Linux Systems Programming with C and C++

DEVNAGA

Contents

The `SIGINT` and `SIGQUIT` are familiar to us as from the keyboard as we usually perform the `ctrl + c` (`SIGINT`) or `ctrl + \` (`SIGQUIT`) combination to stop a program.

Signals are also delivered to a process (running or stopped or waiting) with the help of `kill` command. The manual page (`man kill`) of kill command says that the default and easier version of kill command is the `kill pid`. Where `pid` is the process ID that is found via the `ps` command. The default signal is `SIGTERM (15)`. Alternatively a signal number is specified to the `kill` command such as `kill -2 1291` making a delivery of `SIGINT(2)` signal to the process ID 1291.

The linux also provide a mechanism for sysadmins to control the processes via two powerful and unmaskable signals `SIGKILL` and `SIGSTOP`. They are fatal and the program terminates as soon as it receives them. This allows admins to kill the offending or bad processes from hogging the resources.

The linux also provide us a set of system calls API to use the signals that are delivered to the process. Some of the most important API are `sigaction` and `signal`. Lately `signalfd` is introduced that delivers the signals in a synchronous fashion to safely control the signals that occur at unknown or surprisingly.

handling of the signals:

1. ignore the signal - SIG_IGN (ignore the signal by specifying this)
2. perform the handler function execution - create a callback handler that gets called when signal occurs
3. default action - is default for each process when created - kernel creates a default handler for each process

Signals are asynchornous and must be handled. So the system call interface provides an API to handle the signals. Below described are some of the system calls.

signal and sigaction

`signal` is a system call, that allows the program to handle the signal when it occurs.

prototype:

```
sighandler_t signal(int signum, sighandler_t handler);
```

The first argument is the signal number. The second argument is the signal handler callback.

An example of system call is shown below.

```
void signal_callback(int sig)
{
    printf("signal handler\n);
}

int main()
{
    signal(SIGINT, signal_callback);

    sleep(2);
}
```

above program registers `SIGINT` that's `ctrl + c` combination. A `ctrl + c` key combination is pressed when the program is running, it invokes the `signal_callback`.

to set the signal to default we must use `SIG_DFL` in the callback argument. Such as,

```
signal(SIGINT, SIG_DFL);
```

Sigaction The `sigaction` system call is more sophisticated system call for signal handling. The prototype of `sigaction` system call is as follows,

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

The structure `sigaction` is shown as below,

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

The first argument is the signal number. The second argument is the structure `sigaction`.

The `sa_flags` contain the following information. It provides a various set of flags but only the following list is useful.

1. `SA_SIGINFO` - the callback `sa_sigaction` is used when flags are set to this.

2. 0 - a default `sa_handler` should be set

Key differences A very good stackoverflow question here tells us why `sigaction` is better than `signal`. The key differences are:

1. the `signal()` does not necessarily block other signals from arriving while the current signal is being executed. Thus when more than one signal occur at the same time, it becomes more problematic to understand and perform actions. If it is on the same data, this might even get more complex. The `sigaction()` blocks the other signals while the handler is being executed.
2. As soon as the `signal()` handler is executed, the `signal()` sets the default handler to `SIG_DFL` which may be `SIGINT / SIGTERM / SIGQUIT` and the handler must reset the function back to itself so that when the signal occur again, the signal can be handled back. However, with the signal() allowing the handler to be called even though the handler is already executing, this will implicate a serious problem where in which the small window between the register and default would let the program go into `SIG_DFL`.

sigwaitinfo

sigtimedwait

sigwait

The system call `sigwait` is used to wait for signal until one of the signals specified in the signal mask are pending.

```
int sigwait(const sigset_t *set, int *sig);
```

the second argument `sig` contain the returned signal number.

The `sigwait` returns 0 on success and returns a positive error on failure.

signalfd

signalfd is a new system call introduced by the linux kernel. The signalfd system call returns a file descriptor associated with the signal. This file descriptor is then used to wait on the select, poll or epoll system calls. Unlike the `signal` or `sigaction` the signals are queued in the socket buffer and can be read on the socket.

The prototype of the `signalfd` is as follows.

```
int signalfd(int fd, const sigset_t *mask, int flags);
```

To use the `signalfd` one must include `<sys/signalfd.h>`.

If the fd argument is -1, the `signalfd` returns a new file descriptor. If fd argument is not -1, then the fd that is returned from previous calls to the `signalfd` must be given as an argument.

