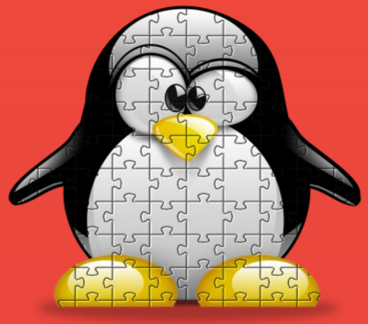
ANITA B.ORG 2023

GRACE HOPPER CELEBRATION

THE WAY FORWARD

LINUX KERNEL HACKING WORKSHOP



Tuba Yavuz
Associate Professor
ECE Department
FICS & Nelms Institute
University of Florida
tuba@ece.ufl.edu

Outline

- **What is this workshop about?**
- What is a device driver & how do drivers work in the Linux Kernel?
- Which system APIs get involved?
- Hands-on Activity: Writing & Testing a Character-special Device Driver
- Wrap-up
- Questions & Answers

Why the Linux Kernel?

- It is very popular; powering a wide range of systems from mobile phones, routers, and edge devices to personal computers, high-end servers, and the cloud
- It is open-source & one of the most complex piece of software on earth
- It has a very active community

Why hacking the Linux Kernel?

- Hacking can be performed for a good or a bad cause
- **Ethical hacking (the good)** is about finding vulnerabilities in a system and responsibly disclosing them to the developers & the vendors.
  - Changing the functionality of a system by adding new components.
- Unethical hacking (the bad) is about finding vulnerabilities in a system and exploiting them to inflict harm on the users of that system.

Importance of system programming

- Every single computing platform relies on some system code
  - (real-time) operating system/kernel, libraries, etc.
- Testing system code is more challenging than applications
- Learning system APIs and their side-effects takes time
- Vulnerabilities & bugs in system code may have a high cost
- Computer science and engineering curriculums could be improved to provide more exposure to system programming

Research and Educational Interest

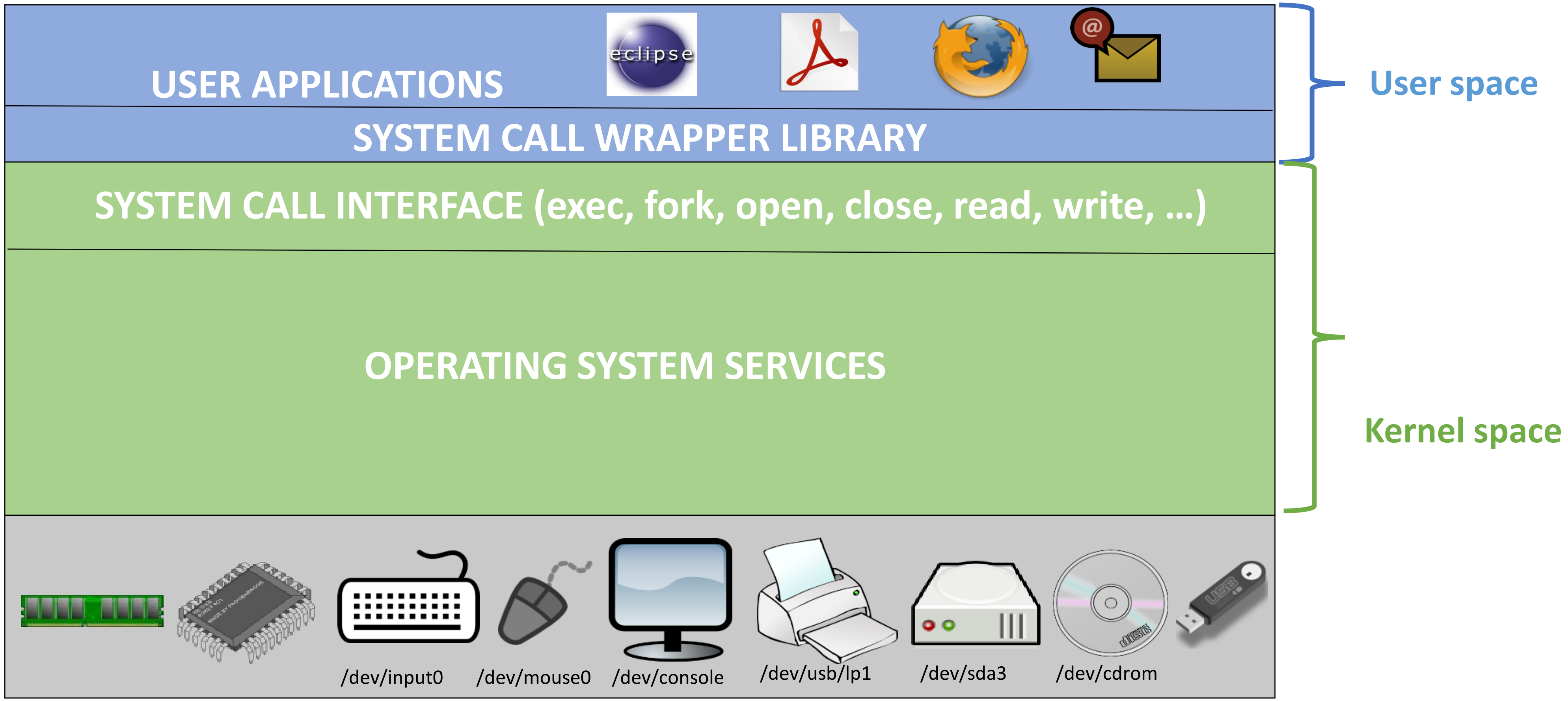
- I received my Ph.D. in computer science from the University of California, Santa Barbara in 2004.
- I work in the intersection of formal methods, program analysis, and system security.
- I am an Associate Professor & the director of the System Reliability Lab at the University of Florida.
- My long-term career goal is to develop scalable techniques for detecting reliability and security issues in real-world system code & use these techniques in developing a workforce empowered by system programming and/or system analysis skills.
- As an ethical hacker and with the help of my research tools, I was able to detect vulnerabilities in the Linux kernel and in other systems code.
- I hope this workshop can provide some inspiration about learning more about system programming and getting involved in system development and/or analysis.

Outline

- What is this workshop about?
- **What is a device driver & how do drivers work in the Linux Kernel?**
- Which system APIs get involved?
- Hands-on Activity: Writing & Testing a Character-special Device Driver
- Wrap-up
- Questions & Answers

What is an Operating System (OS)/Kernel?

How do User Procs. & Kernel communicate ?



What is a device driver?

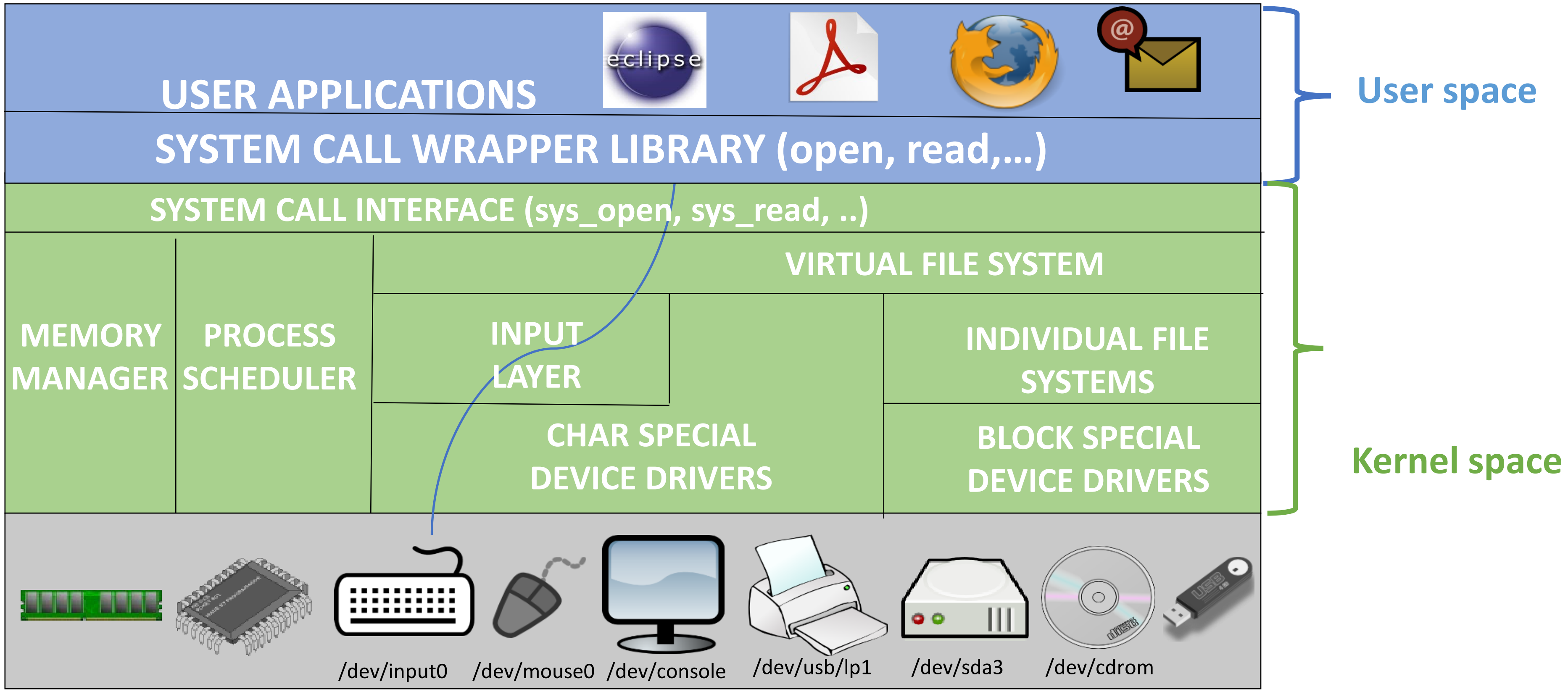
- A device driver is a piece of software that includes functionality to initialize, configure, and perform Input/Output with a device or a class of devices.
- Device drivers typically form one of the lower-level software layer within the operating system or the kernel.
- Some device drivers get statically linked with the kernel and others get loaded at runtime when the device gets connected to the host device the first time.

