Διάλεξη 1

Λειτουργικό σύστημα: Παρεμβάλλεται ανάμεσα στο χρήστη (πχ. μια εφαρμογή που τρέχει) και την μηχανή (hardware) όταν το ζητήσει ο χρήστης, ή όταν υπάρξει κάποιο interrupt (πχ. ένα hardware interrupt από την μεριά της μηχανής). Ειδάλλως, η εφαρμογή μιλάει απευθείας στην μηχανή.

Πυρήνας Linux: Είναι κώδικας που εκτελείται σε kernel mode (έχει πλήρη έλεγχο του συστήματος σε αυτό το mode), είτε ως αποτέλεσμα κάποιου interrupt είτε επειδή το ζήτησε ο χρήστης. Επίσης υπάρχουν και kernel threads τα οποία τρέχουν περιοδικά για να διατηρούν την λειτουργικότητα του πυρήνα, πχ κάποιος daemon.

Interrupts

- **Timer interrupt:** Λήγει το κβάντο χρόνου που αντιστοιχεί σε μια διεργασία που έτρεχε. Έτσι, παίρνει τον έλεγχο ο kernel που κάνει την χρονοδρομολόγηση, σταματάει την διεργασία και διαλέγει ποια άλλη διεργασία θα εκτελεστεί.
- **Device Interrupt:** Interrupt που προκαλείται επειδή κάτι συνέβη σε μία συσκευή της μηχανής.

Τα Android είναι παραλλαγή των Linux

Διαχειρίζεται την μνήμη και την εικονική μνήμη του συστήματος.
Η Virtual Memory χωρίζεται σε user space και kernel space.

copy_to_user(__user *dst, *src, size) : kernel space -> user space
copy_from_user(*dst, __user *src, size) : user space -> kernel space

Σημείωση μετά την εξέταση (άσκηση 3): Όταν έχουμε να αντιγράψουμε μνήμη από/προς δείκτη στο user space, πρέπει πάντα να χρησιμοποιούμε copy_from/to_user() και **όχι memcpy()**. Μεταξύ άλλων ελέγχων, οι copy_from/to_user() συναρτήσεις ελέγχουν αν ο δείκτης στο userspace είναι valid, δηλαδή πχ. αν δείχνει όντως σε μια υπαρκτή διεύθυνση του userspace και όχι σε κάποια διεύθυνση στο kernel space. Η memcpy() δεν κάνει κανέναν τέτοιο έλεγχο, και καλό θα ήταν να χρησιμοποιείται για αντιγραφή δεδομένων που βρίσκονται μόνο μέσα στο kernel space.

Επομένως, με την copy_from/to_user() και τους ελέγχους της αποφεύγουμε kernel crashes ή security holes.

Σημείωση μετά την εξέταση (άσκηση 3): Όταν κάνουμε *copy_from_user()* κάποιο *struct* αντικείμενο της μορφής:

```
struct mystruct {
        int x;
        int *y;
}
```

, τότε πρακτικά αντιγράφουμε το αντικείμενο αυτο, από το user space στο kernel space. ΌΜΩΣ, πρέπει να είμαστε προσεκτικοί, διότι στην περίπτωση του δείκτη *y*, δεν έχουμε κάνει copy το περιεχόμενο της διεύθυνσης που δείχνει ο y, αλλά μόνο την ίδια την διεύθυνση. **Η διεύθυνση αυτή λοιπόν, είναι διεύθυνση στο user space**. Επομένως, αν θέλουμε να φέρουμε το περιεχόμενο της θέσης μνήμης αυτής στο kernel space, πρέπει να ξανακάνουμε: *copy_from_user(..., y, sizeof(*y))*.
Το ίδιο ισχύει πρακτικά και για την περίπτωση του copy_to_user(), όταν ένας δείκτης y δείχνει τώρα στο kernel space.

Linux Tools

- **strace**: Εργαλείο τυπώνει όλα τα system calls που γίνονται όταν εκτελείται μια εντολή. πχ: *strace cat foo* μας δείχνει όλα τα system calls για να παρουσιαστεί το περιεχόμενο του foo στο terminal.
- **vmstat**: Δείχνει στατιστικά για την virtual memory του συστήματος. Μερικά από αυτά είναι πόση είναι ελεύθερη ή σε χρήση, πόσες διεργασίες περιμένουν να πάρουν χρησιμοποιήσουν την μηχανή κλπ.

System Calls: Πραγματοποιούν την μετάβαση εκτέλεσης κώδικα από user space σε kernel space. Η διαδικασία που ακολουθείται είναι η εξής:

1) Γίνεται το system call.
2) Έλεγχος ορισμάτων ώστε να εξακριβωθεί αν ο χρήστης έχει το δικαίωμα να κάνει αυτό που ζητάει με το system call.
3) Εκτέλεση της διεργασίας.
4) Return το αποτέλεσμα στον χρήστη.

Linux Syscall Commands σε C:

- open(“/path/to/device”, flags)
- read(int fd, unsigned char * dst_buf, size_t bytes)
- write(int fd, unsigned char * src_buf, size_t bytes)
- fstat: Επιστρέφει πληροφορίες και στατιστικά για αρχείο που δίνουμε σαν όρισμα.
- kmalloc: Κλήση για δέσμευση “συνεχούς” φυσικής μνήμης στον χώρο πυρήνα. Επιστρέφει δείκτη στην αρχή του τμήματος που δεσμεύτηκε.
- kzalloc: Παρόμοια με την kmalloc, μόνο που εδώ, το περιεχόμενο της μνήμης τίθεται 0, πριν επιστραφεί ο δείκτης.
- vmalloc: Κλήση για δέσμευση “συνεχούς” (contiguous blocks of memory) εικονικής μνήμης στον χώρο πυρήνα. Αυτό το συνεχές κομμάτι εικονικής μνήμης, μπορεί να μην

είναι συνεχές στην πραγματική μνήμη, γιαυτό και το να δουλεύουμε με μνήμη που έχει δεσμευτεί με vmalloc μπορεί να αποδειχθεί λιγότερο αποδοτικό.
- kfree: Κλήση για αποδέσμευση μετά απο kmalloc.

Οι read() και write() παίρνουν σαν όρισμα file descriptor που είναι int και περιγράφει τον τύπο του αρχείου από/προς το οποίο διαβάζει/γράφει. Ο file descriptor είναι 1 στην περίπτωση που το αρχείο είναι ο terminal.

Μια διεργασία (process) βρίσκεται είτε σε **user mode** είτε σε **kernel mode** (όπου έχει πλήρη έλεγχο του συστήματος).

