Nie zwraca ona żadnej wartości.

int nlmsg_unicast(struct sock *sk, struct sk_buff *skb, u32 portid) - funkcja, która wysyła komunikat Netlink do pojedynczego odbiorcy. Jak argumenty wywołania przyjmuje ona, kolejno: adres struktury opisującej gniazdo Netlink, adres struktury opisującej bufor pakietu i numer portu. Funkcja zwraca w przypadku pomyślnego zakończenia `0` lub wartość mniejszą od zera w przeciwnym przypadku.

2.1.2. API przestrzeni użytkownika

Dla procesów użytkownika API do obsługi komunikacji przy pomocy gniazd Netlik jest zdefiniowane w bibliotece `libnl`, choć istnieją również inne biblioteki umożliwiające taką komunikację. Wspomniana biblioteka jest jednak zalecanym standardem, dlatego zostanie w tej instrukcji częściowo opisana. Bardziej szczegółowy jej opis można znaleźć na stronie o adresie `https://www.infradead.org/~tgr/libnl/`. Funkcje, które zostaną opisane w tej części instrukcji są zdefiniowane w plikach nagłówkowych `netlink/netlink.h`, `netlink/socket.h`, `netlink/msg.h` oraz `netlink/atrr.h`. Większość z tych funkcji jest odpowiedniczkami odpowiednich funkcji z przestrzeni jądra.

struct nl_sock *nl_socket_alloc(void) - funkcja gniazdo Netlink i zwraca adres struktury typu `struct nl_sock`, która je opisuje, lub `NULL`, jeśli gniazda nie uda się utworzyć.

void nl_socket_free(struct nl_sock *sk) - funkcja zamyka gniazdo związane ze strukturą typu `struct nl_sock`, której adres jest jej przekazywany jako argument wywołania.

void nl_socket_disable_seq_check(struct nl_sock *sk) - funkcja wyłącza sprawdzanie numeru sekwencyjnego komunikatów dla gniazda związanego ze strukturą, której adres jest jej przekazywany jako pierwszy argument wywołania. Jej wywołanie jest konieczne, jeśli komunikacja między procesem użytkownika, a modułem jądra nie jest prowadzona według wzorca żądanie-odpowiedź.

void nl_socket_enable_auto_ack(struct nl_sock *sk) - funkcja włącza automatyczne potwierdzanie komunikatów dla gniazda związanego ze strukturą, której adres został jej przekazany jako argument wywołania. Tryb potwierdzania jest domyślnie włączony.

void nl_socket_disable_auto_ack(struct nl_sock *sk) - funkcja wyłącza tryb automatycznego potwierdzania komunikatów dla gniazda związanego ze strukturą, której adres jest jej przekazany jako argument wywołania.

void nl_socket_enable_msg_peek(struct nl_sock *sk) - funkcja włącza tryb sprawdzania rozmiaru następnego komunikatu dla gniazda związanego ze strukturą, której adres jest jej przekazywany jako argument wywołania. Ten tryb jest domyślnie włączony.

void nl_socket_disable_msg_peek(struct nl_sock *sk) - funkcja wyłącza tryb sprawdzania rozmiaru następnego komunikatu dla gniazda związanego ze strukturą, której adres jest jej przekazywany jako argument wywołania.

int nl_connect(struct nl_sock *sk, int bus - funkcja łączy gniazdo związane ze strukturą, której adres jest przekazywany jako pierwszy argument jej wywołania z szyną, której identyfikator (numer) jest przekazywany jej jako drugi argument wywołania. W przypadku wystąpienia wyjątku funkcja zwraca wartość mniejszą od zera, a zero w przypadku pomyślnego zakończenia łączenia.

int nl_send_auto(struct nl_sock *sk, struct nl_msg *msg) - funkcja wysyła komunikat Netlink przez gniazdo. Jako argumenty wywołania przyjmuje ona adres na strukturę związaną z gniazdem i adres struktury komunikatu. Ta funkcja zwraca zero lub dodatnia liczbę, jeśli wysyłanie zakończy się pomyślnie, lub wartość mniejszą od zera w przeciwnym przypadku.

int nl_send_simple(struct nl_sock *sk, int type, int flags, void *buf, size_t size) - jeśli wysyłany komunikat ma prostą strukturę, np. tylko jeden atrybut, to zamiast `nl_send_auto()` można użyć właśnie tej funkcji. Przyjmuje ona jako pierwszy argument wywołania adres struktury związanej z gniazdem, przez które wiadomość będzie wysłana, jako drugi argument identyfikator typu tej wiadomości, jako trzeci flagi wiadomości (najczęściej jest to `NLM_F_REQUEST`), jako czwarty adres bufora z komunikatem, a jako piąty rozmiar komunikatu. Zwraca ona wartości w ten sam sposób, co `nl_send_auto_complete()`.

int nl_socket_modify_cb(struct nl_sock *sk, enum nl_cb_type type, enum nl_cb_kind kind, nl_recvmsg_msg_cb_t func, void *arg) - funkcja ta pozwala zarejestrować dla gniazda Netlink funkcję, która będzie wywoływana na drodze wywołania zwrotnego jeśli nadejdzie przez gniazdo komunikat. Zarejestrowana funkcja będzie odpowiedzialna za interpretację odebranego komunikatu. Opisywana funkcja jako pierwszy argument wywołania przyjmuje adres struktury związanej z gniazdem, dla którego funkcja wywołania zwrotnego będzie rejestrowa. Jako drugi argument przyjmuje ona najczęściej stałą `NL_CB_VALID`, a jako trzeci argument stałą `NL_CB_CUSTOM`. Czwartym argumentem jest wskaźnik na funkcję wywołania zwrotnego. Jej prototyp jest następujący:

```
int nl_recvmsg_msg_cb(struct nl_msg *msg, void *arg)
```

Ostatnim argumentem opisywanej funkcji jest adres zmiennej zawierającej dane dla funkcji wywołania zwrotnego (najczęściej `NULL`. Zarówno opisywana funkcja, jak i funkcja wywołania zwrotnego powinny zwracać zero w przypadku pomyślnego wykonania i wartość mniejszą od zera w przeciwnym przypadku.

int nl_recvmsgs_default(struct nl_sock * sk) - funkcja ta wstrzymuje swoje działanie na gnieździe związanym ze strukturą, której adres jest jej przekazywany przez parametr, tak długo, aż pojawi się komunikat Netlink. Jeśli zajdzie takie zdarzenie, to ta funkcja uruchomi opisaną wcześniej funkcję wywołania zwrotnego. Opisywana funkcja zwraca zero po pomyślnym zakończeniu lub wartość mniejszą od zera w przeciwnym przypadku.

