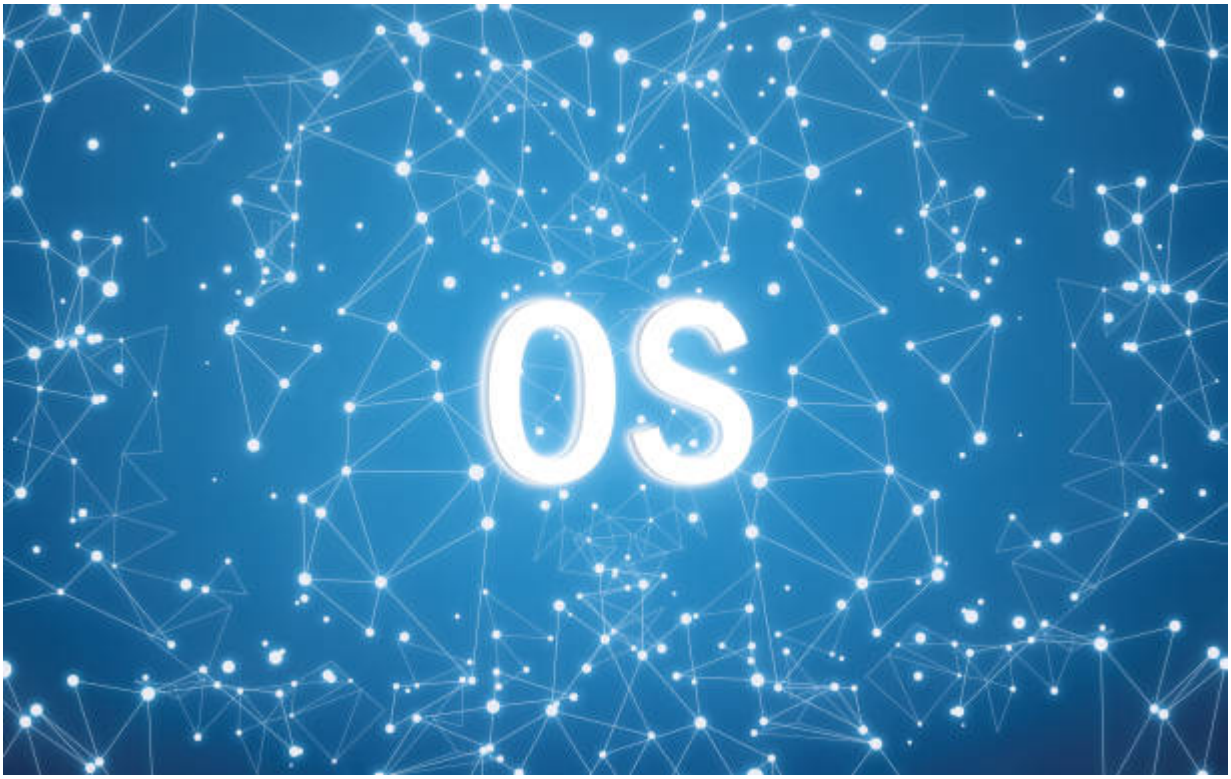
Εργαστήριο Λειτουργικών Συστημάτων

Χριστόπουλος Νικόλαος: 03117065
Γιαννούλης Παναγιώτης: 03117812
31 Ιανουαρίου 2020



Εργαλείο Chat με χρήση του BSD Sockets API

Το πρώτο ζητούμενο του παρόντος εργαστηρίου είναι η δημιουργία ενός εργαλείου chat με την χρήση του BSD Sockets API. Αρχικά, τα δύο άκρα θα επικοινωνούν μέσω του πρωτοκόλλου TCP και τα δεδομένα που θα μεταφέρονται πάνω από την TCP/IP επικοινωνία θα κρυπτογραφούνται. Θα δούμε στη συνέχεια πως θα επιτευχθεί η κρυπτογράφηση.

Η αρχιτεκτονική του συστήματος μας αποτελείται από n clients και έναν server ως ενδιάμεσο που λαμβάνει ένα μήνυμα από κάποιο client και το στέλνει στους υπόλοιπους. Δηλαδή, όλοι

οι clients έχουν συνδεθεί στον server και επικοινωνούν αποκλειστικά και μόνο μαζί του. Στην παρακάτω εικόνα βλέπουμε την "σχέση" client-server:

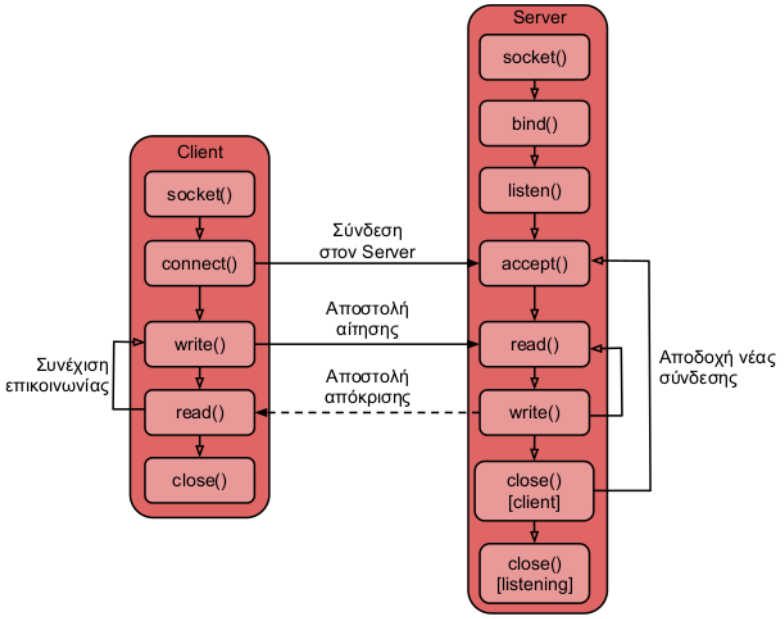


Figure 1: Architecture of the system

Όλοι, λοιπόν, δημιουργούν ένα socket. Ο server "ακούει" σε μια συγκεκριμένη θύρα και οι clients συνδέονται σε αυτήν. Οι δύο αυτές ενέργειες φαίνονται στα παρακάτω κομμάτια κώδικα:

```
if ((server_sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
  }
  fprintf(stderr, "Created TCP socket\n");

  /* Bind to a well-known port */
  memset(&sa, 0, sizeof(sa));
  sa.sin_family = AF_INET;
  sa.sin_port = htons(TCP_PORT);
  sa.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(server_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(server_sd, TCP_BACKLOG) < 0) {
      perror("listen");
      exit(1);
    }
```

Αξίζει να παρατηρήσουμε ότι στο UNIX τα πάντα είναι αρχεία, άρα και η δημιουργία ενός socket επιστρέφει έναν file descriptor. Επιπλέον, όπως αναφέρθηκε και στην αρχή ο τύπος του socket είναι SOCK_STREAM που επιβεβαιώνει την αξιόπιστη μεταφορά ρεύματος από bytes (TCP connection). Η συνάρτηση bind() δένει το socket fd με μια οικογένεια διευθύνσεων (εδώ IPv4) σε μία θύρα. Τέλος ο server μας με την συνάρτηση listen() είναι έτοιμος να δεχθεί συνδέσεις.