Process Control Block (struct task_struct): Είναι ένα struct που δημιουργείται όταν η διεργασία δημιουργείται και περιέχει πληροφορίες για την διεργασία αυτή. Οι πληροφορίες αυτές περιλαμβάνουν δεδομένα για το identification της διεργασίας (όπως το pid της), για το state της (πχ αποθηκεύεται η κατάσταση των registers της CPU όταν η process διακόπτεται, ώστε να ξαναξεκινήσει κανονικά) καθώς και πληροφορίες για το πως να ελεγχθεί αυτή η διεργασία.

struct page_struct: Βασική μονάδα διαχείρισης μνήμης, η σελίδα.

Σκέψεις για kernel mode: Όταν βρίσκεσαι σε kernel mode και προγραμματίζεις, δεν υπάρχει κάποιος να σε πληροφορήσει ότι πχ δεν μπορείς να πας σε εκείνη την θέση μνήμης (επειδή πιθανόν να μην υπάρχει). Οπότε πρέπει να είμαστε πολύ προσεκτικοί! Μπορείς να χαλάσεις κάτι στο σύστημα σου.

Για τον λόγο αυτό, αν θέλουμε να τροποποιήσουμε τον πυρήνα δουλεύουμε με εικονικές μηχανές (αυτό θα γίνει και στις ασκήσεις 2 και 3). Οι VMs τρέχουν σε user mode στο πραγματικό μηχάνημα, οπότε είμαστε ασφαλείς.

/proc filesystem: Είναι ένα **virtual(!) file system** που περιέχει ξεχωριστά directories για κάθε process που τρέχει αυτή την στιγμή στο μηχάνημα. Το /proc δημιουργείται όταν κάνει boot η μηχανή και καταστρέφεται κατά το shut down. Ότι αρχείο περιέχει δεν είναι αρχείο στην πραγματικότητα (**έχει μέγεθος 0B**). Μπορεί να περιγραφεί σαν ένα παράθυρο προς τον kernel του linux, όπου περιέχονται χρήσιμες πληροφορίες για τα processes.

Κάθε process με συγκεκριμένο PID έχει και το δικό της directory μέσα στο /proc filesystem. Το directory αυτό βρίσκεται στην διεύθυνση **/proc/PID/**. Εκεί μέσα περιέχονται αρχεία με χρήσιμες πληροφορίες για το process, όπως για παράδειγμα τα **/proc/PID/status, /proc/PID/root, /proc/PID/cwd** κλπ κλπ.

Για να κάνουμε list τα processes, κάνουμε: **ps -aux**.

Riddle

chmod: Terminal εντολή για αλλαγή των permissions σε ένα file. Καλείται ως εξής:

chmod permissions file_name

παράδειγμα: *chmod u-rw file.txt* που σημαίνει ότι ο user δεν μπορεί πλέον να διαβάζει και να γράφει στο file.txt

Process Image: Είναι ένα instance του process, ένα executable file το οποίο περιέχει όλα τα απαραίτητα αρχεία προκειμένου το λειτουργικό σύστημα να το φορτώσει στον επεξεργαστή, ώστε να μπορέσει να εκτελεστεί.

exec(): Είναι μια οικογένεια συναρτήσεων στην C που δέχονται ως βασικό όρισμα ένα binary executable και αντικαθιστούν το process image του παρόντος προγράμματος με αυτό της διεργασίας του executable. Ως αποτέλεσμα, μέσα στο παρόν πρόγραμμα, εκτελείται το binary executable που δώσαμε ως όρισμα στην exec().
παράδειγμα: *execvp(args[0], args);*

alarm(): Εντολή στην C που δέχεται αριθμό δευτερολέπτων ως όρισμα και στέλνει SIGALARM στην καλούσα διεργασία μετά από αυτά τα δευτερόλεπτα. Η default συμπεριφορά του SIGALARM είναι να τερματίσει την διεργασία.

ltrace: Εντολή στο terminal που αποτυπώνει τα library calls, τις μεταβλητές περιβάλλοντος (που χρειάζεται ή χρησιμοποιεί) κλπ κλπ μιας διεργασίας.

dup2(old_fd, new_fd): Η εντολή αυτή δημιουργεί τον file descriptor new_fd που είναι ένα copy του file descriptor old_fd. Έτσι είναι το ίδιο να χρησιμοποιούμε οποιονδήποτε από τους 2 τους.

pipe: Είναι ένας δίαυλος επικοινωνίας μεταξύ 2 (συγγενικών) processes. Ένα process γράφει στο write-end του pipe και το άλλο process διαβάζει από το read-end του pipe (ροή δεδομένων προς μια μόνο κατεύθυνση). Αν θέλουμε να έχουμε αμφίδρομη επικοινωνία με pipes, κατασκευάζουμε 2 pipes. Πως χρησιμοποιούμε τα pipes:

int fd[2]; *//κατασκευάζουμε 2 ints που θα είναι οι file descriptors των read/write ends*
//fd[0] -> read και fd[1] -> write
if (pipe(fd) < 0) *//καλούμε την pipe() για να κατασκευάσει το pipe*
exit(1);

- Όταν ένα process διαβάζει ή γράφει από αυτό το pipe, μπορούμε να φανταστούμε ότι αυτό το process διαβάζει ή γράφει σε ένα virtual file το οποίο δεν υπάρχει στην πραγματικότητα.
- Αν ένα process προσπαθήσει να διαβάσει από το pipe χωρίς να έχει γραφτεί τίποτα, τότε περιμένει (μπλοκάρει) μέχρι να γραφτεί κάτι.
- Το pipe είναι προφανώς μια **FIFO** δομή.

inode: Είναι ένα data structure (struct) που περιέχει όλες τις απαραίτητες πληροφορίες για ένα file, όπως πχ τύπος του file, **ποια block πιάνει στην μνήμη**, permissions πάνω σε αυτό το file, **το inode number**, τους major/minor numbers αν μιλάμε για /dev file κλπ.
Το **inode number** είναι ένας μοναδικός αριθμός (σε όλο το filesystem) ο οποίος αναφέρεται σε ένα συγκεκριμένο inode.
Μπορείς να δεις τι περιέχει ένα inode με την εντολή: *stat filename*