struct nl_msg *nlmsg_alloc(void) -funkcja przydziela pamięć i zwraca adres struktury opisującej komunikat do wysłania. Jeśli przydział się nie powiedzie zwraca wartość `NULL`.

void nlmsg_free(struct nl_msg *msg) - funkcja zwalnia pamięć przydzieloną na strukturę komunikatu przez `nlmsg_alloc()`.

struct nlmsghdr *nlmsg_put(struct nl_msg *msg, uint32_t port, uint32_t seqnr, int nlmsg_type, int payload, int nlmsg_flags) - funkcja dodaje nagłówek do komunikatu, inicjując dodatkowo jego pola. Jak pierwszy argument wywołania przyjmuje ona adres struktury komunikatu, zwrócony przez `nlmsg_alloc()`, jako drugi numer portu, jako trzeci numer sekwencyjny, jako czwarty stałą określającą typ wiadomości, jako piąty rozmiar treści komunikatu (wszystkich atrybutów), a jako szósty flagę, z jaką wysłany zostanie komunikat. Funkcja zwraca adres nagłówka, lub `NULL`, jeśli nagłówka nie uda się dodać do komunikatu.

int nla_put(struct nl_msg *msg, int attrtype, int attrlen, const void *data) - funkcja ta dodaje atrybut do komunikatu. Jako pierwszy argument wywołania przyjmuje ona adres struktury komunikatu, jako drugi typ atrybutu, jako trzeci rozmiar jego rozmiar, a jako czwarty wskaźnik na zmienną przechowującą dane tego atrybutu. W bibliotece `libnl` zdefiniowano te same stałe, co w przestrzeni jądra do określania typów atrybutów.

`int nlmsg_parse(struct nlmsghdr *nlh, int hdrlen, struct nlattr *tb[], int maxtype, struct nla_policy *policy)` - funkcja, która jest używana do interpretacji (parsowania) atrybutów w otrzymanym komunikacie. Jako pierwszy argument przyjmuje ona adres nagłówka komunikatu, jako drugi rozmiar tego nagłówka, jako trzeci tablicę struktur typu `struct nlattr` opisujących poszczególne atrybutu, jako czwarty maksymalny typ atrybutu, a jako piąty tablicę struktur typu `struct nla_policy` opisującą kolejność i format atrybutów. Jeśli komunikat posiada tylko jeden atrybut, to ostatni argument może być adresem struktury opisującej ten atrybut. Typy `struct nlattr` jest zdefiniowany podobnie jak w przestrzeni jądra, a definicja typu `struct nla_policy` podana została w listingu 6.

`uint8_t nla_get_u8(struct nlattr *hdr)` - funkcja zwraca liczbę ośmiobitową będącą wartością atrybutu. Jako argument wywołania przyjmuje ona adres struktury opisującej atrybut. Najczęściej jest to określony element tablicy przekazywanej funkcji `nlmsg_parse()` jako trzeci argument wywołania.

`uint16_t nla_get_u16(struct nlattr *hdr)` - funkcja zwraca liczbę szesnastobitową będącą wartością atrybutu. Jako argument wywołania przyjmuje ona adres struktury opisującej atrybut. Najczęściej jest to określony element tablicy przekazywanej funkcji `nlmsg_parse()` jako trzeci argument wywołania.

`uint32_t nla_get_u32(struct nlattr *hdr)` - funkcja zwraca liczbę trzydziestodwubitową będącą wartością atrybutu. Jako argument wywołania przyjmuje ona adres struktury opisującej atrybut. Najczęściej jest to określony element tablicy przekazywanej funkcji `nlmsg_parse()` jako trzeci argument wywołania.

`uint64_t nla_get_u64(struct nlattr *hdr)` - funkcja zwraca liczbę sześćdziesięcioczterobitową będącą wartością atrybutu. Jako argument wywołania przyjmuje ona adres struktury opisującej atrybut. Najczęściej jest to określony element tablicy przekazywanej funkcji `nlmsg_parse()` jako trzeci argument wywołania.

`char *nla_get_string(struct nlattr *hdr)` - funkcja ta zwraca adres ciągu znaków, który jest wartością atrybutu. Jako argument wywołania przyjmuje ona adres struktury opisującej atrybut. Najczęściej jest to określony element tablicy przekazywanej funkcji `nlmsg_parse()` jako trzeci argument wywołania.

Listing 6: Typ strukturalny `struct nla_policy`

```
1 struct nla_policy {
2         uint16_t        type;
3         uint16_t        minlen;
4         uint16_t        maxlen;
5 };
```

2.2. API Genetlink

Podobnie jak API gniazd Netlink, również API gniazd Genetlink zostanie opisane z podziałem na to dostępne dla jądra i dla przestrzeni użytkownika. Sekcja 2.2.1 zawiera opis tego pierwszego, a 2.2.2 tego drugiego.

2.2.1. API Genetlink dla jądra

W kodzie źródłowym modułu jądra, który używa gniazd Genetlink należy włączyć plik nagłówkowy `net/genetlink.h`, w którym zdefiniowane jest API jądra tych gniazd. Jedną z podstawowych struktur, jakie należy w tym module stworzyć i zainicjować jest struktura typu `struct genl_family` opisująca rodzinę, która, jak wyjaśniono to wcześniej w instrukcji, jest formą wirtualnej szyny. Definicja wspomnianego typu jest zawarta w listingu 7.

Listing 7: Typ strukturalny `struct genl_family`

```
struct genl_family {
        unsigned int            id;
        unsigned int            hdrsize;
        char                    name[GENL_NAMSIZ];
        unsigned int            version;
        unsigned int            maxattr;
        bool                    netnsok;
        bool                    parallel_ops;
        int                     (*pre_doit)(const struct genl_ops *ops,
                                            struct sk_buff *skb,
                                            struct genl_info *info);
        void                    (*post_doit)(const struct genl_ops *ops,
                                             struct sk_buff *skb,
                                             struct genl_info *info);
        int                     (*mcast_bind)(struct net *net, int group);
        void                    (*mcast_unbind)(struct net *net, int group);
        struct nlattr **        attrbuf;        /* private */
        const struct genl_ops * ops;            /* private */
        const struct genl_multicast_group *mcgrps; /* private */
        unsigned int            n_ops;          /* private */
        unsigned int            n_mcgrps;       /* private */
        unsigned int            mcgrp_offset;   /* private */
        struct list_head        family_list;    /* private */
        struct module           *module;
};
```

Struktury tego typu zawierają dużo pól, ale jeśli moduł ma odbierać i nadawać komunikaty w trybie unikastowym, to należy zainicjować tylko kilka:

id - pole określające identyfikator rodziny. Najlepiej przypisać mu wartość makra `GENL_ID_GENERATE`, bo wtedy jądro samo wygeneruje dla niego unikatową wartość.

hdrsize - rozmiar nagłówka występującego po nagłówku Genetlink. Zazwyczaj przypisuje mu się wartość 0, chyba, że chcemy stworzyć protokół wyższej warstwy osadzony w komunikatach Genetlink.

name - ciąg znaków będący nazwą rodziny. Musi on być unikatowy i jest bardzo ważny, gdyż identyfikator w polu `id` może się zmieniać w zależności od komputera, na którym moduł będzie uruchomiony, ale nazwa rodziny musi być wszędzie ta sama.

version - liczba oznaczająca numer wersji protokołu Genetlink,

maxattr - liczba atrybutów dla komunikatu przesyłanego za pomocą rodziny,

module - temu polu można (opcjonalnie) przypisać wartość makra `THIS_MODULE`.