Αυτό που πρέπει να μας προβληματίσει είναι ότι ο server έχει αναλάβει πολλές λειτουργίες που πρέπει να διαχειριστεί ταυτόχρονα. Για παράδειγμα, μπορεί να χρειαστεί να λάβει κάποιο μήνυμα από έναν client και επιπλέον να αποδεχθεί μια καινούργια σύνδεση με κάποιον άλλον ή μπορεί να λάβει ταυτόχρονα μηνύματα από πολλούς clients ή όλα τα παραπάνω μαζί. Χωρίς να έχει σημασία ποια από αυτές λειτουργίες θα γίνει πρώτη πρέπει να είμαστε σίγουροι ότι θα πραγματοποιηθούν όλες. Με άλλα λόγια, θα πρέπει να παρακολουθεί τον δικό του file descriptor(για καινούργιες συνδέσεις) και ταυτόχρονα όλους τους file descriptors των συνδεόμενων clients(για τυχόν νέα μηνύματα). Επιπλέον από την πλευρά του ο client οφείλει να ελέγχει δύο file descriptors, ο πρώτος είναι ο δικός του σε περίπτωση που λάβει κάποιο μήνυμα από τον server. Ο δεύτερος είναι το stdin στο οποίο ο user πληκτρολογεί τα μηνύματα. Για την επίτευξη αυτού του στόχου χρησιμοποιούμε την **select()**. Η select() μάς δίνει τη δυνατότητα να παρακολουθούμε πολλαπλούς file descriptors. Ο τρόπος; Δημιουργούμε ένα set και τοποθετούμε όλους τους fd σε αυτό. Από εκεί και πέρα η select() αναμένει κάποια ενέργεια σε κάποιον από αυτούς και μας ενημερώνει όταν πρέπει. Μας πληροφορεί φυσικά για το ποιος fd είναι έτοιμος για κάποια λειτουργία I/O. Άρα, για τον server έχουμε τα εξής:

```
/* Loop forever, accept()ing connections */
 for (;;) {
    // clear socket set
    FD_ZERO(&inset);

    // add server sd to set
    FD_SET(server_sd, &inset);
    max_sd = server_sd;

    for (int i = 0; i < MAX_CLIENTS; i++) {
      client_sd = client_socket[i];

      if (client_sd > 0)
        FD_SET(client_sd, &inset);

      if (client_sd > max_sd)
        max_sd = client_sd;
      }

      action = select(max_sd + 1, &inset, NULL, NULL, NULL);
      if ((action < 0) && (errno!=EINTR)) {
```

```
        perror("select");
        exit(1);
      }

      // incoming connection
      if (FD_ISSET(server_sd, &inset)) {
        if ((newsd = accept(server_sd, (struct sockaddr *)&sa, &len)) <
   0) {
          perror("accept");
          exit(1);
        }
        if (!inet_ntop(AF_INET, &sa.sin_addr, addrstr, sizeof(addrstr)))
   {
          perror("could not format IP address");
          exit(1);
        }
        fprintf(stderr, "Incoming connection from %s:%d\n",
              addrstr, ntohs(sa.sin_port));
        for (int i = 0; i < MAX_CLIENTS; i++) {
          if (client_socket[i] == 0) {
            client_socket[i] = newsd;
            break;
          }
        }

      }

      //incomming message from a client socket
      for (int i = 0; i < MAX_CLIENTS; i++) {

          client_sd = client_socket[i];
          if (FD_ISSET(client_sd, &inset))
          .
          .
          .
```

Στην αρχή το set μας είναι κενό οπότε τοποθετούμε τον server_sd και τους client_sd. Έπειτα, περιμένουμε κάποιο action. Αν αυτό έρθει από τον server socket descriptor σημαίνει ότι κάποιος καινούργιος client θέλει να συνδεθεί μαζί μας, οπότε κάνουμε το accept και ενημερώνουμε τον πίνακα με τους clients. Αλλιώς έχουμε καινούργιο μήνυμα από κάποιον από τους clients. Θα δείξουμε στη συνέχεια - αναλύοντας και την κρυπτογράφηση - τι γίνεται σε αυτή την περίπτωση. Όσον αφορά τον client οι δύο εναλλακτικές είναι οι παρακάτω:

```
// receiving message
if (FD_ISSET(client_sd, &inset)) {
  ....

}

// sending message
if (FD_ISSET(STDIN_FILENO, &inset)) {
```

```
  ....
}
```

Κρυπτογράφηση με χρήση του οδηγού συσκευής cryptodev

Για την κρυπτογράφηση των δεδομένων θα χρησιμοποιήσουμε τον οδηγό συσκευής crypto dev-linux. Κάνοντας κλήσεις ioctl() στο ειδικό αρχείο /dev/crypto ο client θα έχει τη δυνατότητα να κρυπτογραφήσει τα δεδομένα του πριν τα μεταφέρει στον server. Ομοίως, οι άλλοι clients θα μπορούν πάλι χρησιμοποιώντας την συσκευή να αποκρυπτογραφήσουν τα αντίστοιχα δεδομένα (με τη χρήση εννοείται ενός κοινού κλειδιού -συμμετρική κρυπτογράφηση-). Αρχικά, προηγείται το άνοιγμα του αρχείου /dev/crypto και στη συνέχεια η δημιουργία ενός CIOCGSESSION στο οποίο έχουν οριστεί κάποιες παράμετροι όπως το key, το cipher κ.α.

```
sess.cipher = CRYPTO_AES_CBC;
sess.keylen = KEY_SIZE;
sess.key = key;


if (ioctl(cfd, CIOCGSESSION, &sess)) {
  perror("ioctl(CIOCGSESSION)");
  return 1;
}
```

Έπειτα είμαστε έτοιμοι για την κρυπτογράφηση:

```
struct crypt_op cryp;
unsigned char   encrypted[size];
//initialize to zero cryp struct
memset(&cryp, 0, sizeof(cryp));

/*
* Encrypt data.in to data.encrypted
*/
cryp.ses = sess.ses;
cryp.len = size;
cryp.src = arr;
cryp.dst = encrypted;
cryp.iv = iv;
cryp.op = COP_ENCRYPT;

if (ioctl(cfd, CIOCCRYPT, &cryp)) {
  perror("ioctl(CIOCCRYPT)");
  return 1;
}
```

Για την κρυπτογράφηση χρησιμοποιούμε την δομή **crypt_op** που περιέχει πεδία όπως source, destination, operation. Τα όνοματα αυτών ανταποκρίνονται και στην λειτουργία τους.

