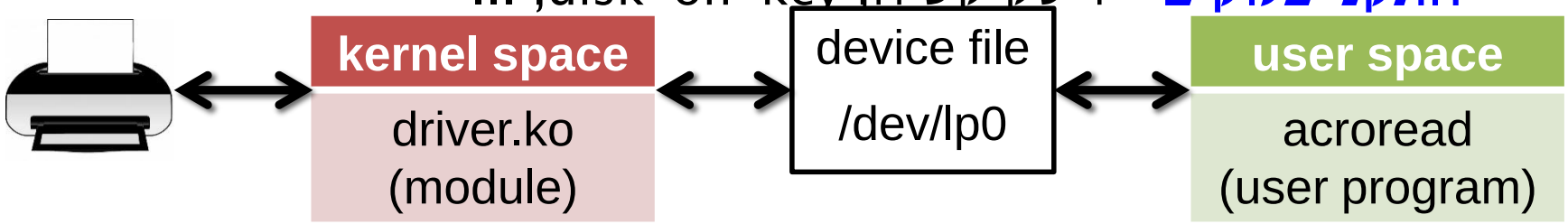
תרגול 4

רצף האתחול (boot sequence) בלינוקס
מודולים בלינוקס (loadable kernel modules)
התקני תווים (character devices)
דרייברים (device drivers)
מימוש דרייברים באמצעות מודולים

TL;DR

- **מודולים** בלינוקס הם תוספות קוד לגרעין שניתן לטעון **בזמן ריצה.**
  - לא צריך להדר מחדש את הגרעין, לא צריך לאתחל את המחשב.

- מודולים משמשים להוספת פונקציונליות חדשה לגרעין, לדוגמה דרייברים להתקני חומרה שמחברים/מנתקים מהמערכת:
  - **התקני תווים** – מקלדת, עכבר, מדפסת, ...
  - **התקני בלוקים** – דיסק קשיח, disk-on-key, ...



רצף האתחול בלינוקס

Linux Boot Sequence

אתחול המחשב

- כאשר המחשב כבוי, מערכת ההפעלה, היישומים וכל הקבצים של המשתמש נשמרים על הדיסק.
- כאשר המשתמש מדליק את המחשב (באמצעות לחיצה על כפתור POWER/ON), מערכת ההפעלה בדרך כלל לא נמצאת עדיין בזיכרון ולכן צריך לטעון אותה מהדיסק.

- יש פה פרדוקס: כדי לטעון את מערכת ההפעלה מהדיסק לזיכרון, צריך מערכת הפעלה שיודעת לגשת להתקנים כמו הדיסק.
- הפרדוקס נפתר באמצעות תהליך אתחול הדרגתי של המחשב: סדרת רכיבי תוכנה שהמחשב מבצע כאשר המשתמש מדליק אותו.

מה קורה לאחר הדלקת המחשב?

קוד ה-BIOS נטען לזיכרון והמעבד מתחיל להריץ אותו

ה-BIOS טוען את הסקטור הראשון של הדיסק (MBR) לזיכרון

קוד ה-MBR טוען את ה-boot loader

ה-boot loader טוען את גרעין לינוקס

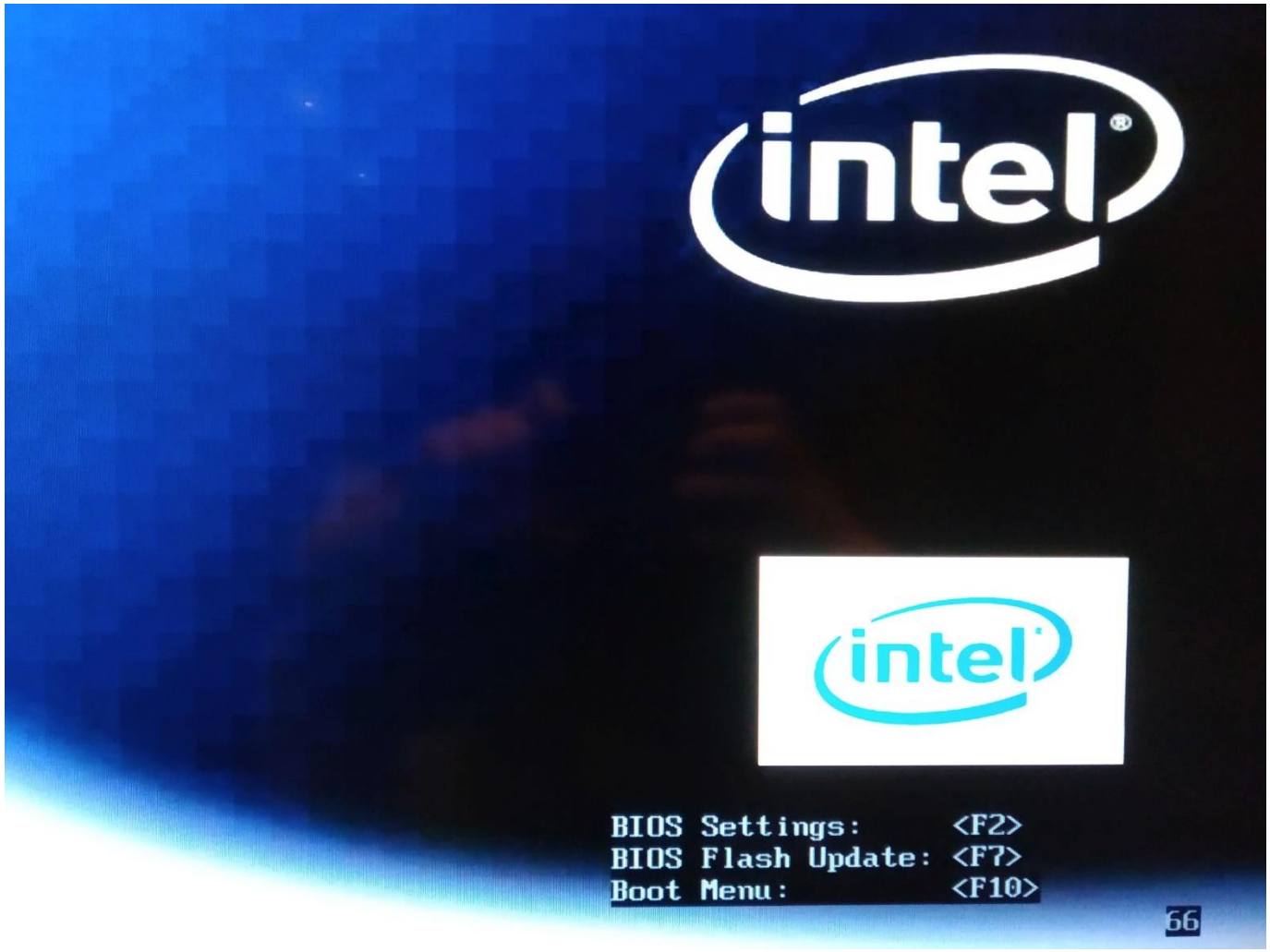
גרעין לינוקס טוען את תהליך init ומריץ אותו

BIOS (Basic Input/Output System)

- התוכנה הראשונה שמופעלת לאחר הדלקת המחשב.
- קוד ה-BIOS צרוב בזיכרון ייעודי בלוח האם ואז נטען לכתובת קבועה בזיכרון באופן אוטומטי ברגע הדלקת המחשב.

