Lecture 8 - Introduction to Linux Kernal Module Programming

מה זה מערכת ההפעלה ?

מתמקדים בגישה טכנולוגית ולא עסקית. (לדוגמא windows נמכר יחד עם freecell explorer ו windows מדיה) ברור שאנחנו לא מדברים על freecell כשאנחנו מדברים על מערכת הפעלה.

אנחנו צריכים סט שרותים מינימלי שמקשר בין החומרה לתהליכים שרצים.

Kernel מתווך בין כל התהליכים בחומרה.

(זה ממש בקיצור קיצור קיצור אבל אתם בדר"כ לא רוצים חפירות אז אני זורם)

כשהתחילו מערכות ההפעלה, אחת השאלות שהייתה בולטת היא לגבי גודל הקרנל. ישנם שני סוגים.

Microkernel (גרעין קטן) – רק המינימום ההכרחי של הפונקציונליות ימצא בקרנל והשאר יבוצע בתהליכים חיצונים. משפר את האבטחה, יש סיכוי יותר קטן לבאגים ב- kernel אלא רק ב-user space באגים בסביבת משתמש הרבה פחות בעייתים מבחינת השפעה ויכולת ולכן אנחנו פחות חשופים.

בנוסף קל יותר לאתר ולפתור באגים כשהם לא ב-kernel. (כלומר קרנל קטן – פחות באגים בקרנל– פחות סיכון)

Monolithic kernel (גרעין גדול) – kernel גדול מרוויח ביצועים - פעולת החלפת התהליכים שקורת לעיתים תכופות יותר ב- **microkernel** היא פעולה יקרה מאוד. מנגד, ב - kernel גדול כל הפעולות מבוצעות בתוך ה-kernel עצמו בלי מעורבות של תהליכים חיצוניים וללא החלפות. **החיסרון** הגדול הוא רגישות לבאגים, באג במנהל התקנים עלול לגרום לקריסת המערכת כולה, כמו כן ליבות גדולות מייקרות את ההחלפות כאשר הן כבר קורות.

Linux התחילו ישירות מ **monolithic kernel** שיכול לתמוך בכל סוגי החומרה, **יצרו kernels** מאוד גדולים. בהמשך (לפני בערך 30-25 שנה) לינוקס תמך במודולים כלומר לטעון ולשחרר חלקים מהקרנל בזמן אמת ועל ידי זה להקטין את הקרנל.

המודל של **windows** התחיל מ**microkernel** **במעלה הדרך ניסו להגדיל אותו ובכך שיפרו בחלק מהמשימות (לדוגמא משחקים) מהביצועים.** כיום שתי המערכות היא מהסוג **hybrid kernel** שמאפשרת לנהל הכנסה והוצאה של דברים לתוך ה- kernel וממנו ובכך להגדיל אותו לצורך שיפור ביצועים או להקטין אותו למינימום הנדרש בסביבה.

בקורס נלמד לכתוב kernel modules במערכת לינוקס.

המדריך שאיתו נעבוד בפרק זה – The Linux Kernel Module Programming Guide

מומלץ מאוד לקחת את כל דוגמאות קוד במדריך עד פרק 6, לקמפל וללמוד.

(עד פרק 6 לשיעור הזה. בעקרון למדנו עוד פרקים וגם אותם מומלץ לקמפל ולהריץ)

Kernel modole

```
/*
 * hello-1.c - The simplest kernel module.
 */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

int init_module(void)
{
    pr_info("Hello world 1.\n");

    /* A non 0 return means init_module failed; module can't be loaded. */
    return 0;
}

void cleanup_module(void)
{
    pr_info("Goodbye world 1.\n");
}

MODULE_LICENSE("GPL");
```

נשים לב כי ישנם מספר שינויים בעבודה עם **kernel module** לעומת **user space** שעבדנו בו עד כה.

עיקר השוני נובע מכך שה-**kernel** מגיב לבקשות ולא מריץ תוכנית ספציפית וגם כי הרבה מהפקודות שאנו מכירים פונות אל ה-**kernel** מה שלא רלוונטי במקרה של בניית **kernel module**.

Init_module - אתחול המודול (חליף main) – זה לא ממש נכון – כי שוב המודול לא מבצע את ההוראות במיין ויוצא. הוא למעשה רץ ועונה על בקשות. init module זה מחליף ל dllmain בוינדוס או attribute (constructor) של gcc.

cleanip_module – הורדת המודול.