Ως source ορίζεται ο buffer με τα δεδομένα που θα κρυπτογραφηθούν/αποκρυπτογραφηθούν και ως destination ο buffer με το αποτέλεσμα της κρυπτογράφησης/αποκρυπτογράφησης. iv είνα ένας initialized vector που χρησιμοποιείται για την κρυπτογράφηση. Μέσω της ioctl χρησιμοποιούμε την συσκευή /dev/crypto για την εκάστοτε λειτουργία. Τέλος, για την μεταφορά των δεδομένων από τον client στον server χρησιμοποιούμε αρχικά την read() για να διαβάσουμε από το stdin το μήνυμα του χρήστη και αφού το κρυπτογραφήσουμε το γράφουμε στον socket descriptor μας ώστε να ενημερωθεί ο server και να το λάβει.

```
n = read(1, buf, DATA_SIZE-strlen(format2)-1);
if (n < 0) {
        perror("read");
        exit(1);
}
buf[n] = '\0';
/*
now we have to encrypt the message and send it to server
*/

snprintf(message, strlen(buf) + strlen(format2)+1, "\033[0m%s", buf);
encrypt(cfd, message, DATA_SIZE);

// send message to server
if (insist_write(client_sd, message, DATA_SIZE) != DATA_SIZE) {
  perror("write");
  exit(1);
}
```

Αντίστοιχα, όταν λαμβάνουμε κάποιο μήνυμα (αυτό έχει γραφτεί στον client sd) το διαβάζουμε από τον socket descriptor και το εκτυπώνουμε στο stdout ώστε να το δει ο χρήστης.
Ο server πρέπει τα μηνύματα που λαμβάνει να τα στέλνει σε όλους τους άλλους. Το τελευταίο σημείο που πρέπει να προσέξουμε είναι το πως αναγνωρίζει ο server ότι κάποιος client έχει αποσυνδεθεί. Ο server αντιλαμβάνεται πως κάποιος client έχει αποχωρήσει από το chat όταν υπάρχει ενέργεια στον αντίστοιχο client sd αλλά δεν υπάρχουν δεδομένα να διαβαστούν οπότε επιστρέφονται 0 χαρακτήρες.

```
if ((n = read(client_sd, buf, DATA_SIZE)) == 0) {
      getpeername(client_sd, (struct sockaddr*) &sa, &len);
      printf("Host with ip:port %s:%d disconnected\n", inet_ntop(AF_INET,
    &sa.sin_addr, addrstr, sizeof(addrstr)), ntohs(sa.sin_port));
      close(client_sd);
      client_socket[i] = 0;
}
```

Αν υπάρχουν δεδομένα στέλνονται στους υπόλοιπους.

```
for (int k = 0; k < MAX_CLIENTS; k++) {
  if (client_socket[k] == 0 || client_socket[k] == client_sd) {
                  continue;
  }
  send_to_sd = client_socket[k];
  insist_write(send_to_sd, buf, DATA_SIZE+1);
}
```

Συσκευή κρυπτογράφησης για QEMU-KVM

Ως τρίτο και τελευταίο μέρος της εργαστηριακής άσκησης είναι η δημιουργία μιας εικονικής συσκευής κρυπτογράφησης για εικονικές μηχανές που εκτελούνται στον QEMU. Συγκεκριμένα, καλούμαστε να χρησιμοποιήσουμε την τεχνική της **παραεικονικοποίησης** και του προτύπου **virtio** ώστε να συνεργαστούμε με τον host για να χρησιμοποιήσουμε την πραγματική συσκευή /dev/crypto με τελικό στόχο την καλύτερη επίδοση. Η αρχιτεκτονική του συστήματος είναι η εξής:

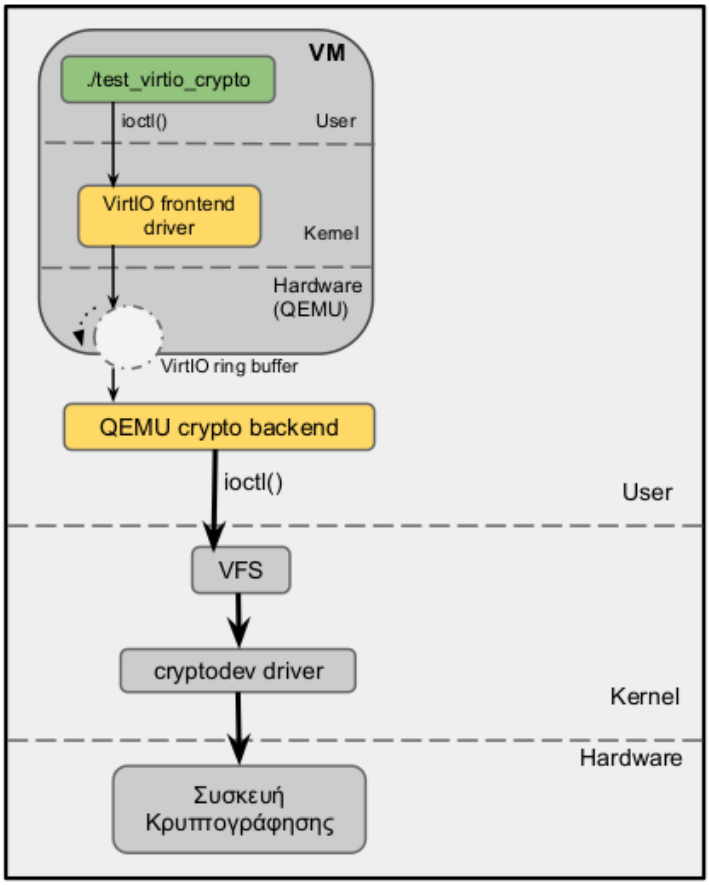


Figure 2: Architecture of the system

Ουσιαστικά, στην συγκεκριμένη υλοποιήση οι κλήσεις ioctl() που χρησιμοποιεί ο χρήστης του guest μηχανήματος θα πρέπει να μεταφέρονται κατάλληλα στον hypervisor μέσω της συσκεύσης virtio και συγκεκριμένα των virtqueues ώστε ο τελευταίος να καλεί την πραγματική συσκευή.

Υλοποίηση:

Πριν αναφερθούμε στην υλοποιήση αυτή καθ' αυτή, είναι σημαντικό να κατανοήσουμε την διαδρομή που ακολουθούν τα δεδομένα και τα 3 διαφορετικά επίπεδα στα οποία θα βρίσκονται. Το πρώτο εξ' αυτών είναι το user space του guest μηχανήματος σε μία διέργασια. Το δεύτερο είναι το kernel space του guest μηχανήματος και τελευταίο επίπεδο αποτελεί ο Hypervisor, ο qemu δηλαδή που βρίσκεται στο user space του host μηχανήματος (εννοείται στη συνέχεια με τη χρήση του /dev/crypto μεταφερόμαστε και στο kernel space του host μηχανήματος, απλώς αυτό το επίπεδο το αναλαμβάνει cryptodev driver που είναι ήδη υλοποιημένος). Μιλάμε, λοιπόν, για 3 εντελώς διαφορετικές καταστάσεις και αντιλαμβανόμαστε εξαρχής ότι η διαδικασία μετάβασης των δεδομένων από την μία κατάσταση στην άλλη δεν είναι και τόσο εύκολη υπόθεση. Ας ξεκινήσουμε όμως από τις virtio συσκεύες που θα μας βοηθήσουν στη μετάβαση από τον guest στον host και το αντίστροφο.

Virtio Συσκευές στον QEMU

Θα δημιουργήσουμε μια virtio cryptodev συσκευή. Αυτή η συσκευή ορίζεται και στον qemu και στον guest. Στον qemu όλες οι virtio συσκευές βρίσκονται στην κλάση TYPE_VIRTIO_PCI. Με την έναρξη του guest μηχανήματος αναγνωρίζεται η virtio συσκευή που έχουμε ορίσει μέσω της παραμέτρου -device. Αρχικοποιούνται τα βασικά πεδία της κλάσης virtio της συσκευής μέσω της **virtio_cryptodev_class_init** και έπειτα καλείται η συνάρτηση **virtio_cryptodev_realize** που δημιουργεί την virtio συσκευή (δηλαδή τις απαραίτητες δομές της) και προσθέτει σε αυτήν μια virtio ουρά που θα χρησιμοποιηθεί για την επικοινωνία guest-qemu.

```
static void virtio_cryptodev_realize(DeviceState *dev, Error **errp)
{
    VirtIODevice *vdev = VIRTIO_DEVICE(dev);

    DEBUG_IN();

    virtio_init(vdev, "virtio-cryptodev", VIRTIO_ID_CRYPTODEV, 0);
    virtio_add_queue(vdev, 128, vq_handle_output);
}
```

Άρα, με την έναρξη του qemu έχει οριστεί πλήρως η συσκευή μας, το struct virtio_device και το virtqueue.