The mask argument is similar to the one that we pass to the `sigprocmask` system call. This allows the `signalfd` to create an `fd` out for the mask. As the mask is created, the signals in the mask should be blocked with the `sigprocmask`. This allows the correct functionality of the `signalfd`.

The `flags` argument is usually set to 0. It is much similar to the O_NONBLOCK flag options of other system calls.

Include `<sys/signalfd.h>` to use the `signalfd` system call.

Below is an example of the `signalfd` system call. Download here

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/signalfd.h>
#include <string.h>

int main(int argc, char **argv)
{
    int sfd;
    sigset_t mask;

    sigemptyset(&mask);
    sigaddset(&mask, SIGQUIT);
    sigaddset(&mask, SIGINT);

    if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0) {
        printf("afiled to sigprocmask\n");
        return -1;
    }

    sfd = signalfd(-1, &mask, 0);
    if (sfd < 0) {
        printf("failed to get signalfd\n");
        return -1;
    }

    while (1) {
        struct signalfd_siginfo sf;
        int ret;

        memset(&sf, 0, sizeof(sf));
        ret = read(sfd, &sf, sizeof(sf));
        if (ret != sizeof(sf)) {
            printf("invalid length of siginfo %d received\n", ret);
            return -1;
        }
```

```
        printf("pid %d signal value: %d signal code: %d\n",
                        sf.ssi_pid,
                        sf.ssi_signo,
                        sf.ssi_code);

        if (sf.ssi_signo == SIGQUIT) {
            printf("received termination signal\n");
        } else if (sf.ssi_signo == SIGINT) {
            printf("received interrupt\n");
        } else {
            printf("invalid signal %d\n", sf.ssi_signo);
        }
    }

    close(sfd);

    return 0;
}
```

The signals are first masked by the `sigaddset` system call and then blocked with `sigprocmask`. The mask is then given to the `signalfd` system call. The signals are then queued to the socket fd returned. This can be waited upon the `read` or `select` system call.

The kernel returns a variable of the form `signalfd_siginfo` upon read.The structure then contains the signal that is occured. Here's some contents in the `signalfd_siginfo`.

```
struct signalfd_siginfo {
    uint32_t ssi_signo;
    ...
    uint32_t ssi_pid;
    ....
};
```

The `ssi_signo` contains the signal number that is occured. The `ssi_pid` is the process that sent this signal.

Example: signalfd basic example

sigaddset

`sigaddset` adds the signal to a set. The prototype is as follows.

```
int sigaddset(sigset_t *set, int signo);
```

The `signo` gets added to the `set`. Multiple calls of the `sigaddset` on the same

`set` would add the signals to the set. The function is mostly used in generating a signal mask for the `sigprocmask`. See more about `sigprocmask` in the below sections.

usually, the `sigaddset` is called this way:

```
sigset_t set;

sigaddset(&set, SIGINT); // ignore SIGINT
```

ignores the signal `SIGINT`.

To setup a group of signals, the `sigaddset` can be allowed to call in a loop. For example,

```
int i = 0;
int signal_list[] = {SIGINT, SIGQUIT, SIGALRM};
sigset_t set;

// setup signal mask for the signals in signal_list
while (i < sizeof(signal_list) / sizeof(signal_list[0])) {
    sigaddset(&set, signal_list[i]);
}
```

sigfillset

`sigfillset` initializes the `set` to full, including all the signals. The prototype is as follows.

```
int sigfillset(sigset_t *set);
```

sigemptyset

`sigemptyset` clears the signal mask. The initialization of the set of type `sigset_t` is usually done with the `sigemptyset`. Before any calls to `sigaddset` the set of type `sigset_t` must be cleared with `sigemptyset`.

The above examples calls of `sigaddset` must use `sigemptyset`. So the fixed code examples look like the below,

```
sigset_t set;

sigemptyset(&set); // clear the signal set

sigaddset(&set, SIGINT); // ignore SIGINT


int i = 0;
```

```
int signal_list[] = {SIGINT, SIGQUIT, SIGALRM};
sigset_t set;

sigemptyset(&set); // clear the signal set

// setup signal mask for the signals in signal_list
while (i < sizeof(signal_list) / sizeof(signal_list[0])) {
    sigaddset(&set, signal_list[i]);
}
```

sigismember

`sigismember` validates if the given signal is with in the set. Prototype is as follows,

```
int sigismember(sigset_t *set, int no);
```

usually, the calling example looks like the following way,

```
sigismember(&set, SIGALRM);
```

below is one of the examples of using both `sigfillset` and `sigismember`.

