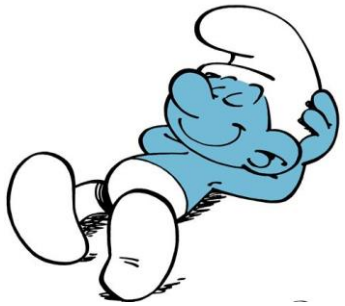
Operating Systems

Recitation 10

Plan

- Short intro on kernel programming
- Device drivers (Linux)
  - Kernel modules
  - Hello world
  - Character device

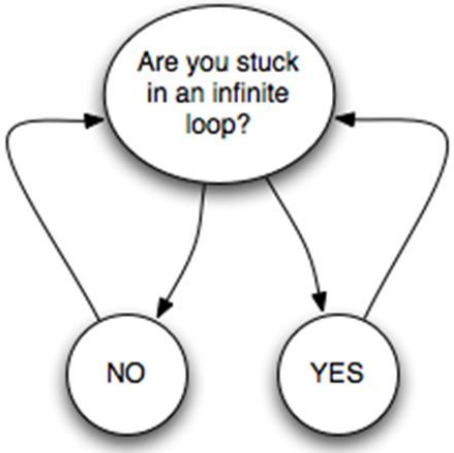
Life is good in “Userland”



- Process context
- Use standard C library (glibc) calls
  - `<stdlib.h>,<math.h>,<stdio.h>,<string.h>,…`
  - various user-level operations translated to system calls
  - “magically” get service from OS
- Can’t touch system memory
  - Because of CPU modes, segments,…

Kernel “wonderland”

- Can’t use standard libraries!
  - C standard library implemented in User space
  - Library implementations may vary and change regardless of kernel
  - Functions may call kernel functions - may enter infinite loop



Kernel programming

- Kernel provides various functions of its own
  - NO - `malloc(), printf()`
  - YES - `vmalloc/kmalloc(), printk()`
  - Some implementations of user space libraries. Found in include/linux (example linux/string.h)
- Bugs are critical → potential crash of entire system (kernel panic) ☹
  - [In user space we have kernel to protect us]

Access user space memory

- Even though we run in kernel mode, it is highly recommended not to directly access user space addresses
  - What if user address illegal?
- Instead use special functions
  - `put_user(x, p)`
  - `get_user(x, p)`

Device drivers

- **Hide** hardware details
  - Provide well-defined interface for kernel/user
  - Manages hardware
- Impractical to have all possible drivers
  - building entire kernel every time not sensible too
- Result: most drivers are **loadable** modules
  - Loaded after kernel boots
- **Modular** == can be loaded dynamically during runtime!
- Only loaded by privileged users

Linux Device Drivers

- Character device
  - Abstraction for stream of bytes, read and written directly ***without buffering***
  - Example: serial ports, monitor, keyboard
- Block device
  - Read and written in multiples of block, **random access**
  - Example: disk, cdrom
  - /dev/sdc
- Network device
  - Abstraction for data packets. Specially handled in Linux networking stack
  - Example: ethernet card

Abstractions

- Abstraction and interface for character and block devices
  - One device file = one device
  - “Drive device” = access file: open, close, read, write, lseek, etc.
- In order to talk to the kernel, driver registers with subsystems to respond to “events”
  - opening of a file, a page fault, the plugging in of a new USB device, etc.
  - Example soon, but first…

Hello world!

```
/* module has access to kernel data structures. Some parts
available only with this macro */
#define __KERNEL__
#define MODULE
#include <linux/module.h> // included for all kernel modules
#include <linux/init.h>   // included for __init and __exit macros
 // loader
static int __init hello_init(void)
{
    printk("Hello world!\n");
    return 0;    // 0 == success
}
```

Hello world!

```
// unloader
static void __exit hello_cleanup(void)
{
    printk("Cleaning up module.\n");
}

module_init(hello_init);
module_exit(hello_cleanup);
```

Compile & load

- Makefile

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

- **Load**: sudo insmod hello.ko
- **Verify**: sudo lsmod
- **Unload**: sudo rmmod hello
- Can view printouts in kernel log using *dmesg*