What is a device in the Linux Kernel?

- A device is a special file in the Linux Kernel
- Like regular files, they appear in the file system hierarchy
- Unlike regular files, they do not store data on the file system
  - The data flows into and/or from the actual device
- The device driver is responsible for communicating with the device
  - It receives the data from the device
  - It sends data to the device

Device Drivers in the Linux Kernel

Device Drivers & Subsystems in the Linux Kernel



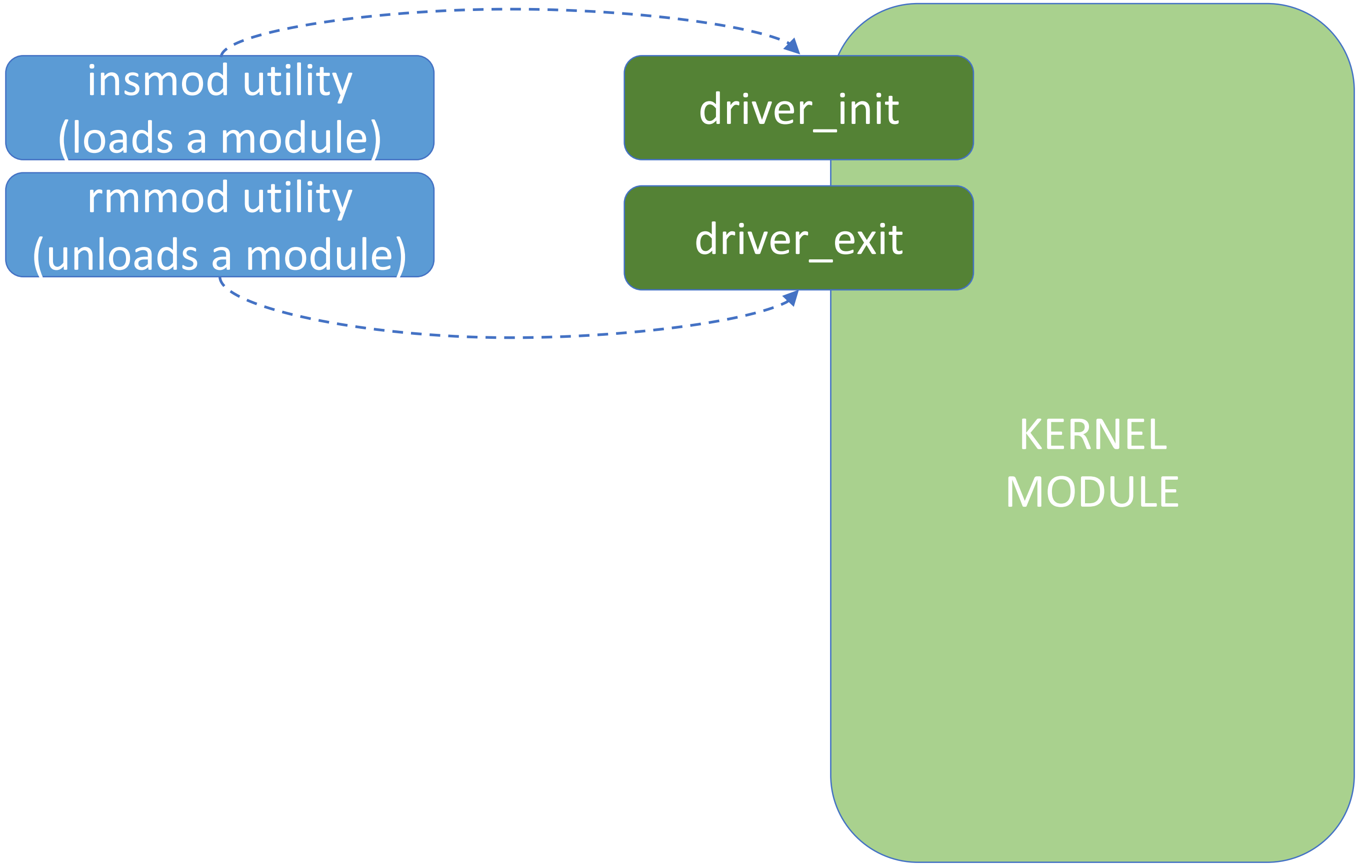
Outline

- What is this workshop about?
- What is a device driver & how do drivers work in the Linux Kernel?
- **Which system APIs get involved?**
- Hands-on Activity: Writing & Testing a Character-special Device Driver
- Wrap-up
- Questions & Answers

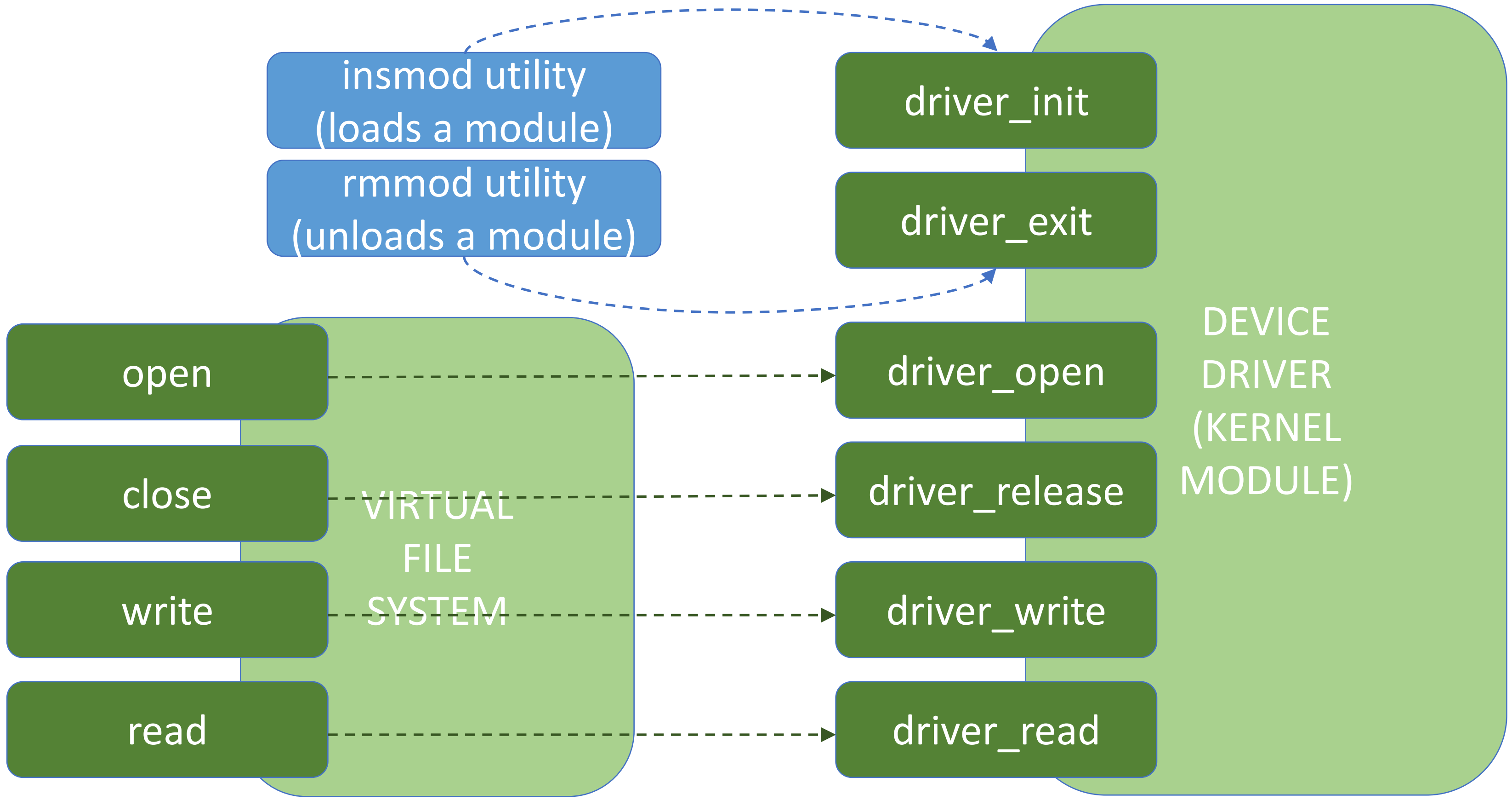
Writing a device driver

- A device driver is a **kernel module** with well defined entry points
- An init function that gets executed at module load time
- An exit function that gets executed at module unload time
- Other functions that serve as entry point to some kernel layer
- Uses kernel API to allocate memory, to print to kernel logs, to register data structures, and so on.