```
/**
 * sigismember and sigfillset example
 *
 * Author: Devendra Naga (devendra.aaru@gmail.com)
 *
 * LICENSE MIT
 */

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

int main()
{
    sigset_t set;

    sigfillset(&set);

    if (sigismember(&set, SIGALRM)) {
        printf("sigalrm is a member of the set\n");
    } else {
        printf("sigalrm is not a member of the set\n");
    }
```

```
}
```

sigdelset

`sigdelset` deletes the signal from the given set. The prototype is as follows,

```
int sigdelset(sigset_t *set, int no);
```

sigprocmask

The system call `sigprocmask`, used to block certain signals.

The `sigprocmask` prototype is as follows (from the man pages),

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

`how` is defined by one of the following, `SIG_BLOCK` and `SIG_UNBLOCK`.

Below is an example use of `sigprocmask`.

```
/**
 * example sigprocmask
 *
 * Author: Devendra Naga (devendra.aaru@gmail.com)
 *
 * LICENSE MIT
 */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main()
{
    sigset_t set;
    int ret;

    sigemptyset(&set);

    // add SIGINT
    sigaddset(&set, SIGINT);

    ret = sigprocmask(SIG_BLOCK, &set, NULL);
    if (ret != 0) {
        perror("sigprocmask");
        return -1;
    }
```

```
    int count = 0;

    while (1) {
        printf("hello .. %d\n", count ++);
        sleep(1);
        if (count == 5) {
            break;
        }
    }

    ret = sigprocmask(SIG_UNBLOCK, &set, NULL);
    if (ret != 0) {
        perror("sigprocmask");
        return -1;
    }
}
```

In the above example, the signal set of type `sigset_t` is created with `sigemptyset` followed by the `sigaddset`. The signal `SIGINT` being setup in the signal set.

The `sigprocmask` system call is made with `SIG_BLOCK` that effectively blocks the signal `SIGINT` till the loop below executes. The loop is created only for testing purposes to see if the `SIGINT` is actually blocked by holding down `ctrl +c` combination on the keyboard. Till the count of 5, the signal is blocked and a call to the `sigprocmask` with `SIG_UNBLOCK` is made to unblock the signal.

The use case of `sigprocmask` is useful when we have multiple threads and we need to handle signals specifically in one of the threads / main thread.

since the signals are inherited by the threads / child processes, the signal might be delivered to any thread.

To do this, block signals in all the threads except the thread that signal needs to be handled.

for example, a below code can be used.

```
static void block_term_signals()
{
    sigset_t set;
    int ret;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);
```

```
    ret = sigprocmask(SIG_BLOCK, &set, NULL);
    if (ret < 0) {
        return;
    }
}
```

Simple demonstration of this is as follows:

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

static void block_term_signals()
{
    sigset_t set;
    int ret;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);

    ret = sigprocmask(SIG_BLOCK, &set, NULL);
    if (ret < 0) {
        return;
    }
}

void *thread_callback(void *data)
{
    block_term_signals();

    while (1) {
        sleep(1);
        printf("in %d %s\n", __LINE__, __func__);
    }
}

void *thread_callback_(void *data)
{
    block_term_signals();

    while (1) {
        sleep(1);
        printf("in %d %s\n", __LINE__, __func__);
    }
}
```

```c
void signal_handler(int sig)
{
    printf("in signal handler\n");
    signal(sig, SIG_DFL);
    raise(sig);
}

int main()
{
    pthread_t tid1, tid2;
    sigset_t set;
    int ret;

    signal(SIGINT, signal_handler);

    ret = pthread_create(&tid1, NULL, thread_callback, NULL);
    if (ret < 0) {
        return -1;
    }

    ret = pthread_create(&tid2, NULL, thread_callback_, NULL);
    if (ret < 0) {
        return -1;
    }

    while (1) {
        printf("in main thread\n");
        sleep(1);
    }
}
```

in the program, the main program creates two threads and in each thread the signals are blocked except in main thread. main thread registers the signal handler for the particular signal.

waiting for child processes

When a child process stops the parent process must `reap` it to prevent it from becoming a zombie. The zombie meaning that the process is not taken out from process table but they dont execute and do not utilize the memory or CPU.

When a parent process stops before the child stops, then the child process is called an `orphan` process. Often some process adapts the child process and becomes its parent. This process often is the `init` process.

When a parent waits for the child process by some means and reaps it, the

mand line. It then copies into the `struct ifreq`. The interface name is copied to the `ifr_name` flag, the family type is put into `ifr.ifr_addr.sa_family` and is `AF_INET`. We also need to set the family in the `ifr.ifr_hwaddr.sa_family` member as the same `AF_INET`. The mac is then copied into `ifr.ifr_hwaddr.sa_data` member.