Virtio Συσκευές στο εικονικό μηχάνημα

Τώρα πρέπει να χρησιμοποιήσουμε έναν virtio_driver για να ορίσουμε στον kernel του guest την virtio αυτή συσκευή ώστε να μπορέσουμε να την χρησιμοποιήσουμε. Η συνάρτηση **register_virtio_driver** εγγράφει τον virtio driver στον πυρήνα. Το όρισμα που δέχεται είναι ένα virtio_driver struct που περιέχει τις κατάλληλες συναρτήσεις που θα εκτελούνται όπως probe, remove κ.α.. Η συνάρτηση probe καλείται κάθε φορά που αναγνωρίζεται μια νέα virtio συσκευή στον guest. Επιπλέον μέσω της συνάρτησης **crypto_chrdev_init** ορίζουμε τον driver για τις συσκευές /dev/crypto οι οποίες θα χρησιμοποιούνται από το user space

του εικονικού μηχανήματος. Η διαδικασία είναι γνωστή από την προηγούμενη εργαστηριακή άσκηση. Χρησιμοποιούνται συναρτήσεις όπως **cdev_init, register_chrdev_region, cdev_add**.

Αυτό που μένει, λοιπόν, είναι να αντιστοιχιστεί κάθε virtio device του backend με ένα character device. Για το σκοπό αυτό καλείται η συνάρτηση probe για κάθε virtio συσκευή που έχει οριστεί στον qemu και η οποία ταιριάζει με τα χαρακτηριστικά της με βάση τον id_table. Σ' αυτή τη συνάρτηση δεσμεύουμε την κατάλληλη μνήμη για τη δομή του character device του frontend, τον συνδέουμε με το αντίστοιχο virtio device, τον προσθέτουμε στη λίστα με όλες τις virtio συσκευές, και του δίνουμε τον επόμενο minor number.

```
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. */
/* ?? */
INIT_LIST_HEAD(&crdev->list);
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);
```

Δομές

Για τις συσκευές /dev/crypto πρέπει να υλοποιήσουμε τις συναρτήσεις του character device driver ώστε να λειτουργούν με τον επιθυμητό τρόπο. Πριν από αυτό πρέπει να δούμε τις σημαντικότερες δομές που χρησιμοποιούνται στον οδηγό:

```
/**
 * Device info.
 **/
struct crypto_device {
  /* Next crypto device in the list, head is in the crdrvdata struct */
  struct list_head list;

  /* The virtio device we are associated with. */
  struct virtio_device *vdev;

  struct virtqueue *vq;

  struct semaphore lock;
  /* The minor number of the device. */
  unsigned int minor;
};


/**
 *  Crypto open file.
 **/
struct crypto_open_file {
  /* The crypto device this open file is associated with. */
  struct crypto_device *crdev;

  /* The fd that this device has on the Host. */
  int host_fd;
};
```

Η πρώτη όπως λέει και το όνομά της αντιστοιχίζεται με μια συσκευή crypto. Η δεύτερη αφορά σε όλα τα ανοιχτά αρχεία τέτοιων συσκευών. Στη συνέχεια θα αντιληφθούμε την λειτουργία των πεδίων και των δύο struct.
Η πρώτη συνάρτηση που υλοποιούμε είναι η **open()**. Η δομή που ακολουθήθηκε για την Virtqueue της συσκευής virtio-cryptodev-pci για κάθε κλήση συστήματος που υλοποιεί ο οδηγός μας είναι η παρακάτω:

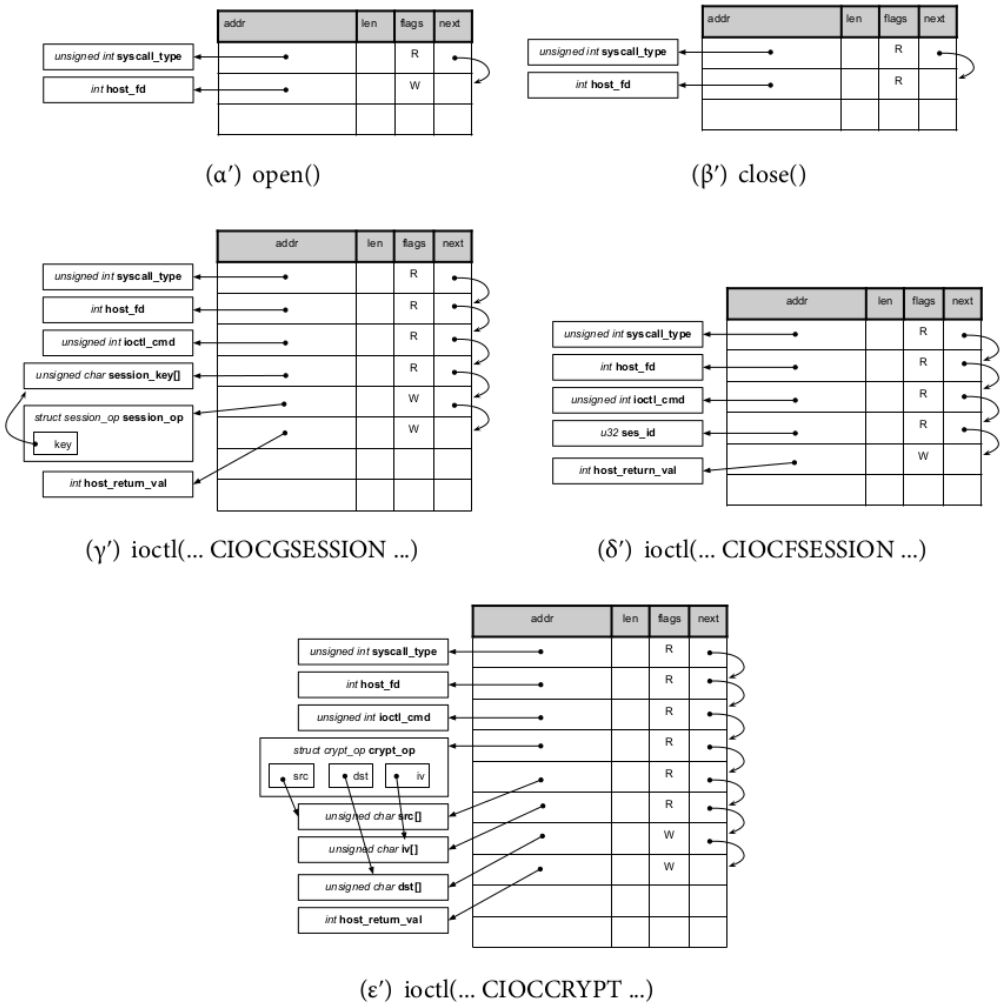


Figure 3: Η προτεινόμενη δομή της Virtqueue

Συναρτήσεις:

Η **open()** θέλουμε να μεταφέρει το μήνυμα του ανοίγματος μιας συσκευής /dev/crypto στον hypervisor(backend), εκείνος να κάνει το open της πραγματικής συσκευής και να μεταφέρει τον αντίστοιχο file descriptor στον guest(frontend). Για την μεταφορά των δεδομένων μεταξύ frontend και backend χρησιμοποιούμε **scatter-gather lists**. Έτσι, στην open μεταφέρουμε τον τύπο της κλήσης συστήματος και επιπλέον μια λίστα που αφορά στον file descriptor στον οποίο θα γράψει το backend την απάντηση (host_fd).

