Άσκηση 3

Ζητούμενο 1 (sockets)

Τα ***sockets*** είναι μία προγραμματιστική διεπαφή που επιτρέπει την επικοινωνία διεργασίων που δεν εκτελούνται απαραίτητα στο ίδιο σύστημα. Σε περίπτωση που έχουμε διεργασίες εκτελούμενες σε διαφορετικούς υπολογιστές, κρίνεται απαραίτητη η χρήση ενός δικτυακού πρωτοκόλλου. Χρησιμοποιούνται τα ***TCP/IP*** και ***TCP/UDP***, ανάλογα με το αν θέλουμε επικοινωνία με ή χωρίς σύνδεση.

Χρήσιμες Δομές

Μία δομή παρουσιάζει ενδιαφέρον. Πρόκειται για τη δομή ***struct sockaddr***. Πρόκειται για τη δομή που χρησιμοποιείται για την αποθήκευση της ***IP Adress*** και της ***Port*** με την οποία είναι συσχετισμένο ένα socket. Σημειώνεται πως, κανονικά, υπάρχουν δύο τύποι τέτοιων structs, ανάλογα με το αν αναφερόμαστε σε IPv4 (***struct sockaddr_in***)ή σε IPv6 (***struct sockaddr_in6***). Τα πεδία τους είναι τα ίδια αλλά υπάρχει διαφοροποίηση ως προς το μέγεθος τους (λόγω του ότι στο ένα πρωτόκολλο έχουμε διευθύνσεις ***32 bits*** και στο άλλο ***128 bits***). Για αυτό το λόγο, όλα τα ***syscalls*** για sockets που θα δούμε παρακάτω έχουν ένα πεδίο που δηλώνει το μέγεθος του struct ώστε να γίνεται κατανοητό ποιος από τους δύο τύπους δομών χρησιμοποιείται.

System Calls

Η αντιμετώπιση των sockets σε προγραμματιστικό επίπεδο είναι παρόμοια με αυτή των αρχείων, με τη διαφορά ότι χρησιμοποιούνται κάποιες επιπλέον κλήσεις συστήματος. Οι κλήσεις αυτές είναι:

socket: Η συγκεκριμένη κλήση συστήματος χρησιμοποιείται για τη δημιουργία ενός καινούριου socket. Επιστρέφει έναν ακέραιο (***socket descriptor***), ο οποίος έχει ρόλο εντελώς αντίστοιχο αυτού που έχουν οι file descriptors για τα συμβατικά αρχεία. Συγκεκριμένα, είναι:

int sockfd = socket(domain, type, protocol)

Το ***domain*** είναι ***PF_INET*** για ***IPv4***, ***PF_INET6*** για ***IPv6*** και ***PF_LOCAL*** ή ***PF_UNIX*** για επικοινωνία στο τοπικό σύστημα. Το ***type*** είναι ***SOCK_STREAM*** αν θέλουμε σύνδεση και ***SOCK_DGRAM*** αν δεν θέλουμε σύνδεση. Το πρωτόκολλο το βάζουμε 0 προκειμένου να γίνεται η default επιλογή με βάση τις τιμές των προηγούμενων πεδίων. Παρακάτω έχουμε ένα παράδειγμα από το ***master socket*** ενός ***server*** το οποίο ακούει για εισερχόμενες συνδέσεις (είναι non-blocking, οπότε έχει ένα επιπλέον flag στο πεδίο type):

if ((sd = socket(PF_INET, SOCK_STREAM|SOCK_NONBLOCK, 0)) < 0) {

```
        perror("socket");
        exit(1);
}
fprintf(stderr, "Created TCP socket\n");
```

bind: Χρησιμοποιείται από το server προκειμένου να συσχετιστεί το master socket με ένα συγκεκριμένο port και να δέχεται δεδομένα από οποιαδήποτε διεύθυνση IP στέλνει στο συγκεκριμένο port. Συγκεκριμένα, είναι:

```
int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Ένα παράδειγμα χρήσης είναι:

```
struct sockaddr_in sa;
memset(&sa, 0, sizeof(sa));
memset(&conns, 0, sizeof(conns));
sa.sin_family = AF_INET;
sa.sin_port = htons(TCP_PORT);
sa.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
        perror("bind");
        exit(1);
}
fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);
```

connect: Χρησιμοποιείται στην περίπτωση που θέλουμε να έχουμε σύνδεση. Είναι:

```
int connect(int sockfd, const struct sockaddr *addr,  socklen_t addrlen);
```

Ένα παράδειγμα χρήσης είναι:

```
if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
        perror("connect");
        exit(1);
}
fprintf(stderr, "Connected.\n");
```

Αν θέλουμε επικοινωνία μέσω ***UDP*** χωρίς σύνδεση, χρησιμοποιούμε:

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr
*dest_addr, socklen_t addrlen);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr,
socklen_t *addrlen);
```

(Το πεδίο flag γενικά θα το βάζουμε 0.)

accept: Χρησιμοποιείται από το server για την αποδοχή εισερχόμενων συνδέσεων. Παίρνει, μεταξύ άλλων, το master socket, το οποίο όμως δεν συσχετίζεται με τη συγκεκριμένη σύνδεση, αλλά μπορεί να χρησιμοποιηθεί για την ίδια δουλειά και μετά. Αντιθέτως, με τη συγκεκριμένη σύνδεση συσχετίζεται ένα καινούριο socket που επιστρέφεται μετά την ολοκλήρωση της accept. Ένα παράδειγμα χρήσης είναι:

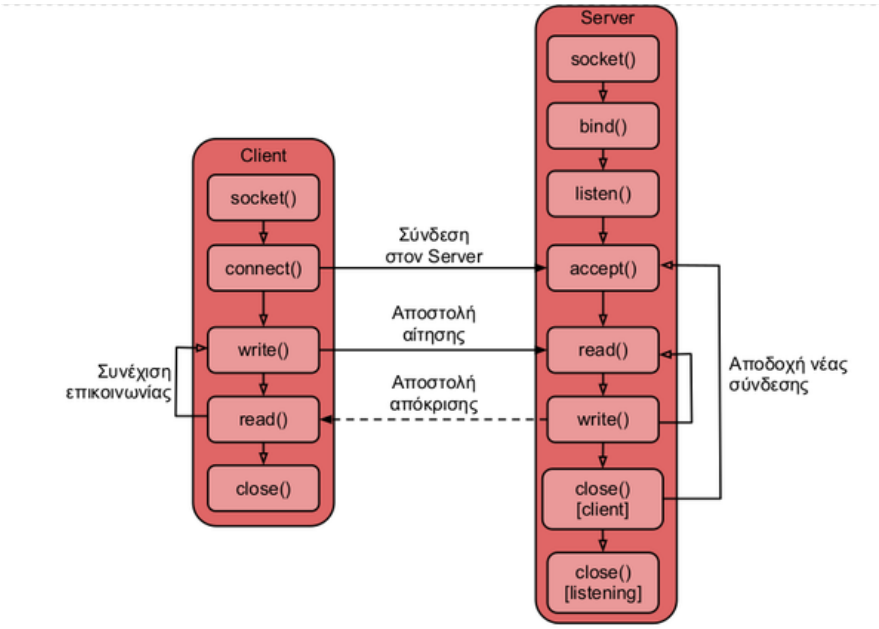
```
if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
        if ((errno==EAGAIN)||(errno==EWOULDBLOCK)) continue;
        perror("accept");
        exit(1);
}
```

select: Χρησιμοποιείται όταν περιμένουμε μηνύματα από περισσότερους του ενός file descriptors. Ένα παράδειγμα φαίνεται παρακάτω, αν και η λειτουργία του γίνεται καλύτερα κατανοητή βλέποντας τον συνολικό κώδικα.

```
if (select(nfds,&rfds,NULL,NULL,NULL)==-1) {
        if (errno==EBADF) printf("Its a bad f d \n");
        perror("select");
        exit(1);
}
```

Πρωτόκολλο Επικοινωνίας

Εδώ φαίνεται σχηματικά η χρήση των sockets στα πλαίσια του γενικότερου πρωτοκόλλου επικοινωνίας client-server.



Παρουσιάζουμε πρώτα τα σχετικά με το ζητούμενο 2 και μετά δίνουμε το συνολικό κώδικα όπου όλα αυτά φαίνονται.

Ζητούμενο 2 (cryptodev)

Το ***cryptodev*** είναι ένας οδηγός συσκευής που χρησιμοποιείται για τη διαχείριση ειδικού υλικού που προορίζεται για την επιτάχυνση της διαδικασίας της κρυπτογράφησης. Η λειτουργία του βασίζεται κατά βάση σε κλήσεις ***ioctl*** μέσω των οποίων διευκρινίζεται ποια είναι η επιθυμητή λειτουργία (κρυπρογράφηση/αποκρυπτογράφηση). Παρακάτω δίνεται ο συνολικός κώδικας για επικοινωνία μέσω sockets, όπου φαίνεται η χρήση του cryptodev.

socket-client.c

```
/*
 * socket-client.c
 * Simple TCP/IP communication using sockets
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 */

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include "socket-common.h"
/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;

        while (cnt > 0) {
                ret = write(fd, buf, cnt);
```

```
            if (ret < 0)
                return ret;
            buf += ret;
            cnt -= ret;
        }

        return orig_cnt;
}

static int running;

static void handler (int signum)
{
        running=0;
}


int main(int argc, char *argv[])
{
        struct sigaction sigact;
        running=1;
        sigact.sa_handler=handler;
        sigact.sa_flags=SA_RESTART;
        sigaction(SIGINT,&sigact,NULL);
        int sd, port;
        ssize_t n;
        char buf[buff_size];
        char *hostname;
        struct hostent *hp;
        struct sockaddr_in sa;
        fd_set rfds;

        if (argc != 3) {
            fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
            exit(1);
        }
        hostname = argv[1];
        port = atoi(argv[2]); /* Needs better error checking */

        /* Create TCP/IP socket, used as main chat channel */
        if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
            perror("socket");
            exit(1);
        }
```

```
fprintf(stderr, "Created TCP socket\n");

/* Look up remote hostname on DNS */
if ( !(hp = gethostbyname(hostname))) {
        printf("DNS lookup failed for host %s\n", hostname);
        exit(1);
}

/* Connect to remote TCP port */
sa.sin_family = AF_INET;
sa.sin_port = htons(port);
memcpy(&sa.sin_addr.s_addr, hp->h_addr, sizeof(struct in_addr));
fprintf(stderr, "Connecting to remote host... "); fflush(stderr);
if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
        perror("connect");
        exit(1);
}
fprintf(stderr, "Connected.\n");

/* Be careful with buffer overruns, ensure NUL-termination */
/* Say something... */
FD_ZERO(&rfds);
FD_SET(0,&rfds);
FD_SET(sd,&rfds);

while(running) {
        FD_ZERO(&rfds);
        FD_SET(0,&rfds);
        FD_SET(sd,&rfds);
        if (select(sd+1,&rfds,NULL,NULL,NULL)< 0) {
                if(errno==EINTR) {
                        printf("\n Interrupted by Signal. Terminating connection... \n");
                        continue;
                }
                else
                perror("select");
                exit(1);
        }
        if (FD_ISSET(0,&rfds)) {
                n = read(0, buf, sizeof(buf)-1);

                if (n < 0) {
                        perror("read");
                        exit(1);
```

```
                }
                buf[n]='\0';
                if (insist_write(sd, buf, n) != n) {
                        perror("write");
                        exit(1);
                }
                continue;
        }
        if (FD_ISSET(sd,&rfds)) {
                n = read(sd, buf, sizeof(buf)-1);
                if (n < 0) {
                        perror("read");
                        exit(1);
                }
                buf[n]='\0';
                if (insist_write(1, buf, n) != n) {
                        perror("write");
                        exit(1);
                }
                continue;
        }
    }
    /*
     * Let the remote know we're not going to write anything else.
     * Try removing the shutdown() call and see what happens.
     */
    printf("Program shutting down due to Ctrl+C\n");
    if (shutdown(sd, SHUT_WR) < 0) {
        perror("shutdown");
        exit(1);
    }
    return 0;
}
```