Hard Links: Είναι κάτι σαν pointers σε files. Τα hard links συγκεκριμένα, σε αντίθεση με τα soft links (symlinks), συνεχίζουν να αναφέρονται στο target file τους ακόμα κι αν αυτό μετακινηθεί κάπου αλλού μέσα στο file system. Αυτό συμβαίνει επειδή τα **hard links παίρνουν τον inode number του target file**, οπότε συνεχίζουν να έχουν πρόσβαση σε αυτό ακόμα κι αν αυτό μετακινηθεί μέσα στο filesystem. Δημιουργούμε hard link με την εξής εντολή:

- *ln [original filename] [link name]*

Ακόμα, δημιουργούμε symlinks προσθέτοντας το option -s:

- *ln -s [original filename] [link name]*

Τα symlinks είναι πρακτικά references σε αρχεία ή directories (τα hard links δεν μπορούν να κάνουν reference directories). Σε αντίθεση με τα hard links, αν μετακινηθεί το original αρχείο από την θέση που δείχνει το symlink, τότε το symlink σπάει και λέμε ότι έχουμε dangling link.

unlink(): Πρακτικά διαγράφει ένα όνομα, που δέχεται ως όρισμα, από το filesystem. Πιθανώς να διαγράψει το ίδιο το file αν αυτό το name είναι το τελευταίο link που υπάρχει προς αυτό το file. Αν το όρισμα είναι ένα soft link τότε διαγράφεται απλά το soft link. Αν το file που πάει να διαγράψει είναι ανοικτό από κάποιο άλλο process, τότε αυτό το process έχει πρόσβαση ακόμα στο file, μέχρι να τερματίσει την λειτουργία του.

lseek(): Είναι ένα syscall-εντολή (στην C) με την οποία μπορούμε να μετακινούμε τους read/write pointers ακόμα και πέρα από το τέλος του αρχείου (sparse file). Παράδειγμα χρήσης:

- *lseek(fd, 1073741824, SEEK_SET);*

Η παραπάνω εντολή μετακινεί τους pointers στο byte 1073741824.

mmap(): Κάνει map έναν file descriptor (αρχείο, pipe, συσκευή κλπ) σε κάποια θέση εικονικής μνήμης του process και επιστρέφει την θέση μνήμης αυτή.

ftruncate(2): Αλλάζει το μέγεθος ενός file, που γίνεται specify από file descriptor. Το offset δίνεται σε bytes.
Παράδειγμα: *ftruncate(fd1, 32768);*
Αν το αρχικό αρχείο ήταν μεγαλύτερο από το τελικό, τα extra data στο τέλος του αρχείου χάνονται. Αν το αρχικό ήταν μικρότερο, τότε τα νέα data στο τελικό αρχείο θα έχουν τιμή “\0”.

Σημείωση: Τα pids που δίνονται σε ένα καινούριο process (μετά από ένα fork()) αυξάνονται κάθε φορά κατά 1 (σε σχέση με τον πατέρα).

Διάλεξη 2 (Εργαστηριακή Άσκηση 2)

Drivers

Υπάρχουν διαφορετικά είδη συσκευών hardware οι οποίες διαβάζουν και στέλνουν δεδομένα:

- **Character devices**: Σειριακές/παράλληλες θύρες, κάρτες ήχου
- **Block devices**: Σκληροί δίσκοι, CD-ROM
- **Network devices**: Κάρτες δικτύου

Σε αυτήν την εργαστηριακή άσκηση, επικεντρωνόμαστε σε **character devices**.

Για διαφορετικού είδους συσκευές έχουμε και διαφορετικού είδους drivers.

/dev filesystem

Πρόκειται για τον χώρο στον οποίο “βλέπουμε” τις συσκευές που είναι συνδεδεμένες, ως αρχεία. Πρόκειται για ένα filesystem παρόμοιο με το /proc filesystem με την έννοια ότι τα αρχεία που περιέχει δεν είναι πραγματικά, δεν έχουν φυσική υπόσταση (δεν καταλαμβάνουν χώρο στην μνήμη). Είναι αρχεία που αντιπροσωπεύουν αυτές τις συσκευές. Μια εφαρμογή στο περιβάλλον χρήστη αποκτά πρόσβαση σε μια συσκευή κάνοντας system calls (read(), write(), open() κλπ κλπ) προς το αρχείο του /dev filesystem που της αντιστοιχεί.

Τα files στο /dev filesystem έχουν το καθένα από ένα Major και Minor number (κάτι σαν identifiers).

Major: Καθορίζει ποιος driver θα χρησιμοποιηθεί αν μια διεργασία ζητήσει να διαβάσει από το συγκεκριμένο αρχείο-συσκευή. Κάνει την σύνδεση της συσκευής με τον κατάλληλο driver.

Minor: Καθορίζει την λειτουργικότητα του ίδιου του driver. Το χρησιμοποιεί ο driver για να καταλάβει σε ποια συσκευή να μιλήσει, μιας και ο ίδιος driver μπορεί να κάνει δουλειά για διαφορετικές συσκευές.

Μπορούμε να δούμε τους Major και Minor numbers με: *ls -l /dev*

Struct file_operations ενός driver

Βρίσκεται μέσα στον driver και μοιάζει με dictionary, ουσιαστικά είναι ένα struct. Περιέχει έναν δείκτη για κάθε μέθοδο που υλοποιείται μέσα στον driver.

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

```
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*open) (struct inode *, struct file *);
        .
        .
        .
};
```

Επικοινωνία εφαρμογής με character device

Όταν μια εφαρμογή θέλει να επικοινωνήσει με μια συσκευή (να στείλει ή να λάβει αρχεία), τότε επικοινωνεί μαζί της χρησιμοποιώντας system calls προς το αρχείο της συσκευής (βρίσκεται στο /dev filesystem). Τότε, **ο kernel κάνει redirect το system call** προς την αντίστοιχη μέθοδο (στο struct file_operations) του driver που αντιστοιχεί σε αυτή την συσκευή.

Συγκεκριμένα, στο inode ενός αρχείου-συσκευής του /dev filesystem περιέχονται και οι major/minor numbers του αρχείου αυτού. Οπότε, για να γίνουμε πιο σαφείς, αν χρησιμοποιήσουμε την system call: open() για την συσκευή-αρχείο dev/ttyUSB1, τότε ο kernel κοιτάει τον major number του “αρχείου”, από αυτόν **καταλαβαίνει σε ποιον driver να απευθυνθεί**, μέσω του **file_operations** του driver βρίσκει ότι θα πρέπει να κληθεί η αντίστοιχη μέθοδος του driver: open(), και την καλεί.

