*Netlink is a Linux kernel socket interface, which is used for inter-process communication between the user space and the kernel, and also between different user space processes. It reduces dependence on system calls, ioctls and proc files. Netlink also helps preserve kernel purity.*

There are various ways by which processes in the user space can communicate with the kernel. These are system calls, ioctl and the proc file system. The problem with system calls is that they are linked statically to kernel code. So, any new feature that is to be provided has to be compiled with the kernel, but with dynamic linking modules such as device drivers, any feature that these modules want to provide cannot always be preconfigured as a system call. Similarly, for every new feature, it is hard to provide communication through the file system. All these communication mechanisms also require that the processes initiate the communication  the kernel cannot initiate the communication.

****Netlink socket****  
Netlink socket is a communication mechanism used between the user space processes and also for communication between processes and the kernel. It can also be used for communication between user space threads and the kernel. It is a full duplex communication mechanism, that is, the kernel itself can initiate the communication. One of the advantages of this mechanism is that, in the user space, popular socket APIs that software programmers are familiar with are used for Netlink communication, so no new study is required. Netlink sockets are easier to add than system calls, ioctls and proc filesall of which will start polluting the kernel. If these are added for every new feature, then later on, it becomes difficult to remove these features — a problem kernel developers are facing currently in the case of the proc file system. In case of Netlink sockets, only a protocol type macro needs to be inserted in the*netlink.h file*, which resides in*include/uapi/linux/netlink.h,* and processes and the kernel can start communicating immediately through the socket API. Netlink socket is an asynchronous communication method, that is, it queues the messages to be sent in the receivers Netlink queue. One of the features of a Netlink socket is that it also supports multicast communication, i.e., one process can send a message to a Netlink group address, and many processes can listen on this group address. Since in the user space it is implemented through the socket API, this is an easy-to-use communication mechanism.

****The basics of Netlink sockets****  
To use Netlink sockets in code, a standard socket API is used, which is as follows:

|  |
| --- |
| int socket(int domain, int type, int protocol); |

Here, *domain* specifies the protocol family used for communication which is defined in *sys/socket.h;* domain in the case of Netlink is AF\_NETLINK.  
*type* specifies the way in which communication is done. In the case of Netlink, SOCK\_RAW or SOCK\_DGRAM can be used.  
*protocol* specifies which Netlink feature is to be used. Various features are specified in *include/uapi/linux/netlink.h*, which are NETLINK\_GENERIC, NETLINK\_ROUTE, NETLINK\_FIREWALL, etc. You can also add a custom Netlink protocol easily by adding the macro in this file.  
For each Netlink protocol type, up to 32 multicast groups can be specified in code. A multicast group in Netlink is a 32-bit bitmask, where each bit represents a group. Using this multicast feature, multiple processes and kernel modules can communicate with each other with a lesser number of system calls.  
To understand Netlink sockets in the user space, the following data structures need to be understood.

|  |
| --- |
| struct sockaddr\_nl;  struct nlmsghdr;  struct iovec;  struct msghdr;    struct sockaddr\_nl (include/uapi/linux/netlink.h)  {  \_\_kernel\_sa\_family\_t nl\_family;  unsigned short nl\_pad;  \_\_u32 nl\_pid;  \_\_u32 nl\_groups;    }; |

In the above code, lets look at what certain terms stand for.

* *nl\_family:* This is the protocol family to be used, which is AF\_NETLINK.
* *nl\_pad:* This is used for padding.
* *nl\_pid:* This is the identification or the local address of the process. It is used if a process wants to send a unicast message to other processes or the kernel.
* *nl\_groups:* This is a 32-bit bitmask used for multicast communication.
* *nl\_pid* can be the PID of the process, which can be initialised as follows:

|  |
| --- |
| struct sockaddr\_nl addr;    addr.nl\_pid = getpid(); |

If, in a process, each thread wants its own Netlink socket, then *nl\_pid* can be initialised to:

|  |
| --- |
| addr.nl\_pid = pthread\_self() << 16 | getpid(); |

, or it can be initialised to simple numbers as:

|  |
| --- |
| addr.nl\_pid = 1; |

Or any algorithm can be used to assign it a unique value.  
nl\_groups are used for multicast communication. Each bit in this field is a multicast address. Any process which needs to listen on a particular group should set the bit.  
As an example, if a process wants to listen on multicast addresses 3 and 5, then the bits are stored as follows:

|  |
| --- |
| addr.nl\_groups = 1<<3 | 1<<5; |

If, for example, a process wants to send data to multicast group 3, then it will initialise the *nl\_groups* field as follows:

|  |
| --- |
| addr.nl\_groups = 1 << 3; |

If the process wants to send to both the 3 and 5 groups, then *nl\_groups* will be initialised as follows:

|  |
| --- |
| addr.nl\_groups = 1<<3 | 1<<5; |

*nl\_pid* is used to identify a single process or kernel and *nl\_groups* is used to identify multiple processes or kernel modules, where*nl\_pid* = 0 is a special address, which is the kernel address.  
The kernel requires each Netlink message to include the Netlink message header. Thus a Netlink message is a combination of a message header and message payload. An application allocates a buffer long enough to store both header and payload. The starting of the buffer holds the Netlink message header and it is followed by the payload. So just by typecasting the buffer address with the header structure, the header can be accessed, after which there is the payload. The header structure (*include/uapi/linux/netlink.h*) is as follows:

|  |
| --- |
| struct nlmsghdr  {  \_\_u32 nlmsg\_len;  \_\_u32 nlmsg\_type;  \_\_u32 nlmsg\_flags;  \_\_u32 nlmsg\_seq;  \_\_u32 nlmsg\_pid;  }; |

In the code above, lets look at what certain terms mean:

* + *\* nlmsg\_len:* This is the length of the message to be transferred, including the header length.
  + *\* nlmsg\_type:* This is the type of message that is being transferred and is used by applications. This field is not used by the kernel.
  + *nlmsg\_flags:* This is used to give additional information.
  + *nlmsg\_seq:* This is the sequence number of the message and is used by applications. This field is not used by the kernel.
  + *nlmsg\_pid:* This is the identification of the process which sends the message and is used by applications. This field is not used by the kernel.

A Netlink message is a buffer that holds both the Netlink header and the Netlink payload. The buffer is passed to the Netlink core through *iovec* structure. The structure (*include/uapi/linux/uio.h*) definition is as follows:

|  |
| --- |
| struct iovec{  void \_\_user \*iov\_base;  \_\_kernel\_size\_t iov\_len;  }; |

In the above code, *iov\_base* holds the base address of the Netlink message buffer, and*iov\_len* holds the length of the Netlink message buffer, which is the size of the Netlink header and payload.  
Socket messages are sent through the *sendmsg* API, which requires the *msghdr* structure as a parameter. The following fields of *struct msghdr* are useful:

|  |
| --- |
| struct msghdr  {  void \*msg\_name;  int msg\_namelen;  struct iovec \*msg\_iov;  \_\_kernel\_size\_t msg\_iovlen;  //other fields not discussed  }; |

In the above code:

* *msg\_name* is the base address of the *struct sockaddr\_nl* variable, which holds information about the destination address.
* *msg\_namelen* is the length of the structure, which is pointed by the*msg\_name* field.
* *msg\_iov* is the address of the *iovec* structure which holds the netlink message buffer.
* *msg\_iovlen* is the length of the netlink message buffer.

****Process-to-process unicast communication****  
*****Unicast sender example:***** The following header file needs to be included in an application:

|  |
| --- |
| #include <sys/socket.h>  #include <linux/netlink.h> |

First, the application has to create a Netlink socket, which is done through the socket API as follows:

|  |
| --- |
| int fd = socket(PF\_NETLINK, SOCK\_RAW, NETLINK\_GENERIC); |

After creating the Netlink socket, the application has to bind the socket with the unique address as follows:

|  |
| --- |
| struct sockaddr\_nl src\_addr;    //AF\_NETLINK socket protocol  src\_addr.nl\_family = AF\_NETLINK;    //application unique id  src\_addr.nl\_pid = 1;    //specify not a multicast communication  src\_addr.nl\_groups = 0;    //attach socket to unique id or address  bind(fd, (struct sockaddr \*)&src\_addr, sizeof(src\_addr)); |

After binding the socket with the unique address, the application has to define the destination address, message header, message payload, *iovec* structure and send the message using the *sendmsg* API as follows:

|  |
| --- |
| //total netlink message length  #define NLINK\_MSG\_LEN 1024    struct sockaddr\_nl dest\_addr;    dest\_addr.nl\_family = AF\_NETLINK;    //destination process id  dest\_addr.nl\_pid = 2;    dest\_addr.nl\_groups = 0;    //allocate buffer **for** netlink message which is message header + message payload  struct nlmsghdr \*nlh =(struct nlmsghdr \*) malloc(NLMSG\_SPACE(NLINK\_MSG\_LEN)); |