The sample command to set the mac is below.

```
[root@localhost devnaga]# ./a.out enp0s25 00:ff:31:ed:ff:e1

[root@localhost devnaga]# ifconfig enp0s25
enp0s25: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 00:ff:31:ed:ff:e1  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xf0600000-f0620000
```

Get Interface Index

To get the interface index of a particular network interface, `SIOCGIFINDEX` is used.

Below is an example of getting an index from a particular interface. Download here

```c
#include <stdio.h>
#include <sys/socket.h>
#include <net/if.h>
#include <linux/ioctl.h>
#include <errno.h>
#include <string.h>
#include <sys/ioctl.h>

int main(int argc, char **argv)
{
    int fd;

    if (argc != 2) {
        fprintf(stderr, "<%s> interface name\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
```

```
    }

    struct ifreq ifr;

    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);

    int ret;

    ret = ioctl(fd, SIOCGIFINDEX, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to get ifindex %s\n", strerror(errno));
        return -1;
    }

    printf("interface index for [%s] is %d\n", argv[1], ifr.ifr_ifindex);

    return 0;
}
```

Set Interface Flags

Below example is another one of `SIOCGIFFLAGS`. Download here

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    struct ifreq ifr;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
```

```
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(fd, SIOCGIFFLAGS, &ifr);
    if (ret < 0) {
        return -1;
    }

    printf("ifname : %s\n", argv[1]);
    if (ifr.ifr_flags & IFF_UP) {
        printf("UP\n");
    }

    if (ifr.ifr_flags & IFF_BROADCAST) {
        printf("BROADCAST\n");
    }

    if (ifr.ifr_flags & IFF_LOOPBACK) {
        printf("LO\n");
    }

    if (ifr.ifr_flags & IFF_RUNNING) {
        printf("RUNN\n");
    }

    if (ifr.ifr_flags & IFF_PROMISC) {
        printf("PROMISC\n");
    }

    if (ifr.ifr_flags & IFF_MULTICAST) {
        printf("MCAST\n");
    }

    return 0;
}
```

there is another way to get the interface index. This can be done using the `if_nametoindex` function.

Below is an example of `if_nametoindex`. Download here

```
#include <stdio.h>
#include <net/if.h>
```

```c
int main(int argc, char **argv)
{
    int id;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    id = if_nametoindex(argv[1]);
    printf("index %d\n", id);

    return 0;
}
```

Get IPv4 Address

The ioctl flag `SIOCGIFADDR` is used to get interface ipv4 address. Below is the example program.

```c
#include <stdio.h>
#include <string.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <netinet/in.h>

int main(int argc, char **argv)
{
    int fd;

    if (argc != 2) {
        printf("<%s> <ifname>\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    struct ifreq ifr;
    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_addr.sa_family = AF_INET;
    strcpy(ifr.ifr_name, argv[1]);
```

```
    int ret;

    ret = ioctl(fd, SIOCGIFADDR, &ifr);
    if (ret < 0) {
        printf("failed to ioctl\n");
        return -1;
    }

    char *ip_addr = inet_ntoa(((struct sockaddr_in *)(&ifr.ifr_addr))->sin_addr);
    if (ip_addr)
        printf("ipaddr %s\n", ip_addr);

    return 0;
}
```

The function `if_nametoindex` is declared in `net/if.h`.

Get Broadcast Address

The ioctl `SIOCGIFBRDADDR` allows to get the broadcast address of network interface.

Below is an example. Download here

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <errno.h>

int main(int argc, char **argv)
{
    struct ifreq ifr;
    char *braddr;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }
```

```
    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        fprintf(stderr, "failed to socket %s\n", strerror(errno));
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));
    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(fd, SIOCGIFBRDADDR, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to ioctl %s\n", strerror(errno));
        return -1;
    }

    braddr = inet_ntoa(((struct sockaddr_in *)&(ifr.ifr_broadaddr))->sin_addr);
    if (!braddr) {
        fprintf(stderr, "failed to inet_ntoa %s\n", strerror(errno));
        return -1;
    }

    printf("broadcast %s\n", braddr);

    close(fd);

    return 0;
}
```

Get Network Mask

when the network interface does not have an IP address the above ioctl might fail with an error `cannot assign requested address`. It may be useful in cases of diagnostics.

The ioctl `SIOCGIFNETMASK` is used to get the network mask of an interface.