Pozostałe pola tej struktury są prywatne, tzn. nie powinny być zmieniane w module, lub nie są istotne w przypadku komunikacji unikastowej. Kolejną istotną strukturą jest struktura typu `struct genl_ops`. Listing 8 zawiera definicję tego typu.

Listing 8: Typ strukturalny `struct genl_ops`

```
struct genl_ops {
        const struct nla_policy *policy;
        int                     (*doit)(struct sk_buff *skb,
                                        struct genl_info *info);
        int                     (*dumpit)(struct sk_buff *skb,
                                          struct netlink_callback *cb);
        int                     (*done)(struct netlink_callback *cb);
        u8                      cmd;
        u8                      internal_flags;
        u8                      flags;
};
```

Struktura ta pozwala określić operacje dla gniazda Genetlink. Jeśli takich operacji ma być kilka, to należy określić tablicę takich struktur. Pola struktury typu `struct genl_ops` mają następujące znaczenie:

policy - wskaźnik na strukturę lub tablicę takich struktur, które określają format, czyli kolejność oraz typy atrybutów w komunikacie,

doit - wskaźnik na funkcję wywołania zwrotnego, która odpowiedzialna jest za odbiór zwykłych komunikatów. Musi ona mieć nagłówek zgodny z następującym wzorcem:

```
int doit(struct sk_buff *skb, struct genl_info *infor)
```

Definicja typu struktury `genl_info` jest zawarta w listingu 9.

dumpit - wskaźnik na funkcję obsługującą komunikaty wieloczęściowe. Nie będzie w tej instrukcji dokładniej opisywana.

done - wskaźnik na funkcję, która jest wywoływana po odebraniu ostatniej części komunikatu wieloczęściowego. Nie będzie w tej instrukcji dokładniej opisywana.

cmd - identyfikator polecania, które jest przenoszone przez komunikat Genetlink.

internal_flags - używane przez rodzinę do określania wewnętrznych flag, nie należy go modyfikować w module,

flags - pole określające flagi, można mu przypisać wartość `0`.

Listing 9: Typ strukturalny `struct genl_info`

```
1  struct genl_info {
2          u32                     snd_seq;
3          u32                     snd_portid;
4          struct nlmsghdr *       nlhdr;
5          struct genlmsghdr *     genlhdr;
6          void *                  userhdr;
7          struct nlattr **        attrs;
8          possible_net_t          _net;
9          void *                  user_ptr[2];
10         struct sock *           dst_sk;
11 };
```

Struktury typu `struct genl_info` opisują właściwości odebranego komunikatu Genetlink. Znaczenie najważniejszych pól struktury tego typu jest następujące:

snd_seq - numer sekwencyjny,

snd_portid - numer portu nadawcy,

nlhdr - adres nagłówka Netlink,

genlhdr - adres nagłówka Genetlink,

userhdr - adres nagłówka protokołu użytkownika osadzonego w komunikacie Genetlink,

nlattr - tablica wskaźników na atrybuty komunikatu,

Obsługę komunikacji Genetlink w przestrzeni jądra zapewniają między innymi następujące podprogramy:

`genl_register_family_with_ops(family, ops)` - makro rejestrujące rodzinę Genetlink. Jako pierwszy argument otrzymuje strukturę opisującą rodzinę, a jako drugi tablicę struktur, lub adres pojedynczej struktury opisującej operacje. Zwraca `0` jeśli jego działanie zakończy się pomyślnie lub wartość mniejszą od zera w przeciwnym przypadku.

`int genl_unregister_family(struct genl_family *family)` - funkcja, która wyrejestrowuje rodzinę Genetlink. Jako argument wywołania przyjmuje adres struktury opisującej rodzinę do wyrejestrowania.

`struct net *genl_info_net(struct genl_info *info)` - funkcja `inline`, która umożliwia uzyskanie adresu struktury typu `struct net` ze struktury typu `genln_info`, której adres jest jej przekazywany w wywołaniu. Struktura typu `struct net` jest potrzebna przy wysyłaniu komunikatu zwrotnego.

`int genlmsg_parse(const struct nlmsghdr *nlh, const struct genl_family *family, struct nlattr *tb[], int maxtype, const struct nla_policy *policy)` - funkcja będąca odpowiedniczką funkcji `nla_parse()`. Jako pierwszy argument przyjmuje adres struktury nagłówka Netlink, jako drugi adres struktury rodziny, jako trzeci tablicę wskaźników na struktury opisujące atrybuty komunikatu, jako czwarty maksymalny typu komunikatu, a jako piąty tablicę na strukturę opisującą format argumentu lub tablicę takich struktur.

`void *genlmsg_put(struct sk_buff *skb, u32 portid, u32 seq, struct genl_family *family, int flags, u8 cmd)` - funkcja pozwalająca dodać nagłówek Genetlink do komunikatu. Nie zwraca ona żadnej wartości. Jako pierwszy argument ta funkcja przyjmuje adres bufora z komunikatem, następnie jako drugi argument pobiera numer portu, jako trzeci numer sekwencyjny komunikatu, na który odpowiedzią będzie przygotowywany przez nią komunikat. Czwartym argumentem tej funkcji jest adres struktury opisującej rodzinę, piątym flagi komunikatu, a szóstym identyfikator polecenia.

`void genlmsg_end(struct sk_buff *skb, void *hdr)` - funkcja `inline`, która jest odpowiedniczką `nlmsg_end()`. Jako pierwszy argument przyjmuje adres bufora z komunikatem. Drugim argumentem jest adres na strukturę nagłówka użytkownika. Może on mieć wartość `NULL`.

`nt genlmsg_unicast(struct net *net, struct sk_buff *skb, u32 portid)` - funkcja, która jest odpowiedniczką `nlmsg_unicast()`. Jako pierwszy argument przyjmuje adres zwrócony przez funkcję `genl_info_net()`, jako drugi adres bufora z komunikatem do wysłania, a jako trzeci numer portu.