socket-server.c

```
/*
 * socket-server.c
 * Simple TCP/IP communication using sockets
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 */
#define MAX_CONN 3
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
#include <errno.h>
#include <time.h>

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>
#include <netinet/in.h>

#include "socket-common.h"

/* Convert a buffer to upercase */
void toupper_buf(char *buf, size_t n)
{
        size_t i;

        for (i = 0; i < n; i++)
                buf[i] = toupper(buf[i]);
}

/* Insist until all of the data has been written */
ssize_t insist_write(int fd, const void *buf, size_t cnt)
{
        ssize_t ret;
        size_t orig_cnt = cnt;
```

```
        while (cnt > 0) {
                ret = write(fd, buf, cnt);
                if (ret < 0)
                        return ret;
                buf += ret;
                cnt -= ret;
        }

        return orig_cnt;
}

int main(void)
{
        time_t t = time(NULL);
        struct tm tm = *localtime(&t);

        fd_set rfds;
        int nfds,nconn,i,j;
        char buf[buff_size];
        char bufout[buff_size];
        char addrstr[INET_ADDRSTRLEN];
        int sd, newsd;
        int conns[MAX_CONN];
        ssize_t n;
        socklen_t len;
        struct sockaddr_in sa;
        len = sizeof(struct sockaddr_in);

        /* Make sure a broken connection doesn't kill us */
        signal(SIGPIPE, SIG_IGN);

        /* Create TCP/IP socket, used as main chat channel */
        if ((sd = socket(PF_INET, SOCK_STREAM|SOCK_NONBLOCK, 0)) < 0) {
                perror("socket");
                exit(1);
        }
        fprintf(stderr, "Created TCP socket\n");

        /* Bind to a well-known port */
        memset(&sa, 0, sizeof(sa));
        memset(&conns, 0, sizeof(conns));
        sa.sin_family = AF_INET;
        sa.sin_port = htons(TCP_PORT);
        sa.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
if (bind(sd, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
        perror("bind");
        exit(1);
}
fprintf(stderr, "Bound TCP socket to port %d\n", TCP_PORT);

/* Listen for incoming connections */
if (listen(sd, TCP_BACKLOG) < 0) {
        perror("listen");
        exit(1);
}
FD_ZERO(&rfds);
nfds=sd+1;
FD_SET(sd,&rfds);
nconn=MAX_CONN;
/* Loop forever, accept()ing connections */
fprintf(stderr, "Waiting for an incoming connection...\n");
while(1) {
        nfds=sd+1;
        FD_ZERO(&rfds);
        //remake fdset
        FD_SET(sd,&rfds);
        for(i=0;i<nconn;i++){
                if (conns[i]==0) continue;
                else FD_SET(conns[i],&rfds);
                if (nfds<(conns[i]+1)) nfds=conns[i]+1;
        }
        if (select(nfds,&rfds,NULL,NULL,NULL)==-1) {
                if (errno==EBADF) printf("Its a bad f d \n");
                perror("select");
                exit(1);
        }
        if (FD_ISSET(sd,&rfds)) {
        /* Accept an incoming connection */
                if ((newsd = accept(sd, (struct sockaddr *)&sa, &len)) < 0) {
                        if ((errno==EAGAIN)||(errno==EWOULDBLOCK)) continue;
                        perror("accept");
                        exit(1);
                }
                bzero(&sa,len);
                if (getpeername(newsd,(struct sockaddr *)&sa, &len)<0) {
                        perror("getpeername");
                        exit(1);
                }
```

```
                if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr))) {
                        perror("could not format IP address");
                        exit(1);
                }
                for(i=0;i<nconn;i++) {
                        if (conns[i]==0) {conns[i]=newsd;break;}
                }
                if (i==nconn) {
                //too many connections, you cant connect now
                        strncpy(bufout,"Too many connected clients right now. Try again
later.\n",sizeof(bufout)-1);
                        bufout[sizeof(bufout)-1]='\0';
                        n=strlen(bufout);
                        if (insist_write(newsd, bufout, n) != n) {
                                perror("write to remote peer failed");
                                exit(1);
                        }
                        continue;
                }
                FD_SET(newsd,&rfds);
                if (newsd+1>nfds) nfds=newsd+1;
                t = time(NULL);
                tm = *localtime(&t);
                n=sprintf(bufout, "[%d-%d-%d]User from [%s:%d] has joined the
chat.\n",tm.tm_hour,tm.tm_min,tm.tm_sec,addrstr, ntohs(sa.sin_port));
                for(i=0;i<nconn;i++) {
                        if(conns[i]==0) continue;
                        newsd=conns[i];
                        if (insist_write(newsd, bufout, n) != n) {
                                perror("write to remote peer failed");
                                exit(1);
                        }
                }
                continue;
        }
        for (i=0;i<nconn;i++) {
                if (conns[i]==0) continue;
                newsd=conns[i];
                if (!FD_ISSET(newsd,&rfds)) continue;
                n = read(newsd, buf, sizeof(buf));
                if (n <= 0) {
                        if (n < 0)
                                perror("read from remote peer failed");
                        else
```

```
                        //      fprintf(stderr, "Peer went away\n");
                                bzero(&sa,len);
                                if (getpeername(newsd,(struct sockaddr *)&sa, &len)<0) {
                                        perror("getpeername");
                                        exit(1);
                                }

                                if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr,
sizeof(addrstr))) {
                                        perror("could not format IP address");
                                        exit(1);
                                }
                                t = time(NULL);
                                tm = *localtime(&t);
                                n=sprintf(bufout, "[%d-%d-%d]User from [%s:%d] has
left the chat.\n",tm.tm_hour,tm.tm_min,tm.tm_sec,addrstr, ntohs(sa.sin_port));
                                for(j=0;j<nconn;j++) {
                                        if (conns[j]==0) continue;
                                        newsd=conns[j];
                                        if (insist_write(newsd, bufout, n) != n) {
                                                perror("write to remote peer failed");
                                                exit(1);
                                        }
                                }
                                newsd=conns[i];
                                conns[i]=0;
                                FD_CLR(newsd,&rfds);
                                if (close(newsd) < 0)
                                        perror("close");
                                newsd=-1;
                                break;
                        }
                        buf[n]='\0';
                        bzero(&sa,len);
                        if (getpeername(newsd,(struct sockaddr *)&sa, &len)<0) {
                                perror("getpeername");
                                exit(1);
                        }
                        if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr))) {
                                perror("could not format IP address");
                                exit(1);
                        }
                        t = time(NULL);
                        tm = *localtime(&t);
```

```
                        n=sprintf(bufout, "[%d-%d-%d][%s:%d] said:
%s",tm.tm_hour,tm.tm_min,tm.tm_sec,addrstr, ntohs(sa.sin_port),buf);
                        for(j=0;j<nconn;j++) {
                                if (conns[j]==0) continue;
                                newsd=conns[j];
                                if (insist_write(newsd, bufout, n) != n) {
                                        perror("write to remote peer failed");
                                        exit(1);
                                }
                        }
                        break;
                }
        }

        /* This will never happen */
        return 1;
}
```