Modules στα Linux

Ένα module στα Linux είναι ένα κομμάτι κώδικα, το οποίο μπορεί να γίνει load και unload δυναμικά στον kernel, ακόμα και την στιγμή που αυτός τρέχει. Αφού φορτωθεί, έχει τα ίδια δικαιώματα με τον υπόλοιπο κώδικα στον kernel. Δηλαδή, έχει πρόσβαση στις routines του πυρήνα πχ kmalloc(), έχει την δυνατότητα να διαχειρίζεται system calls που έρχονται από το περιβάλλον χρήστη, κλπ κλπ. Τα modules είναι .ko (kernel object) αρχεία, τα οποία γίνονται load και unload με τις εντολές: *insmod module_name* και *rmmod module_name*. Επίσης η lsmod μας κάνει list όλα τα modules στον kernel.
Τα modules βρίσκονται στο directory: */lib/modules/kernel_version*

Ο κώδικας του lunix που θα μας δοθεί για την 2η εργαστηριακή είναι σε μορφή module. Κάνοντας make, παίρνουμε ένα .ko αρχείο. Αυτόν τον κώδικα πρέπει να τον επεκτείνουμε πριν τον κάνουμε make και insmod μέσα στον πυρήνα.

Μηχανισμοί Συγχρονισμού:

- Spinlocks
- Semaphores
- Ατομικές εντολές

Spinlocks

Τα spinlocks είναι εργαλεία για mutual exclusion δύο ή περισσοτέρων διεργασιών από κάποιο “αρχείο”. Παίρνουν δύο τιμές, locked και unlocked και συνήθως υλοποιούνται με ένα bit.

Παράδειγμα χρήσης spinlocks:
spinlock_t lock;
spin_lock_init(&lock);
spin_lock_irq(&lock);
spin_unlock_irq(&lock);

Χαρακτηριστικά των spinlocks:

- Αν μια διεργασία πάει να πάρει στην κατοχή της ένα spinlock (με την συνάρτηση spin_lock_*()) και το βρει κλειστό, **τότε κάνει spin (ελέγχει συνεχώς αν άνοιξε το lock), μέχρι αυτό να απελευθερωθεί** από την διεργασία που το κρατάει. Προσοχή: Δεν κάνει sleep. Συνεχίζει να χρησιμοποιεί τον επεξεργαστή κάνοντας συνεχώς poll για το αν απελευθερώθηκε το spinlock ή όχι.
- Όταν μια διεργασία κλείσει ένα spinlock, καλό θα ήταν αυτή η διεργασία να μην μπορεί να διακοπεί από kernel preemption, ή hardware και software interrupts. Επίσης, καλό είναι η διεργασία που έχει κλειστό το spinlock να μην πηγαίνει σε sleep, αν δεν το έχει απελευθερώσει. Ο κώδικας μέσα στο critical section δηλαδή, δεν πρέπει να στέλνει για κανέναν λόγο το process σε sleep.
- Ο χρόνος που μια διεργασία κρατάει ένα spinlock κλειστό πρέπει να είναι όσο το δυνατόν μικρότερος, για να αποφευχθούν κολλήματα.

Note: Ένα **critical section** είναι ένα τμήμα κώδικα το οποίο πρέπει να εκτελεστεί από 1 process την φορά (πχ. το τμήμα ανάμεσα σε spin_lock() και spin_unlock()).

Semaphores

Είναι ακόμα ένα εργαλείο συγχρονισμού, αυτή την φορά για πολλές διεργασίες, που μπορεί να χρησιμοποιηθεί και ως εργαλείο mutual exclusion, αν ο μέγιστος αριθμός διεργασιών που έχουν πρόσβαση σε ένα αρχείο είναι 1. Η λογική μοιάζει με αυτή των spinlocks, μόνο που τώρα, αν το lock είναι κλειστό, η διεργασία που ζητά πρόσβαση στο αρχείο (μέσω της syscall) πέφτει για ύπνο μέχρι το lock να απελευθερωθεί.

Παράδειγμα:
struct semaphore lock;
sema_init(&lock, max_number_of_processes);
down_interruptible(&lock);
up(&lock);

Σημείωση μετά την εξέταση 1 (άσκηση 2 και 3): Μία από τις βασικές διαφορές ανάμεσα σε spinlocks και semaphores είναι ότι τα spinlocks είναι κατάλληλα να χρησιμοποιηθούν για περιπτώσεις που εκτελείται κώδικας πυρήνα σε interrupt context. Παραδείγματος χάρη, όταν εκτελείται κώδικας πυρήνα (τον οποίο πχ. έχουμε γράψει εμείς) που κάνει handle ένα hardware interrupt, και θέλει πρόσβαση σε ένα critical section, τότε χρησιμοποιούμε spinlocks και όχι semaphores, διότι αν το lock είναι κλειδωμένο εκείνη τη στιγμή, δεν υπάρχει κάποια διεργασία χρήστη να πέσει για ύπνο μέχρι αυτό να ξεκλειδωθεί. Αφού ο κώδικας αυτός εκτελείται λόγω interrupt και όχι επειδή το ζήτησε κάποια διεργασία χρήστη.

Στην περίπτωση της άσκησης 3, χρησιμοποιήθηκαν semaphores διότι υλοποιήσαμε file operations τα οποία εκτελούνται μόνο αν κάποια διεργασία ΧΡΉΣΤΗ το ζητήσει (μέσω κάποιου syscall). Δεν υπήρχε περίπτωση ο κώδικας αυτός (μέσα στα file operations) να εκτελεστεί σε interrupt context.

Σημείωση μετά την εξέταση 2 (άσκηση 2): Στην δική μας περίπτωση, χρησιμοποιούμε semaphores μέσα στην lunix_chrdev_read() για να έχουμε mutual exclusion μεταξύ διεργασιών χρήστη που πάνε να αποκτήσουν πρόσβαση στο ίδιο state struct. Αυτές οι διεργασίες, δεν μπορεί να είναι δύο cat που εκτελούνται παράλληλα σε δύο διαφορετικά terminals. Διότι αυτά τα cat, καλούν ξεχωριστά την lunix_chrdev_open() η οποία κάνει allocate διαφορετικά state structs για τις lunix_chrdev_read() που θα ακολουθήσουν. Αντίθετα, αυτές οι διεργασίες μπορεί να είναι ένας πατέρας με το forked παιδί του (το παιδί γίνεται fork() μετά την lunix_chrdev_open() του πατέρα). Γιατί σε εκείνη την περίπτωση, το παιδί κληρονομεί όλα τα resources του πατέρα (όπως τους fds) και επομένως, αν κάνουν read() παράλληλα, θα επιχειρήσουν να αποκτήσουν πρόσβαση στο ίδιο state struct. Επίσης, αξίζει να σημειωθεί πως η struct file δημιουργείται από τον kernel όταν εκτελείται το syscall open(), ύστερα περνιέται σε όλες τις syscalls (που ακολουθούν μετά την open()) που αφορούν το αρχείο (μέσω του δείκτη *filp) και επομένως δεν είναι μοναδική για κάθε αρχείο στον πυρήνα (σε αντίθεση με το struct inode).