2.2.2. API Genetlink dla procesów użytkownika

Pliki nagłówkowe zawierające API niezbędne do obsługi komunikacji Genetlink przez procesy przestrzeni użytkownika to `netlink/genl/genl.h` i `netlink/genl/ctrl.h`. W tej obsłudze są używane jest także API Netlink. Oto spis najważniejszych podprogramów związanych z obsługą gniazd Genetlink:

`int genl_connect(struct nl_sock *skb)` - funkcja, która jest odpowiedniczką `nl_connect()`.

`int genl_ctrl_resolve(struct nl_sock * sk, const char *name)` - funkcja, która zwraca identyfikator rodziny uzyskany na podstawie przekazanych jej w argumentach wywołania adresów struktury związanej z gniazdem Genetlink i łańcucha znaków będącego nazwą rodziny. Zwraca liczbę ujemną, jeśli nie zakończy się pomyślnie.

`int genl_send_simple(struct nl_sock *sk, int family, int cmd, int version, int flags)` - funkcja, która jest odpowiedniczką `nl_send_simple()`. Wysyła ona prosty komunikat Genetlink. Jako pierwszy argument przyjmuje adres struktury związanej z gniazdem Genetlink, jako drugi identyfikator rodziny, jako trzeci identyfikator polecenia, jako czwarty numer wersji protokołu, a jako piąty flagi komunikatu. Funkcja zwraca `0` w przypadku pomyślnego wykonania, lub liczbę ujemną w przeciwnym przypadku.

`int genlmsg_parse(struct nlmsghdr *nlh, int hdrlen, struct nlattr *tb[], int maxtype, struct nla_policy * policy)` - odpowiedniczka `nlmsg_parse()`.

`struct genlmsghdr* genlmsg_hdr(struct nlmsghdr *nlh)` - funkcja zwraca adres struktury nagłówka Genetlink komunikatu na podstawie przekazanego jej jako argument wywołania adresu nagłówka Netlink komunikatu.

`int genlmsg_len(const struct genlmsghdr *gnlh)` - funkcja zwraca rozmiar treści komunikatu Genetlink, na podstawie przekazanego jej adresu jego nagłówka.

`void* genlmsg_put(struct nl_msg *msg, uint32_t port, uint32_t seq, int family, int hdrlen, int flags, uint8_t cmd, uint8_t version)` - funkcja dodaje nagłówek Genetlink do komunikatu. Przyjmuje ona jako pierwszy argument adres struktury opisującej komunikat Netlink, jako drugi argument port, jako trzeci numer sekwencyjny, jako czwarty identyfikator rodziny, jako piąty rozmiar nagłówka protokołu użytkownika (`0` jeśli taki protokół nie jest używany), jako szósty flagi komunikatu, jako siódmy polecenie, a jako ósmy numer wersji protokołu. Funkcja

zwraca wskaźnik do dodanego nagłówka Genetlink lub `NULL` jeśli jej wywołanie zakończy się niepowodzeniem.

3. Przykład

Przykłady programów użytkownika i modułów jądra korzystających z gniazd Netlink i Genetlink można znaleźć w artykule dostępnym pod adresem `http://people.netfilter.org/pablo/netlink/netlink-libmnl-manual.pdf`, ale są one przeznaczone dla starszych wersji jądra systemu i wymagają dostosowania, aby mogły być użyte z nowszymi. Ponadto programy dla przestrzeni użytkownika używają biblioteki `libmnl` zamiast `libnl`.

Listing 10 zawiera kod źródłowy modułu, który odbiera komunikat Netlink i informacje o nim, wraz z wartością atrybutu, umieszcza w pliku `/proc/netlink`[1]. Kod źródłowy powiązanego z nim programu dla przestrzeni użytkownika zawarty jest w listingu 11.

Listing 10: Moduł jądra odbierający dane z użyciem gnizada Netlink (plik `netlink.c`)

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/fs.h>
#include<linux/seq_file.h>
#include<linux/proc_fs.h>
#include<linux/netlink.h>
#include<net/netlink.h>
#include<net/sock.h>

#define NETLINK_TEST 17

static int error_level;
static char content[128];
static struct sock *sock;

static int netlink_show(struct seq_file *seq, void *v)
{
        seq_printf(seq,content);
        return 0;
}

static int netlink_seq_open(struct inode *inode, struct file *file)
{
        return single_open(file,netlink_show,NULL);
}

static struct file_operations netlink_proc_fops = {
        .owner = THIS_MODULE,
        .open = netlink_seq_open,
        .read = seq_read,
        .llseek = seq_lseek,
        .release = single_release,
};

static void netlink_receive_skb(struct sk_buff *skb)
{
        struct nlmsghdr *nlh;
        uint32_t rlen;

```

[1] Proszę nie mylić z plikiem `/proc/net/netlink`, który jest plikiem standardowo tworzonym przez jądro systemu i zawiera informacje statystyczne o podsystemie Netlink.