Ζητούμενο 3 (virtio)

Πρόκειται για πρωτόκολλο ανάπτυξης drivers με σκοπό την εκμετάλλευση δυνατοτήτων ***paravirtualization*** που προσφέρουν ***hypervisors*** όπως το ***QEMU***. Έτσι επιταχύνονται διάφορες λειτουργίες που, σε περιβάλλον πλήρους εικονικοποίησης, θα έπρεπε να υλοποιηθούν εξ' ολοκλήρου σε επίπεδο λογισμικού. Η ιδέα είναι ότι, όταν πραγματοποιείται ένα ***syscall*** από μία διεργασία χώρου χρήστη μίας εικονικής μηχανής με σκοπό την πρόσβαση σε κάποια περιφερειακή συσκευή, ο πυρήνας της εικονικής μηχανής (που για τον ***host*** βρίσκεται σε χώρο χρήστη), κάνει ένα ***hypercall*** στο ***VMM*** (χώρος χρήστη ***host***), που με τη σειρά του κάνει syscall στον πυρήνα του host.
Για να υλοποιηθούν τα προηγούμενα τα δεδομένα που πρέπει να περαστούν στον host αποθηκεύονται σε κάποιους buffers. Αυτοί απεικονίζονται με ***mmap*** σε κάποιες διευθύνσεις που βλέπει ο host. Μετά την επεξεργασία των δεδομένων, αυτά στέλνονται πίσω στον ***guest***. Για την υλοποίηση του πρωτοκόλλου γίνεται χρήση ***scatter-gather lists***.
Αρχίκα, δίνεται ο κώδικας του ***frontend***.

Open

```
static int crypto_chrdev_open(struct inode *inode, struct file *filp)
{
        int ret = 0;
        int err;
        unsigned int len, num_out, num_in;
        struct crypto_open_file *crof;
        struct crypto_device *crdev;
        unsigned int *syscall_type;
        int *host_fd;
        unsigned long flags;
        struct scatterlist syscall_type_sg, host_fd_sg, *sg[2];

        debug("Entering");

        syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
        *syscall_type = VIRTIO_CRYPTO_SYSCALL_OPEN;
        host_fd = kzalloc(sizeof(*host_fd), GFP_KERNEL);
        *host_fd = -1;

        crof = NULL;

        ret = -ENODEV;
        if ((ret = nonseekable_open(inode, filp)) < 0)
            goto fail;

        /* Associate this open file with the relevant crypto device. */
```

```
                crdev = get_crypto_dev_by_minor(iminor(inode));
                if (!crdev) {
                        debug("Could not find crypto device with %u minor",
                              iminor(inode));
                        ret = -ENODEV;
                        goto fail;
                }

                crof = kzalloc(sizeof(*crof), GFP_KERNEL);
                if (!crof) {
                        ret = -ENOMEM;
                        goto fail;
                }
                crof->crdev = crdev;
                crof->host_fd = -1;
                filp->private_data = crof;

                /**
                 * We need two sg lists, one for syscall_type and one to get the
                 * file descriptor from the host.
                 **/
                num_out = 0;
                num_in = 0;
                sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
                sg[num_out++] = &syscall_type_sg;
                sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
                sg[num_out + num_in++] = &host_fd_sg;

                spin_lock_irqsave(&crdev->lock, flags);
                ret = virtqueue_add_sgs(crdev->vq, sg, num_out, num_in,
&syscall_type_sg, GFP_ATOMIC);
                if (ret < 0) {
                        spin_unlock_irqrestore(&crdev->lock, flags);
                        debug("Could not add buffers to the vq.");
                        goto fail;
                }
                virtqueue_kick(crdev->vq);

                /**
                 * Wait for the host to process our data.
                 **/
                while (virtqueue_get_buf(crdev->vq, &len) == NULL); // busy-wait
loop
                spin_unlock_irqrestore(&crdev->lock, flags);
```

```
                        /* If host failed to open() return -ENODEV. */
                        debug("Backend returned file descriptor %d", *host_fd);
                        if (*host_fd < 0) ret = -ENODEV;
                        crof->host_fd = *host_fd;

fail:
                        kfree(syscall_type);
                        kfree(host_fd);
                        debug("Leaving with ret = %d", ret);
                        return ret;
}
```