Διεργασίες σε Sleep

Οι συναρτήσεις *wait_event_*(myqueue, condition)* βάζουν διεργασίες χρήστη σε sleep, μέσα σε μια ουρά αναμονής queue. Όταν, σε κάποιον άλλο κώδικα πυρήνα κληθεί η *wake_up_*(&myqueue)*, τότε όλες οι διεργασίες του queue ξυπνάνε και μπαίνουν σε κατάσταση ready.

Σημείωση: Προτιμότερες είναι οι *wait_event_interruptible(queue, condition)* και *wake_up_interruptible(&queue)* οι οποίες ξυπνούν τις sleeping διεργασίες αν έρθει κάποιο interrupt από τον χρήστη.

Στην περίπτωση της άσκησης 2 (lunix), χρησιμοποιούσαμε την συνάρτηση *wait_event_interruptible(sensor->wq, condition)* για να στείλουμε για ύπνο τις διεργασίες χρήστη που κάλεσαν την *read()* την ώρα που δεν υπήρχαν νέα δεδομένα να διαβάσουν. Οι διεργασίες θα ήταν σε sleep mode μέχρι να έρθουν τα νέα δεδομένα. Τότε, καλείται η

lunix_sensor_update() η οποία ανανεώνει το sensor struct με τα νέα δεδομένα και ύστερα ξυπνάει όλες τις διεργασίες του *sensor->wq* με *wake_up_interruptible(&sensor->wq)*.

Σημείωση μετά την εξέταση: ΕΡΏΤΗΣΗ: Που χρειάζεται, λοιπόν, το condition που παίρνει ως όρισμα η wait_event(); Αφού οι διεργασίες θα ξυπνήσουν όταν είναι να ξυπνήσουν, δηλαδή όταν κληθεί μια wake_up(myqueue) από τον πυρήνα.
ΑΠΆΝΤΗΣΗ: Όταν οι διεργασίες μπουν σε κατάσταση ready και πάλι, και ύστερα ο scheduler πάει να τις κάνει running, τότε εξετάζεται αν ικανοποιούν το condition που δώσαμε ως όρισμα. Αν το ικανοποιούν, τότε γίνονται running. Αν δεν το ικανοποιούν, τότε πέφτουν και πάλι για ύπνο. Αυτό χρησιμεύει όταν δεν θες να ξυπνήσουν όλες οι διεργασίες που κοιμούνται στο queue, αλλά μόνο μερικές επιλεγμένες. Στην δική μας περίπτωση βέβαια, το condition θα μπορούσε να είναι σκέτο true.

printk()

Είναι μια συνάρτηση που χρησιμοποιείται για τύπωση μηνυμάτων και debugging, σε επίπεδο κώδικα kernel. Παράδειγμα της χρήσης της printk():

printk(KERN_INFO "Message: %s\n", arg);

Μπορούμε να βλέπουμε τι τυπώνει (log) ο πυρήνας με το εργαλείο dmesg. Στην περίπτωση της 2ης εργαστηριακής άσκησης, μια χρήσιμη εντολή είναι η:

dmesg --follow | *grep lunix*

Περιγραφή ζητουμένου της άσκησης

Η συσκευή (δίκτυο από αισθητήρες) περιέχει 3 αισθητήρες: sensor0, sensor1, sensor2 καθένας από τους οποίους μπορεί να εκτελέσει 3 μετρήσεις: battery, temperature, light. Θέλουμε, αντί για ένα ενιαίο dev/ttyUSB1 αρχείο, να υπάρχουν 9 διαφορετικά αρχεία που το καθένα να αντιπροσωπεύει ένα ζευγάρι sensor-μέτρησης. Οι εφαρμογές στο περιβάλλον χρήστη, θα μπορούν να αποκτούν πρόσβαση στα δεδομένα των μετρήσεων της συσκευής κάνοντας read() στο αντίστοιχο file του /dev directory. Επίσης, διαφορετικοί χρήστες θα έχουν διαφορετικά δικαιώματα πάνω σε κάθε αρχείο, δηλαδή σε κάθε ζεύγος sensor-μέτρησης.

Εμείς, αρχικά καλούμαστε να υλοποιήσουμε τον Lunix character device driver, ο οποίος διαβάζει τα δεδομένα από τους Lunix sensor buffers (έχουν προέλθει από την συσκευή και έχουν αποθηκευτεί στους buffers μέσω του ήδη δοσμένου κώδικα) και τα κατανέμει στα αντίστοιχα αρχεία του /dev filesystem, ώστε να τα πάρει η εφαρμογή (περιβάλλον χρήστη) που τα ζήτησε. Επίσης, ο Lunix character device driver περιέχει τον κώδικα για τις μεθόδους που εγείρονται/εκτελούνται όταν ο χρήστης/εφαρμογή κάνει system call προς κάποιο από τα 9 αρχεία.

Σημείωση: Οι εφαρμογές δεν γράφουν σε αυτά τα αρχεία. Μόνο διαβάζουν, αν έχουν το δικαίωμα και αν δεν είναι κλειδωμένο το αρχείο επειδή κάποια άλλη εφαρμογή το χρησιμοποιεί.

Οδηγός εκκίνησης του VM

Ανοίγουμε terminal και τρέχουμε:

1) cd ~/utopia
2) ./utopia.sh

Στο terminal τώρα βλέπουμε το VM που έχει ξεκινήσει. Ανοίγουμε 2ο terminal και **συνδεόμαστε στο VM με SSH**:

1) ssh -p 22223 root@localhost
2) Κωδικός root για χρήστη root, user για χρήστη user.

Σημείωση: Μπορώ να κάνω access τα αρχεία που υπάρχουν μέσα στο VM μέσω του φακέλου: /home/konstantinos/ στο host machine. Αρκεί, σε κάθε session να εκτελώ:
sshfs -o allow_other konstantinos@10.0.2.2:/home/konstantinos /home/user/host
από το "/" directory του VM.