Entry points of a kernel module



Entry points of a device driver



APIs used in a character-special device driver

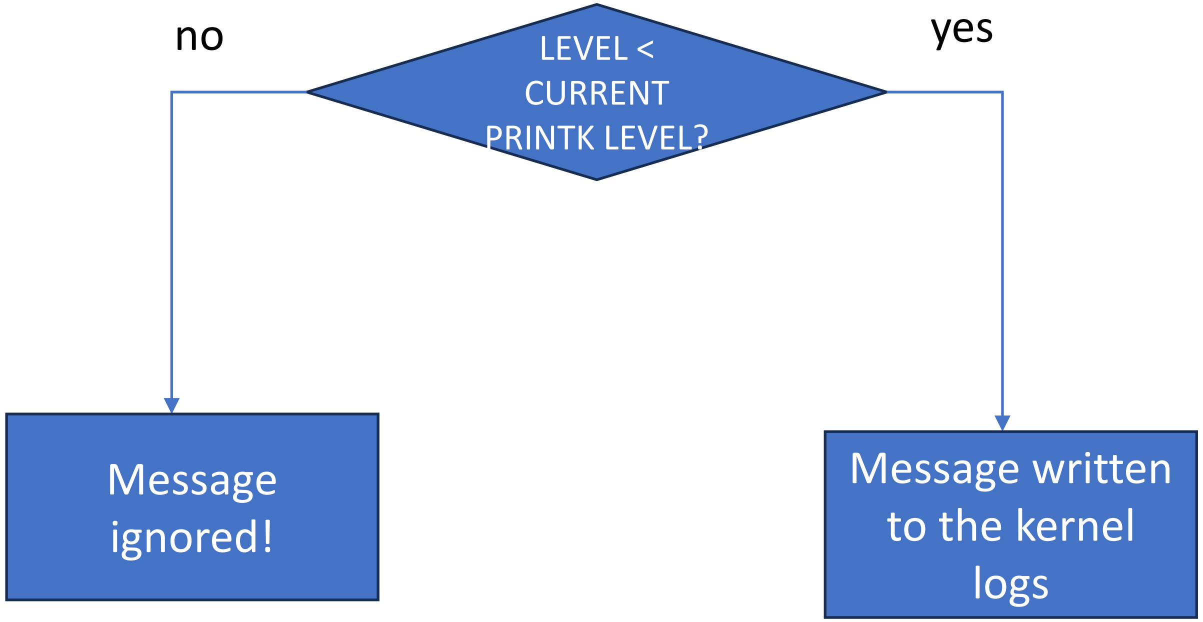
- Printing messages to the kernel logs
- Allocating dynamic memory in the kernel
- Copy data to & from user space
- VFS data structures
- Reserving device (major & minor) numbers
- Creating device nodes
- Registering a character-special device

Printing messages to the kernel logs

- In user space we may use printf to display messages on the terminal
- In kernel space we use printk to write messages to the kernel logs (NOT to the terminal!)
- We can check what is in the kernel logs from user space using the dmesg command (-T option to pretty print the time info):
  - $ **dmesg -T**

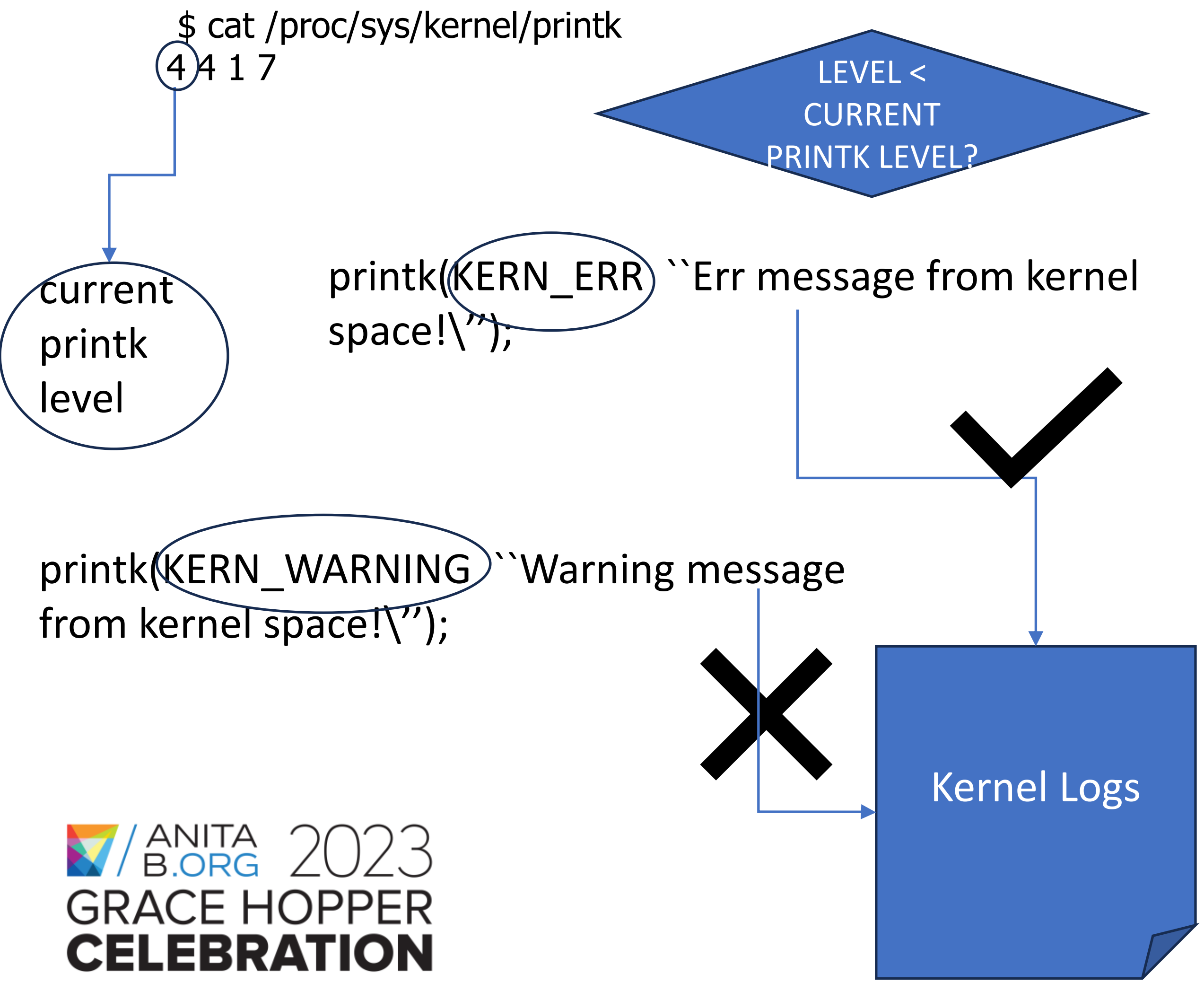
How printk works

- printk(LEVEL Message);



Level	Name	Description
0	KERN_EMERG	An emergency condition; the system is probably dead
1	KERN_ALERT	A problem that requires immediate attention
2	KERN_CRIT	A critical condition
3	KERN_ERR	An error
4	KERN_WARNING	A warning (default log level, if not specified)
5	KERN_NOTICE	A normal, but perhaps noteworthy, condition
6	KERN_INFO	An informational message
7	KERN_DEBUG	A debug message typically superfluous