release

```
static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
                        int ret = 0;
                        struct crypto_open_file *crof = filp->private_data;
                        struct crypto_device *crdev = crof->crdev;
                        unsigned int *syscall_type;
                        unsigned int num_out, len;
                        struct scatterlist syscall_type_sg, host_fd_sg, *sg[2];
                        unsigned long flags;
                        debug("Entering");

                        syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
                        *syscall_type = VIRTIO_CRYPTO_SYSCALL_CLOSE;

                        /**
                         * Send data to the host.
                         **/
                        num_out = 0;
                        sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
                        sg[num_out++] = &syscall_type_sg;
                        sg_init_one(&host_fd_sg, &crof->host_fd, sizeof(crof->host_fd));
                        sg[num_out++] = &host_fd_sg;
                        spin_lock_irqsave(&crdev->lock, flags);
                        ret = virtqueue_add_sgs(crdev->vq, sg, num_out, 0, &syscall_type_sg,
GFP_ATOMIC);
                        if (ret < 0) {
                            spin_unlock_irqrestore(&crdev->lock, flags);
                            debug("Could not add buffers to the vq.");
                            goto fail;
                        }
                        virtqueue_kick(crdev->vq);

                        /**
                         * Wait for the host to process our data.
                         **/
                        while (virtqueue_get_buf(crdev->vq, &len) == NULL); // busy-wait
loop
                        spin_unlock_irqrestore(&crdev->lock, flags);
fail:
                        kfree(crof);
                        kfree(syscall_type);
                        debug("Leaving");
```

```
                return ret;
}
```