Virtual Machines (VMs)

Hypervisors (VMMs)

Ένας **hypervisor** είναι ουσιαστικά ένα λογισμικό που επιτρέπει σε virtual machines να τρέχουν πάνω στο ίδιο physical machine. Ο hypervisor μπορεί να δημιουργεί και να καταστρέφει virtual machines, καθώς επίσης και να διαχειρίζεται τους πόρους που αυτά χρειάζονται. Είναι δηλαδή ο supervisor των guest machines που τρέχουν πάνω στο host machine.

Οι hypervisors χωρίζονται σε 2 κατηγορίες:

1) **Type-1 (bare-metal) hypervisors**: Οι Type-1 hypervisors τρέχουν απευθείας πάνω στο hardware του physical machine και όχι πάνω σε κάποιο OS το οποίο λειτουργεί ως host. Ένας τέτοιος hypervisor διαχειρίζεται τους πόρους του hardware και τους κατανέμει στα host machines που τρέχουν πάνω του. Οποιοδήποτε VM τρέχει πάνω στον hypervisor θεωρείται guest machine.
2) **Type-2 (hosted) hypervisors**: Οι Type-2 hypervisors τρέχουν ως processes πάνω από ένα host OS, ακριβώς όπως ένα απλό πρόγραμμα. Το host OS δηλαδή παρεμβάλλεται ανάμεσα στον hypervisor και στο hardware του host machine και οποιοδήποτε άλλο OS τρέχει πάνω στον hypervisor θεωρείται guest machine. Τέτοιοι hypervisors είναι το **QEMU (που έχει ειδική υποστήριξη για KVM)**, το VirtualBox κλπ.

Kernel-based Virtual Machine (KVM)

Το KVM είναι ένα **module στον Linux kernel** που μετατρέπει το host OS (τον linux kernel) σε Type-1 hypervisor. Το KVM δίνει την δυνατότητα σε guest machines να είναι ταχύτερα, να μιλάνε απευθείας στο hardware, να μοιράζονται τα ίδια resources, να χρησιμοποιούν κοινές βιβλιοθήκες κλπ.

Συγκεκριμένα, έστω ένα VM που τρέχει μέσω QEMU και έστω ότι το guest OS και το host OS είναι σχεδιασμένα να τρέχουν πάνω σε CPUs διαφορετικής αρχιτεκτονικής. Για κάθε απαίτηση του guest OS, το QEMU (που τρέχει ως process στο host machine) πρέπει να μεταφράσει σε επίπεδο λογισμικού όλες τις εντολές από την αρχιτεκτονική του guest machine, στην αρχιτεκτονική του host machine, πράγμα το οποίο είναι πολύ χρονοβόρο.

Αν όμως, το host machine τρέχει linux σε x86 hardware και έχει KVM, και το guest machine τρέχει επίσης linux σε x86 αρχιτεκτονική για την vCPU, τότε χάρη στο KVM του host μπορεί να γίνεται allocate ένα slice του physical CPU ώστε ο κώδικας του guest machine που πρέπει να εκτελεστεί στην vCPU να εκτελείται κατευθείαν στην CPU, χωρίς να απαιτείται ενδιάμεση μετάφραση. Έτσι η απόδοση βελτιώνεται αισθητά.

Διάλεξη 3 (Εργαστηριακή Άσκηση 3)

IP (Internet Protocol)

Κάθε access point (ή router/modem/switch) έχει μοναδική διεύθυνση μήκους 32 ή 128 bit, για τις εκδόσεις IPv4 ή IPv6 αντίστοιχα. Όλες οι συσκευές/υπολογιστές που είναι συνδεδεμένες στο access point έχουν την Public IP Address του access point και από μία μοναδική Private IP Address το καθένα, η οποία είναι valid μόνο μέσα στο private network.

Όταν συνδέονται πολλοί υπολογιστές σε ένα network, τότε το access point αντιστοιχίζει το κάθε μηχάνημα και σε ένα port ώστε να μπορούν να δημιουργούν ξεχωριστά κανάλια επικοινωνίας με τον έξω κόσμο, χρησιμοποιώντας την ίδια IP address.

Πολύ γενικά, όταν στέλνονται πακέτα με χρήση του TCP/IP, τότε τα headers περιέχουν source IP και Port, destination IP και Port (η οποία είναι συνήθως η 80 ή η 443 όταν το destination είναι ο server κάποιου site για παράδειγμα).

TCP (Transmission Control Protocol)

Είναι protocol πάνω από το IP που εξασφαλίζει ότι τα πακέτα θα φτάσουν στο απέναντι άκρο και με την σωστή σειρά.

Από την άλλη, το UDP είναι protocol που δεν εξασφαλίζει ούτε ότι θα φτάσουν, ούτε και την σειρά.

Sockets

Ορίζουν έναν δίαυλο επικοινωνίας μεταξύ processes, όπως το pipe, μόνο που αυτή την φορά τα processes μπορούν και να επικοινωνούν remotely, (από διαφορετικά μηχανήματα) μέσω του δικτύου. Τα sockets είναι πρακτικά τα endpoints αυτού του διαύλου, ο οποίος είναι bidirectional by default.

- Με την εντολή **socket(3**) ένα πρόγραμμα δημιουργεί ένα socket σύμφωνα με τα flags που δίνει ως ορίσματα. Η εντολή αυτή **επιστρέφει έναν file descriptor** τον οποίο χρησιμοποιεί η διεργασία για να ακούει και να μιλάει (με read() και write() κανονικά) στον δίαυλο.
- Με την εντολή **bind()** αντιστοιχίζουμε έναν file descriptor σε μια συγκεκριμένη διεύθυνση (host και port).
- Με την εντολή **listen(2)** καθορίζουμε ότι το socket που δίνεται ως όρισμα (με file descriptor) θα χρησιμοποιηθεί για να ληφθούν connection requests. Με το όρισμα int backlog, καθορίζουμε το πόσα connection requests μπορούν να περιμένουν στο queue για να γίνουν αποδεκτά από την εντολή accept().
- Με την εντολή **accept()** καθορίζουμε ότι το πρώτο connection request στο queue γίνεται accepted. Δίνουμε ως όρισμα τον αρχικό file descriptor (που δημιουργήθηκε με την socket(), μετά έγινε bind() με μια διεύθυνση και μπήκε σε listening state με την listen()) και επιστρέφεται ένας νέος file descriptor που αναφέρεται σε socket έτοιμο για επικοινωνία. Το socket αυτό έχει τις ιδιότητες του προηγούμενου.
- Με την εντολή **close(fd)**, ένας εκ των 2 άκρων κλείνει το socket και σταματά η επικοινωνία.
  Note: Ακόμα κι αν κάνουμε *close()* το socket στο οποίο ακούμε για νέα connections, δεν κλείνουμε ολοκληρωτικά το socket. Επομένως, αν ξεκινήσουμε και πάλι τον server, μπορούμε να δούμε *Unable to bind: Address already in use*. Αυτό συμβαίνει επειδή το λειτουργικό σύστημα είναι υπεύθυνο να κλείσει το socket, και αυτό μπορεί να πάρει μερικά λεπτά. Δες εδώ.