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

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, host_fd, sizeof(*host_fd));
sgs[num_out + num_in++] = &host_fd_sg;
```

Έπειτα, με τη συνάρτηση **virtqueue_add_sgs** προσθέτουμε στην ουρά virtqueue όλες τις λίστες που έχουμε ορίσει. Η **virtqueue_kick** αναλαμβάνει να ενημερώσει το backend ότι υπάρχουν καινούργια δεδομένα. Με την **virtqueue_get_buf** λαμβάνουμε τις απαντήσεις από το backend. Όσο αυτό είναι NULL αναμένουμε. Πρέπει να προσέξουμε έδω ότι οι παραπάνω συναρτήσεις πρέπει να εκτελεσθούν ατομικά. Αυτό συμβαίνει γιατί έχουμε μία ουρά (virtqueue) και δεν είναι δυνατόν δυο διεργασίες που έχουν ανοίξει το ίδιο device (π.χ. /dev/cryptodev0) να προσθέτουν ταυτόχρονα στοιχεία στην ουρά αυτή. Γι' αυτό το λόγο χρησιμοποιούμε έναν σημαφόρο.

```
if(down_interruptible(&crdev->lock)) {
    ret = -ERESTARTSYS;
    debug("open: down_interruptible");
    goto fail;
}
err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                        &syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(vq);
while (virtqueue_get_buf(vq, &len) == NULL)
    /* do nothing */;


up(&crdev->lock);
```

Όσον αφορά την **ioctl()** που εκτελεί τις εντολές για την κρυπτογράφηση ακολουθείται ακριβώς ίδια διαδικασία/λογική. Υπάρχουν όμως μερικά λεπτά σημεία που πρέπει να προσέξουμε και να αναφερθούμε σε αυτά.

Για την ioctl() πρέπει να μεταφέρουμε τον host_fd και την εντολή ioctl που θα γίνει. Επιπλέον, ανάλογα με την εντολή πρέπει να μεταφερθούν και κάποιες δομές τις οποίες λαμβάνουμε από το user space. Ας δούμε για παράδειγμα την περίπτωση της **CIOCCRYPT**. Για αρχή το πεδίο arg της ioctl περιέχει το struct crypt_op. Για να λάβουμε με ασφάλεια το struct αυτό πρέπει να χρησιμοποιήσουμε την **copy_from_user()**. Όμως, μας αρκεί αυτό; Αν κοιτάξουμε ένα βήμα παραπέρα θα καταλάβουμε ότι αυτό δεν είναι αρκετό γιατί μέσα στη δομή υπάρχουν pointers της εικονικής μνήμης της διεργασίας σε διάφορα πεδία όπως cryp.iv, cryp.src. Και αυτοί, λοιπόν, οι pointers πρέπει να γίνουν dereference με ασφάλεια ώστε να ολοκληρωθεί επιτυχώς η διαδικασία της μεταφοράς των δεδομένων. Παρακάτω φαίνεται ακριβώς αυτό σε μορφή κώδικα:

```
if (copy_from_user(cryp, (struct crypt_op*) arg, sizeof(struct crypt_op))
   ) {
      return -EFAULT;
}
```

```
data_size = cryp->len;
cryp_dst = kzalloc(data_size, GFP_KERNEL);
cryp_iv = kzalloc(VIRTIO_CRYPTODEV_BLOCK_SIZE, GFP_KERNEL);
cryp_src = kzalloc(data_size, GFP_KERNEL);

sg_init_one(&crypto_sg, cryp, sizeof(struct crypt_op));
sgs[num_out++] = &crypto_sg;

if (copy_from_user(cryp_src, cryp->src, data_size)) {
      return -EFAULT;
}
sg_init_one(&cryp_src_sg, cryp_src, data_size);
sgs[num_out++] = &cryp_src_sg;

if (copy_from_user(cryp_iv, cryp->iv, VIRTIO_CRYPTODEV_BLOCK_SIZE)) {
      return -EFAULT;
}

sg_init_one(&cryp_iv_sg, cryp_iv, VIRTIO_CRYPTODEV_BLOCK_SIZE);
sgs[num_out++] = &cryp_iv_sg;
```

Επιπλέον, κάποιος μπορεί να αναρωτηθεί πώς γνωρίζουμε το μέγεθος της εισόδου που θέλουμε να κρυπτογραφήσουμε ώστε να πάρουμε τον σωστό αριθμό bytes μέσω της copy_from_user. Η δομή **crypt_op** έχει το πεδίο **len** που μας παρέχει αυτή την πληροφορία. Έπειτα, ακολουθούμε την ίδια διαδικασία για την ενημέρωση του backend και την παραλαβή των απαντήσεων. Στο συγκεκριμένο σημείο τίθεται το εξής ερώτημα, το οποίο αποτελεί και βήμα για να μεταφερθούμε στην υλοποιήση του backend. Το backend που βρίσκεται στο user space του host μηχανήματος κάνει κάποιο ασφαλές copy των δεδομένων που μεταφέρονται μέσω των scatter gather lists από το guest μηχάνημα; Η απάντηση είναι **όχι**. Δεν χρειάζεται να συμβεί αυτό μιας και το εικονικό μηχάνημα έχει δημιουργηθεί από τον ίδιο τον qemu. Ο τελευταίος έχει δεσμεύσει την απαραίτητη μνήμη για αυτό το μηχάνημα, την οποία ελέγχει πλήρως(είναι, δηλαδή, δική του μνήμη). Επομένως, οι σελίδες της μνήμης που μεταφέρονται από τον guest περιέχονται στην μνήμη του qemu. Ας δούμε όμως πως το backend λαμβάνει αυτά τα δεδομένα και πως δίνει τις καταλλήλες απαντήσεις.

Αρχικά, κάθε φορά που υπάρχουν νέα δεδομένα στην ουρά καλείται η vq_handle_output(). Έπειτα, η συνάρτηση **virtqueue_pop(vq, sizeof(VirtQueueElement))** παίρνει από την ουρά τις λίστες. Συγκεκριμένα, επιστρέφεται η δομή VirtQueueElement που έχει ως πεδία τα **in_sg / out_sg** (κ.α.), δηλαδή, τις εισερχόμενες και εξερχόμενες ουρές. Οι πρώτες είναι αυτές στις οποίες θα γραφτεί η απάντηση, ενώ από τις δεύτερες λαμβάνουμε τα αιτήματα από το frontend. Τα πεδία in_sg και out_sg είναι τύπου iovec. Η δομή αυτή περιέχει τον pointer(iov_base) στα δεδομένα αλλά και το μέγεθος των δεδομένων. Έτσι, πρώτα απ' όλα παίρνουμε το syscall_type και με βάση αυτό εκτελούμε την κατάλληλη κλήση συστήματος. Στην open, γίνεται το άνοιγμα του αρχείου /dev/crypto στο πραγματικό μηχάνημα, επιστρέφεται ο host_fd, τον οποίο εγγράφουμε στην πρώτη θέση των in_sgs σύμφωνα με την σύμβαση της εικόνας 3.