Here, NLMSG\_SPACE is the macro that gives an aligned length for the Netlink message.

|  |
| --- |
| //netlink message length  nlh->nlmsg\_len = NLMSG\_SPACE(NLINK\_MSG\_LEN);    //src application unique id  nlh->nlmsg\_pid = 1;    nlh->nlmsg\_flags = 0;    //copy the payload to be sent  strcpy(NLMSG\_DATA(nlh), Hello Process); |

Here, the NLMSG\_DATA macro is used to access the address of the payload.

|  |
| --- |
| //fill the iovec structure  struct iovev iov;    //netlink message header base address  iov.iov\_base = (void \*)nlh;    //netlink message length  iov.iov\_len = nlh->nlmsg\_len;    //define the message header **for** message sending  struct msghdr msg;    msg.msg\_name = (void \*)&dest\_addr;    msg.msg\_namelen = sizeof(dest\_addr);    msg.msg\_iov = &iov;    msg.msg\_iovlen = 1;  //send the message  sendmsg(fd, &msg, 0); |

****Unicast receive example:**** In case of the receiver, first the Netlink socket will be created using a socket API, as it was in the case of the sender.  
Then, like the sender, the receiver will bind its socket with the unique address, which will be the same as in the case of the sender. *src\_addr.nl\_pid* should be initialised as follows:

|  |
| --- |
| //receiver address or id  src\_addr.nl\_pid = 2; |

*dest\_addr* will be used to receive the data which does not need to be initialised. In case of *nlmsghdr*, this structure is just cleared as follows:

|  |
| --- |
| memset(nlh, 0, NLMSG\_SPACE(NLINK\_MSG\_PAYLOAD)); |

The rest of the code will be similar to the senders code, but instead of the *sendmsg* API, the *recvmsg* API will be used as follows:

|  |
| --- |
| recvmsg(fd, &msg, 0); |

This API will block until the message is received, after which the *nlmsghdr* variable *nlh* will get updated with the message header and payload, where the latter can be accessed as NLMSG\_DATA(nlh) which will be a pointer to the payload.

****Process-to-process multicast communication****  
In case of receivers for multicast communication, the *sockaddr\_nl* structure should be initialised as follows:

|  |
| --- |
| struct sockaddr\_nl src\_addr;  //initialize the protocol as Netlink family    src\_adr.nl\_family = AF\_NETLINK;    //assign the unique id to each application, here 2 is assigned **for** example  src\_addr.nl\_pid = 2;    //assign multicast addresses on which the process wants to listen, **for** example all the process wants to listen on multicast address 3 and 5  src\_addr.nl\_groups = 1<<3 | 1<<5; |

The rest of the code is similar to the unicast receiver code.  
In case of a sender for multicast communication, the destination *sockaddr\_nl* structure should be initialised as follows:

|  |
| --- |
| struct sockaddr\_nl dest\_addr;    //initialize the protocol as Netlink family  dest\_addr.nl\_family = AF\_NETLINK;    //suppose process wants to send multicast message to all process with multicast group 3  dest\_addr.nl\_groups = 1<<3; |

****Kernel Netlink implementation****  
The kernel space API for Netlink is different from that for user space. To create a Netlink socket in the kernel, the following API is used:

|  |
| --- |
| struct sock\* netlink\_kernel\_create(int unit,  void (\*input)(struct sock \*sock, int len)); |

In the above code:

* + *unit* is the protocol type, which is defined in*include/uapi/linux/netlink.h;* for example, NETLINK\_GENERIC.
  + *input* is the function pointer, which is called when the application sends data to the kernel with a unit type protocol.