- תפקידי ה-BIOS:
  - לזהות את התקני החומרה המחוברים למערכת.
  - לבדוק שההתקנים הבסיסיים (מסך, מקלדת, ...) פועלים בצורה תקינה.
  - לעבור על רשימה מוגדרת מראש של התקנים, ולחפש התקן המאפשר אתחול (bootable device).
  - אם לא נמצא אף התקן כזה, ה-BIOS רושם הודעת שגיאה למסך ומפסיק את תהליך האיתחול.
  - אם נמצא התקן כזה, ה-BIOS טוען את הסקטור הראשון של ההתקן למקום קבוע בזיכרון.

example BIOS screen



MBR (master boot record)

- דיסק קשיח הוא התקן איחסון המחולק לסקטורים בגודל 512 בתים.
  - קריאה/כתיבה מהדיסק נעשית בכפולות של סקטורים.
- אם הדיסק הוא התקן המאפשר אתחול (bootable device), אז הסקטור הראשון בדיסק נקרא MBR (master boot record).
- ה-MBR מכיל קוד אסמבלי בסיסי המשמש לטעינת קוד נוסף: מנהל האתחול (boot loader).

למה לא לשמור את ה-boot loader ב-MBR?

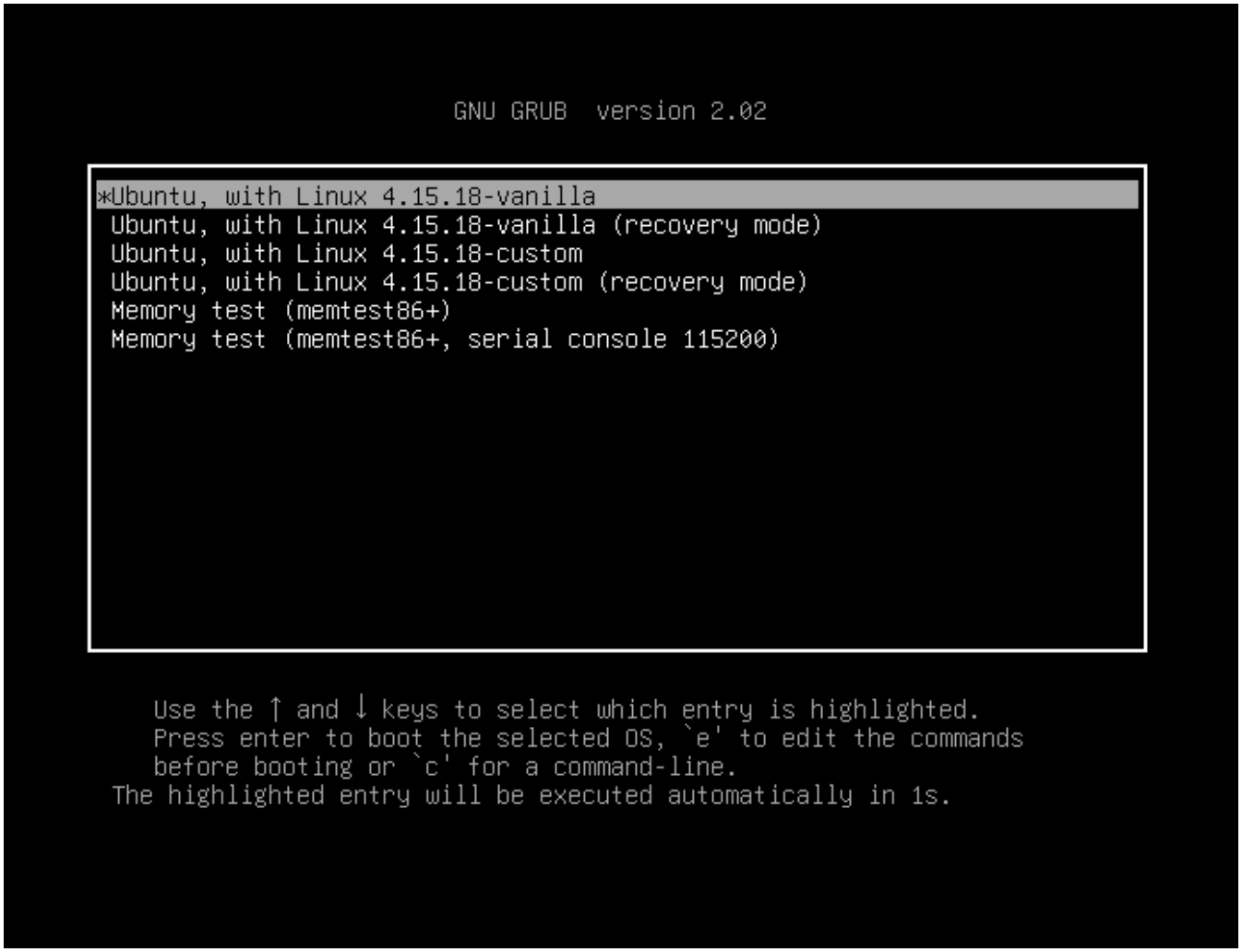
- יש מספר boot loaders נפוצים. מערכות לינוקס משתמשות בדרך כלל ב-GRUB.

GRUB (GRand Unified Bootloader)

- בשקף הבא מופיע דוגמה של תפריט ה-GRUB: הוא מאפשר לבחור את גרעין מערכת ההפעלה אשר ייטען לזיכרון.