```
elem = virtqueue_pop(vq, sizeof(VirtQueueElement));
if (!elem) {
```

```
    DEBUG("No item to pop from VQ :(");
    return;
}

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

syscall_type = elem->out_sg[0].iov_base;
switch (*syscall_type) {
    case VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN:
      DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN");
      DEBUG("hello world");
      int cfd = open("/dev/crypto", O_RDWR);
      host_fd = elem->in_sg[0].iov_base;
      *host_fd = cfd;


      break;
    .
    .
    .
```

Τα ίδια ακριβώς βήματα κάνουμε και για τα υπόλοιπα system calls. Όταν είναι έτοιμες οι απαντήσεις το backend χρησιμοποιεί την συνάρτηση virtqueue_push(vq, elem, 0) για να τις προσθέσει στην ουρά και έπειτα με την virtio_notify(vdev, vq) ενημερώνει το frontend ότι τα δεδομένα είναι έτοιμα μέσω ενός interrupt. Όμως, στο frontend δεν έχει υλοποιηθεί κάποιος handler που θα διαχειριστεί αυτό το interrupt. Συγκεκριμένα, η vq_has_data() είναι αυτή που καλείται να τρέξει σε interrupt context, αλλά στη συγκεκριμένη υλοποιήση δεν την εκμεταλλευόμαστε. Γι' αυτό το λόγο χρησιμοποιούμε το loop while (virtqueue_get_buf(vq, &len) == NULL). Έτσι, αποφεύγουμε την χρήση των interrupts που θα έκανε λίγο πιο περίπλοκη την άσκηση.
Τέλος, τις απαντήσεις που παίρνουμε - το αποτέλεσμα μιας κρυπτογράφησης για παράδειγμα - πρέπει να τις μεταφέρουμε και πάλι στο user space του guest. Χρησιμοποιούμε, λοιπόν, την συνάρτηση copy_to_user. Επίσης για κάθε κλήση συστήματος αναμένουμε και ένα return value ώστε να ενημερωθούμε αν ολοκληρώθηκαν επιτυχώς τα αντίστοιχα calls ή αν προέκυψε κάποιο error το οποίο πρέπει να μεταφέρουμε και στο user space, ώστε να ενημερωθεί η αρχική διεργασία. Παρακάτω παραθέτουμε όλους τους κώδικες που υλοποιήσαμε για την εργαστηριακή άσκηση.

```
/*
 * crypto-chrdev.c
 *
 * Implementation of character devices
 * for virtio-cryptodev device
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 *
```

```
 */
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/wait.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>

#include "crypto.h"
#include "crypto-chrdev.h"
#include "debug.h"

#include "cryptodev.h"

/*
 * Global data
 */
struct cdev crypto_chrdev_cdev;

/**
 * Given the minor number of the inode return the crypto device
 * that owns that number.
 **/
static struct crypto_device *get_crypto_dev_by_minor(unsigned int minor)
{
  struct crypto_device *crdev;
  unsigned long flags;

  debug("Entering");

  spin_lock_irqsave(&crdrvdata.lock, flags);
  list_for_each_entry(crdev, &crdrvdata.devs, list) {
    if (crdev->minor == minor)
      goto out;
  }
  crdev = NULL;

out:
  spin_unlock_irqrestore(&crdrvdata.lock, flags);

  debug("Leaving");
  return crdev;
}

/*************************************
 * Implementation of file operations
 * for the Crypto character device
 *************************************/

static int crypto_chrdev_open(struct inode *inode, struct file *filp)
```

```
{
  int ret = 0;
  int err;
  unsigned int len;
  struct crypto_open_file *crof;
  struct crypto_device *crdev;
  unsigned int *syscall_type;
  int *host_fd;
  struct scatterlist syscall_type_sg, host_fd_sg, *sgs[3];
  unsigned int num_out, num_in;
  struct virtqueue *vq;




  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;
  }
  vq  = crdev->vq;
  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.
   **/
  /* ?? */
  debug("Entering");

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

  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, host_fd, sizeof(*host_fd));
  sgs[num_out + num_in++] = &host_fd_sg;
  /**
   * Wait for the host to process our data.
   **/
  /* ?? */
  // lock lock lock
  if(down_interruptible(&crdev->lock)) {
    ret = -ERESTARTSYS;
    debug("open: down_interruptible");
    goto fail;
  }
  err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                          &syscall_type_sg, GFP_ATOMIC);
  virtqueue_kick(vq);
  while (virtqueue_get_buf(vq, &len) == NULL)
    /* do nothing */;



  up(&crdev->lock);

  crof->host_fd = *host_fd;

  /* If host failed to open() return -ENODEV. */
  /* ?? */
  if (crof->host_fd == -1) {
    return -ENODEV;
  }
  debug("ola popa man moy ola komple me hostfd = %d", *host_fd);

  kfree(host_fd);
  kfree(syscall_type);

fail:
  debug("Leaving");
  return ret;
}

static int crypto_chrdev_release(struct inode *inode, struct file *filp)
{
  int ret = 0;
  int err;

  struct crypto_open_file *crof = filp->private_data;
  struct crypto_device *crdev = crof->crdev;
  struct virtqueue *vq = crdev->vq;
  struct scatterlist syscall_type_sg, host_fd_sg, host_ret_val_sg, *sgs
```

```
 [3];
unsigned int *syscall_type;
unsigned int num_out, num_in, len;
int *host_fd, *host_ret_val;


debug("Entering");

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


host_fd = kzalloc(sizeof(*host_fd), GFP_KERNEL);
host_ret_val = kzalloc(sizeof(*host_ret_val), GFP_KERNEL);

num_out = 0;
num_in = 0;
/**
 * Send data to the host.
 **/
/* ?? */


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

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

sg_init_one(&host_ret_val_sg, host_ret_val, sizeof(*host_ret_val));
sgs[num_out + num_in++] = &host_ret_val_sg;
/**
 * Wait for the host to process our data.
 **/
/* ?? */
if(down_interruptible(&crdev->lock)) {
  ret = -ERESTARTSYS;
  debug("open: down_interruptible");
  goto fail;
}
err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                        &syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(vq);
while (virtqueue_get_buf(vq, &len) == NULL)
  /* do nothing */;
up(&crdev->lock);

if (*host_ret_val) {
  return -ENODEV;
}
```

```
  kfree(syscall_type);
  kfree(host_fd);
  kfree(host_ret_val);
  kfree(crof);
  debug("Leaving");
  return ret;

}

static long crypto_chrdev_ioctl(struct file *filp, unsigned int cmd,
                                unsigned long arg)
{
  long ret = 0;
  int err, data_size;
  struct crypt_op *temp_cryp;
  struct crypto_open_file *crof = filp->private_data;
  struct crypto_device *crdev = crof->crdev;
  struct virtqueue *vq = crdev->vq;
  struct scatterlist syscall_type_sg, ioctl_cmd_sg, session_sg,
   sess_id_sg,
                     crypto_sg, host_fd_sg, host_ret_val_sg, sess_key_sg,
    cryp_src_sg,
                     cryp_iv_sg, cryp_dst_sg, *sgs[8];
  unsigned int num_out, num_in, len;
  unsigned char *sess_key, *cryp_src, *cryp_iv, *cryp_dst;
  unsigned int *syscall_type, *ioctl_cmd;
  struct session_op *sess;
  struct crypt_op *cryp;
  int *host_fd, *host_ret_val;
  __u32 *ses_id;
  cryp_dst = NULL;
  ses_id = NULL;
  cryp_iv = NULL;
  cryp_src = NULL;
  data_size = 0;
  debug("Entering");
  /**
   * Allocate all data that will be sent to the host.
   **/
  sess = kzalloc(sizeof(struct session_op), GFP_KERNEL);
  cryp = kzalloc(sizeof(struct crypt_op), GFP_KERNEL);

  host_fd = kzalloc(sizeof(*host_fd), GFP_KERNEL);
  syscall_type = kzalloc(sizeof(*syscall_type), GFP_KERNEL);
  ioctl_cmd = kzalloc(sizeof(*ioctl_cmd), GFP_KERNEL);
  host_ret_val = kzalloc(sizeof(*host_ret_val), GFP_KERNEL);