So to create a Netlink socket in the kernel module with NETLINK\_GENERIC protocol type,*netlink\_kernel\_create* is called as follows:

|  |
| --- |
| struct sock\* nlink    nlink = netlink\_kernel\_create(NETLINK\_GENERIC, receive\_func);    receive\_func **for** example, is implemented as follows:    void receive\_func(struct sock \*sock, int len)  {  struct sk\_buff \*buffer;  struct nlmsghdr \*nlh;    **while**((buffer = skb\_dequeue(&buffer->receive\_queue)) != NULL)  {  nlh = (struct nlmsghdr \*)buffer->data;    //access the data through NLMSG\_DATA(nlh)  }  } |

*receive\_func* function is called in the *sendmsg* system call context. If the task that is to be done with the received message is small, then it can be done in *receive\_func;* but if it is not small, then it can block other system calls and can cause delays in the application. So to avoid this, kernel threads can be later used to process the message. For this purpose, the *skb\_recv\_datagram* API can be used as follows:

|  |
| --- |
| struct sk\_buff \*buffer;  int error;    buffer = skb\_recv\_datagram(nlink, 0, 0, &error);    In the above code:  \* nlink = struct sock\* variable is returned by netlink\_kernel\_create.  \* buffer = will be the buffer that will contain the Netlink message when the skb\_recv\_datagram wakes up. |

After this call, the calling thread will block and will have to be woken up through *wake\_up\_interruptible* in*receive\_func* callback as follows:

|  |
| --- |
| void receive\_func(struct sock \*buffer, int len)  {  // this will be wake up the thread which has called skb\_recv\_datagram  wake\_up\_interruptible(bufffer->sleep);  } |

After the thread has woken up, data can be accessed as follows:

|  |
| --- |
| struct sk\_buff \*buffer;  int error;  struct nlmsghdr \*nlh;    //here the thread will sleep till the message is received, after message is received receive\_func is called which will wake up this thread  buffer = skb\_recv\_datagram(nlink, 0, 0, &error);    //access the data through buffer variable  nlh = (struct nlmsghdr \*)buffer->data;    //access the data through NLMSG\_DATA macro in kernel  printk(Message received %s\n, NLMSG\_DATA(nlh)); |

To close the Netlink socket allocated, *sock\_release* is called as follows:

|  |
| --- |
| sock\_release(&nlink->socket); |

where nlink is the struct sock \* variable returned by*netlink\_kernel\_create api.*  
For sending unicast and multicast messages from the kernel to the process, the following are the APIs:

|  |
| --- |
| //unicast message sending from kernel  int netlink\_unicast(struct sock \*ssk, struct sk\_buff \* skb, u32 pid, int nonblock); |

In the above code:

* *ssk* is the struct sock \* returned by*netlink\_kernel\_create*.
* *skb* is the buffer which holds the message.
* *pid* is the ID or address of the process to which the message is to be sent.
* *Nonblock* is the variable to decide whether to block if the process is not present.

|  |
| --- |
| //multicast message sending from kernel  int netlink\_broadcast(struct sock \*ssk, struct sk\_buff \*skb, u32 pid, u32 group, int allocation); |

In the above code:

* + *group* is the multicast group to which the message is to be sent. This is similar to the *nl\_groups* field in the*sockaddr\_nl* structure.
  + *allocation* is GFP\_ATOMIC if called from the interrupt context or GFP\_KERNEL if called from the kernel thread. This is due to the fact that the kernel requires multiple buffer allocations to clone a multicast message.

Let’s end with an example of message sending in the kernel, as follows:

|  |
| --- |
| #define NLINK\_MSG\_SIZE 1024    //allocate netlink socket  struct sock \*nlink = netlink\_kernel\_create(NETLINK\_GENERIC, receive\_func);    //allocate socket buffer **for** message  struct sk\_buff \*skb = alloc\_skb(NLMSG\_SPACE(NLINK\_MSG\_SIZE), GFP\_KERNEL);    //get the header pointer  nlh = (struct nlmsghdr \*) skb->data;    //update source header parameters  nlh->nlmsg\_len = NLMSG\_SPACE(NLINK\_MSG\_SIZE);    //kernel id is 0  nlh->nlmsg\_pid = 0;    nlh->nlmsg\_flags = 0;    //copy the data  strcpy(NLMSG\_DATA(nlh), Hello);    //update this **if** kernel belongs to multicast group  NETLINK\_CB(skb).groups = 0;  //kernel id is 0  NETLINK\_CB(skb).pid = 0;    //use this **in** **case** of unicast message  NETLINK\_CB(skb).dst\_pid = 2;    //update this **in** **case** of multicast /message with multicast address  NETLINK\_CB(skb).dst\_groups = 0;    //unicast the message to process with /process address 2  netlink\_unicast(nlink, skb, 2, MSG\_DONTWAIT);    //use this **if** multicast is to be **done**example is multicast address 3  //netlink\_broadcast(nlink, skb, 0, 1<<3, GFP\_KERNEL); |