- תפריט ה-GRUB שימושי כאשר מפתחים גרעין לינוקס חדש (כפי שתעשו בשיעורי הבית):
  - נניח כי עידכנתם את קוד הגרעין של לינוקס ושמרתם אותו בתמונה /boot/vmlinuz-4.15.18-**custom** על הדיסק.
  - לאחר מכן ניסיתם לטעון את התמונה הזו לזיכרון ומערכת ההפעלה קרסה ☹ לכולנו יש באגים לפעמים...
  - כעת תוכלו להפעיל מחדש את המחשב ולטעון תמונה אחרת ותקינה, למשל /boot/vmlinuz-4.15.18-**vanilla** מתוך הדיסק.
  - לאחר עליית מערכת ההפעלה, תוכלו לנסות ולתקן את התמונה /boot/vmlinuz-4.15.18-**custom** על הדיסק.

example GRUB menu



טעינת גרעין לינוקס

- גם גרעין לינוקס נטען בשלבים:

1. GRUB טוען לזיכרון תמונה דחוסה של הגרעין, אשר נקראת בדרך כלל bzImage או vmlinuz, ואז מחלץ אותה.
   - התמונה הדחוסה של גרסת לינוקס 4.15 שוקלת בערך 8MB.
2. לצד תמונת הגרעין, GRUB טוען לזיכרון גם את מערכת הקבצים הראשונית: מערכת קבצים קטנה בשם initramfs או initrd.
   - initrd = initial RAM disk
   - initramfs = initial RAM file-system
3. גרעין לינוקס מריץ את התוכנית init/ מתוך מערכת הקבצים הראשונית.
   - init/ היא בדרך-כלל סקריפט shell.

טעינת גרעין לינוקס

```
Loading Linux 4.15.18-vanilla ...
Loading initial ramdisk ...
```

מערכת הקבצים הראשונית

- מערכת הקבצים הראשונית מכילה את המודולים הנחוצים עבור גרעין לינוקס כדי לחפש ולטעון את מערכת הקבצים האמיתית.
  - לדוגמה: מודולים הממשים דרייברים של הדיסק או כרטיס הרשת.
  - שימו לב: מערכת הקבצים הראשונית לא מכילה את כל המודולים הקיימים, אלא רק את החיוניים שבהם. שאר המודולים נמצאים במערכת הקבצים האמיתית, ולאחר שהיא תעלה גרעין לינוקס יוכל לטעון גם אותם (במידת הצורך).

4. התוכנית init/ טוענת את הדרייברים הנחוצים ומרכיבה (mounts) את מערכת הקבצים האמיתית במקום מערכת הקבצים הראשונית.
5. לאחר שעלתה מערכת הקבצים האמיתית, גרעין לינוקס קורא לתוכנית /sbin/init .
   - היום /sbin/init הוא קישור לתוכנית /lib/systemd/systemd .

>> dmesg –H

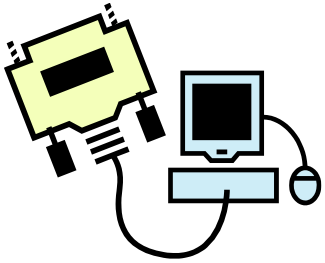
- פקודת dmesg מאפשרת לקרוא הודעות שהודפסו למסך בתהליך האיתחול לאחר שהמערכת הופעלה:

```
[Apr24 13:01] Linux version 4.15.18-custom (student@pc) (gcc version 7.5.0 (Ubuntu 7
[  +0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.15.18-custom root=UUID=3c7375
[  +0.000000] KERNEL supported cpus:
[  +0.000000]   Intel GenuineIntel
[  +0.000000]   AMD AuthenticAMD
[  +0.000000]   Centaur CentaurHauls
[  +0.000000] Disabled fast string operations
[  +0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers
[  +0.000000] x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
[  +0.000000] x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
[  +0.000000] x86/fpu: Supporting XSAVE feature 0x008: 'MPX bounds registers'
[  +0.000000] x86/fpu: Supporting XSAVE feature 0x010: 'MPX CSR'
[  +0.000000] x86/fpu: xstate_offset[2]:  576, xstate_sizes[2]:  256
[  +0.000000] x86/fpu: xstate_offset[3]:  832, xstate_sizes[3]:   64
[  +0.000000] x86/fpu: xstate_offset[4]:  896, xstate_sizes[4]:   64
[  +0.000000] x86/fpu: Enabled xstate features 0x1f, context size is 960 bytes, usin
[  +0.000000] e820: BIOS-provided physical RAM map:
[  +0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009e7ff] usable
[  +0.000000] BIOS-e820: [mem 0x000000000009e800-0x000000000009ffff] reserved
[  +0.000000] BIOS-e820: [mem 0x00000000000dc000-0x00000000000fffff] reserved
[  +0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000bfecffff] usable
[  +0.000000] BIOS-e820: [mem 0x00000000bfed0000-0x00000000bfefefff] ACPI data
[  +0.000000] BIOS-e820: [mem 0x00000000bfeff000-0x00000000bfeffffff] ACPI NVS
:
```

מודולים בלינוקס

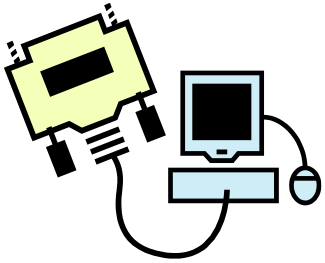
Loadable Kernel Modules

מודולים (modules)



- **מודולים** מאפשרים להוסיף לגרעין לינוקס, **בזמן ריצה**, קטעי קוד חדשים.

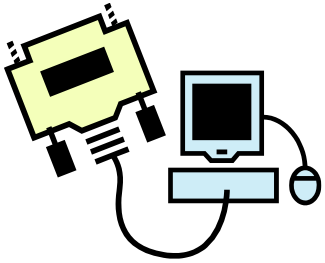
- מודול הוא ספריה משותפת (shared library) הנטענת (מקושרת) בזמן ריצה ופועלת במצב גרעין (כלומר CPL==0).
- הפקודה **insmod** טוענת מודול חדש.
- הפקודה **rmmod** פורקת מודול שנטען בעבר.
- הפקודה **lsmod** מציגה את רשימת המודולים הפעילים (כלומר, שנטענו ע"י המשתמש).



למה משמשים מודולים?

- מודולים משמשים בעיקר על מנת להוסיף תמיכה בהתקני חומרה (devices) ע"י דרייברים (drivers):
  - התקני תווים – מקלדת, עכבר, מדפסת, ...
  - התקני בלוקים – דיסק קשיח, disk-on-key, ...
  - התקני רשת – נראה בתרגולים הבאים.

- למעשה, מודולים רצים בהרשאות גרעין ולכן הם יכולים להוסיף ולעדכן כל פונקציונליות של הגרעין:
  - להוסיף מערכות קבצים חדשות.
  - להוסיף קריאות מערכת חדשות ו/או לשנות קריאות מערכת קיימות.