Γενικότερα, κάθε φορά που κάνει *accept()* ένας server (ακούει σε ένα socket για νέα connections) ένα αίτημα ***connect()*** από έναν client, δημιουργεί ένα ξεχωριστό socket (επιστρέφεται νέος fd από την accept) για αυτήν την συγκεκριμένη επικοινωνία με αυτόν τον client.

Δηλαδή στην δική μας περίπτωση, υπάρχει ένα server socket στο οποίο δέχεται αιτήματα *connect()* ο server από διάφορους clients, και απο κει και πέρα δημιουργείται ένα καινούριο socket, με την *accept()*, για κάθε ξεχωριστή επικοινωνία client-server.

select()

Είναι ένα syscall που παρέχει ο linux kernel ώστε ένα πρόγραμμα να μπορεί να “επικοινωνεί” με πολλούς διαφορετικούς file descriptors. Η select, στην δική μας περίπτωση κλήθηκε ως εξής:

rc = select(maxsd + 1, &read_set, NULL, NULL, &timeout);

- Το read_set είναι ένα set (τύπου fd_set) το οποίο περιέχει ένα σύνολο από file descriptors από τους οποίους περιμένουμε να διαβάσουμε κάποιο input. Αν κάποιους από αυτούς γίνει ready, τότε η select ξεμπλοκάρει επιστρέφοντας το πλήθος των fd που είναι έτοιμοι για ανάγνωση. Αφού η select επιστρέψει, το read_set έχει πειραχτεί in place, ώστε να περιέχει μόνο τους fds οι οποίοι είναι έτοιμοι προς ανάγνωση. Με παρόμοια λογική δουλεύει η select() και στην περίπτωση των write_set και except_set τα οποία είναι NULL εδώ.

Note 1: Αν περάσει το timeout και δεν έρθει κάποιο connection, η select “ξεμπλοκάρει” και επιστρέφει 0.
Note 2: Μπορούμε να τσεκάρουμε αν κάποιος fd ανήκει σε ένα fd_set κάνοντας πχ:
FD_ISSET(fd, &read_set);
Note 3: Αν χρησιμοποιούμε έναν fd για επικοινωνία (πχ. μέσω sockets) με κάποιον client και οι client κλείσει το connection (πχ, τερματίζει, ή close(fd)), τότε η select “ξεμπλοκάρει” και ο fd τώρα περιέχεται στο read_set. Επομένως ύστερα, κάνοντας κάποιο read(fd,...) που θα μας επιστρέψει 0, μπορούμε να καταλάβουμε ότι ο client έχει φύγει και να τον αφαιρέσουμε από το master_set μας, όπου φυλάμε όλα τα connections.
Note 4: Μπορούμε να αφαιρέσουμε έναν fd από ένα set με:
FD_CLR(fd, &master_set);

ioctl()

Είναι ένα syscall που παρέχει ο linux kernel ώστε οι εφαρμογές που τρέχουν σε user space να μπορούν να αποκτήσουν πρόσβαση και να χειριστούν συσκευές που είναι συνδεδεμένες στον υπολογιστή. Συγκεκριμένα, επειδή υπάρχει μια ευρεία γκάμα εξωτερικών συσκευών, τα linux δεν γνωρίζουν πως μπορείς να αποκτήσεις πρόσβαση σε όλες αυτές τις συσκευές. Επομένως, προσφέρουν την ioctl() στον χρήστη, ώστε να την καλεί απέναντι σε κάποιο ειδικό αρχείο στο dev filesystem (δίνοντας ως όρισμα τον fd του αρχείου, ο οποίος επιστρέφεται μετά από μια open) και ύστερα να εγείρεται ο αντίστοιχος driver για την συσκευή. Ο driver τρέχει το αντίστοιχο file operation, όπως έχουμε δει μέχρι τώρα, και έτσι ικανοποιεί το αίτημα του χρήστη.Η ioctl() καλείται ως εξής:

*ioctl(int fd, int request, void * …)*

- **fd** είναι ο file descriptor του ειδικού αρχείου
- το **request** είναι ένας κωδικός ο οποίος αντιστοιχίζεται σε κάποια λειτουργία της συσκευής
- το 3ο όρισμα είναι ένας **void pointer** (pointer σε θέση μνήμης οποιουδήποτε τύπου) ο οποίος είναι προαιρετικός και εξαρτάται από τον τύπο του 2ου ορίσματος (request). Το

3ο όρισμα δηλαδή μπορεί να περιέχει την πληροφορία για το τι configuration ακριβώς θέλουμε να αλλάξει στην συσκευή, και τι τιμή να πάρει.

Privilege levels (Rings)

Στα linux, έχουμε 4 διαφορετικά privilege levels (0 έως 3). Στο ring 3 τρέχουν οι user apps, ενώ στο Ring 0, που είναι και το πιο προνομιούχο level, τρέχουν οι διεργασίες του πυρήνα (kernel mode).

Όταν μια user app εκτελεί μια μη επιτρεπτή εντολή στο ring 3, τότε η εντολή γίνεται trap και αναλαμβάνει το λειτουργικό που τρέχει σε ring 0. Μία τέτοια περίπτωση μπορεί να είναι: διαίρεση με το 0 (interrupt από τον επεξεργαστή), πρόσβαση σε μη έγκυρη θέση μνήμης, εκτέλεση syscalls (software interrupt από το app).

Προνομιούχες εντολές είναι οι εντολές που μπορούν να εκτελεστούν μόνο από τον πυρήνα (kernel mode), στο ring 0. Αν μια διεργασία χρήστη προσπαθήσει να εκτελέσει μια προνομιούχο εντολή, τότε γίνεται trap και αναλαμβάνει ο kernel.

Μη προνομιούχες είναι οι εντολές που μπορούν να εκτελεστούν σε όλα τα privilege levels. Παράδειγμα τέτοιων εντολών είναι οι bitwise operations ή οι **αριθμητικές εντολές**, οι οποίες εκτελούνται κατευθείαν στην CPU και σε χώρο χρήστη.