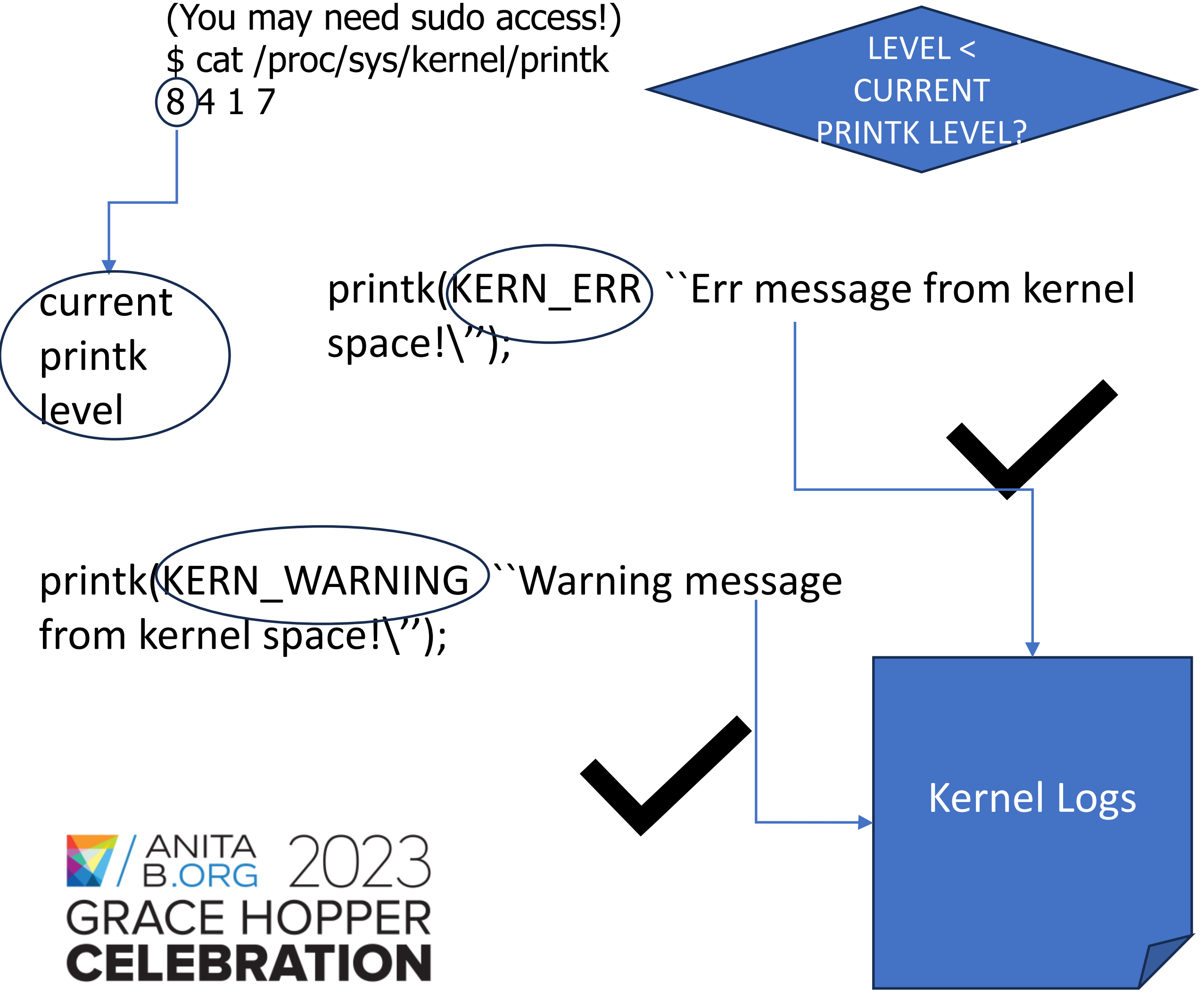
How to check the current printk level



Level	Name	Description
0	KERN_EMERG	An emergency condition; the system is probably dead
1	KERN_ALERT	A problem that requires immediate attention
2	KERN_CRIT	A critical condition
3	KERN_ERR	An error
4	KERN_WARNING	A warning (default log level, if not specified)
5	KERN_NOTICE	A normal, but perhaps noteworthy, condition
6	KERN_INFO	An informational message
7	KERN_DEBUG	A debug message typically superfluous

How to change the current printk level

```
$ echo 8 > /proc/sys/kernel/printk
(You may need sudo access!)
$ cat /proc/sys/kernel/printk
8 4 1 7
```



Level	Name	Description
0	KERN_EMERG	An emergency condition; the system is probably dead
1	KERN_ALERT	A problem that requires immediate attention
2	KERN_CRIT	A critical condition
3	KERN_ERR	An error
4	KERN_WARNING	A warning (default log level, if not specified)
5	KERN_NOTICE	A normal, but perhaps noteworthy, condition
6	KERN_INFO	An informational message
7	KERN_DEBUG	A debug message typically superfluous

Allocating dynamic memory in the kernel

- In kernel space, we can use kmalloc to allocate dynamic memory.
- Similar to malloc, the first argument specifies the size in bytes
- Unlike malloc, kmalloc has a 2nd argument to specify the context it is executed in. For our activity, we will use GFP_KERNEL.
- Example: char *buf = kmalloc(100, GFP_KERNEL);
  - size = 100 bytes
  - GFP_KERNEL means if needed the current process can be put to sleep until memory becomes available
- The allocated memory can be accessed by the kernel only & is physically contiguous.

Copy data to & from user space

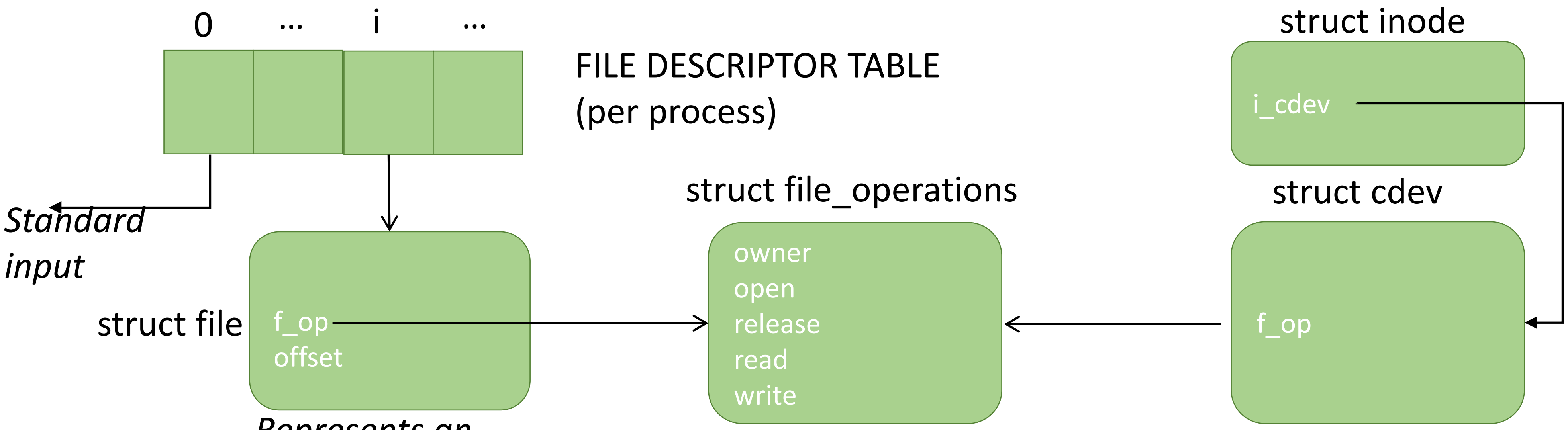
- Kernel code can copy data to & from user space buffers
- Since we cannot trust addresses provided from user space when a system call is issued, we need help from the kernel to check if it is safe to use such addresses.
- To safely copy data from kernel space to user space:
  - unsigned long copy_to_user (void __user * *to*, const void * *from*, unsigned long *n*);
  - return value: 0 on success
- To safely copy data from user space to kernel space:
  - unsigned long copy_from_user (void * to, const void __user * from, unsigned long n);
  - return value: 0 on success

Virtual File System (VFS) data structures

User process opens a device to do I/O

`Int tux_filedesc = open("/dev/tux", ...);`
`read(tux_filedesc , ...); or write(tux_filedesc , ...);`

A file descriptor no is a handle to a file/device. Once a device is opened, we can use it to read/write the file/device.



Represents a file in the file system and stores metadata about the file

Represents an open file within a user process

Represents file specific implementation of the file operations

Represents a character-special device

Major & minor numbers

- The kernel uniquely identifies a device using a combination of the major and minor numbers
  - The major number represents the device driver
  - The minor number represents the device supported by a device driver
  - dev_t devno = MKDEV(major, minor)
  - MAJOR(devno), MINOR(devno)

Reserving device (major & minor) numbers

int register_chrdev_region(dev_t first, unsigned int count, char *name)

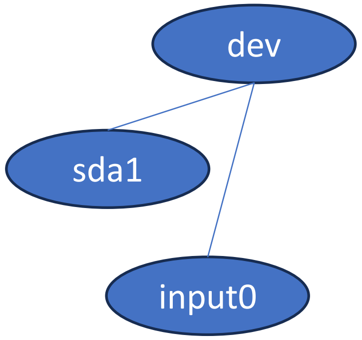
- first: the first device no that's registered
- count: number of device no's registered
- name: device name

- Example
  - major = 500, minor = 0, count = 2
  - register_chrdev_region(MKDEV(500, 0), 2, "tuxdriver");
  - if successul, two device nos are registered for driver tuxdriver:
    - MKDEV(500, 0)
    - MKDEV(500,1)

Creating device nodes

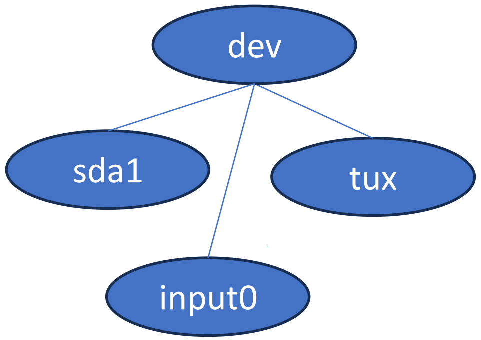
- Once we know the device number(s) to use, we can create the device nodes on the file system
- To create device nodes from user space:

Before



	device node name	character-special device	major number	minor number
$ mknod	/dev/tux	c	500	0