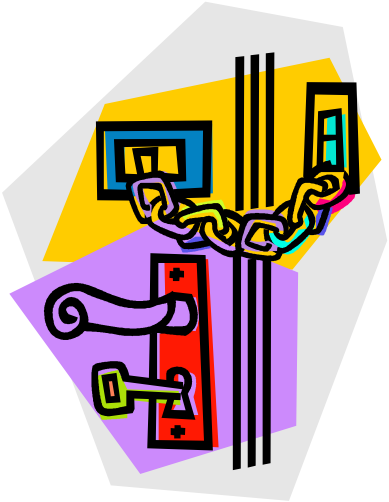


יתרונות השימוש במודולים

1. ניתן להוסיף יכולות חדשות לגרעין מבלי לקמפל אותו ומבלי לאתחל (reboot) את המערכת.
2. מודולים מפותחים בנפרד מהגרעין ← פחות שורות קוד בגרעין ← זמן קומפילציה קצר יותר.
   - אין צורך לקמפל פונקציונליות מיותרת, למשל דרייברים לחומרה שלא נמצאת ברשותנו.
3. הגרעין תופס פחות זכרון.
   - המשתמש טוען לזיכרון רק מודולים שהוא זקוק להם.
   - ניתן לפרוק מודולים שאינם בשימוש ולשחרר זיכרון.
4. ניתן להוסיף בעתיד תמיכה בחומרה חדשה שעדיין לא קיימת.

ענייני אבטחה

- מודול רץ במצב גרעין ולכן יש לו גישה לכל מבני הנתונים בגרעין.
- חשוב להימנע מחורי אבטחה במודול:
  - למשל: הקפדה על אתחול משתנים, בדיקת תקינות קלט משתמש כדי למנוע buffer overflow וכולי.
- כמו כן, לינוקס מגבילה טעינת מודולים למשתמשים מורשים (root) בלבד.



דוגמת קוד: מודול ראשון

```
#include <linux/module.h> // always required
#include <linux/kernel.h> // for printk
```

נקראת בטעינת המודול

```
int init_module(void) {
  printk("Hello World!\n");
  return 0;
}
```

האם אפשר לקרוא ל-
printf במקום printk?

נקראת בטעינת המודול

```
void cleanup_module(void) {
  printk("Goodbye cruel world!\n");
}
```

בניית המודול

- קובץ makefile לדוגמה:

```
obj-m += hello.o
all:
 make -C /lib/modules/$(shell uname -r)/build
 M=$(PWD) modules
clean:
 make -C /lib/modules/$(shell uname -r)/build
 M=$(PWD) clean
```

- חייבת להיות התאמה מלאה בין גרסת לינוקס עליה נבנה המודול לבין גרסת לינוקס בה הוא רץ.
  - חוסר תאימות עשוי לגרום לבעיות בזמן ריצה.

טעינת המודול

```
>> make
>> sudo insmod ./hello.ko
Hello world!
>> lsmod
Module          Size   Used by
hello           868    0   (unused)
...
...
>> sudo rmmod hello
Goodbye cruel world!
```

העברת פרמטרים למודול

- הגדרת הפרמטרים בקוד המודול:

```
#include <linux/moduleparam.h>
int iValue=0;    // 0 is the default value
char *sValue;
module_param(iValue, int, S_IRUGO);
module_param(sValue, charp, S_IRUGO);
```

- העברת פרמטרים בטעינת המודול:

```
>> insmod ./params.ko iValue=3
 sValue="hello"
```

העברת פרמטרים למודול

- מודולים יכולים לקבל פרמטרים מהמשתמש בזמן טעינתם.
- הפרמטרים מוגדרים בקוד באמצעות המאקרו module_param.
  - המאקרו צריך להופיע מחוץ לפונקציה. בד"כ ממוקם בתחילת הקוד.
  - פרמטר ראשון – המשתנה שיכיל את הפרמטר, פרמטר שני – סוג הפרמטר, פרמטר שלישי – הרשאות גישה לקובץ המתאים ב-sysfs (לא רלוונטי כרגע).
  - יש להגדיר לכל פרמטר ערך ברירת מחדל.
- סוגי פרמטרים לדוגמה: byte, short, int, charp, bool, invbool.
- ניתן להשתמש במאקרו MODULE_PARM_DESC כדי להוסיף תיאור לפרמטר.
  - כלי ניהול אוטומטיים יכולים לקרוא את התיאור (אפשר גם עם modinfo).

גישה לנתוני גרעין

- למודול יש גישה למבני הנתונים של הגרעין במידה והוא מצרף את הקבצים המתאימים (ע"י #include).

```
#include <linux/sched.h>

int init_module(void)
{
  printk("The process is \"%s\" (pid %d)\n",
    current->comm, current->pid);
  return 0;
}
```

comm הינו שדה השומר את שם התוכנית המתבצעת. מה שם התכנית שיודפס במקרה זה?

התקני תווים
Character Devices

התקנים (devices) ודרייברים (drivers)

- **התקנים:**
  - מיוצגים ע"י קבצים מיוחדים בנתיב dev/ במערכת הקבצים.
  - המשתמש עובד מול ההתקן באמצעות **הממשק הסטנדרטי** לעבודה מול קבצים – קריאות המערכת open(), read(), write().
  - לדוגמה: כדי להשתמש במדפסת יש לפתוח את הקובץ dev/lp0/ ואז לכתוב אליו את הטקסט להדפסה.

- **דרייבר (מנהל התקן):** פועל בהרשאות גרעין וממפה את קריאות המערכת הללו לפעולות ספציפיות להתקן.
  - לדוגמה: הדרייבר של המדפסת "מדבר" עם המדפסת בפקודות ספציפיות עבורה (איפה להדפיס על הנייר, מתי מסתיימת שורה של הדפסה, ...).
  - דרייבר הוא שכבת תוכנה החוצצת בין ההתקן לבין האפליקציה כדי לספק אבסטרקציה לפעולת ההתקן הספציפי.

התקני תווים ובלוקים

התקן תווים
(character devices)

- התקן שניגשים אליו כאל רצף של בתים.
  - לרוב משמשים **להעברת** מידע. לדוגמה: מסך, מקלדת.
  - בדרך כלל ניתן לגשת להתקן תווים רק באופן סדרתי (ולא אקראי).

התקן בלוקים
(block devices)

- התקן שניתן לגשת אליו רק בכפולות של בלוק (למשל 512 בתים).
  - לרוב משמשים **לאחסון** מידע. לדוגמה: דיסק קשיח, דיסק נשלף (disk on key).
  - התקן בלוקים מאפשר גישה אקראית למידע שבו.
  - לינוקס מוסיפה שכבה נוספת (**page cache**) ומאפשרת לקרוא מהתקני בלוקים גם בתים בודדים.