ioctl

```
static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
                unsigned long arg)
{
                long ret = 0;
                int err;
                int *host_ret;
                uint32_t *ses_id;
                struct session_op *sess;
                struct crypt_op *cryp;
                struct crypto_open_file *crof = filp->private_data;
                struct crypto_device *crdev = crof->crdev;
                struct virtqueue *vq = crdev->vq;
                struct scatterlist sess_id_sg,syscall_type_sg, cmd_sg, session_sg,
host_fd_sg, ret_sg, ses_id_sg,
                        cryp_src_sg, cryp_dst_sg, cryp_iv_sg, cryp_op_sg, seskey_sg,
                                *sgs[8];
                unsigned int num_out, num_in, len, *cmd_ptr;
                unsigned long flags;
                unsigned char *ses_key, *src, *dst=NULL, *iv;
                unsigned int *syscall_type;

                printk(KERN_CRIT "Entering");

                syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
                *syscall_type = VIRTIO_CRYPTO_SYSCALL_IOCTL;
                host_ret = kzalloc(sizeof(*host_ret), GFP_KERNEL);
                cmd_ptr = kzalloc(sizeof(*cmd_ptr), GFP_KERNEL);
                ses_id = kzalloc(sizeof(*ses_id), GFP_KERNEL);
                src=NULL;
                dst=NULL;
                iv=NULL;
                ses_key=NULL;
                *cmd_ptr = cmd;

                num_out = 0;
                num_in = 0;

                sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
                sgs[num_out++] = &syscall_type_sg;
```

```
                sg_init_one(&host_fd_sg,&crof->host_fd,sizeof(crof->host_fd));
                sgs[num_out++]=&host_fd_sg;

                sess = kzalloc(sizeof(*sess), GFP_KERNEL);
                if (!sess) {
                    return -ENOMEM;
                }

                cryp = kzalloc(sizeof(*cryp), GFP_KERNEL);
                if (!cryp) {
                    return -ENOMEM;
                }

                switch (cmd) {
                case CIOCGSESSION:
                    debug("CIOCGSESSION");
                    sg_init_one(&cmd_sg, cmd_ptr, sizeof(*cmd_ptr));
                    sgs[num_out++] = &cmd_sg;
                    if (copy_from_user(sess, (struct session_op*) arg, sizeof(struct
session_op))){
                            debug("copy_from_user");
                            return -EFAULT;
                    }
                    ses_key = kzalloc(sess->keylen*sizeof(char), GFP_KERNEL);
                    if (!ses_key) {
                            return -ENOMEM;
                    }
                    if(copy_from_user(ses_key, sess->key, sizeof(char)*sess->keylen)){
                            debug("copy_from_user");
                            return -EFAULT;
                    }
                    sg_init_one(&seskey_sg, ses_key, sizeof(char)*sess->keylen);
                    sgs[num_out++] = &seskey_sg;

                    sg_init_one(&session_sg, sess, sizeof(*sess));
                    sgs[num_out + num_in++] = &session_sg;

                    sg_init_one(&ret_sg, host_ret, sizeof(*host_ret));
                    sgs[num_out + num_in++] = &ret_sg;

                    break;

                case CIOCFSESSION:
                    debug("CIOCFSESSION");
```

```
                sg_init_one(&cmd_sg, cmd_ptr, sizeof(*cmd_ptr));
                sgs[num_out++] = &cmd_sg;

                if(copy_from_user(ses_id, (uint32_t*)arg, sizeof(*ses_id))){
                        debug("copy_from_user");
                        return -EFAULT;
                }
                sg_init_one(&sess_id_sg, ses_id, sizeof(*ses_id));
                sgs[num_out++] = &sess_id_sg;
                sg_init_one(&ret_sg, host_ret, sizeof(host_ret));
                sgs[num_out + num_in++] = &ret_sg;
                break;

        case CIOCCRYPT:
                debug("CIOCCRYPT");
                sg_init_one(&cmd_sg, cmd_ptr, sizeof(*cmd_ptr));
                sgs[num_out++] = &cmd_sg;
                if(copy_from_user(cryp, (struct crypt_op*)arg, sizeof( struct
crypt_op))){
                        debug("copy_from_user");
                        return -EFAULT;
                }

                sg_init_one(&cryp_op_sg, cryp, sizeof(*cryp));
                sgs[num_out++] = &cryp_op_sg;

                src = kzalloc(cryp->len*sizeof(char), GFP_KERNEL);
                if (!src) {
                        return -ENOMEM;
                }

                if(copy_from_user(src, cryp->src, cryp->len*sizeof(char))){
                        debug("copy_from_user");
                        return -EFAULT;
                }

                sg_init_one(&cryp_src_sg, src, cryp->len*sizeof(char));
                sgs[num_out++] = &cryp_src_sg;

                iv = kzalloc(16*sizeof(char), GFP_KERNEL);
                if (!iv) {
                        return -ENOMEM;
                }
```

```
                    if(copy_from_user(iv, cryp->iv, 16*sizeof(char))){
                        debug("copy_from_user");
                        return -EFAULT;
                    }

                    sg_init_one(&cryp_iv_sg, iv, cryp->len*sizeof(char));
                    sgs[num_out++] = &cryp_iv_sg;


                    dst = kzalloc(cryp->len*sizeof(char), GFP_KERNEL);
                    if (!dst) {
                        return -ENOMEM;
                    }

                    sg_init_one(&cryp_dst_sg, dst, cryp->len*sizeof(char));
                    sgs[num_out + num_in++] = &cryp_dst_sg;

                    sg_init_one(&ret_sg, host_ret, sizeof(host_ret));
                    sgs[num_out + num_in++] = &ret_sg;
                    break;

                default:
                    debug("Unsupported ioctl command");

                    break;
                }

                spin_lock_irqsave(&crdev->lock, flags);

                err = virtqueue_add_sgs(vq, sgs, num_out, num_in, &syscall_type_sg,
GFP_ATOMIC);
                if (err < 0) {
        spin_unlock_irqrestore(&crdev->lock, flags);
        debug("Could not add buffers to the vq.");
        return -EINVAL;
                }
                printk(KERN_CRIT "about to notify backend\n");
                virtqueue_kick(vq);
                printk(KERN_CRIT "backend has been notified");
                while (virtqueue_get_buf(vq, &len) == NULL)
                    /* do nothing */;
                printk(KERN_CRIT "backend has sent us data");
                spin_unlock_irqrestore(&crdev->lock,flags);
```

```
                        switch(cmd){
                        case CIOCGSESSION:
                            debug("CIOCGSESSION");
                            if((*host_ret<0)||(copy_to_user((struct session_op*)arg,
sess,sizeof(struct session_op)))){
                                    debug("CIOCGSESSION");
                                    return -1;
                            }
                            break;
                        case CIOCFSESSION:
                            debug("CIOCFSESSION");
                            if((*host_ret<0)){
                                    debug("CIOCFSESSION");
                                    return -1;
                            }
                            break;
                        case CIOCCRYPT:
                            debug("CIOCCRYPT");
                            if((*host_ret<0)||(copy_to_user(((struct crypt_op*) arg)->dst,
dst, cryp->len*sizeof(char)))){
                                    debug("CIOCCRYPT, with %d", err);
                                    return -1;
                            }
                            break;
                        }

                        kfree(syscall_type);
                        kfree(host_ret);
                        kfree(cmd_ptr);
                        kfree(ses_id);
                        kfree(sess);
                        kfree(cryp);
                        kfree(ses_key);
                        kfree(src);
                        kfree(dst);
                        kfree(iv);
                        debug("Leaving");

                        return *host_ret;
}
```

probe

```
/**
 * This function is called each time the kernel finds a virtio device
 * that we are associated with.
 */
static int virtcons_probe(struct virtio_device *vdev)
{
                        int ret = 0;
                        struct crypto_device *crdev;

                        debug("Entering");

                        crdev = kzalloc(sizeof(*crdev), GFP_KERNEL);
                        if (!crdev) {
                            ret = -ENOMEM;
                            goto out;
                        }

                        crdev->vdev = vdev;
                        vdev->priv = crdev;

                        crdev->vq = find_vq(vdev);
                        if (!(crdev->vq)) {
                            ret = -ENXIO;
                            goto out;
                        }

                        /* Other initializations. */
                        spin_lock_init(&crdev->lock);

                        /**
                         * Grab the next minor number and put the device in the driver's list.
                         **/
                        spin_lock_irq(&crdrvdata.lock);
                        crdev->minor = crdrvdata.next_minor++;
                        list_add_tail(&crdev->list, &crdrvdata.devs);
                        spin_unlock_irq(&crdrvdata.lock);
                        debug("Got minor = %u", crdev->minor);

                        debug("Leaving");

out:
                        return ret;
```

```
}
```