Ευαίσθητες είναι οι εντολές που ανάλογα με την αρχιτεκτονική του συστήματος μπορούν να είναι προνομιούχες ή μη προνομιούχες.

Virtualization Techniques

Full Virtualization

1) **Trap and Emulate**: Οι μη-προνομιούχες του guest εκτελούνται κατευθείαν στην CPU, οι προνομιούχες γίνονται trap από τον VMM και αυτός “μεταφέρει” τις εντολές στον host και πάλι πίσω. Όλες οι ευαίσθητες θεωρούνται προνομιούχες.
2) **Binary Translation**
3) **Hardware-assisted Virtualization**: Το KVM παρέχει τέτοιου είδους virtualization. Κάνει expose τα virtualization extensions του επεξεργαστή του φυσικού μηχανήματος στο VM (το οποίο τρέχει με QEMU).

Paravirtualization

Ο guest έχει έναν τροποποιημένο kernel ώστε να γνωρίζει ότι τρέχει πάνω σε έναν host. Έτσι μπορεί να εκτελεί συγκεκριμένα hypercalls προς τον VMM, εκφράζοντας έτσι τις προθέσεις που έχει απέναντι στο hardware, και αποκτά έλεγχο στο hardware απευθείας, χωρίς να εμπλακεί ο πυρήνας του host. Ως αποτέλεσμα, όταν δουλεύουμε μέσα σε ένα τέτοιο VM, έχουμε αυξημένη

ταχύτητα σε IO λειτουργίες, όπως πρόσβαση στη μνήμη κλπ. Το KVM παρέχει τέτοιου είδους virtualization.

Hybrid Virtualization

Έτσι ονομάζεται ο συνδυασμός hardware-assisted (για να μιλήσει γρήγορα στην CPU) και paravirtualization (για γρήγορες IO λειτουργίες, όπως επικοινωνία με την μνήμη). Αυτού του είδους το virtualization το παρέχει το KVM.

VirtIO

Είναι ένα γενικό framework που υλοποιεί paravirtualization ανάμεσα σε host και guest. Στην δική μας περίπτωση, υπάρχει μια εικονική κρυπτογραφική συσκευή την οποία βλέπει ο guest και μέσω αυτής, μιλάει “έμμεσα” στην “πραγματική” κρυπτογραφική συσκευή: *crypto/dev* του host.

Πρακτικά, το VirtIO αποτελείται από ένα *frontend* και ένα *backend*, τα οποία τρέχουν σε guest και host αντίστοιχα. Ο χρήστης μέσα στο VM, κάνει ioctl() calls προς μια εικονική συσκευή και το frontend μεταφέρει το αίτημα του χρήστη μέσω buffers στο backend. Πιο συγκεκριμένα, οι buffers αυτοί είναι μια **ουρά από buffers** που ονομάζεται **VirtQueue**, και το frontend προσθέτει buffers σε αυτήν την ουρά. Το backend μεταφράζει την απαίτηση του χρήστη σε ioctl() call προς την πραγματική συσκευή, και επιστρέφει τα αποτελέσματα στο frontend μέσω του VirtQueue.

User in VM -> virtual device -> frontend -> VirtQueue -> backend -> real device
User in VM <- virtual device <- frontend <- VirtQueue <- backend <- real device

VirtIO - Backend

Για να πάρουμε το στοιχείο του VirtQueue που μας “έσπρωξε” το frontend:
*VirtQueueElement *elem = virtqueue_pop(vq, sizeof(VirtQueueElement));*

Για να πάρουμε τον δείκτη στην τιμή της 1ης readable από το backend scatterlist.
unsigned int syscall_type = elem->out_sg[0].iov_base;*

Για να πάρουμε τον δείκτη στην τιμή της 1ης writable από το backend scatterlist.
host_return_value = elem->in_sg[0].iov_base;

Για να πειράξουμε in place την writable τιμή:
**host_return_value = close(fd);*

Για να στείλουμε τις αλλαγές πίσω στο frontend (εδώ ξυπνάει το frontend αν περιμένει με while)
virtqueue_push(vq, elem, 0);

Προαιρετικό: Δημιουργεί interrupt στο frontend
virtio_notify(vdev, vq);

Εντολές για 3ή άσκηση

Host

1) cd ~/crypto/cryptodev/cryptodev-linux/
2) make && sudo make install
3) sudo insmod cryptodev.ko
4) cd ~/qemu-3.0.0
5) patch -p1 < ../crypto/virtio-cryptodev/backend/qemu-3.0.0_helpcode.patch
6) cp ../crypto/virtio-cryptodev/backend/virtio-cryptodev.c hw/char/virtio-cryptodev.c
7) make && make install
8) cd ~/utopia && ./utopia.sh -device virtio-cryptodev-pci

Guest

1) ssh -p 22223 root@localhost
2) cd / && sshfs -o allow_other konstantinos@10.0.2.2:/home/konstantinos /home/user/host
3) cd /home/user/host/crypto/virtio-cryptodev/frontend && chmod +x initialize.sh && chmod +x crypto_dev_nodes.sh && ./initialize.sh
4) ./test_crypto /dev/cryptodev0
5) cd ../chat && make clean && make
6) ./crypto-server /dev/cryptodev0
7) cd /home/user/host/crypto/virtio-cryptodev/chat && ./crypto-client localhost 35001 /dev/cryptodev0

Watch TCP packages: tcpdump tcp port 35001 -vvv -XXX -i lo

Εντολές φλοιού/system calls

chmod: Αλλαγή δικαιωμάτων στο αρχείο, επιλογές r ανάγνωση,w εγγραφή, x εκτέλεση, u για τον user, a για όλους.
ps -aux: Εμφάνιση διεργασιών
ltrace/strace: Εκτέλεση προγράμματος με trace βιβλιοθηκών/κλήσεων συστήματος.
dup2 (f1, f2): Redirect τον file descriptor f2 στον f1.

ln, ln -s: Δημιουργία hard/soft link. Με hard link γίνεται assign το ίδιο Inode, δηλαδή δείχνουν στην ίδια διεύθυνση φυσικής μνήμης, παραμένουν συνδεδεμένοι ακόμα και αν αλλάξει η θέση του αρχείου. Soft link είναι σαν shortcut στα windows, κάθε ένα έχει διαφορετικό Inode που κάνει reference το ίδιο αρχείο.
lseek(fd, offset, whence): Μετακινεί το file offset στον fd. whence = SEEK_SET ακριβώς offset bytes, SEEK_CUR offset bytes από την αρχή του αρχείου, SEEK_END offset bytes από τη θέση που είναι εκείνη τη στιγμή.