התקני תווים

- התקן תווים מאופיין ע"י שני מספרים:
- **מספר ראשי (major number)** – מזהה את הדרייבר המקושר להתקן.
- **מספר משני (minor number)** – מזהה את ההתקן הספציפי המקושר לאותו דרייבר (יכולים להיות מספר התקנים, למשל מספר עכברים, המנוהלים ע"י אותו דרייבר).

```
>> ls -l /dev
crw-rw-rw- 1 root root 1,  3 Aug 31 10:33  null
crw------- 1 root root 10, 1 May 12 10:33  psaux
crw------- 1 root tty  4,  1 May 12 10:33  tty1
crw-rw-rw- 1 root root 1,  5 Aug 31 10:33  zero
```

התקני תווים מסומנים ע"י תו c בעמודה הראשונה

התקני תווים פיקטיביים

- לינוקס מספקת גם התקני תווים פיקטיביים שאינם קשורים לחומרה אמיתית, כולם בעלי מספר ראשי 1.

<table>
<tr><th>device file</th><th>read()</th><th>write()</th></tr>
<tr><td>/dev/null</td><td>מחזירה מיד EOF
(end of file)</td><td rowspan="2">מצליחה ולא עושה דבר
(המידע שנשלח נזרק)</td></tr>
<tr><td>/dev/zero</td><td rowspan="2">מחזירה רצף אפסים
באורך המבוקש</td></tr>
<tr><td>/dev/full</td><td>מחזירה מיד ENOSPC
(no space left on device)</td></tr>
<tr><td>/dev/random
/dev/urandom
/dev/arandom</td><td>מחזירה רצף בתים
אקראי שנוצר בזמן ריצה
מתוך "רעש"</td><td>כותבת לרצף הבתים האקראי</td></tr>
</table>

שאלות לווידוא הבנה

- מה מבצעת הפקודה?

```
>> some_program > /dev/null
```

- מריצה את התוכנית `some_program` בחזית ומשתיקה את כל ההדפסות שלה לערוץ הפלט הסטנדרטי.
  - כלומר התוכנית לא תדפיס למסך (אלא אם כן יש הדפסות ל-STDERR).

- מה מבצעת הפקודה?

```
>> cat /dev/zero > file.txt
```

- מייצרת קובץ אינסופי באורכו מלא באפסים.
  הפעולה לא תעצור עד שנשלח סיגנל הריגה עם CTRL+C.

יצירת התקן חדש

shell utility

mknod <NAME> <TYPE> <MAJOR> <MINOR>

- <u>פעולה:</u> יוצרת קובץ התקן חדש.
  - הפעולה דורשת הרשאות root.
- <u>פרמטרים:</u>
  - NAME – שם הקובץ החדש שייצג את ההתקן.
  - TYPE – סוג ההתקן (c – התקן תווים, b – התקן בלוקים).
  - MAJOR – המספר הראשי של הדרייבר המפעיל את ההתקן.
  - MINOR – המספר המשני של ההתקן (מספר בין 0 ל–255).
- התוכנית mknod ממומשת באמצעות קריאת המערכת mknod().

דוגמה: יצירת קובץ התקן חדש

```
>> mknod /dev/myDev c 254 0
```

- הפקודה תיצור קובץ בשם /dev/myDev , המייצג התקן תווים, המנוהל ע"י הדרייבר הרשום עם מספר ראשי 254. המספר המשני של ההתקן הוא 0.

- התקנים חדשים נוצרים כברירת מחדל עם הרשאות כתיבה ליוצר ההתקן (לרוב root) והרשאות קריאה לשאר המשתמשים.
  - במידת הצורך ניתן לשנות את הרשאות אלה לאחר יצירת ההתקן.

- ניתן להסיר התקן באופן דומה למחיקת קובץ רגיל:

```
>> rm /dev/myDev
```

הפסקה

דרייברים (מנהלי התקנים)
Device Drivers

תזכורת: file descriptors

- תזכורת: קריאת המערכת open() מקצה כניסה חדשה במקום הפנוי הראשון ב-FDT של התהליך.
- הכניסה מצביעה על אובייקט מטיפוס struct file:

```
struct file {
   ...
   loff_t  f_pos;
   ...
   void *private_data;
   ...
   struct file_operations* *f_op;
};
```

מצביע למיקום הקריאה/הכתיבה הנוכחי

שדה המאותחל ל-NULL ומאפשר לדרייבר לשמור מידע נוסף (אם הוא צריך)

זה למעשה הדרייבר

פעולות על התקנים

- מערכת ההפעלה מגדירה אוסף פעולות שניתן לבצע על קבצים באמצעות קריאות מערכת read(), write(), ...
  - בפרט זהו גם אוסף הפעולות שניתן לבצע על התקן תווים.

- כל קובץ פתוח מצביע למבנה נתונים מסוג **file_operations**, שהוא מערך של מצביעים לפונקציות המממשות את אותן הפעולות.
  - **f_op** הוא השדה המצביע למבנה זה ב-file struct.
  - מצביע NULL מייצג פונקציה לא ממומשת, או מימוש ברירת מחד

מי מממש את הפעולות הללו?

- גישה מונחית עצמים:
  - הקובץ הוא האובייקט.
  - המתודות של האובייקט מוגדרות ע"י אוסף הפונקציות ב-fops.

שלבי הפונקציה sys_read()

```
ssize_t sys_read(unsigned int fd,
                 char * buf, size_t count) {
  ...
  struct file * file = fget(fd);
  if (!file)
    return -EBADF;
  if (!(file->f_mode & FMODE_READ))
    return -EINVAL;
  if (file->f_op && file->f_op->read != NULL)
    return file->f_op->read(file, buf,
                    count, &file->f_pos);
  ...
}
```

שדות חשובים ב-file operations

- **open** – מצביע לפונקציה לפתיחת ההתקן.
  - אם מאותחל ל-NULL, פעולת open() תמיד תצליח.

```
int (*open) (struct inode *, struct file *);
```

- **release** – מצביע לפונקציה לשחרור ההתקן.
  - קריאת המערכת close() לא גוררת בהכרח קריאה ל-release(). אם ה-file object משותף (למשל, לאחר fork()), תתבצע קריאה ל-release() **רק לאחר שכל העותקים של ההתקן נסגרו.**
  - כמו במקרה של open(), ניתן לאתחל את הפונקציה ל-NULL.

```
int (*release) (struct inode *, struct file *);
```