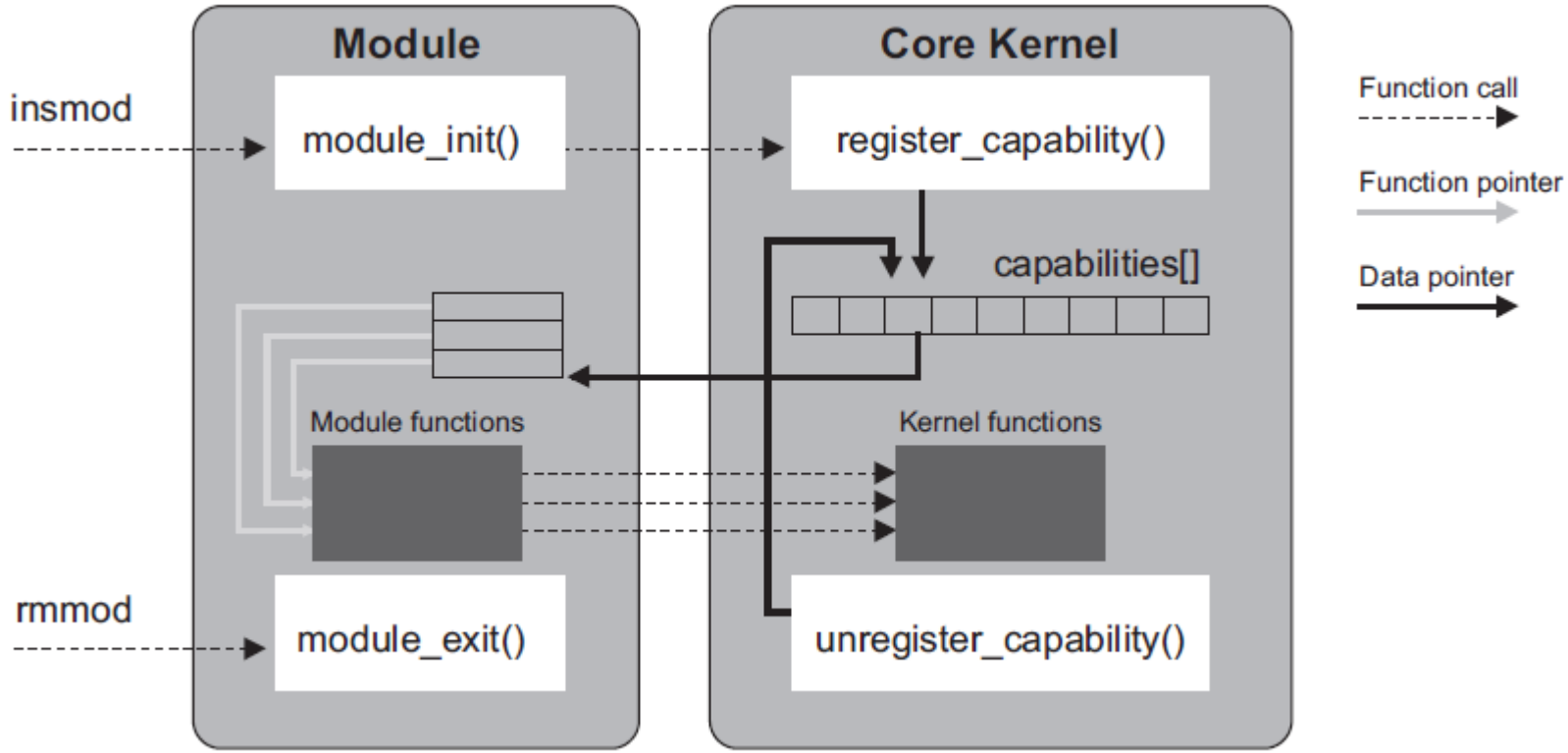
Hello world!