After



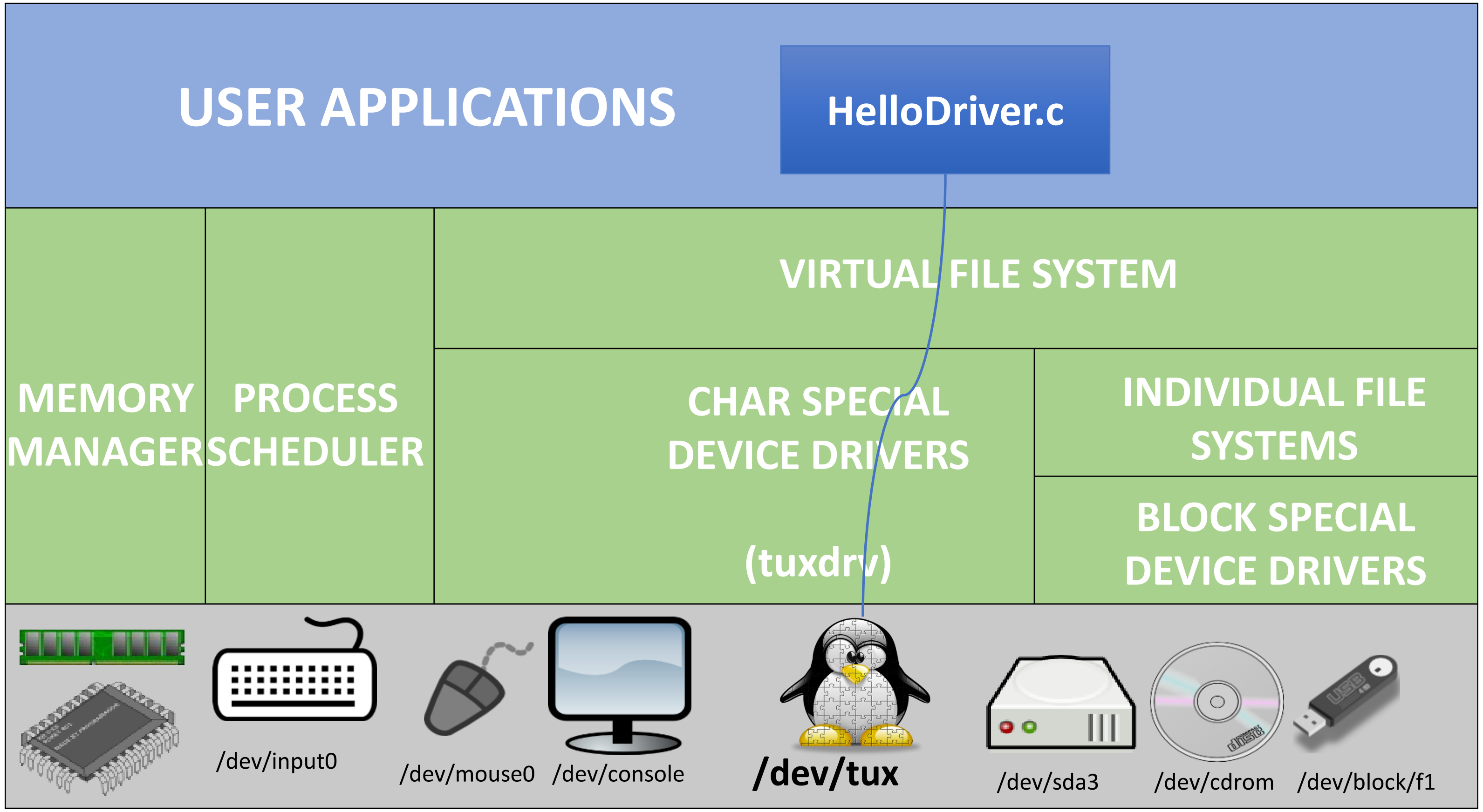
Registering a character-special device

- First a cdev data structure needs to be created and initialized:
  - cdev_alloc(): returns a pointer to struct cdev
- Then cdev must be initialized to point to the file operations:
  - cdev_init(struct cdev *, struct file_operations *);
- To register it with the kernel, we also need the device number:
  - cdev_add(struct cdev *, dev_t first, int count);
- When we are done, we will recycle it:
  - cdev_del(struct cdev *);

Outline

- What is this workshop about?
- What is a device driver & how do drivers work in the Linux Kernel?
- Which system APIs get involved?
- **Hands-on Activity: Writing & Testing a Character-special Device Driver**
- Wrap-up
- Questions & Answers

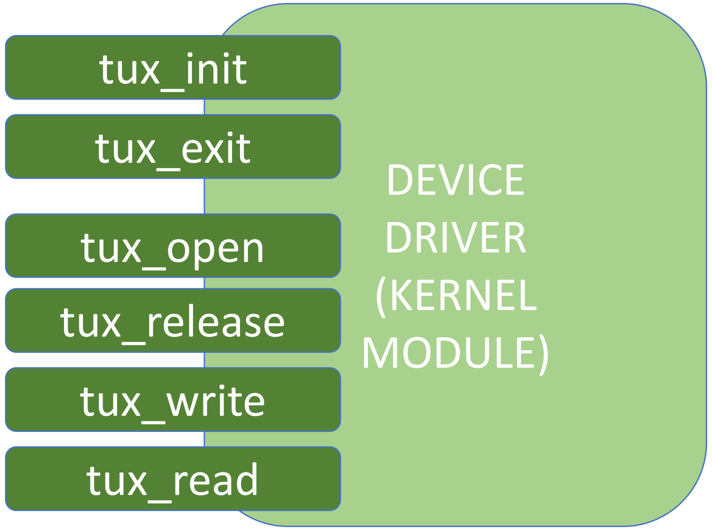
This is what we are going to do...



Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 0.a

- Install VirtualBox from https://www.virtualbox.org/
- Create a virtual machine (VM) Ubuntu instance.
- You will need to download the .iso file for an Ubuntu version (latest one is recommended) on to your host machine.
- You can find the iso files from https://ubuntu.com/download/desktop .When you try to run the virtual machine instance for the first time, you will be asked for the .iso file for installing Ubuntu.

Hands-on Activity, Step 0.b

Once you have the VM instance ready, install the following software on your VM if you do not already have the make & gcc:

a. make (sudo apt install make)
b. gcc (sudo apt install gcc)

Hands-on Activity, Step 0.c

You will need to use sudo when executing most of the commands, e.g., sudo command …

If you do not have sudo access on your VM you might instead use su to enter the root mode once and execute all commands without worrying about using sudo:

$ su
#root:user> command …

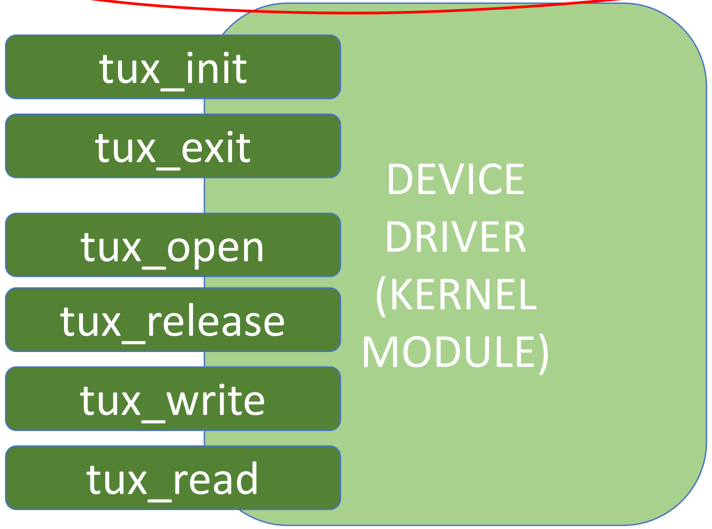
Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 1: Implement the device driver



Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 1

- On your VM, create a new directory on your file system and let APATH denote the full path to this directory.
- Create tuxdriver.c under APATH using your favorite editor
  - Feel free to customize the printk messages

Linux header files to include

```
#include <linux/module.h>   /* for modules */
#include <linux/fs.h>       /* file_operations */
#include <linux/uaccess.h>  /* copy_(to,from)_user */
#include <linux/init.h>     /* module_init, module_exit */
#include <linux/slab.h>     /* kmalloc */
#include <linux/cdev.h>     /* cdev utilities */
```

Constant & Global Variable Declarations

```
#define TUXDEV_NAME "tux"
#define ramdisk_size (size_t)(16)
static char *ramdisk; static dev_t first;
static unsigned int count = 1;
static int tux_major = 500, tux_minor = 0;
static struct cdev *tux_cdev;
MODULE_LICENSE("GPL v2");
```

Initialization of File Operations

```
static int tux_open(struct inode *inode, struct file *file);
static int tux_release(struct inode *inode, struct file *file);
static ssize_t tux_read(struct file *file, char __user *buf, size_t lbuf, loff_t *ppos);
static ssize_t tux_write(struct file *file, const char __user *buf, size_t lbuf, loff_t *ppos);
static const struct file_operations tux_fops = {
	.owner = THIS_MODULE,
	.read = tux_read,
	.write = tux_write,
	.open = tux_open,
	.release = tux_release,
};
```

What tuxdriver does at load time..

```
static int __init tux_init(void) {
   ramdisk = kmalloc(ramdisk_size, GFP_KERNEL);
   first = MKDEV(tux_major, tux_minor);
   register_chrdev_region(first, count, "tuxdriver");
   tux_cdev = cdev_alloc();
   cdev_init(tux_cdev, &tux_fops);
   cdev_add(tux_cdev, first, count);
   printk(KERN_INFO "Succeeded in registering tux cdev using major no %d and minor no %d\n",
                              tux_major, tux_minor);
   return 0;
}
module_init(tux_init);
```

What tuxdriver does at unload time..

```
static void __exit tux_exit(void)
{
        cdev_del(tux_cdev);
        unregister_chrdev_region(first, count);
        printk(KERN_INFO "\ntux unregistered\n");
        kfree(ramdisk);
}
module_exit(tux_exit);
```

What tuxdriver does on opening/closing a tux dev

```
static int tux_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO " OPENING device: %s:\n\n", TUXDEV_NAME);
        return 0;
}

static int tux_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO " CLOSING device: %s:\n\n", TUXDEV_NAME);
        return 0;
}
```

What tuxdriver does upon writing on a tux dev

```
static ssize_t tux_write(struct file *file, const char __user * buf, size_t lbuf, loff_t * ppos) {
    int nbytes;
    if ((lbuf + *ppos) > ramdisk_size) {
        printk(KERN_INFO "trying to write past end of device, aborting because this is just a stub!\n");
        return 0;
    }
    nbytes = lbuf - copy_from_user(ramdisk + *ppos, buf, lbuf);
    *ppos += nbytes;
    printk(KERN_INFO "\n WRITING tux, nbytes=%d, pos=%d\n", nbytes, (int)*ppos);
    return nbytes;
}
```