- **flush** – מצביע לפונקציה לניקוי החוצצים וכתיבת המידע בהם ישירות להתקן.
  - מופעלת כל פעם שתהליך סוגר העתק של התקן מסוים. במידה ומאותחל ל-NULL, מערכת ההפעלה לא תבצע את הפעולה.

```
int (*flush) (struct file *);
```

שדות חשובים ב-file operations

- **read** – מצביע לפונקציה לקריאה מההתקן.
  - במידה ומאותחל ל-NULL, קריאת המערכת read תחזיר EINVAL-.

```
ssize_t (*read) (struct file *, char *,
                 size_t, loff_t *);
```

- **write** – מצביע לפונקציה לכתיבה להתקן.
  - במידה ומאותחל ל-NULL, קריאת המערכת write תחזיר EINVAL-.

```
ssize_t (*write) (struct file *, const char *,
                  size_t, loff_t *);
```

- **llseek** – מצביע לפונקציה לשינוי המיקום הנוכחי בקובץ.
  - ישפיע על פעולות read/write.
  - מחזיר את המיקום החדש בקובץ.

```
loff_t (*llseek) (struct file *, loff_t, int);
```

שדות חשובים ב-file operations

- **ioctl** – משמש להעברת פקודות ייחודיות להתקן. במידה ומאותחל ל-NULL, קריאת המערכת ioctl תחזיר EINVAL-.

```
int (*ioctl) (struct inode *, struct file *,
unsigned int cmd_id, unsigned long arg);
```

קריאת המערכת ioctl()

```
int ioctl(int fd, int cmd, …);
```

- <u>פעולה:</u> מאפשרת פקודות בקרה ייחודיות להתקן.
  - יש להשתמש באפשרות זאת רק אם לא ניתן לספק מענה הולם במסגרת הפונקציות הקיימות. לדוגמה: פקודת **eject** לכונן הדיסקים.
- <u>פרמטרים:</u>
  - fd – מתאר קובץ (שהתקבל כתוצאה של open()).
  - cmd – מספר סידורי של פקודה (יועבר כמו שהוא לפונקציה הממשת).
  - … – פרמטר אופציונלי (המהדר לא יבדוק אם הוא הועבר או לא).
    - הפרמטר האופציונלי עשוי להיות ערך שלם או מצביע. מאחר והמהדר לא מבצע כל בדיקה, הפונקציה הממשת את ioctl צריכה לוודא כי הערך שהמשתמש העביר הוא ערך חוקי המתאים להגדרת הפונקציה.
    - בדרך כלל מתייחסים לפרמטר האופציונלי כ-char *argp.
- <u>ערך חזרה:</u> תלוי בכותב הדרייבר (ערך 1- מסמן שגיאה).

מימוש דרייברים באמצעות מודולים

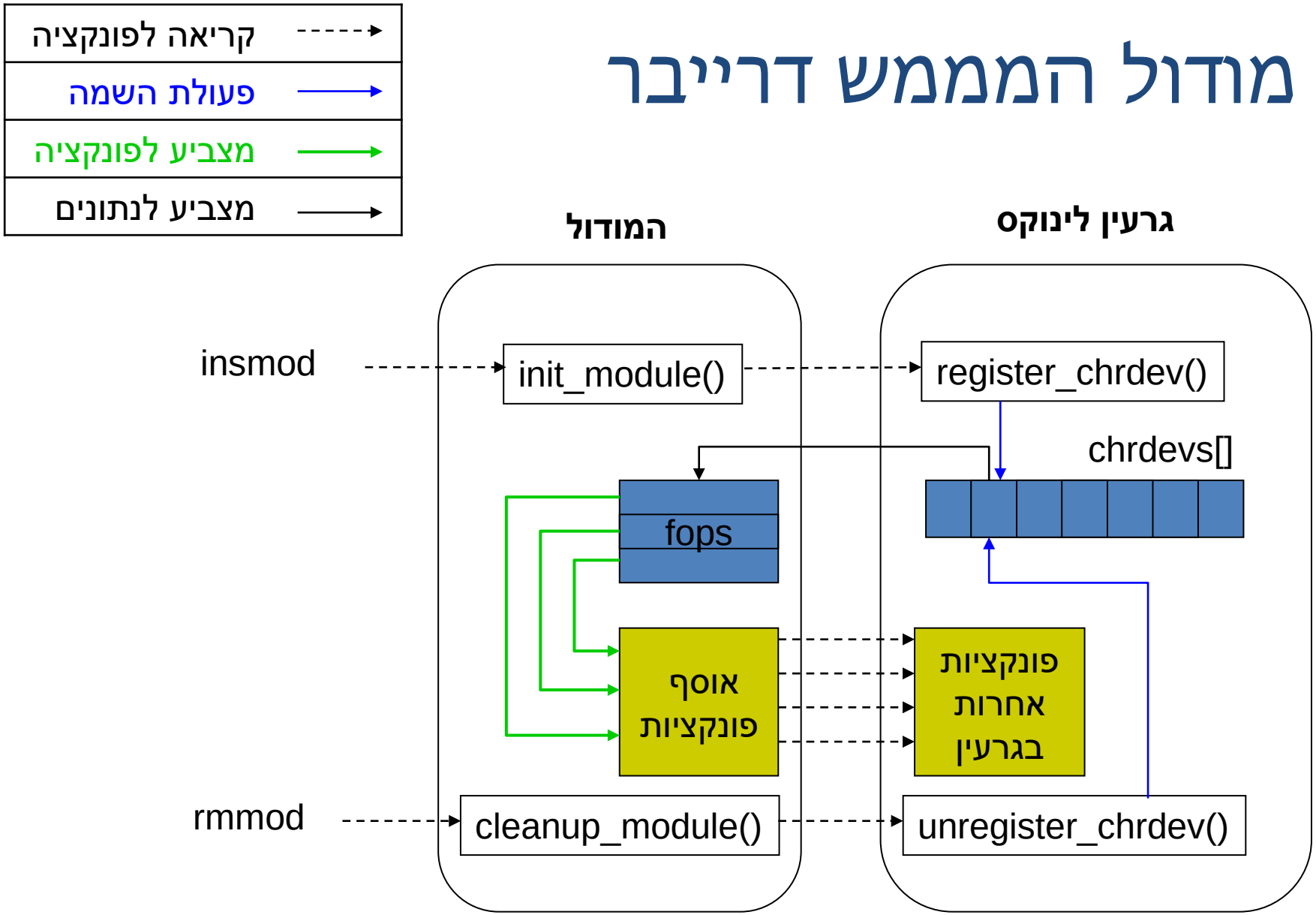
דרייברים בתור מודולים

- ניתן לבנות דרייבר בתור **מודול** (כלומר, בנפרד משאר הגרעין) ואז "לחבר" אותו בזמן ריצה בשעת הצורך.