```
40          nlh = nlmsg_hdr(skb);
41          rlen = NLMSG_ALIGN(nlh->nlmsg_len);
42          sprintf(content,
43          "Received message: \"%s\", sequence number: %u\n",(char *)NLMSG_DATA(nlh),nlh->nlmsg_seq);
44          skb_pull(skb,rlen);
45  }
46
47  static struct netlink_kernel_cfg netlink_configuration = {
48          .groups = 0,
49          .input = netlink_receive_skb,
50          .cb_mutex = NULL,
51          .bind = NULL,
52          .unbind = NULL,
53          .compare = NULL,
54  };
55
56  static int __init netlink_start(void)
57  {
58          if(proc_create("netlink",0,NULL,&netlink_proc_fops)==NULL) {
59                  printk(KERN_ALERT "Error creating procfs entry!\n");
60                  error_level=1;
61                  return -1;
62          }
63          sock = netlink_kernel_create(&init_net,NETLINK_TEST, &netlink_configuration);
64          if(!sock) {
65                  printk(KERN_ALERT "Error creating netlink socket!\n");
66                  error_level=2;
67                  return -2;
68          }
69          return 0;
70  }
71
72  static void __exit netlink_stop(void)
73  {
74          if(error_level!=1)
75                  remove_proc_entry("netlink",NULL);
76          if(error_level!=2)
77                  netlink_kernel_release(sock);
78  }
79
80  module_init(netlink_start);
81  module_exit(netlink_stop);
82  MODULE_LICENSE("GPL");
83  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
84  MODULE_DESCRIPTION("A user space to kernel space simple communication module");
85  MODULE_VERSION("1.0");
```

Wiersze 1-8 kodu źródłowego opisywanego modułu zawierają instrukcje włączające pliki nagłówkowe. W wierszach 3-5 dodane są plik nagłówkowe związane z obsługą systemu `procfs`, a w wierszach 6-8 pliki nagłówkowe związane z obsługą komunikacji Netlink. W wierszu nr 10 zdefiniowana jest stała określająca identyfikator szyny Netlink, która zostanie użyta do komunikacji. Wiersz nr 12 zawiera zmienną, która będzie zawierała kod wykonania poszczególnych funkcji wywoływanych w module. Wiersz nr 13 zwiera definicję bufora dla pliku z systemu plików `procfs`, a w wierszu nr 14 zadeklarowany jest wskaźnik na strukturę gniazda sieciowego. Wiersze 16-20 zawierają definicję funkcji, która będzie wywoływana przy odczycie pliku z systemu plików `procfs`. Ponieważ obsługa tych plików była tematem instrukcji czwartej, to elementy modułu z nią związane nie będą tu dokładniej opisywane. Wiersze 22-25 zawierają z kolei definicję funkcji wywoływanej przy otwieraniu wspomnianego pliku. Struktura operacji dla pliku z systemu plików `procfs` jest zadeklarowana i zainicjowana w wierszach 27-33. Funkcja `netlink_receive_skb()`, zdefiniowana w wierszach 35-45, będzie wywoływana zwrotnie po nadesłaniu

komunikatu Netlink z przestrzeni użytkownika. Jej zadaniem będzie reakcja na pojawienie się tego komunikatu i interpretacja jego treści. Posiada ona jeden parametr, przez który otrzymuje adres bufora zawierającego komunikat. W wierszu nr 37 został zadeklarowany lokalny wskaźnik na nagłówek komunikatu Netlink, a w wierszu nr 38 zmienna lokalna, która będzie przechowywała rozmiar tego komunikatu. Obie te wartości zapisywane są we wspomnianych zmiennych odpowiednio w wierszu nr 40 i nr 41. Następnie przy pomocy `sprintf()` zapisuje do bufora pliku z systemu plików `procfs` komunikat, który zawiera ciąg znaków otrzymany przez gniazdo Netlink z przestrzeni użytkownika. Ponieważ moduł spodziewa się tylko jednego atrybuty wiadomości, który właśnie jest tym ciągiem, to po prostu uzyskuje jego adres jako adres danych komunikatu Netlink, korzystając z makra `NLMSG_DATA`. W pliku z systemu plików `procfs` zapisywany jest także numer sekwencyjny komunikatu Netlink, pobrany z jego nagłówka. Ponieważ funkcja `netlink_receive_skb()` nie korzysta z funkcji `netlink_recv_skb()`, to musi bezpośrednio wywołać funkcję `skb_pull()`, która usuwa z bufora odebrany komunikat. Jako argumenty wywołania przyjmuje ona wskaźnik do bufora oraz rozmiar usuwanych danych. Zwraca ona adres kolejnych danych do odebrania z bufora, ale ta wartość jest w funkcji `netlink_recv_skb()` ignorowana. W wierszach 47-54 deklarowana i inicjowana jest struktura konfiguracji dla gniazda Netlink. Większość jej pól jest inicjowana wartością `NULL`. Wyjątkiem są pola `groups`, do którego jest zapisywana wartość `0`, bowiem moduł nie używa komunikacji multikastowej, oraz `input`, do którego zapisywany jest adres funkcji `netlink_recv_skb()`. W wierszu nr 58 wywoływana jest funkcja tworząca plik `/proc/netlink`, a w wierszu 63 wywoływana jest funkcja tworząca gniazdo Netlink. Do tej funkcji, jako argumenty wywołania są przekazywane, odpowiedni, adres struktury związanej z listą urządzeń sieciowych, stała będąca identyfikatorem szyny oraz adres struktury konfiguracyjnej gniazda Netlink. W wierszu nr 75 usuwany jest z systemu plik `/proc/netlink`, a w wierszu nr 77 wywoływana jest funkcja zamykająca gniazdo Netlink.

Listing 11 zawiera kod źródłowy programu użytkownika, który nadaje komunikat Netlink do jądra.

Listing 11: Program użytkowy wysyłający dane z użyciem gniazda Netlink (plik `nl.c`)

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<netlink/netlink.h>
4
5  #define NETLINK_TEST 17
6  #define MSG_STR 0x11
7
8  int main(void)
9  {
10         struct nl_sock *ns;
11         char msg[] = "Hello netlink";
12         ns = nl_socket_alloc();
13         if(!ns) {
14                 fprintf(stderr,"Błąd tworzenia gniazda netlink: %s\t%d.\n",__FILE__,__LINE__-2);
15                 return -1;
16         }
17         printf("Mój port: %u\n",nl_socket_get_local_port(ns));
18         printf("Port odbiorcy: %u\n",nl_socket_get_peer_port(ns));
19         printf("Długość wiadomości: %u\n",sizeof(msg));
20
21         nl_socket_disable_seq_check(ns);
22         nl_socket_disable_auto_ack(ns);
23         nl_socket_disable_msg_peek(ns);
24
25         int result = nl_connect(ns,NETLINK_TEST);
26          if(result<0) {
27                 fprintf(stderr,"Błąd łączenia gniazda netlink: %s\t%d\t%d.\n",__FILE__,__LINE__-2,result);
28                 nl_socket_free(ns);
29                 return -2;
30         }
31
```

```
32          result = nl_send_simple(ns,MSG_STR,NLM_F_REQUEST,(void *)msg,sizeof(msg));
33          if(result<0)
34                  fprintf(stderr,"Błąd wysyłania wiadomości: %s\t%d\t%d\n",__FILE__,__LINE__,result);
35          else
36                  printf("Wysłano %d bajtów wiadomości.\n",result);
37
38          nl_socket_free(ns);
39          return 0;
40  }
```

Wiersz nr 3 w tym kodzie źródłowym zawiera instrukcję włączającą plik nagłówkowy biblioteki `libnl` udostępniający API do obsługi komunikacji Netlink. W wierszu nr 5 zdefiniowano stałą będącą identyfikatorem szyny. Jest on taki sam, jak w przestrzeni jądra. W wierszu nr 6 zdefiniowano stałą określającą typ komunikatu, który nie jest standardowym typem. Program będzie wysyłał do jądra komunikat zawierający pojedynczy atrybut będący łańcuchem znaków. Zmienna przechowująca ten łańcuch jest zdefiniowana i zainicjowana w wierszu nr 11. W wierszu nr 10 jest z kolei zadeklarowany wskaźnik na strukturę gniazda Netlink. Gniazdo to tworzonej jest w wierszu nr 12. Proszę zwrócić uwagę, że do obsługi wyjątków zostały wykorzystane w tym programie makra `__FILE__` oraz `__LINE__`. Pierwsze z nich określa nazwę pliku, w którym pojawił się wyjątek, a drugie numer wiersza. W programie wartość tego drugiego marka jest pomniejszana o dwa, gdyż wyjątek pojawia się przy wywołaniu funkcji znajdującym się dwa wiersze wcześniej od tego, w którym to makro jest użyte.