Below is an example of `SIOCGIFNETMASK`. Download here

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
```

```c
int main(int argc, char **argv)
{
    char *netmask;
    struct ifreq ifr;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));
    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(fd, SIOCGIFNETMASK, &ifr);
    if (ret < 0) {
        return -1;
    }

    netmask = inet_ntoa(((struct sockaddr_in *)&(ifr.ifr_netmask))->sin_addr);
    if (!netmask) {
        return -1;
    }

    printf("netmask %s\n", netmask);

    close(fd);

    return 0;
}
```

Get Network Interface List

Below is an example to get the network interface list (that have the ipv4 address).

```c
struct interface_info {
    char ifname[24];
    char ifaddr[80];
```

```
};

int get_interfaces(std::vector<interface_info> &ifinfo)
{
    struct ifconf ifc;
    struct ifreq *req;
    char data[4096];
    int ret;
    int fd;

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    ifc.ifc_len = sizeof(data);
    ifc.ifc_buf = (caddr_t)(data);
    ret = ioctl(fd, SIOCGIFCONF, &ifc);
    if (ret < 0) {
        goto err;
    }

    req = (struct ifreq *)data;
    while ((char *)req < data + ifc.ifc_len) {
        switch (req->ifr_addr.sa_family) {
            case AF_INET:
                interface_info ifi;
                char *intf_addr;

                strcpy(ifi.ifname, req->ifr_name);
                intf_addr = inet_ntoa(((struct sockaddr_in *)(&req->ifr_addr))->sin_addr);
                if (!intf_addr) {
                    return -1;
                }
                strcpy(ifi.ifaddr, intf_addr);
                ifinfo.push_back(ifi);
            break;
        }
        req = (struct ifreq *)((char *)req + sizeof(*req));
    }

    ret = 0;

    close(fd);

err:
```

```
    return ret;
}
```

Set Interface Name

There is also a way to change the interface name. This is done using the `SIOCSIFNAME` ioctl.

Below is the example that demonstrates the `SIOCSIFNAME`. Download here

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if_arp.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int sock;
    int ret;
    struct ifreq ifr;

    if (argc != 3) {
        fprintf(stderr, "%s <ifname> <new-ifname>\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        return -1;

    memset(&ifr, 0, sizeof(struct ifreq));

    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(sock, SIOCGIFFLAGS, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to get interface flags for %s\n", argv[1]);
        return -1;
    }
```

```
    ifr.ifr_flags &= ~IFF_UP;

    ret = ioctl(sock, SIOCSIFFLAGS, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set interface flags for %s\n", argv[1]);
        return -1;
    }

    strcpy(ifr.ifr_newname, argv[2]);

    ret = ioctl(sock, SIOCSIFNAME, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set interface name for %s\n", argv[1]);
        perror("ioctl");
        return -1;
    }

    strcpy(ifr.ifr_name, argv[2]);

    ifr.ifr_flags |= IFF_UP;

    ret = ioctl(sock, SIOCSIFFLAGS, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set interface flags for %s\n", argv[1]);
        return -1;
    }

    close(sock);

    return 0;
}
```

The programs makes the interface go down, otherwise we cannot change the name of the interface. The interface is made down using the `SIOCSIFFLAGS` ioctl and sets up the interface name and makes the interface up again using the `SIOCSIFFLAGS`.

VLANs

VLANs are used to logically divide the network into many interfaces although there is only one physical interface available. In linux VLANs can be created on with the `vlan` utility or programmatically.

We will look at programmatically creating VLANs here.

VLANs are generally associated with the physical name and an ID. There can be 0 4095 VLAN Ids. The interface must be a real interface in order to create a

VLAN.

Below program is used to add a VLAN interface.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if_vlan.h>
#include <linux/sockios.h>

int main(int argc, char **argv)
{
    struct vlan_ioctl_args ioctl_args;
    int sock;
    int ret;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        return -1;
    }

    memset(&ioctl_args, 0, sizeof(ioctl_args));
    strcpy(ioctl_args.device1, argv[1]);
    ioctl_args.u.VID = atoi(argv[2]);
    ioctl_args.cmd = ADD_VLAN_CMD;

    ret = ioctl(sock, SIOCSIFVLAN, &ioctl_args);
    if (ret < 0) {
        perror("ioctl");
        return -1;
    }

    close(sock);

    return 0;
}
```

running `./a.out enp2s0 4` creates an interface `enp2s0.4`.

```
enp2s0.4: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether 58:8a:5a:0a:6b:2e  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```