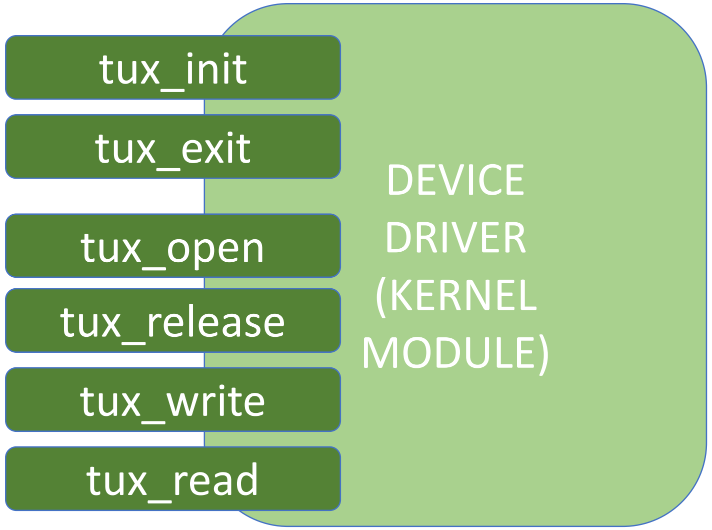
What tuxdriver does upon reading from a tux dev

```
static ssize_t tux_read(struct file *file, char __user * buf, size_t lbuf, loff_t * ppos) {
   int nbytes;
    if ((lbuf + *ppos) > ramdisk_size) {
       printk(KERN_INFO "trying to read past end of device, aborting because this is just a stub!\n");
       return 0;
    }
    nbytes = lbuf - copy_to_user(buf, ramdisk + *ppos, lbuf);
    *ppos += nbytes;
    printk(KERN_INFO "\n READING from tux, nbytes=%d, pos=%d\n", nbytes, (int)*ppos);
    return nbytes;
}
```

Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 2.a

Check to see if you have the kernel header files on your system:

a. $ ls -l /usr/src/linux-headers-$(uname -r)
   i. If you see some files including a Makefile, it means you already have the linux header files. If not (No such file or directory), get the linux header files:
   ii. $ sudo apt-get install linux-headers-$(uname -r)
   iii. You can execute the above ls -l command to see if this was successful.

Hands-on Activity, Step 2.b

Create a very simple Makefile under APATH

a. You can use your favorite editor. We use the pico or nano editor in the examples
b. pico Makefile
   - You just need a single line in your Makefile:
   - obj-m += tuxdriver.o
   - This line says that tuxdriver.o will be one of the modules that will be generated in the current directory.

Hands-on Activity, Step 2.c

Now, let's use the Makefile of the kernel to build the module for our driver. Assuming you are under APATH:

a. make -C /usr/src/linux-headers-$(uname -r) M=$PWD modules
b. Note that -C tells the make utility to go to that directory and use the Makefile in that directory. With M=$PWD, it tells make to come back to the current directory to build the modules target. Remember in the simple Makefile you created, with the line obj-m += tuxdriver.o, we just listed our driver as one of the kernel modules to be built.

Hands-on Activity, Step 2.d

Check if the build was successful. If you can see tuxdriver.ko under APATH then YES:

```
$ ls -l tuxdriver.ko
```

- If you get compilation error regarding a missing kernel header file, e.g., generated/autoconf.h then you better remove Linux header files and reinstall
  - $ sudo apt remove linux-headers-$(uname –r)
  - $ sudo apt-get install linux-headers-$(uname –r)

Compiling tuxdriver on the VM

ghchandson [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

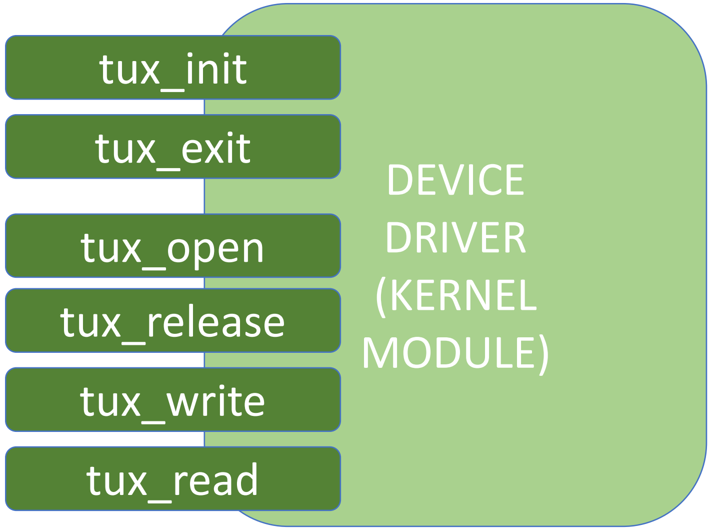
root@ghchandson: /home/ghchandson/GHCHandson/tuxdriver

```
root@ghchandson:/home/ghchandson/GHCHandson/tuxdriver# ls
Makefile  tuxdriver.c
root@ghchandson:/home/ghchandson/GHCHandson/tuxdriver# make -C /usr/src/linux-headers-$(uname -r) M=$PWD modules
make: Entering directory '/usr/src/linux-headers-4.15.0-112-generic'
  CC [M]  /home/ghchandson/GHCHandson/tuxdriver/tuxdriver.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ghchandson/GHCHandson/tuxdriver/tuxdriver.mod.o
  LD [M]  /home/ghchandson/GHCHandson/tuxdriver/tuxdriver.ko
make: Leaving directory '/usr/src/linux-headers-4.15.0-112-generic'
root@ghchandson:/home/ghchandson/GHCHandson/tuxdriver# ls
Makefile  modules.order  Module.symvers  tuxdriver.c  tuxdriver.ko  tuxdriver.mod.c  tuxdriver.mod.o  tuxdriver.o
root@ghchandson:/home/ghchandson/GHCHandson/tuxdriver#
```

Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

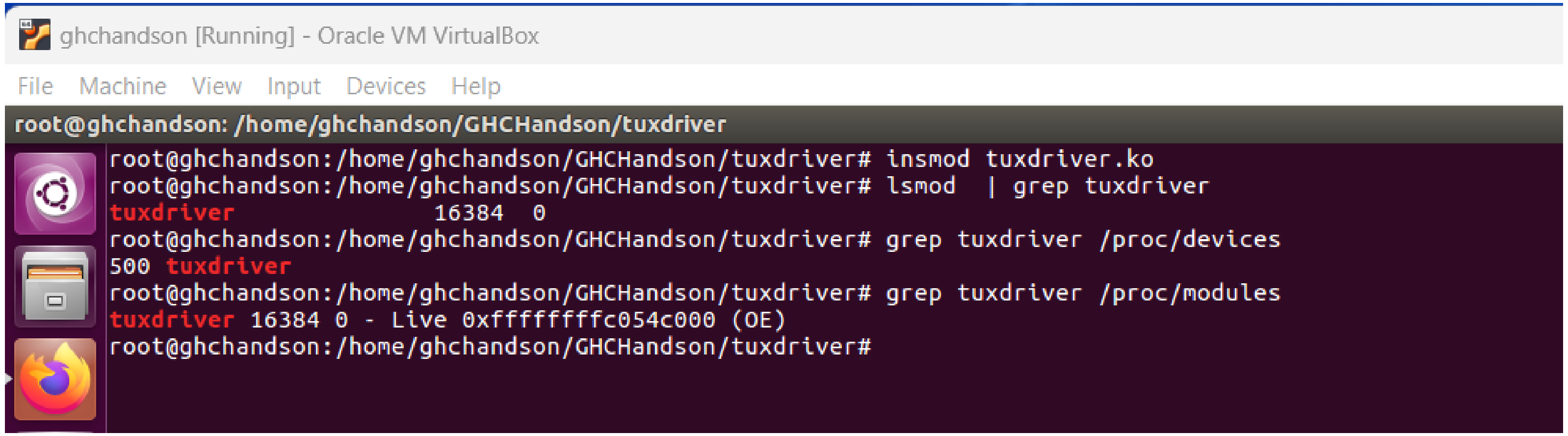
Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 3