W wierszach nr 17 i nr 18 program wypisuje na ekranie numery portów. Są to, odpowiednio, port nadawcy i port odbiorcy. Można je uzyskać ze struktury gniazda poprzez wywołania, odpowiednio, funkcji `nl_socket_get_local_port()` i `nl_socket_get_peer_port()`.

W wierszach 21-23 wyłączane jest sprawdzanie numeru sekwencyjnego komunikatu, automatycznego potwierdzania oraz uzyskiwania informacji o rozmiarze nieodebranego komunikatu.

W wierszu nr 25 gniazdo jest łączone z szyną. W wierszu nr 32 przez to samo gniazdo jest wysyłany komunikat. Funkcja `nl_send_simple_auto()`, która przyprowadza tę operację jako pierwszy argument przyjmuje adres struktury gniazda, jako drugi identyfikator typu komunikatu. Jako trzeci argument przyjmuje ona flagę komunikatu Netlink, a jako czwarty adres zmiennej z komunikatem. Ostatnim argumentem jej wywołania jest rozmiar tej zmiennej.

W wierszu nr 38 programu gniazdo Netlink jest zamykane.

Listing 12 zawiera treść pliku konfiguracyjnego dla narzędzia `make`, aby skompilowało ono oba kody źródłowe. Proszę zwrócić uwagę, że wprowadzono do niego nową regułę o nazwie `default`, której wykonanie powoduje skompilowanie modułu jądra, ale wymaga wcześniejszego wykonania reguły `nl`, która z kolei kompiluje program użytkownika. W poleceniu wywołującym kompilator GCC użyto wywołania polecenia `pkg-config` z flagami `--cflags --libs` i argumentem `libnl-3.0`. Pierwsza flaga doda do wywołania kompilatora flagę informującą go o katalogu, w którym znajdują się niezbędne do kompilacji programu pliki nagłówkowe biblioteki `libnl`, a druga doda flagę dla programu konsolidującego (ang. *linker*), aby ten wiedział, że będzie używana biblioteka ładowana dynamicznie `libnl` w wersji 3.0. Proszę również zwrócić uwagę, że stworzono regułę `disclean`, która pozwala usunąć nie tylko plik ze skompilowanym modułem, ale również plik ze skompilowanym programem użytkownika.

Listing 12: Plik `Makefile` do kompilacji plików `netlink.c` i `nl.c`

```
1  obj-m := netlink.o
2  KERNELDIR = /lib/modules/$(shell uname -r)/build
3
4  default: netlink.c  nl
5          $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
6
7  nl: nl.c
8          gcc -Wall -o nl nl.c $(shell pkg-config --cflags --libs libnl-3.0)
9
10 distclean:
11         $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
12         rm -f nl
```

Po skompilowaniu obu plików należy załadować moduł do jądra systemu, a następnie uruchomić program `nl`. Po jego wykonaniu można zobaczyć informację o przesłanym przed niego do jądra komunikacie używając polecenia:

```
cat /proc/netlink
```

Program `nl` można również uruchomić w trybie debugowania, nakazując mu wypisanie zawartości przesyłanych komunikatów. Ten efekt można uzyskać przy pomocy następującego polecenia:

```
NLCB=debug ./nl
```

Dodatkowo, funkcje z biblioteki `libnl` mogą wypisywać na ekran informacje diagnostyczne, jeśli program zostanie uruchomiony następująco:

```
NLDBG=2 ./nl
```

Wartości jakie może przyjmować zmienna środowiskowa `NLDBG` są następujące (po myślniku podano opis ich znaczenia):

`0` - debugowanie wyłączone (wartość domyślna),

`1` - ostrzeżenia, informacje o ważniejszych zdarzeniach, powiadomienia,

`2` - mniej lub bardziej ważnie informacje debugowania,

`3` - informacje o powtarzających się zdarzeniach,

`4` - nawet najmniej ważne informacje,

Proszę zwrócić uwagę, że każda następna wartość oznacza poziom logowania rozszerzający poprzedni, a więc wyświetlający na ekranie więcej informacji. Oba opisane polecenia można połączyć np. tak:

```
NLCB=debug NLDBG=2 ./nl
```

Listing 13 zawiera kod źródłowy modułu jadra, który odbiera komunikat Genetlink zawierający ciąg znaków wysłany przez proces z przestrzeni użytkownika. Kod źródłowy programu realizowanego przez ten proces znajduje się w listingu 14.

Listing 13: Moduł jądra odbierający dane z użyciem gniazda Genetlink (plik `genetlink.c`)

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/fs.h>
#include<linux/seq_file.h>
#include<linux/proc_fs.h>
#include<linux/netlink.h>
#include<net/netlink.h>
#include<net/genetlink.h>
#include<net/sock.h>

#define TEST_GENETLINK_CMD 1
#define TEST_GENETLINK_A_MSG 1
#define TEST_GENETLINK_A_MAX 1

static int error_level;
static char content[128];

static struct genl_family test_genetlink_family = {
        .id = GENL_ID_GENERATE,
        .hdrsize = 0,
        .name = "GENETLINK_TEST",
        .version = 1,
        .maxattr = TEST_GENETLINK_A_MAX,
};

static struct nla_policy test_genetlink_policy[TEST_GENETLINK_A_MAX+1];
```

```
27
28  static int test_genetlink_doit(struct sk_buff *skb, struct genl_info *info)
29  {
30          if(info)
31                  if(info->attrs)
32                          sprintf(content,"Received message: \"%s\", sequence number: %u port ID: %d\n",
33                          (char *)nla_data(info->attrs[TEST_GENETLINK_A_MSG]),info->snd_seq,info->snd_portid);
34          return 0;
35  }
36
37  static struct genl_ops test_genetlink_ops[] =
38  {
39          {
40                  .cmd = TEST_GENETLINK_CMD,
41                  .flags = 0,
42                  .policy = test_genetlink_policy,
43                  .doit = test_genetlink_doit,
44                  .dumpit = NULL,
45          },
46  };
47
48  static int genetlink_show(struct seq_file *seq, void *v)
49  {
50          seq_printf(seq,content);
51          return 0;
52  }
53
54  static int genetlink_seq_open(struct inode *inode, struct file *file)
55  {
56          return single_open(file,genetlink_show,NULL);
57  }
58
59  static struct file_operations genetlink_proc_fops = {
60          .owner = THIS_MODULE,
61          .open = genetlink_seq_open,
62          .read = seq_read,
63          .llseek = seq_lseek,
64          .release = single_release,
65  };
66
67  static int __init genetlink_start(void)
68  {
69          test_genetlink_policy[TEST_GENETLINK_A_MSG].type = NLA_STRING;
70          test_genetlink_policy[TEST_GENETLINK_A_MSG].len = 31;
71          if(proc_create("genetlink",0,NULL,&genetlink_proc_fops)==NULL) {
72                  printk(KERN_ALERT "Error creating procfs entry!\n");
73                  error_level=1;
74                  return -1;
75          }
76          if(genl_register_family_with_ops(&test_genetlink_family,test_genetlink_ops)) {
77                  error_level=2;
78                  return -1;
79          }
80          return 0;
81  }
82
83  static void __exit genetlink_stop(void)
84  {
85          if(error_level!=1)
86                  remove_proc_entry("genetlink",NULL);
```