- ***Code example***

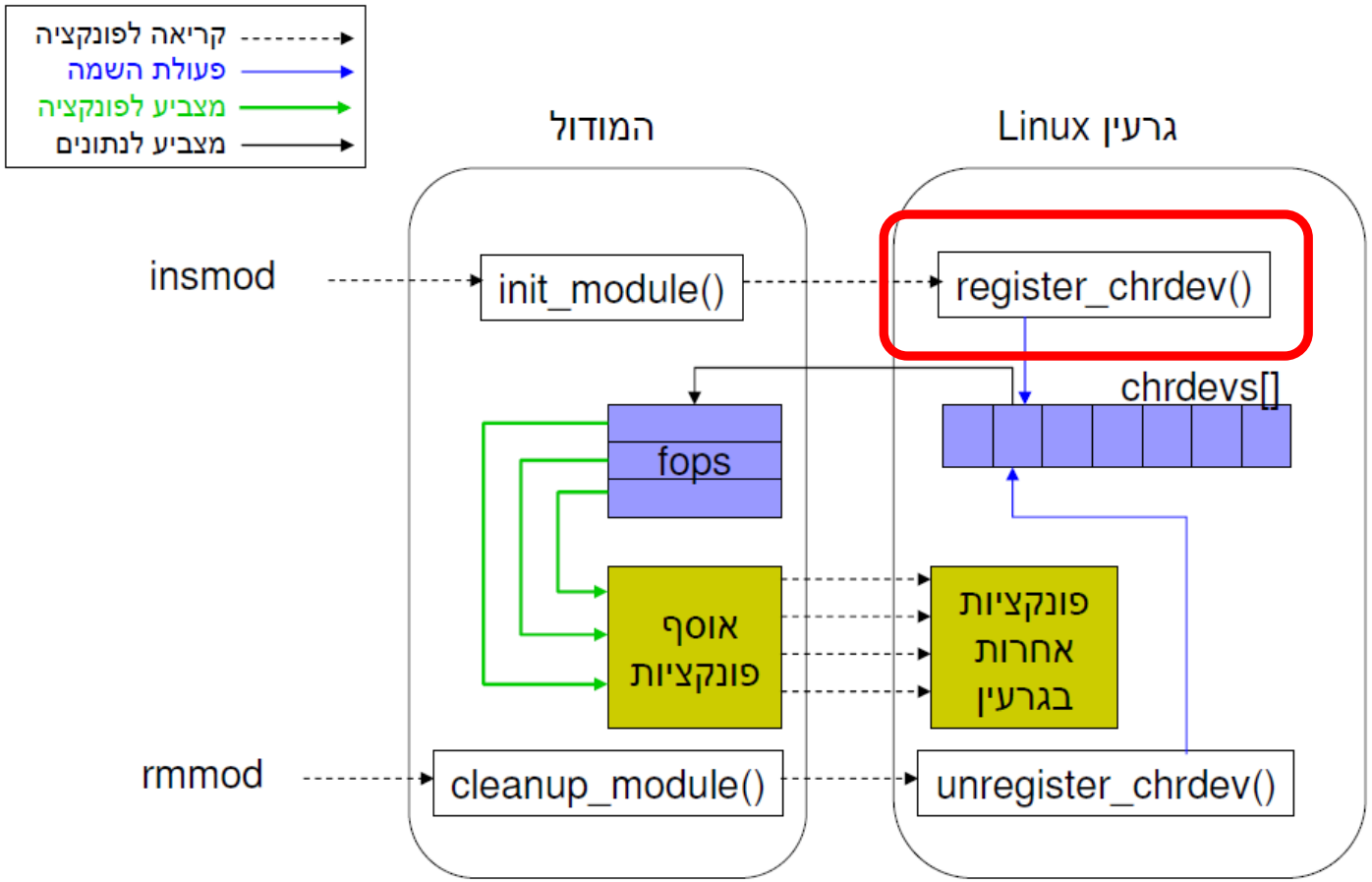
Role of a Module

- Extension of the kernel
- `module_init()` “registers” module’s “capabilities”.
  - calls `register_capability()` function with pointer to a structure containing pointers to functions within the module.
  - `register_capability()` function puts pointer into internal “capabilities” data structure.
- System defines how applications get access to information in capabilities data structure (by using system calls)

Module


Module
Core Kernel
insmod
module_init()
register_capability()
capabilities[]
Module functions
Kernel functions
rmmod
module_exit()
unregister_capability()
Function call
Function pointer
Data pointer