Let’s load our module to the kernel

```
$ sudo insmod tuxdriver.ko
```

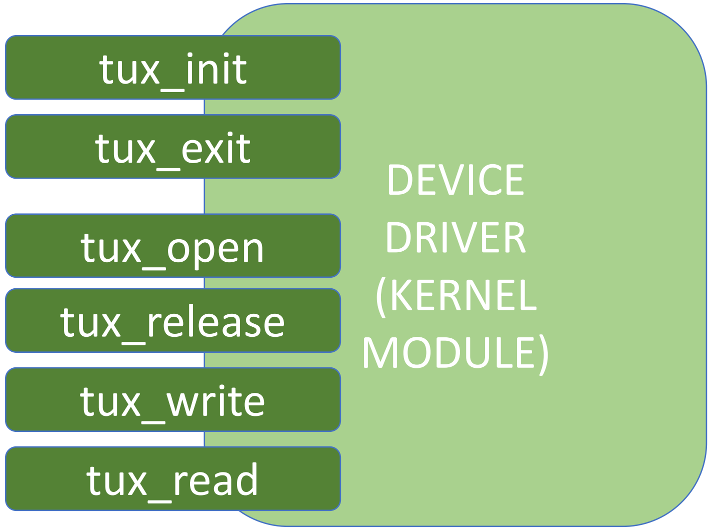
Installing tuxdriver on the VM



Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

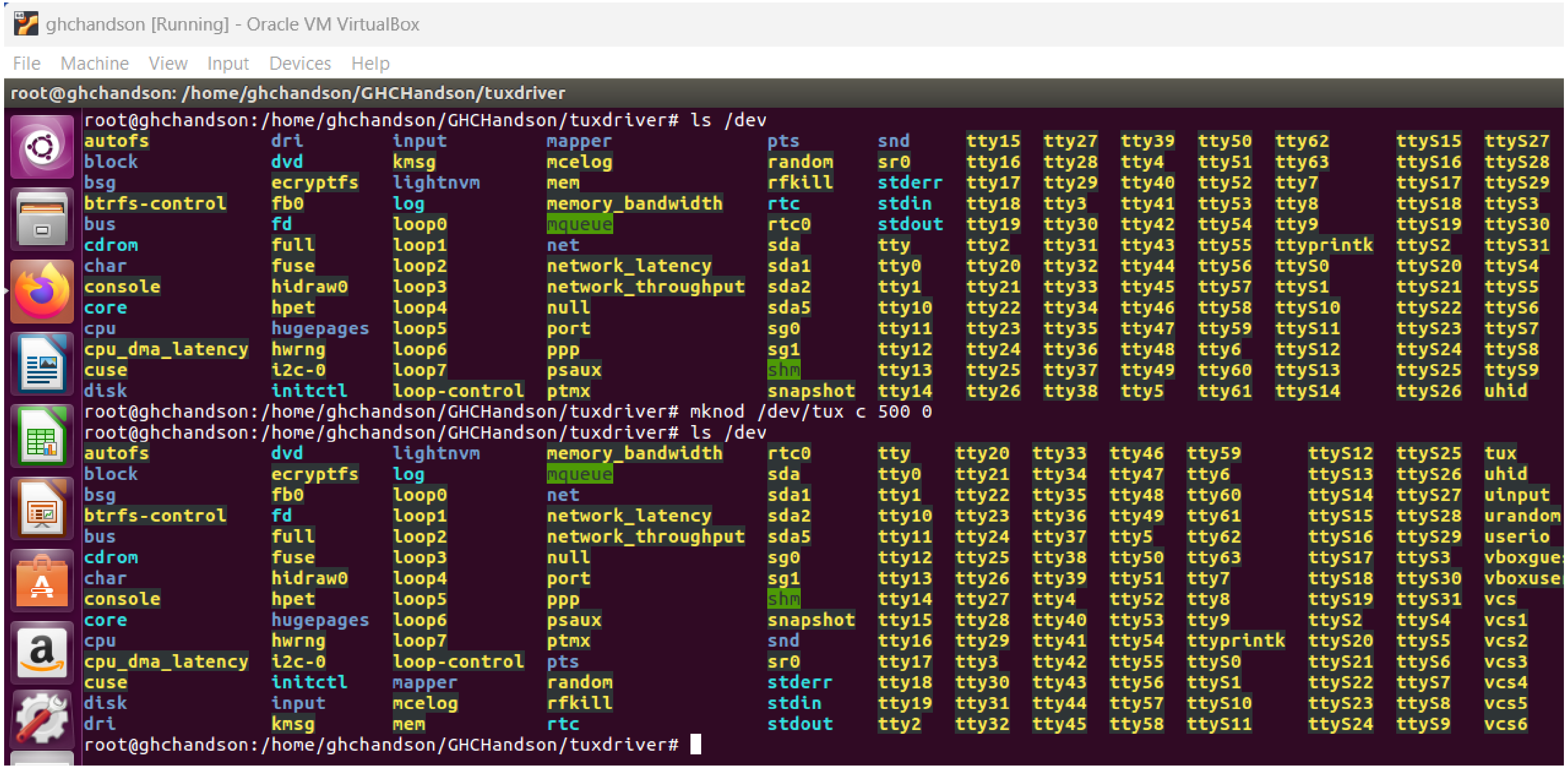
Hands-on Activity, Step 4

Now, let's play with our driver via the VFS Layer. We will first create a node for our hypothetical device tux.

a. $ sudo mknod /dev/tux c 500 0
b. Check if it gets created

   $ ls -l /dev/tux

Creating tux device node on the VM



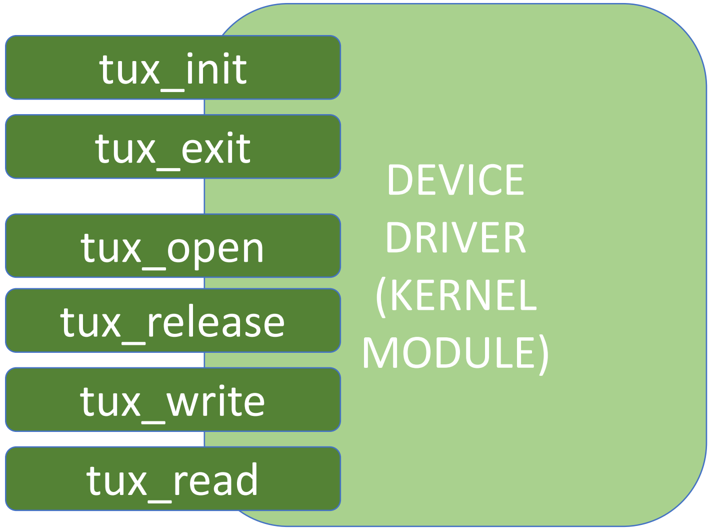
ANITA B.ORG 2023 GRACE HOPPER CELEBRATION

THE WAY FORWARD

Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 5

Now, let's play with our driver or test it using some shell commands.

a. First we will read its initial content, which should be some garbage

```
$ sudo dd if=/dev/tux bs=16 count=1
```

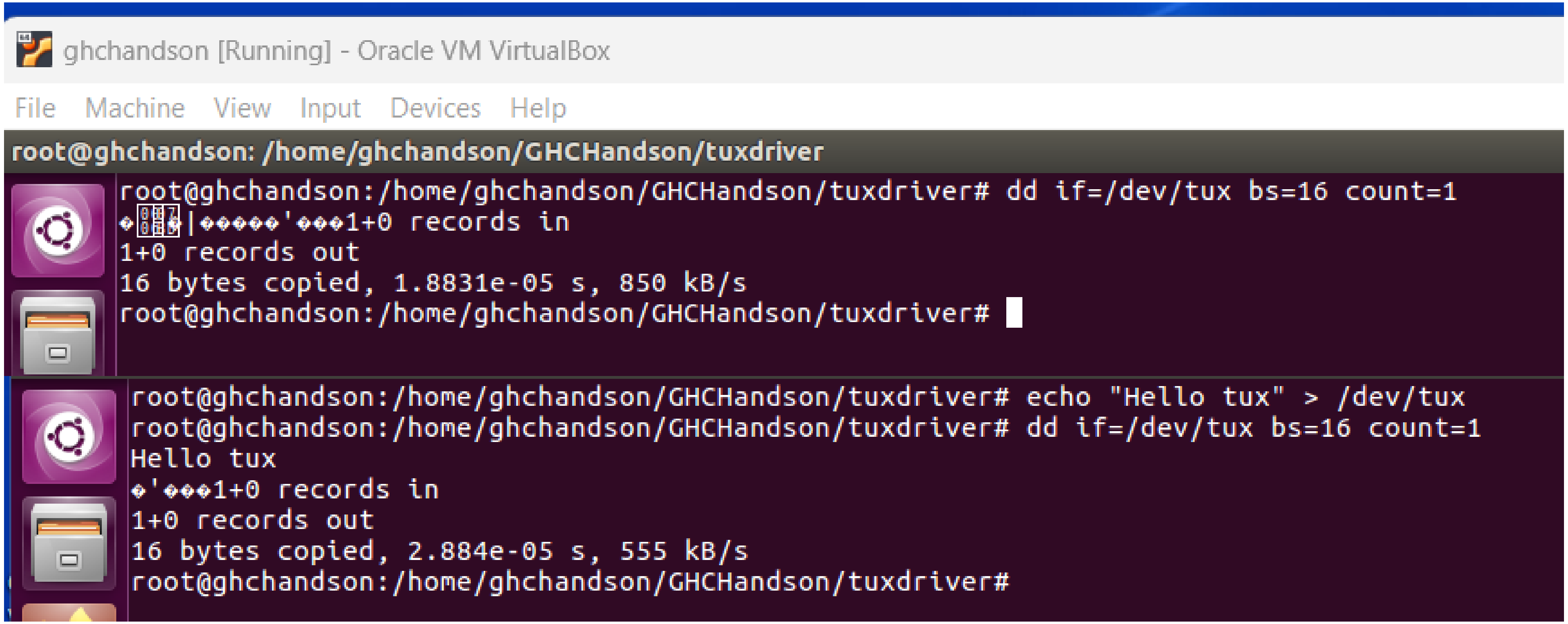
b. Next we will write to it

```
$ sudo echo "Hello tux" > /dev/tux
```

c. Last we will read its updated content

```
$ sudo dd if=/dev/tux bs=16 count=1
```