Ο κώδικας του **backend** είναι:

```
static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
{
                        VirtQueueElement elem;
                        unsigned int *syscall_type;
                        int *host_fd;
                        int* ret;
                        unsigned int* ioctl_cmd;
                        DEBUG_IN();

                        char output_str[100];
                        if (!virtqueue_pop(vq, &elem)) {
                            DEBUG("No item to pop from VQ :(");
                            return;
                        }

                        DEBUG("I have got an item from VQ :)");

                        if ((host_fd = (int *) malloc(sizeof(int))) == NULL) {
                            perror("out of mem");
                            exit(1);
                        }
                        syscall_type = elem.out_sg[0].iov_base;
                        switch (*syscall_type) {
                            case VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN:
                                    DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_OPEN");
                                    host_fd = elem.in_sg[0].iov_base;
                                    *host_fd = open("/dev/crypto", O_RDWR);
                                    if (*host_fd < 0){
                                            DEBUG("I WAS UNABLE TO OPEN /dev/crypto");
                                            perror("open");
                                            return;
                                    }
                                    sprintf(output_str,"I WAS ABLE TO OPEN /dev/crypto
returning %d",*host_fd);
                                    DEBUG(output_str);
                                    break;

                            case VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE:
                                    DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_CLOSE");
                                    host_fd = elem.out_sg[1].iov_base;
                                    if (close(*host_fd) < 0){
```

```
                perror("close");
                return;
            }
            break;

        case VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL:
            DEBUG("VIRTIO_CRYPTO_SYSCALL_TYPE_IOCTL");
            host_fd = elem.out_sg[1].iov_base;
            ioctl_cmd = elem.out_sg[2].iov_base;
            sprintf(output_str,"I GOT IOCTL = %u", *ioctl_cmd);
            DEBUG(output_str);
            switch(*ioctl_cmd){
                case CIOCGSESSION:
                    DEBUG("CIOCGSESSION");
                    struct session_op *session_op =
elem.in_sg[0].iov_base;
                    unsigned char *session_key =
elem.out_sg[3].iov_base;
                    ret=elem.in_sg[1].iov_base;
                    session_op->key = session_key;

                if(ioctl(*host_fd,CIOCGSESSION,session_op)){
                        *ret = -1;
                        perror("ioctl");
                    }
                    else *ret = 0;
                    break;

                case CIOCFSESSION:
                    DEBUG("CIOCFSESSION");
                    int* ses_id = elem.out_sg[3].iov_base;
                    ret=elem.in_sg[0].iov_base;
                    if(ioctl(*host_fd,CIOCFSESSION,ses_id)){
                        perror("ioctl");
                        *ret=-1;
                    }
                    else *ret=0;
                    break;

                case CIOCCRYPT:
                    DEBUG("CIOCRYPT");
                    struct crypt_op* crypt_op =
elem.out_sg[3].iov_base;
                    unsigned char *src = elem.out_sg[4].iov_base;
```

```
                                        unsigned char *iv = elem.out_sg[5].iov_base;
                                        unsigned char *dst = elem.in_sg[0].iov_base;
                                        ret = elem.in_sg[1].iov_base;
                                        crypt_op->src=src;
                                        crypt_op->iv=iv;
                                        crypt_op->dst=dst;

                                        if(ioctl(*host_fd,CIOCCRYPT,crypt_op)){
                                                perror("ioctl");
                                                *ret= -1;
                                        }
                                        else *ret=0;
                                        break;
                                default:
                                        DEBUG("Unrecognised ioctl");
                                        break;
                        }
                        break;
                default:
                        DEBUG("Unknown syscall_type");
                        break;
        }

        virtqueue_push(vq, &elem, 0);
}
```

Θέμα 3 (Επαναληπτική 2016)