- המודול ירשום את הדרייבר במהלך הטעינה שלו.
  - כלומר הפונקציה **init_module()** תקרא לפונקציה **register_chrdev()**.
- **שימו לב:** רישום דרייבר רק מקשר בין דרייבר לבין מספר ראשי. אין קישור ישיר בין דרייבר להתקן.

- המודול ימחק את הרישום בזמן הפריקה שלו.
  - כלומר הפונקציה **cleanup_module()** תקרא לפונקציה **unregister_chrdev()**.

מודול הממשק דרייבר



מערך הדרייברים הרשומים

	name	fops
0		
1		
2		
3		
4		
…		

- המערך []chrdevs שומר את כל הדרייברים הרשומים כרגע במערכת.
- כל תא במערך מכיל לפחות שני שדות:
  - שם הדרייבר.
  - מצביע ל-file_operations.
- האינדקס למערך הוא המספר הראשי של הדרייבר.

רישום דרייבר חדש

פונקצית גרעין, לא קריאת מערכת!

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

- פעולה: רושמת דרייבר חדש להתקני תווים ומקצה לו מספר ראשי.
- מוסיפה רישום שלו לקובץ /proc/devices .
- מוסיפה רישום שלו במערך הדרייברים chrdevs.

- פרמטרים:
  - major – המספר הראשי אותו רוצים להקצות לדרייבר.
  - name – שם הדרייבר כפי שיופיע ב-/proc/devices .
  - fops – מערך של מצביעי פונקציות, המממשים את פעולת ההתקן.
- ערך מוחזר: במקרה של הצלחה יוחזר ערך 0 או חיובי, אחרת 1-.

הקצאת מספר ראשי

- לינוקס תומך ב-512 מספרים ראשיים.
- חלק מהמספרים הראשיים מוקצים באופן סטטי להתקנים נפוצים.
- הקצאה סטטית של מספרים ראשיים יכולה להיות בעייתית אם שני התקנים שונים יבקשו אותו מספר ראשי.

- ניתן ועדיף לבצע **הקצאה דינמית** של מספר ראשי ע"י העברת ערך **0** עבור הפרמטר major.
  - מערכת ההפעלה תחפש מספר ראשי פנוי החל מ-512 ומטה.
  - המספר הראשי שנבחר יהיה ערך החזרה של register_chrdev().
- במקרה של הקצאה סטטית ערך החזרה יהיה 0.
  - ניתן לראות את המספר שהוקצה גם ב-/proc/devices .

>> cat /proc/devices

```
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  ...

Block devices:
  7 loop
  8 sd
  9 md
  ...
```

- /proc/devices הוא קובץ המפרט את כל הדרייברים הרשומים כרגע במערכת.
  - שימו לב שקיים גם מערך נוסף, של דרייברים עבור התקני בלוקים, ולכן יתכנו כפילויות של מספרי major.
- ההתקנים עצמם נשמרים בתור קבצים בתיקיה /dev , ומקושרים לדרייברים באמצעות המספר הראשי שלהם.
  - זכרו כי המשתמש יוצר קובץ התקן עם מספר ראשי מסוים באמצעות פקודת mknod.

הסרת דרייבר רשום

פונקצית גרעין, לא קריאת מערכת!

```
int unregister_chrdev(unsigned int major,
    const char *name);
```

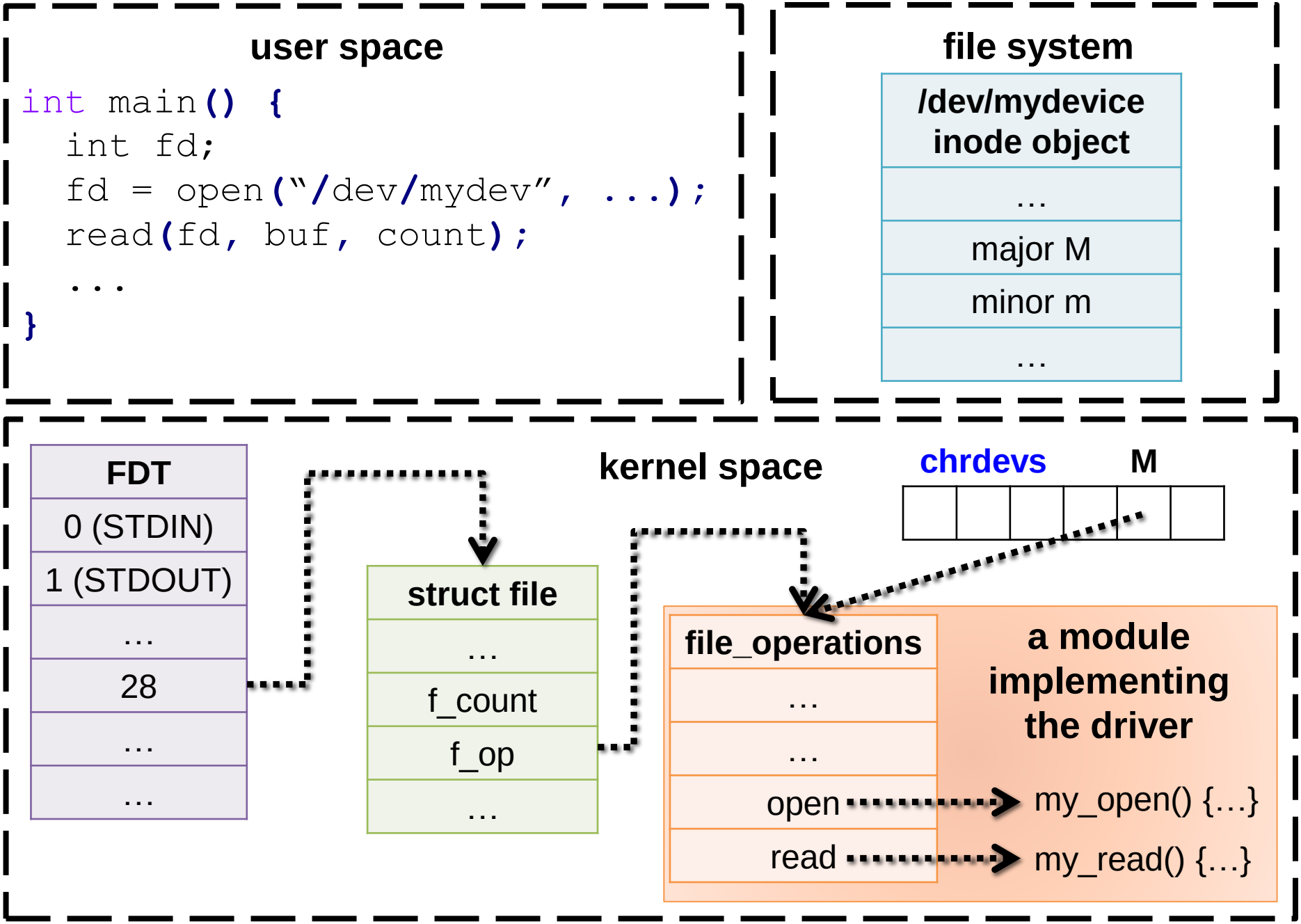
- פעולה: מסירה את הדרייבר ע"י שחרור המספר הראשי שהוקצה לו.
  - אינה מוחקת את קובץ ההתקן מ-/dev/
- פרמטרים:
  - major – המספר הראשי של הדרייבר אותו רוצים להסיר.
  - name – שם הדרייבר, כפי שמופיע ב-/proc/devices .
- ערך מוחזר: במקרה של הצלחה יוחזר ערך 0 או חיובי, אחרת 1-.