Testing tuxdriver using shell commands



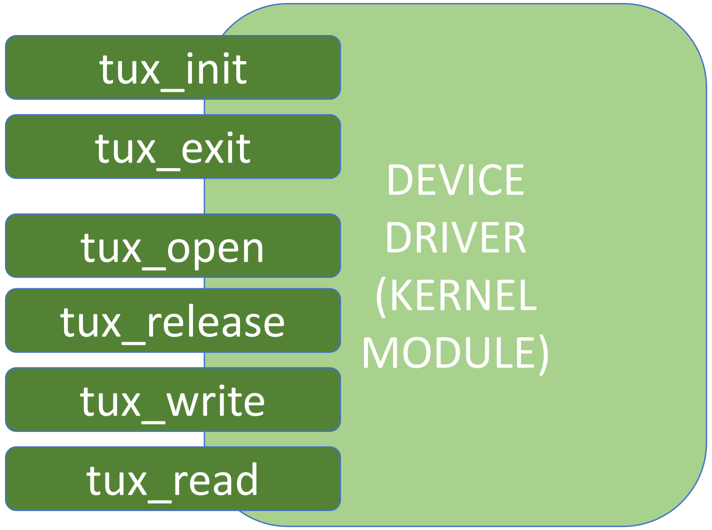
Checking Kernel logs after the first testing

```
[Wed Aug 30 14:55:42 2023] Closing tux as a file
[Wed Aug 30 14:57:09 2023] Opening tux as a file
[Wed Aug 30 14:57:09 2023] WRITING to tux, nbytes=10, pos=10
[Wed Aug 30 14:57:09 2023] Closing tux as a file
[Wed Aug 30 14:57:17 2023] Opening tux as a file
[Wed Aug 30 14:57:17 2023] READING from tux, nbytes=16, pos=16
[Wed Aug 30 14:57:17 2023] Closing tux as a file
root@ghchandson:/home/ghchandson/GHCHandson/tuxdriver# dmesg -T
```

Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 6

- Let's write our testtuxdriver.c that opens the device file and reads & writes.
- Make sure that the driver is loaded.
- Execute HelloDriver.c's executable.
- To check if we could write to the device, let's use the dd (data duplicate) command:
  - sudo dd if=/dev/tux0 bs=10 count=1
    - Here bs denotes block size and count denotes to number of blocks to duplicate

How we test tux

- Open tux the first time (file position pointer reset to the beginning of the file)
- Read 16 bytes to see its initial content (file position pointer points to end of ramdisk)
- Open tux the second time (file position pointer reset to the beginning of the file)
- Write "BYE for now, tux‟ to overwrite the contents (file position pointer points to end of ramdisk)
- Open tux the third time (file position pointer reset to the beginning of the file)
- Read 16 bytes to see its current content

Header files for the user space test code

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define size 16
char user_space_buf[size+1];
char user_space_buf2[size+1];
```

Test code for tux

```
int main(…) {
  int tuxfd = open("/dev/tux", O_RDWR);
  if (tuxfd == -1) {     printf("Could not open tux!\n");     return 1;  }
  printf("Opened tux successfully!\n");
  int numread = read(tuxfd, user_space_buf, size);
  if (numread == 0) {     printf("Could not read from tux!");     return 1;    }
  user_space_buf[numread] = '\0’;
  printf("Read from tux: %s\n", user_space_buf);
  printf("Let's reopen tux to move the position pointer to the beginning\n");
  // or you can implement an lseek entry point for tux and use that instead!
  …
```

Test code for tux (cont'd)

```
…
int tuxfd2 = open("/dev/tux", O_RDWR);
if (tuxfd2 == -1) {     printf("Could not open tux!\n");     return 1;  }
printf("Let's overrite tux's contents!");
int numwrote = write(tuxfd2,"BYE for now, tux",size);
if (numwrote == 0) {     printf("There was a problem writing to tux!\n");  }
…
```

Test code for tux (cont’d)

```
printf("Let's reopen tux to move the position pointer to the beginning\n");
int tuxfd3 = open("/dev/tux", O_RDWR);
if (tuxfd3 == -1) {     printf("Could not open tux!\n");     return 1;  }
numread = read(tuxfd3, user_space_buf2, size);
if (numread == 0) {     printf("Could not read from tux the 2nd time!");     return
1;    }
user_space_buf2[numread] = '\0’;
printf("This is what tux has now: %s\n", user_space_buf2);  printf("That's it
folks!\n");
return 0; } // end main
```

Testing tux using testtuxdriver

```
root@ghchandson:/home/ghchandson/GHCHandson/testtuxdriver# ./testtuxdriver
Opened tux successfully!
Read from tux: Hello tux
�'���
Let's reopen tux to move the position pointer to the beginning
Let's overrite tux's contents!Let's reopen tux to move the position pointer to the beginning
This is what tux has now: BYE for now, tux
That's it folks!
root@ghchandson:/home/ghchandson/GHCHandson/testtuxdriver# dd if=/dev/tux bs=16 count=1
BYE for now, tux1+0 records in
1+0 records out
16 bytes copied, 0.000280247 s, 57.1 kB/s
root@ghchandson:/home/ghchandson/GHCHandson/testtuxdriver#
```

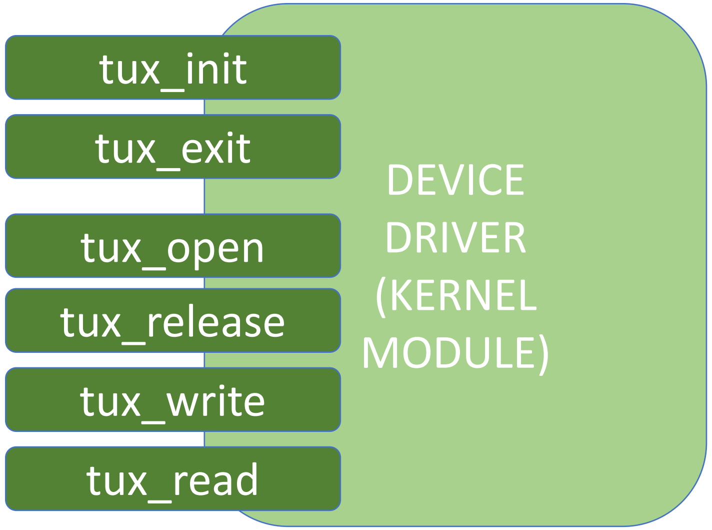
Kernel logs after running testtuxdriver

```
[Wed Aug 30 15:48:06 2023] Opening tux as a file
[Wed Aug 30 15:48:06 2023] READING from tux, nbytes=16, pos=16
[Wed Aug 30 15:48:06 2023] Opening tux as a file
[Wed Aug 30 15:48:06 2023] WRITING to tux, nbytes=16, pos=16
[Wed Aug 30 15:48:06 2023] Opening tux as a file
[Wed Aug 30 15:48:06 2023] READING from tux, nbytes=16, pos=16
[Wed Aug 30 15:48:06 2023] Closing tux as a file
[Wed Aug 30 15:48:06 2023] Closing tux as a file
[Wed Aug 30 15:48:06 2023] Closing tux as a file
[Wed Aug 30 15:48:33 2023] Opening tux as a file
[Wed Aug 30 15:48:33 2023] READING from tux, nbytes=16, pos=16
[Wed Aug 30 15:48:33 2023] Closing tux as a file
root@ghchandson:/home/ghchandson/GHCHandson/testtuxdriver# dmesg -T
```

Putting all major steps together

Step 0: Prepare a virtual machine instance

Step 1: Implement the device driver



Step 2: Compiling the device driver

Step 3: Loading the device driver

Step 4: Creating the device node

Step 5: Testing the driver using shell commands

Step 6: Implementing a user space program to test the driver

Step 7: Hacking the driver to cause a Kernel Panic

Hands-on Activity, Step 7

- Let's change tuxdriver.c to introduce a memory error to observe its side effects.
- Some suggestions to try (one by one):
  - Comment out the line that calls kmalloc to cause NULL pointer dereference (ending in a Kernel Panic/Oops, kind of a **Denial of Service** (DOS) attack)
  - Comment out the if statements that check whether the number of bytes to be read/written to ramdisk goes beyond the end of the buffer
    - Memory out of bounds read (as in the case of the **HEARTBLEED** vulnerability, sensitive data may be leaked)
    - Memory out of bounds write (this may be exploited for **Remote Code Execution**!)
- Recompile the driver each time and test your code!
- Happy hacking!

Outline

- What is this workshop about?
- What is a device driver & how do drivers work in the Linux Kernel?
- Which system APIs get involved?
- Hands-on Activity: Writing & Testing a Character-special Device Driver
- **Wrap-up**
- Questions & Answers

Resources

- Writing Linux Device Drivers book by Jerry Cooperstein
  - Writing Linux Device Drivers: a guide with exercises - Volume 3 | Guide books | ACM Digital Library
- Linux Device Drivers, 3rd edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
  - Linux Device Drivers, Third Edition [LWN.net]
- The Linux Documentation Project (tldp.org)
- The Linux Kernel Archives
- Linux source code (v6.5) - Bootlin

Acknowledgements

- Thanks to my students who have applied the presented content as an in-class activity in my **Advanced Systems Programming** course, which is offered as an online course (undergrad & grad sections) at the University of Florida.
- Thanks to those students who have participated in an earlier version of the Kernel Hacking Workshop at the University of Florida in Spring 2015 & 2017.
- This work has been partially funded by my NSF CAREER Award #CNS-1942235.

ANITA B.ORG 2023
GRACE HOPPER
CELEBRATION

THANK YOU

ANITA B.ORG 2023
GRACE HOPPER CELEBRATION

THE WAY
FORWARD