Όλος ο κώδικας είναι:

```
/* The struct that is being exchanged via the virtqueue. */
struct crypto_buffer {
                        char *input; /* The input string. */
                        char *output; /* The output string. */
                        unsigned int len; /* The length of the input. */
                        char *key; /* The key used for encryption/decryption. */
                        unsigned int key_len;  /* The length of the key. */
};

struct crypto_device {
                        struct virtqueue *vq;
                        struct semaphore vq_lock;
} crypto_dev;

static long virtio_crypto_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
                        struct statterlist cmd_sg, cbuf_sg, cbuf_input_sg, cbuf_output_sg,
cbuf_key_sg, *sgs[5];
                        struct crypto_device *cdev = &crypto_dev;
                        struct crypto_buffer *cbuf;
                        char *input_ptr, *output_ptr, *key_ptr;
                        unsigned int len;
                        long ret = 0;

                        /* Fetch all necessary data from userspace. */
                        cbuf = kzalloc(sizeof(*cbuf), GFP_KERNEL);
                        if (!cbuf) return -ENOMEM;
                        if (copy_from_user(cbuf, (struct crypto_buffer *) arg, sizeof(*cbuf))) {
                            ret = -EFAULT;
                            goto out;
                        }

                        len = cbuf->len;
                        input_ptr = kzalloc(len*sizeof(char), GFP_KERNEL);
                        if (!input_ptr) {
                            ret = -ENOMEM;
                            goto out;
                        }
                        if (copy_from_user(input_ptr, cbuf->input, len)) {
                            ret = -EFAULT;
```

```
                goto out;
            }

            key_ptr = kzalloc(cbuf->key_len*sizeof(char), GFP_KERNEL);
            if (!key_ptr) {
                ret = -ENOMEM;
                goto out;
            }
    if (copy_from_user(key_ptr, cbuf->key, key_len)) {
        ret = -EFAULT;
        goto out;
    }

    output_ptr = kzalloc(len*sizeof(char), GFP_KERNEL);
            if (!output_ptr) {
                ret = -ENOMEM;
                goto out;
            }

            switch (cmd) {
                case ENCRYPT:
                case DECRYPT:
                    sg_init_one(&cmd_sg, &cmd, sizeof(cmd));
                    sg[0] = &cmd_sg;

                    sg_init_one(&cbuf_sg, cbuf, sizeof(*cbuf));
                    sg[1] = &cbuf_sg;

                    sg_init_one(&cbuf_input_sg, input_ptr, len);
                    sg[2] = &cbuf_input_sg;

                    sg_init_one(&cbuf_key_sg, key_prt, key_len);
                    sg[3] = &cbuf_key_sg;

                    sg_init_one(&cbuf_output_sg, output_ptr, len);
                    sg[4] = &cbuf_output_sg;

                    /* Send sgs and notify the host. */
                    down_interruptible(&c_dev->vq_lock);
                    if (virtqueue_add_sgs(c_dev->vq, sgs, 4, 1, cbuf,
GFP_ATOMIC)) {
                        up(&c_dev->vq_lock);
                        ret = -EINVAL;
                        goto out;
```

```
                        }
                        virtqueue_kick(vq);

                        /* Spin on the virtqueue until the buffer is back. */
                        while (virtqueue_get_buf(c_dev->vq, &len) == NULL)
                                /* do nothing */;
                        up(&c_dev->vq_lock);

                    break;

                    default:
                        ret = -EINVAL;
                        goto out;
                }

                /* Copy all necessary data back to userspace. */
                if (copy_to_user((struct crypto_buffer *) arg, cbuf, sizeof(*cbuf))) ret
= -EINVAL;

out:
                kfree(key_ptr);
                kfree(output_ptr);
                kfree(input_ptr);
                kfree(cbuf);
                return ret;
}

void vq_crypto_callback(VirtIODevice *vdev, VirtQueue *vq)
{
                VirtQueueElement elem;
                struct crypto_buffer *cbuf;
                char *input, *output, *key;
                int ret;
                int fd; /* This is an open instance of /dev/crypto on the host. */
                char *input_saved, *output_saved, *key_saved;
                unsigned int cmd;

                if (!virtqueue_pop(vq, &elem))
                    return;

                cmd = *elem.out_sg[0].iov_base;
                cbuf = *elem.out_sg[1].iov_base;
                input = *elem.out_sg[2].iov_base;
                key = *elem.out_sg[3].iov_base;
```

```
                output = *elem.in_sg[0].iov_base;
                cbuf->input = input;
                cbuf->key = key;
                cbuf->output = output;

                /* ioctl() to the host device driver, it is always successful. */
                ret = ioctl(fd, cmd, cbuf);
                if (ret < 0) perror("ioctl");

                virtqueue_push(vq, &elem, 0);
                virtio_notify(vdev, vq);
}
```

Στο frontend έχει γίνει χρήση των ***copy_to_user/copy_from_user*** περισσότερες της μίας φορές. Την πρώτη φορά, είναι ***copy_from_user(cbuf, (struct crypto_buffer *) arg, sizeof(*cbuf))***, όπου αντιγράφουμε από το arg, το οποίο μεταχειριζόμαστε ως διεύθυνση χώρου χρήστη, γεγονός που επιβάλλει τη χρήστη της. Στη συνέχεια, ξαναχρησιμοποιούμε τη συγκεκριμένη συνάρτηση, αντιγράφοντας δεδομένα από πεδία του ***cbuf***, τα οποία είναι δείκτες και παραπέμπουν σε διευθύνσεις που είναι στο χώρο χρήστη. Τέλος, αντιγράφουμε τα δεδομένα πίσω στο χώρο χρήστη με την εντολή ***copy_to_user((struct crypto_buffer *) arg, cbuf, sizeof(*cbuf))***.
Η παραπάνω υλοποίηση πάσχει για τον εξής λόγο. Όταν μία διεργασία περιμένει απάντηση από το backend, έχει το σημαφόρο της δομής, με αποτέλεσμα άλλες διεργασίες που θέλουν να χρησιμοποιήσουν τη συσκευή κρυπτογράφησης να μην μπορούν. Επιπλέον, καθώς περιμένει απάντηση από το backend, πραγματοποιεί μίας μορφής ***busy-wait loop***, γεγονός που επιβαρύνει τον επεξεργαστή. Για τους παραπάνω λόγους, θα ήταν προτιμότερη η εναλλακτική υλοποίηση που προτείνεται στα θέματα, όπου οι διεργασίες κοιμούνται καθώς περιμένουν το backend.
Σε μία τέτοια υλοποίηση, σημαντικό ρόλο έχει η ***virtio_notify***. Η συγκεκριμένη συνάρτηση προκαλεί μία διακοπή όταν προστίθεται κάτι στην ουρά από το backend. Στην υλοποίηση που έχουμε δώσει παραπάνω, αυτό δεν βοηθάει κάπου, αφού η παραλαβή των δεδομένων από το frontend βασίζεται στο busy-wait loop που αναφέρθηκε. Για την εναλλακτική υλοποίηση όμως, είναι απαραίτητη, αφού η ***crypto_recv*** εκτελείται σε ***interrupt context*** και περιμένει κατάλληλο interrupt από το backend για να ενεργοποιηθεί.
Στην πρώτη υλοποίηση μία διεργασία που περιμένει δεδομένα είναι ***running***, ενώ στην δεύτερη είναι ***waiting***.