```
87          if(error_level!=2)
88                  if(genl_unregister_family(&test_genetlink_family))
89                          printk(KERN_ALERT "Error occured while unregistering genetlink family!\n");
90  }
91
92  module_init(genetlink_start);
93  module_exit(genetlink_stop);
94  MODULE_LICENSE("GPL");
95  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
96  MODULE_DESCRIPTION("A user space to kernel space simple genetlink communication module.");
97  MODULE_VERSION("1.0");
```

W kodzie źródłowym modułu w wierszu nr 8 włączany jest plik nagłówkowy związany z obsługa gniazd Genetlink. Pozostałe używane pliki nagłówkowe są takie same, jak w poprzednio opisywanym module.

Wiesz nr 11 zawiera definicję stałej, która określa identyfikator polecenia dla nagłówka Genetlink. Kolejny wiersz (nr 12) zawiera definicję stałej określającej położenie atrybutu tego komunikatu. Będzie używany tylko jeden atrybut, który jest ciągiem znaków. Wiersz nr 13 zawiera definicję stałej określającej liczbę atrybutów (ponownie - jeden).

Wiersze 18-24 zawierają deklarację i inicjację struktury określającej rodzinę używaną w komunikacji Genetlink. Jej polu `id` przypisywana jest wartość makra `GENL_ID_GENERATE`, dzięki czemu jądro automatycznie nada mu wartość pierwszego wolnego identyfikatora rodziny. W komunikacji nie będzie wykorzystywany nagłówek protokołu użytkownika, dlatego polu określającemu jego wielkość (wiersz nr 20) nadawana jest wartość `0`. Pole `version` otrzymuje wartość `1` (wiersz nr 22), co oznacza pierwszą wartość wersji rodziny. Rodzinie jest nadawana nazwa `GENETLINK_TEST` w wierszu nr 21. Liczba atrybutów tej rodziny jest określana w wierszu nr 23. W wierszu nr 26 jest deklarowana tablica struktur opisujących format atrybutów wiadomości Genetlink. W przypadku opisywanego modułu zawiera ona tylko jeden element, bo komunikat będzie miał tylko jeden atrybut. Wartości temu elementowi nadawane są w wierszach nr 69 i 70. Będzie to zatem atrybut będący ciągiem znaków, o maksymalnym rozmiarze wynoszącym `31` bajtów.

Wiersze 28-35 zawierają definicję funkcji `test_genetlink_doit()`, która będzie wywoływana zwrotnie, kiedy proces użytkownika wyśle do jądra komunikat. Treść tego komunikatu funkcja pobierze ze struktury typu `struct genl_info`, na którą adres zostanie jej przekazany. Funkcja po sprawdzeniu, czy adres tej struktury jest różny od `NULL` (wiersz nr 30) oraz po przeprowadzeniu analogicznego testu dla wskaźnika na tablicę atrybutów (wiersz nr 31) zapisze do bufora pliku `/proc/genetlink` informacje o otrzymanym komunikacie, wraz z wartością jego atrybutu. Wartość ta zostanie uzyskana bezpośrednio z tablicy atrybutów (wiersz nr 33) przy pomocy funkcji `nla_data()`. Funkcja ta ma działanie bardzie ogólne od wcześniej opisanych funkcji zwracających wartości atrybutów komunikatu. Zwraca ona adres (wartość typu `void *`) pamięci zawierającej wartość atrybutu, nie określając jej typu. Musi to zrobić programista, stosując rzutowanie. W przypadku opisywanego modułu jest to konwersja na ciąg znaków. Innymi informacjami o komunikacie Genetlink dodawanymi do pliku `/proc/genetlink` są: numer sekwencyjny komunikatu i numer portu. Funkcja `test_genetlink_doit()` kończy swoje działanie zwracając wartość `0`.

Wiersze 37-46 zawierają definicję tablicy operacji związanych z rodziną. Ta tablica również będzie zawierała tylko jeden element, bo tylko jedna operacja została przewidziana dla tej rodziny. Identyfikator tej operacji (polecenia) jest zapisywany w tym elemencie w wierszu nr 40. W wierszu nr 41 zerowane jest pole flag, a w wierszu nr 42 w polu `policy` elementu zapisywany jest adres tablicy określającej format atrybutów. Adres funkcji wywoływanej zwrotnie po otrzymaniu komunikatu Genetlink jest zapisywany w polu `doit` elementu w wierszu nr 43. Wartość `NULL` jest zapisywana z kolei w polu `dumpit`, gdyż moduł nie będzie odbierał komunikatów wieloczęściowych.

Wiersze 48-65 zawierają definicje i deklaracje elementów modułu związanych z obsługą pliku `/proc/genetlink`. Są one zapisana analogicznie, jak ich odpowiedniki w poprzednio opisywanym module i także nie będą tutaj dokładniej opisywane.

W wierszu nr 76 modułu jest rejestrowana rodzina Genetlink. To wystarcza do odebrania komunikatu pochodzącego od procesu użytkownika. W funkcji finalizującej modułu, w wierszu nr 88 ta rodzina jest wyrejestrowywana.

Listing 14 zawiera kod źródłowy programu, który wysyła komunikat Genetlink do modułu jądra.

Listing 14: Program użytkowy wysyłający dane z użyciem gniazda Genetlink (plik `gl.c`)

```c
#include<stdio.h>
#include<string.h>
#include<netlink/netlink.h>
#include<netlink/msg.h>
#include<netlink/attr.h>
#include<netlink/genl/genl.h>
#include<netlink/genl/ctrl.h>