  *syscall_type = VIRTIO_CRYPTODEV_SYSCALL_IOCTL;
  num_out = 0;
  num_in = 0;
```

```
/**
 *  These are common to all ioctl commands.
 **/
sg_init_one(&syscall_type_sg, syscall_type, sizeof(*syscall_type));
sgs[num_out++] = &syscall_type_sg;
/* ?? */
*host_fd = crof->host_fd;
sg_init_one(&host_fd_sg, host_fd, sizeof(*host_fd));
sgs[num_out++] = &host_fd_sg;
/**
 *  Add all the cmd specific sg lists.
 **/
switch (cmd) {
case CIOCGSESSION:
  debug("CIOCGSESSION");
  *ioctl_cmd = IOCTL_CIOGSESSION;
  sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
  sgs[num_out++] = &ioctl_cmd_sg;
  if (copy_from_user(sess, (struct session_op*) arg, sizeof(struct
 session_op))) {
    return -EFAULT;
  }
  sess_key = kzalloc(sess->keylen, GFP_KERNEL);
  if (copy_from_user(sess_key, sess->key, sess->keylen)) {
    return -EFAULT;
  }
  //sess->key = sess_key;
  sg_init_one(&sess_key_sg, sess_key, sess->keylen);
  sgs[num_out++] = &sess_key_sg;
  sg_init_one(&session_sg, sess, sizeof(struct session_op));
  sgs[num_out + num_in++] = &session_sg;
  sg_init_one(&host_ret_val_sg, host_ret_val, sizeof(*host_ret_val));
  sgs[num_out + num_in++] = &host_ret_val_sg;

  break;

case CIOCFSESSION:
  debug("CIOCFSESSION");
  *ioctl_cmd = IOCTL_CIOGFSESSION;
  sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
  sgs[num_out++] = &ioctl_cmd_sg;
  ses_id = kzalloc(sizeof(*ses_id), GFP_KERNEL);

  if (copy_from_user(ses_id, (__u32 *) arg, sizeof(*ses_id))) {
    return -EFAULT;
  }
  sg_init_one(&sess_id_sg, ses_id, sizeof(*ses_id));
  sgs[num_out++] = &sess_id_sg;

  sg_init_one(&host_ret_val_sg, host_ret_val, sizeof(*host_ret_val));
  sgs[num_out + num_in++] = &host_ret_val_sg;
```

```
  break;

case CIOCCRYPT:
  debug("CIOCCRYPT");
  *ioctl_cmd = IOCTL_CIOCRYPT;
  sg_init_one(&ioctl_cmd_sg, ioctl_cmd, sizeof(*ioctl_cmd));
  sgs[num_out++] = &ioctl_cmd_sg;

  if (copy_from_user(cryp, (struct crypt_op*) arg, sizeof(struct
 crypt_op))) {
    return -EFAULT;
  }
  data_size = cryp->len;
  cryp_dst = kzalloc(data_size, GFP_KERNEL);
  cryp_iv = kzalloc(VIRTIO_CRYPTODEV_BLOCK_SIZE, GFP_KERNEL);
  cryp_src = kzalloc(data_size, GFP_KERNEL);

  sg_init_one(&crypto_sg, cryp, sizeof(struct crypt_op));
  sgs[num_out++] = &crypto_sg;

  if (copy_from_user(cryp_src, cryp->src, data_size)) {
    return -EFAULT;
  }
  sg_init_one(&cryp_src_sg, cryp_src, data_size);
  sgs[num_out++] = &cryp_src_sg;

  if (copy_from_user(cryp_iv, cryp->iv, VIRTIO_CRYPTODEV_BLOCK_SIZE)) {
    return -EFAULT;
  }

  sg_init_one(&cryp_iv_sg, cryp_iv, VIRTIO_CRYPTODEV_BLOCK_SIZE);
  sgs[num_out++] = &cryp_iv_sg;

  sg_init_one(&cryp_dst_sg, cryp_dst, data_size);
  sgs[num_out + num_in++] = &cryp_dst_sg;

  sg_init_one(&host_ret_val_sg, host_ret_val, sizeof(*host_ret_val));
  sgs[num_out + num_in++] = &host_ret_val_sg;

  break;

default:
  debug("Unsupported ioctl command");

  break;
}


/**
 * Wait for the host to process our data.
```

```
 **/
/* ?? */
/* ?? Lock ?? */
if(down_interruptible(&crdev->lock)) {
  ret = -ERESTARTSYS;
  debug("open: down_interruptible");
  goto fail;
}
err = virtqueue_add_sgs(vq, sgs, num_out, num_in,
                        &syscall_type_sg, GFP_ATOMIC);
virtqueue_kick(vq);
while (virtqueue_get_buf(vq, &len) == NULL)
  /* do nothing */;
up(&crdev->lock);


switch (cmd) {
  case CIOCGSESSION:
    if (*host_ret_val != 0) {
      return -ENODEV;
    }
    if (copy_to_user((struct session_op*) arg, sess, sizeof(struct
 session_op))) {
      return -EFAULT;
    }
    break;
  case CIOCFSESSION:
    if (*host_ret_val != 0) {
      return -ENODEV;
    }
    kfree(ses_id);
    break;
  case CIOCCRYPT:
    if (*host_ret_val != 0) {
      return -ENODEV;
    }
    temp_cryp = (struct crypt_op*)arg;
    if (copy_to_user(temp_cryp->dst, cryp_dst, data_size)) {
      return -EFAULT;
    }
    kfree(cryp_dst);
    kfree(cryp_iv);
    kfree(cryp_src);
    break;
  default:
    debug("Unsupported ioctl command");

}
kfree(syscall_type);
kfree(host_fd);
kfree(host_ret_val);
```

```
  kfree(ioctl_cmd);
  kfree(sess);
  kfree(cryp);

  debug("Leaving");

  return ret;
}

static ssize_t crypto_chrdev_read(struct file *filp, char __user *usrbuf,
                                  size_t cnt, loff_t *f_pos)
{
  debug("Entering");
  debug("Leaving");
  return -EINVAL;
}

static struct file_operations crypto_chrdev_fops =
{
  .owner          = THIS_MODULE,
  .open           = crypto_chrdev_open,
  .release        = crypto_chrdev_release,
  .read           = crypto_chrdev_read,
  .unlocked_ioctl = crypto_chrdev_ioctl,
};

int crypto_chrdev_init(void)
{
  int ret;
  dev_t dev_no;
  unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

  debug("Initializing character device...");
  cdev_init(&crypto_chrdev_cdev, &crypto_chrdev_fops);
  crypto_chrdev_cdev.owner = THIS_MODULE;

  dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
  ret = register_chrdev_region(dev_no, crypto_minor_cnt, "crypto_devs");
  if (ret < 0) {
    debug("failed to register region, ret = %d", ret);
    goto out;
  }
  ret = cdev_add(&crypto_chrdev_cdev, dev_no, crypto_minor_cnt);
  if (ret < 0) {
    debug("failed to add character device");
    goto out_with_chrdev_region;
  }

  debug("Completed successfully");
  return 0;
```

```
out_with_chrdev_region:
  unregister_chrdev_region(dev_no, crypto_minor_cnt);
out:
  return ret;
}

void crypto_chrdev_destroy(void)
{
  dev_t dev_no;
  unsigned int crypto_minor_cnt = CRYPTO_NR_DEVICES;

  debug("entering");
  dev_no = MKDEV(CRYPTO_CHRDEV_MAJOR, 0);
  cdev_del(&crypto_chrdev_cdev);
  unregister_chrdev_region(dev_no, crypto_minor_cnt);
  debug("leaving");
}
```