בין שתי פקודות אלה המודול פעיל ומגיב לבקשות.

pr_info – מחליף את **printf** שאנו מכירים ומאפשר הדפסה מה-**kernel module** מכיוון שאיננו יכולים להשתמש בספריות. (קורא בשכבה נמוכה יותר לprintk שהיא הגירסא הקרנלית ל printf)

MODULE_LICENSE - באיזה רישיון המודול שלנו נכתב. בקוד פתוח בקרנל לרוב אנו משתמשים ב-**GPL**.

ה-**include** הוא לא **stdio** הרגיל כי אין בו צורך יותר.

קומפיצליה – קומפיצליה רגילה באמצעות ה - **gcc** אינה רלוונטית עוד כי הוא מחפש **headers** וספריות שאין בהם צורך וההדרים שאנחנו משתמשים בהם אינם בסרץ פף)

נעבוד עם **makefile** ונבנה אותו באופן הבא:

```
obj-m += hello-1.o

PWD := $(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

הערה – מי שמתמש בכל מיני סביבות פיתוח חכמות שמחפשות הדרים וכדומה יכול להיות שתראו כל מיני סימנים אדומים שהוא לא מוצא הדרים. אם זה מה שאתם רואים צריך לקנפג את הסביבה או להתעלם.

מוסיפים למודולים (**obj-m**) את המודול החדש שיצרנו **hello-1.o**, מריץ את ה- **make** על עצמו בצורה רקורסיבית כשהוא מוסיף לרשימת המודולים את המודול החדש. לבסוף מתקבל **hello-1.ko** (ko->kernel object).

הרצה של המודול מתבצעת על ידי הרשאות admin באמצעות הsudo.

```
modinfo hello-1.ko
```

At this point the command:

```
sudo lsmod | grep hello
```

should return nothing. You can try loading your shiny new module with:

```
sudo insmod hello-1.ko
```

The dash character will get converted to an underscore, so when you again try:

```
sudo lsmod | grep hello
```

you should now see your loaded module. It can be removed again with:

```
sudo rmmod hello_1
```

Notice that the dash was replaced by an underscore. To see what just happened in the logs:

```
sudo journalctl --since "1 hour ago" | grep kernel
```

ישנם סדר גודל של עשרות אלפים של מודולים ב-**kernel**. לצורך ניהול יעיל של ה-**kernel** על שמות המודולים להיות נבדלים זה מזה. דוגמא בולטת לצורך זה היא היכולת לזהות איזה מודל זורק שגיאה ועוד.

```c
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int __init hello_2_init(void)
{
    pr_info("Hello, world 2\n");
    return 0;
}

static void __exit hello_2_exit(void)
{
    pr_info("Goodbye, world 2\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);

MODULE_LICENSE("GPL");
```

module_init ו-module_exit שני macros שיוצרים את הפונקציות init ו-clean מבלי שרואים אותם וכך לכל מודול יש שם משלו וניתן לנהל מודולים בשמות שונים.

על מנת להקטין את ה- kernel ניתן להשתמש ב-macros, פונקציות שנועדו לצורך אתחול בלבד ואין בהן צורך בהמשך, ישוחרר ע"י ה-kernel לאחר האתחול.

macro _exit – משתמשים בו כדי להגיד שהקוד הזה נועד לצורך הורדת המודול אז אומנם או "יישאר" עד סוף התוכנית אבל הוא לא ייכנס לזיכרון בקונטקס סוויץ.

__initdata - עובד בדומה ל-__init אבל עבור משתני init ולא פונקציות.

```c
static int __init hello_3_init(void)
{
    pr_info("Hello, world %d\n", hello3_data);
    return 0;
}
```

בטוטריאל 4 הוא מגדיר מידע (מי המחבר מה מטרת המודול) שאפשר לשלוף בעזרת modinfo

בטוטריאל 5 הוא מוסיף פרמטרים למודל (כמו argc, argv בתוכנית)

ניתן להוסיף פרמטרים ואת התיאור שלהם:

```
26 module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
27 MODULE_PARM_DESC(myshort, "A short integer");
28 module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
29 MODULE_PARM_DESC(myint, "An integer");
30 module_param(mylong, long, S_IRUSR);
31 MODULE_PARM_DESC(mylong, "A long integer");
32 module_param(mystring, charp, 0000);
33 MODULE_PARM_DESC(mystring, "A character string");
```

לאחר ההגדרה שלהם ניתן לשלוף את המידע עליהם באמצעות הפקודה

$modinfo <module name.ko>

איך נוכל לבנות מודול בכמה קבצים – multi modelling?

הקובץ הראשון יהיה start.c והשני stop.c:

```
1  /*
2   * start.c - Illustration of multi filed modules
3   */
4
5  #include <linux/kernel.h> /* We are doing kernel work */
6  #include <linux/module.h> /* Specifically, a module */
7
8  int init_module(void)
9  {
10     pr_info("Hello, world - this is the kernel speaking\n");
11     return 0;
12 }
13
14 MODULE_LICENSE("GPL");
```

```
1  /*
2   * stop.c - Illustration of multi filed modules
3   */
4
5  #include <linux/kernel.h> /* We are doing kernel work */
6  #include <linux/module.h> /* Specifically, a module  */
7
8  void cleanup_module(void)
9  {
10     pr_info("Short is the life of a kernel module\n");
11 }
12
13 MODULE_LICENSE("GPL");
```

החיבור בניהם יתבצע ב-make file ע"י השורה: startstop-objs := start.o stop.o

Character device drivers

– יכולת לבנות kernel module יותר אטרקטיבי, עם יכולת להגיב.

ב- Unix אנו עובדים באמצעות קבצים, כאשר המשתמש "מושך" קובץ מסוים הוא למעשה פונה באמצעות הפקודות השונות ל-kernel.

ישנן 2 סוגי דרייברי ם העובדות בשיטות שונות - Block device ו- Character device.

<u>הגדרה לא נכונה אבל מסבירה את ההבדלים טוב יותר:</u>

Character device – עובד בשיטה של תו תו.

Block device – עובד בשיטת הבלוקים

<u>הגדרה לא נכונה נוספת:</u>

אם אפשר לעשות חיפוש על ה - device מדובר בשיטת block, אחרת שיטת ה-character.

<u>הגדרה נכונה:</u>

block layer - שכבה במערכת שמנהלת את הקלט והפלט.

block device בקשות שמנוהלות ע"י ה-block layer

הבלוק לאיר יכול לשנות את סדר הבקשות. לבקש דברים שלא ביקשו בקוד אם חושב שזה יעיל יותר (לדוגמא אם ביקשנו 1,2,3,5 אולי עדיף לבקש 1-5 ולזרוק את 4) וכדומה.

לכן יותר קשה (במיוחד בהתחלה) לדבג בלוק דוויס. אנחנו נתחיל מצ'אר דביס – כי זה יותר קל.

, character device – בקשות שהוא לא מנהל. נלמד לעבוד עם Character device כי זה פשוט יותר.

נזכיר כי ב-linux כל העבודה מבוצעת היא באמצעות קבצים.

מבנה file_operations מוגדר ב- include/linux/fs.h, ומכיל מצביעים לפונקציות שהוגדרו על ידי הדרייבר ומבצעות פעולות שונות במכשיר. כל שדה במבנה מתאים לכתובת של פונקציה כלשהי שהוגדרה על ידי הדרייבר

לטיפול בפעולה המבוקשת. לדוגמה, כל מנהל התווים צריך להגדיר פונקציה שקוראת מהמכשיר. מבנה file_operations מכיל את הכתובת של הפונקציה של המודול שמבצעת את הפעולה הזו. כך נראית ההגדרה עבור קרנל 5.4:

```
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 (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
```

חלק מהפעולות אינן מיושמות על ידי מנהל התקן. לדוגמה, מנהל התקן המטפל בכרטיס מסך לא יצטרך לקרוא ממבנה ספריות. יש להגדיר את הערכים שלא נרצה שיהיו ממומשים ל-NULL.

פעולות שאנו מגדירים:

```
struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

על מנת לייצר קובץ חדש ניצור character device חדש ונכניס אותו לkernel תוך כדי קישור ל- file operations שצריכים להיות לו.

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

מחיקה תתבצע ע"י unregister.

```
static int __init chardev_init(void)
{
    major = register_chrdev(0, DEVICE_NAME, &chardev_fops);

    if (major < 0) {
        pr_alert("Registering char device failed with %d\n", major);
        return major;
    }

    pr_info("I was assigned major number %d.\n", major);

    cls = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_NAME);

    return SUCCESS;
}
```

באמצעות major device נוכל לייצר מספר רכיבים מאותו הסוג ובאמצעות המספר להבדיל בניהם.

Device create משייך את הרכיב לקובץ.

Exit – מנטרל את הרכיב ומשחרר את ה-character device.

Put_user/get_user – אם נרצה להעתיק זיכרון מה-kernel ל-user וההפך נשתמש בפקודות אלה. באותה שיטה נוכל לעשות copy_user.

הקוד המלא נמצא במדריך, פרק 6.5. מומלץ להעתיק את הקוד, לקמפל וללמוד היטב את הפקודות.

ניתן לייצר char device לתקשורת עם הקרנל גם מהcommand line בעזרת mknod(1)