#define TEST_GENETLINK_CMD 1
#define TEST_GENETLINK_A_MSG 1

int main(void)
{
        char msg[] = "Hello genetlink";
        struct nl_sock *ns = NULL;
        ns = nl_socket_alloc();
        if(!ns) {
                fprintf(stderr,"Błąd tworzenia gniazda netlink: %s\t%d.\n",__FILE__,__LINE__-2);
                return -1;
        }
        printf("Mój port: %u\n",nl_socket_get_local_port(ns));
        printf("Port odbiorcy: %u\n",nl_socket_get_peer_port(ns));
        printf("Długość wiadomości: %u\n",sizeof(msg));

        nl_socket_disable_seq_check(ns);
        nl_socket_disable_auto_ack(ns);
        nl_socket_disable_msg_peek(ns);

        int result = genl_connect(ns);
         if(result<0) {
                fprintf(stderr,"Błąd łączenia gniazda netlink: %s\t%d\t%d.\n",__FILE__,__LINE__-2,result);
                nl_socket_free(ns);
                return -2;
        }

        int family = genl_ctrl_resolve(ns,"GENETLINK_TEST");
        if(family<0) {
                fprintf(stderr,"Błąd uzyskiwania numeru rodziny: %s\t%d\t%d\n",__FILE__,__LINE__-2,family);
                return -3;
        }
        printf("Numer rodziny: %d\n",family);

        struct nl_msg *message = NULL;
        message = nlmsg_alloc();
        if(message==NULL) {
                fprintf(stderr,"Błąd przydziału pamięci na komunikat %s\t%d\n",__FILE__,__LINE__-2);
                nl_socket_free(ns);
                return -3;
        }

        void *gen_hdr = NULL;
        gen_hdr=genlmsg_put(message,0,NL_AUTO_SEQ,family,0,0,TEST_GENETLINK_CMD,1);
        if(gen_hdr==NULL) {
                fprintf(stderr,"Błąd tworzenia nagłówka komunikatu %s\t%d\n",__FILE__,__LINE__-2);
                nlmsg_free(message);
                nl_socket_free(ns);
```

```
57                  return -4;
58          }
59
60          result = nla_put_string(message,TEST_GENETLINK_A_MSG,msg);
61
62          if(result<0) {
63                  fprintf(stderr,"Błąd dodawania wiadomości do komunikatu %s\t%d\t%d\n",
64                                                                          __FILE__,__LINE__-2,result);
65                  nlmsg_free(message);
66                  nl_socket_free(ns);
67                  return -5;
68          }
69
70          result = nl_send_auto(ns,message);
71          if(result<0) {
72                  fprintf(stderr,"Błąd wysyłania wiadomości do jądra %s\t%d\t%d\n",__FILE__,__LINE__-2,result);
73                  nlmsg_free(message);
74                  nl_socket_free(ns);
75                  return -6;
76          }
77          printf("Wysłano %d bajtów do jądra\n",result);
78
79          nlmsg_free(message);
80          nl_socket_free(ns);
81          return 0;
82  }
```

W tym kodzie, w wierszach nr 6 i nr 7 włączone są pliki nagłówkowe biblioteki `libnl`, umożliwiające obsługę komunikatów Genetlink. W wierszu nr 9 zdefiniowano stałą określającą identyfikator polecenia, a w wierszu nr 8 stałą określającą położenie atrybutu komunikatu. Proszę zwrócić uwagę, że są to takie same stałe, jak te zdefiniowane w kodzie źródłowym modułu jądra. W stosunku do programu użytkownika wysyłającego komunikat Netlink, ten program zawiera kilka dodatkowych elementów. W wierszu nr 36, w zmiennej `family` zapisywany jest uzyskany od jądra systemu identyfikator rodziny. Ten identyfikator pozyskiwany jest przy pomocy nazwy rodziny, która nadana jej jest przez moduł jądra. W wierszu nr 43 tworzona jest struktura komunikatu Netlink. W wierszu nr 52 do tej struktury dodawany jest nagłówek komunikatu Genetlik, a w wierszu nr 60 dodatkowo dodawany jest do komunikatu atrybut będący ciągiem znaków. Komunikat Genetlik jest wysyłany przez program do jądra systemu poprzez wywołanie w wierszu nr 70 funkcji `nl_send_auto()`. Przed zakończeniem programu zwalniana jest pamięć przeznaczona na strukturę komunikatu. Ta operacja jest wykonywana w wierszu nr 79.

Zawartość pliku `Makefile` (listing 15) jest analogiczna jak w przypadku pliku o tej samej nazwie do kompilacji kodów źródłowych dotyczących komunikacji Netlink. W przypadku kodów związanych z komunikacją Genetlik, polecenie `pkg-config` używane w regule `gl` zyskuje dodatkowy argument. Jest nim nazwa biblioteki `libnl-genl-3.0`.

Listing 15: Plik `Makefile` do kompilacji plików `genetlink.c` i `gl.c`

```
1   obj-m := genetlink.o
2   KERNELDIR ?= /lib/modules/$(shell uname -r)/build
3
4   default: genetlink.c gl
5           $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
6
7   gl: gl.c
8           gcc -Wall -o gl gl.c $(shell pkg-config --cflags --libs libnl-3.0 libnl-genl-3.0)
9
10  distclean:
11          $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
12          rm -f gl
```

Po skompilowaniu modułem i programem można się posługiwać analogicznie, jak w przypadku modułu i programu opisanych wcześniej. Proszę pamiętać, że informacje o otrzymanym komunikacie ten moduł jądra będzie zapisywał w pliku `/proc/genetlink`.

Zadania

1. `[3 punkty]` Podziel kod źródłowy programu w pliku `nl.c` na funkcje z parametrami.
2. `[5 punktów]` Zmień moduł i program komunikujące się za pomocą gniazda Netlink, tak aby program wysyłał komunikat zawierający pojedynczy atrybut będący ośmiobitową liczą naturalną, a moduł żeby go poprawnie odbierał.
3. `[7 punktów]` Zmień moduł i program komunikujące się za pomocą gniazda Netlink, tak aby w odpowiedzi na otrzymany komunikat moduł jądra wysłał inny komunikat do procesu użytkownika, a ten żeby go poprawnie odebrał.
4. `[3 punkty]` Podziel kod źródłowy programu w pliku `gl.c` na funkcje z parametrami.
5. `[5 punktów]` Zmień moduł i program komunikujące się za pomocą gniazda Genetlik, tak aby program wysyłał komunikat zawierający pojedynczy atrybut będący ośmiobitową liczą naturalną, a moduł żeby go poprawnie odbierał.
6. `[7 punktów]` Zmień moduł i program komunikujące się za pomocą gniazda Genetlink, tak aby w odpowiedzi na otrzymany komunikat moduł jądra wysłał inny komunikat do procesu użytkownika, a ten żeby go poprawnie odebrał.