רצף השלבים בקריאת המערכת open()

- לסיכום, נלמד כיצד ממומשת קריאת המערכת open().
- במילים אחרות: מה שלבי הפונקציה sys_open() ?

- נעקוב אחרי הדוגמה הבאה:

```
int main() {
  int fd;
  fd = open("/dev/mydev", ...);
  read(fd, buf, count);
  ...
}
```


user space
int main() {
int fd;
fd = open("/dev/mydev", ...);
read(fd, buf, count);
...
}
file system
/dev/mydevice inode object
...
major M
minor m
...
kernel space
FDT
0 (STDIN)
1 (STDOUT)
...
28
...
...
struct file
...
f_count
f_op
...
chrdevs
M
file_operations
...
...
open
read
a module implementing the driver
my_open() {...}
my_read() {...}

שלבי הפונקציה sys_open()

1. ניגשת למערכת הקבצים וקוראת את המספר הראשי **M** והמשני **m** של ההתקן.
2. בודקת כי הדרייבר של ההתקן רשום: במידה ו- **chrdevs**[M]==NULL, תוחזר שגיאה ENODEV.
3. מאתחלת file object חדש (נקרא לו filp) ומקצה כניסה חדשה ב-FDT שמצביעה עליו.
4. מצביעה את filp→f_op לפונקציות של הדרייבר **chrdevs**[M]→f_op.
5. במידה והפונקציה chrdevs[M]→f_op→open() אינה NULL, קוראת לה ומעבירה את filp כפרמטר.
   - הפונקציה open() של הדרייבר תעדכן את filp→f_op כדי להגדיר התנהגות ספציפית להתקן - נראה בהמשך התרגול.
6. לבסוף, מחזירה את ה-FD (הכניסה שהוקצתה עבור הקובץ ב-FDT).

זיהוי התקן ע"י דרייבר

- שימו לב לחתימה של open() :

```
int (*open)(struct inode *, struct file *);
```

- הפונקציה מקבלת כפרמטר את ה-inode המייצג את ההתקן.
- כל קובץ פתוח מיוצג ע"י מבנה נתונים בשם **inode** (פרטים נוספים בתרגול 13). בפרט, גם להתקן תווים יש inode המייצג אותו.
- השדה i_rdev ב-inode מכיל את **המספר הראשי והמשני** של ההתקן.
  - המאקרו **MAJOR(inode->i_rdev)** מחזיר את המספר הראשי.
  - המאקרו **MINOR(inode->i_rdev)** מחזיר את המספר המשני.

דוגמת קוד – מודול הממשק דרייבר (1)

```
#include "linux/module.h"

int major = 0;  /* will hold the driver major number */

struct file_operations my_fops = {
    .open=    my_open,
    .release=    my_release,
    .read=    my_read,
    .write=    my_write,
    .ioctl=    my_ioctl,
};

struct file_operations my_fops2 = {
    .open=    my_open,
    .release=    my_release2,
    .read=    my_read2,
    .write=    my_write2,
    .ioctl=    my_ioctl,
};
```

דוגמת קוד – מודול הממשק דרייבר (2)

```
int init_module( void ) {
  major = register_chrdev(major, “my_module”, &my_fops);

  if( major < 0 ) {
    printk(KERN_WARNING “Bad dynamic major\n");
    return major;
  }
  //do_init();
  return 0;
}

void cleanup_module( void ) {
    unregister_chrdev(major, “my_module”);
    //do clean_up();
}
```

דוגמת קוד – מודול הממשק דרייבר (3)

```
int my_open(struct inode *inode, struct file *filp ) {
    filp->private_data = allocate_private_data();
    if( filp->f_mode & FMODE_READ )
        // handle read opening
    if( filp->f_mode & FMODE_WRITE )
        // handle write opening

    if (MINOR( inode->i_rdev )==2)
        filp->f_op = &my_fops2;
    return 0;
}
```

my_open() מחליפה את fops כתלות במספר המינורי.
שימו לב: fops2 לא רשום בתור דרייבר במערך chrdevs,
אבל הוא עדיין נגיש מתוך קוד המודול.

דוגמת קוד – מודול הממשק דרייבר (4)

```
ssize_t my_read(struct file
*filp, char *buf, size_t count,
loff_t *f_pos) {
  // custom implementation 1
}

ssize_t my_write(struct file
*filp, const char *buf, size_t
count, loff_t *f_pos) {
  // custom implementation 1
}

int my_release(struct inode
*inode, struct file *filp) {
  // custom implementation 1
}
```

```
ssize_t my_read2(struct file
*filp, char *buf, size_t count,
loff_t *f_pos) {
  // custom implementation 2
}

ssize_t my_write2(struct file
*filp, const char *buf, size_t
count, loff_t *f_pos) {
  // custom implementation 2
}

int my_release2(struct inode
*inode, struct file *filp) {
  // custom implementation 2
}
```

דוגמת קוד – מודול הממשק דרייבר (5)

```
int my_ioctl(struct inode *inode,
    struct file *filp,
    unsigned int cmd,
    unsigned long arg) {

    switch( cmd ) {
        case MY_OP1:
            //handle op1;
            break;

        case MY_OP2:
            //handle op2;
            break;

        default:
        return -ENOTTY;
    }
    return 0;
}
```