```
/*
 * Virtio Cryptodev Device
 *
 * Implementation of virtio-cryptodev qemu backend device.
 *
 * Dimitris Siakavaras <jimsiak@cslab.ece.ntua.gr>
 * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
 * Konstantinos Papazafeiropoulos <kpapazaf@cslab.ece.ntua.gr>
 *
 */

#include "qemu/osdep.h"
#include "qemu/iov.h"
#include "hw/qdev.h"
#include "hw/virtio/virtio.h"
#include "standard-headers/linux/virtio_ids.h"
#include "hw/virtio/virtio-cryptodev.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <crypto/cryptodev.h>


#define DATA_SIZE       16384
#define BLOCK_SIZE      16
#define KEY_SIZE        24

static uint64_t get_features(VirtIODevice *vdev, uint64_t features,
                             Error **errp)
{
    DEBUG_IN();
```

```
    return features;
}

static void get_config(VirtIODevice *vdev, uint8_t *config_data)
{
    DEBUG_IN();
}

static void set_config(VirtIODevice *vdev, const uint8_t *config_data)
{
    DEBUG_IN();
}

static void set_status(VirtIODevice *vdev, uint8_t status)
{
    DEBUG_IN();
}

static void vser_reset(VirtIODevice *vdev)
{
    DEBUG_IN();
}

static void vq_handle_output(VirtIODevice *vdev, VirtQueue *vq)
{
    VirtQueueElement *elem;
    unsigned int *syscall_type;
    int *host_fd, *ioctl_cmd, *host_ret_val, *ses_id;
    struct session_op *sess_ptr;
    struct crypt_op  *cryp_ptr;
    DEBUG_IN();

    elem = virtqueue_pop(vq, sizeof(VirtQueueElement));
    if (!elem) {
        DEBUG("No item to pop from VQ :(");
        return;
    }

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

    syscall_type = elem->out_sg[0].iov_base;
    switch (*syscall_type) {
    case VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN:
        DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_OPEN");
  DEBUG("hello world");
  int cfd = open("/dev/crypto", O_RDWR);
  host_fd = elem->in_sg[0].iov_base;
  *host_fd = cfd;

  DEBUG("host_fd epityxiaaaa");
```

```
        break;

    case VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE:
        DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_CLOSE");
        host_fd = elem->out_sg[1].iov_base;
        host_ret_val = elem->in_sg[0].iov_base;
        if ((*host_ret_val = close(*host_fd))) {
          perror("close(fd)");
        }

        break;

    case VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL:
        DEBUG("VIRTIO_CRYPTODEV_SYSCALL_TYPE_IOCTL");
        host_fd = elem->out_sg[1].iov_base;
        ioctl_cmd = elem->out_sg[2].iov_base;


        switch (*ioctl_cmd) {
    case 0:
      DEBUG("we are in ciogsession bitches");
      sess_ptr = elem->in_sg[0].iov_base;
      sess_ptr->key = elem->out_sg[3].iov_base;
      host_ret_val = elem->in_sg[1].iov_base;
      if ((*host_ret_val = ioctl(*host_fd, CIOCGSESSION, sess_ptr))) {
          perror("ioctl(CIOCGSESSION)");

        }

      DEBUG("EEE AN GINEI KAI AYTO OLA YPERPOPA");
      break;
    case 1:
      ses_id = elem->out_sg[3].iov_base;
      host_ret_val = elem->in_sg[0].iov_base;
      if ((*host_ret_val = ioctl(*host_fd, CIOCFSESSION, ses_id))) {
        perror("ioctl(CIOCFSESSION)");
      }
      break;
    case 2:
      DEBUG("we are in ciocrypt");
      cryp_ptr = (struct crypt_op *) elem->out_sg[3].iov_base;
      cryp_ptr->src = elem->out_sg[4].iov_base;
      cryp_ptr->iv = elem->out_sg[5].iov_base;
      cryp_ptr->dst = elem->in_sg[0].iov_base;
      host_ret_val = elem->in_sg[1].iov_base;

      if ((*host_ret_val = ioctl(*host_fd, CIOCCRYPT, cryp_ptr))) {
        perror("ioctl(CIOCCRYPT)");
      }
      DEBUG("OKAY CRYPTOGRAPHY HAPPENED, WE ARE GOOD");
```

```
      DEBUG("OLA POPA AGAIN");

      break;
    default:
      DEBUG("NOT VALID IOCTL command");
      break;
    }









        /* ?? */
        /*unsigned char *output_msg = elem->out_sg[1].iov_base;
        unsigned char *input_msg = elem->in_sg[0].iov_base;
        memcpy(input_msg, "Host: Welcome to the virtio World!", 35);
        printf("Guest says: %s\n", output_msg);
        printf("We say: %s\n", input_msg);*/
        break;

    default:
        DEBUG("Unknown syscall_type");
        break;
    }

    virtqueue_push(vq, elem, 0);
    virtio_notify(vdev, vq);
    g_free(elem);
}

static void virtio_cryptodev_realize(DeviceState *dev, Error **errp)
{
    VirtIODevice *vdev = VIRTIO_DEVICE(dev);

    DEBUG_IN();

    virtio_init(vdev, "virtio-cryptodev", VIRTIO_ID_CRYPTODEV, 0);
    virtio_add_queue(vdev, 128, vq_handle_output);
}

static void virtio_cryptodev_unrealize(DeviceState *dev, Error **errp)
{
    DEBUG_IN();
}

static Property virtio_cryptodev_properties[] = {
    DEFINE_PROP_END_OF_LIST(),
};

static void virtio_cryptodev_class_init(ObjectClass *klass, void *data)
```

```
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    VirtioDeviceClass *k = VIRTIO_DEVICE_CLASS(klass);

    DEBUG_IN();
    dc->props = virtio_cryptodev_properties;
    set_bit(DEVICE_CATEGORY_INPUT, dc->categories);

    k->realize = virtio_cryptodev_realize;
    k->unrealize = virtio_cryptodev_unrealize;
    k->get_features = get_features;
    k->get_config = get_config;
    k->set_config = set_config;
    k->set_status = set_status;
    k->reset = vser_reset;
}

static const TypeInfo virtio_cryptodev_info = {
    .name          = TYPE_VIRTIO_CRYPTODEV,
    .parent        = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtCryptodev),
    .class_init    = virtio_cryptodev_class_init,
};

static void virtio_cryptodev_register_types(void)
{
    type_register_static(&virtio_cryptodev_info);
}

type_init(virtio_cryptodev_register_types)
```