Character device driver



* Note - Some changes made since kernel 2.6. But old way still applies

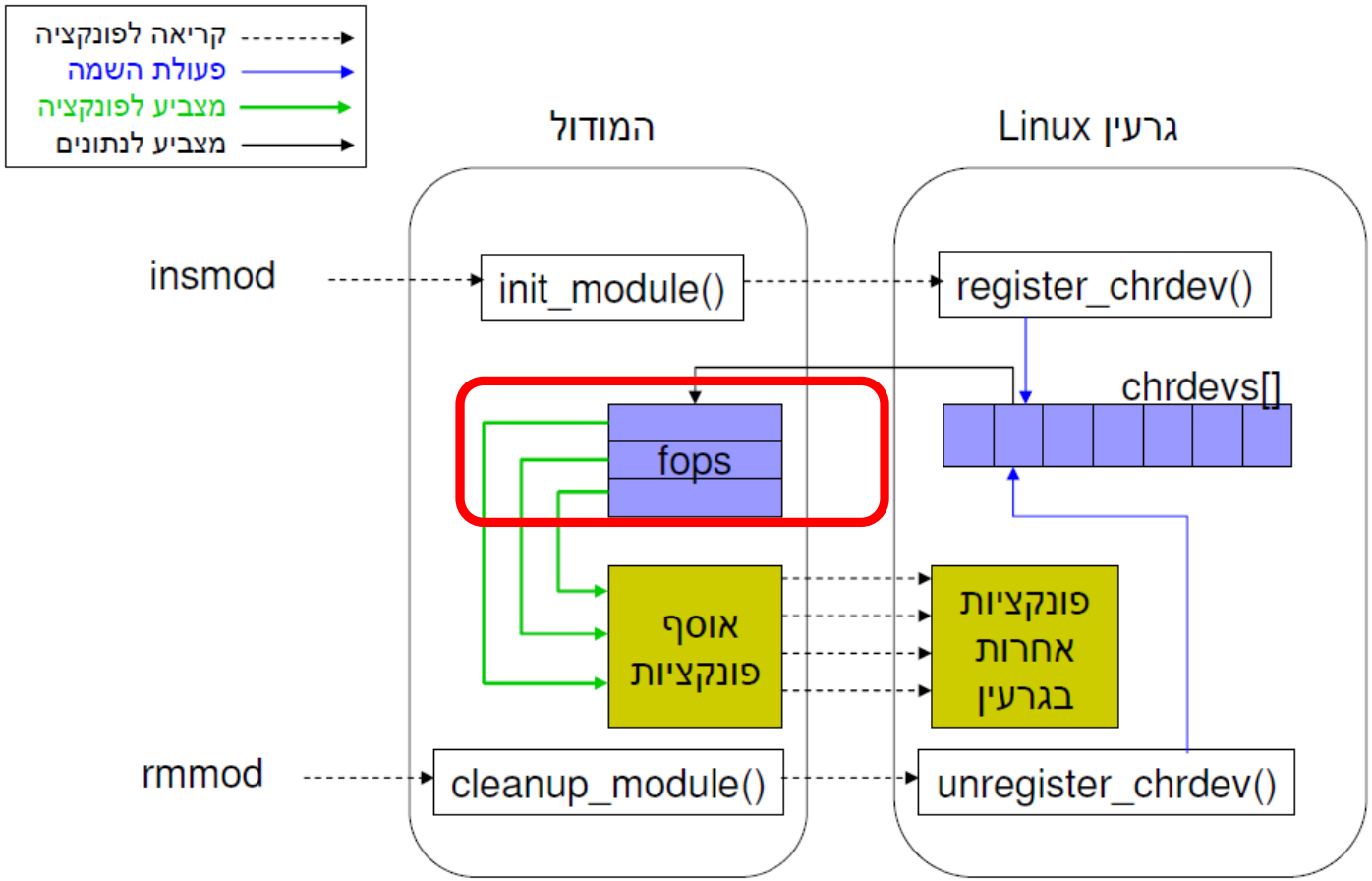
Character device driver

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

- major – desired major number. 0 for dynamic allocation
- name – device name (/proc/devices)
- fops – driver fops struct

- Returns – 0 on success, -1 otherwise
  (**or dynamic major num…**)

Character device driver



Operations on device

- Character device = special file
- OS defines which file operations can be performed on it
  - Invoked by system calls
- `struct file_operations` – pointers to functions
  - Instance usually named `fops`
  - Can use `NULL` for functions not implemented
  - Defined in `<linux/fs.h>`

Operations on device

```
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
ssize_t (*read) (struct file *, char *,
                 size_t, loff_t *);
ssize_t (*write) (struct file *, const char
*,
                  size_t, loff_t *);
loff_t (*llseek) (struct file *, loff_t,
int);
int (*ioctl) (struct inode *, struct file *,
              unsigned int, unsigned long);
```

file_operations
open()/release()

- On open() the Linux operating system:
  - Initializes struct file, internally represents open device/file
  - Calls open from file_operations (if exists)
- Important fields in struct file:
  - `mode_t f_mode; /* (FMODE_READ, FMODE_WRITE) */`
  - `loff_t f_pos;`
  - `unsigned int f_flags; /* (O_RDONLY, O_NONBLOCK,…, */`
  - `struct file_operations *f_op;`
  - `void *private_data;`

file_operations
`open()/release()`

- Device driver `open()` responsible for
  - Verifying device still working
  - any driver-specific initialization (if opened for the first time)
- `release()` called on close or exit
  - Release any data specifically allocated by open

file_operations
read()/write()

```
ssize_t read(struct file *filp, char *buff,
             size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char
*buff,
              size_t count, loff_t *offp);
```

- filp - representing open file
- buff - buffer in user space
- count - bytes to read/write
- offp - offset in file

- Returns – number of bytes read/written.
  - offp needs to be updated accordingly…

file_operations `ioctl()`

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

- For special commands not covered by existing functions (read, write,…)
  - Considered bad programming to use it otherwise
- inode – device inode file
- filp – open device file
- cmd – command number
- arg – optional general purpose
- Returns – depends on implementation, <0 on error

file_operations `ioctl()`

```
int ioctl(int fd,int cmd,…)
```

- User space equivalent
- fd – device file descriptor
- cmd - command number
  - Usually from driver's header file…
- … - optional arguments (driver specific)
- return – depends on implementation, <0 on error

Creating a new device

`mknod <name> <type> <major> (minor)`

- Device driver not enough!
  - It's only code
- Need to have a device file to be driven
- mknod creates the device file + conenction
  - name – new device file name
  - type – c (char), b (block)
  - major – to determine the driver
  - minor – between 0 and 255
  - `sudo mknod /dev/simple_char_dev c 250 0`
  - make sure we have permissions..
- Can remove file with rm

Character device

- ***Code example***