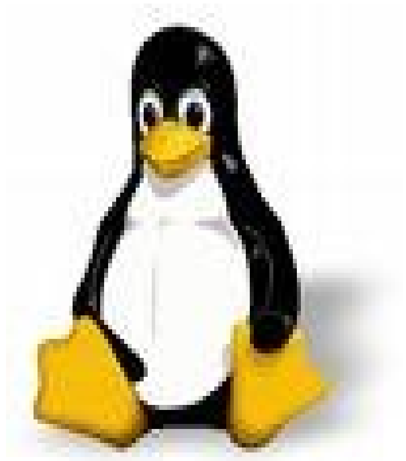


Introduction to Device Drivers

Santosh Sam Koshy

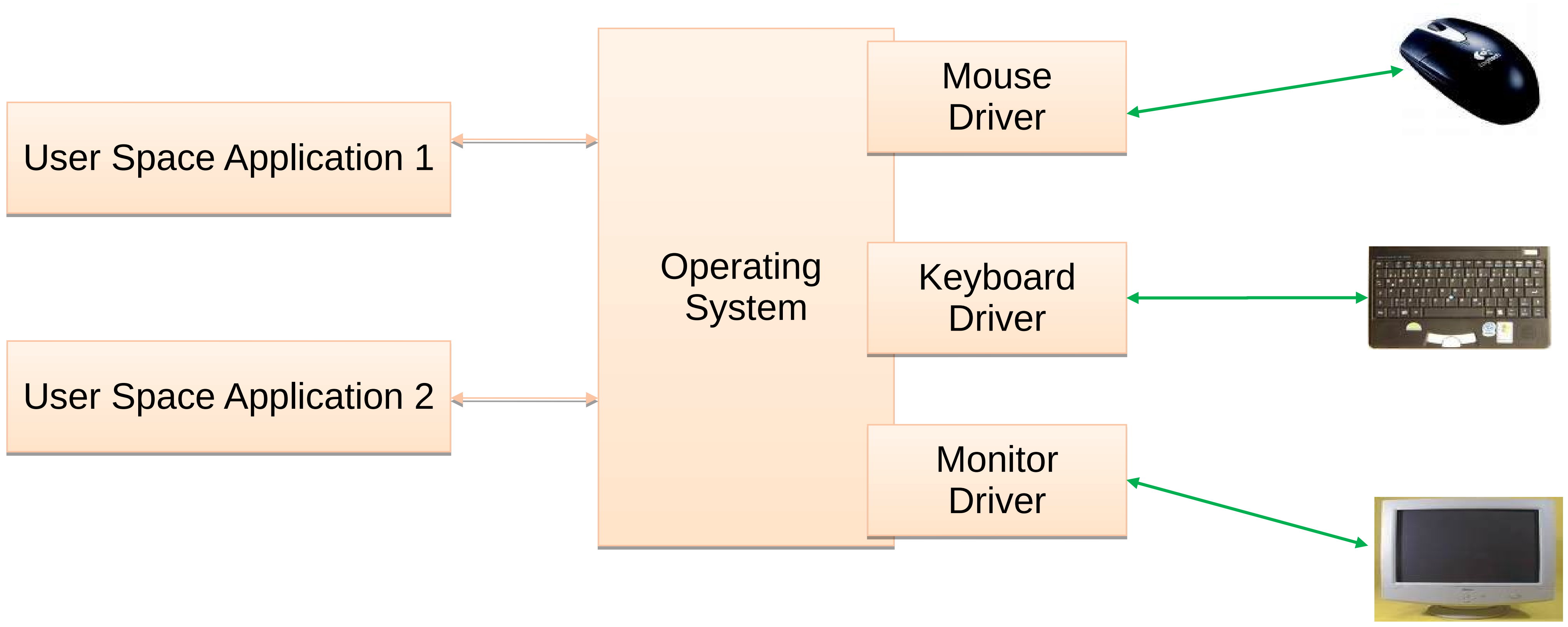
JD, C-DAC, Hyderabad

santoshk@cdac.in

Recap

- Module Initialization
- Module De-Initialization
- Compilation & Loading a Module
- The Makefile
- Module Parameters
- Kernel Symbol Table

Device Drivers



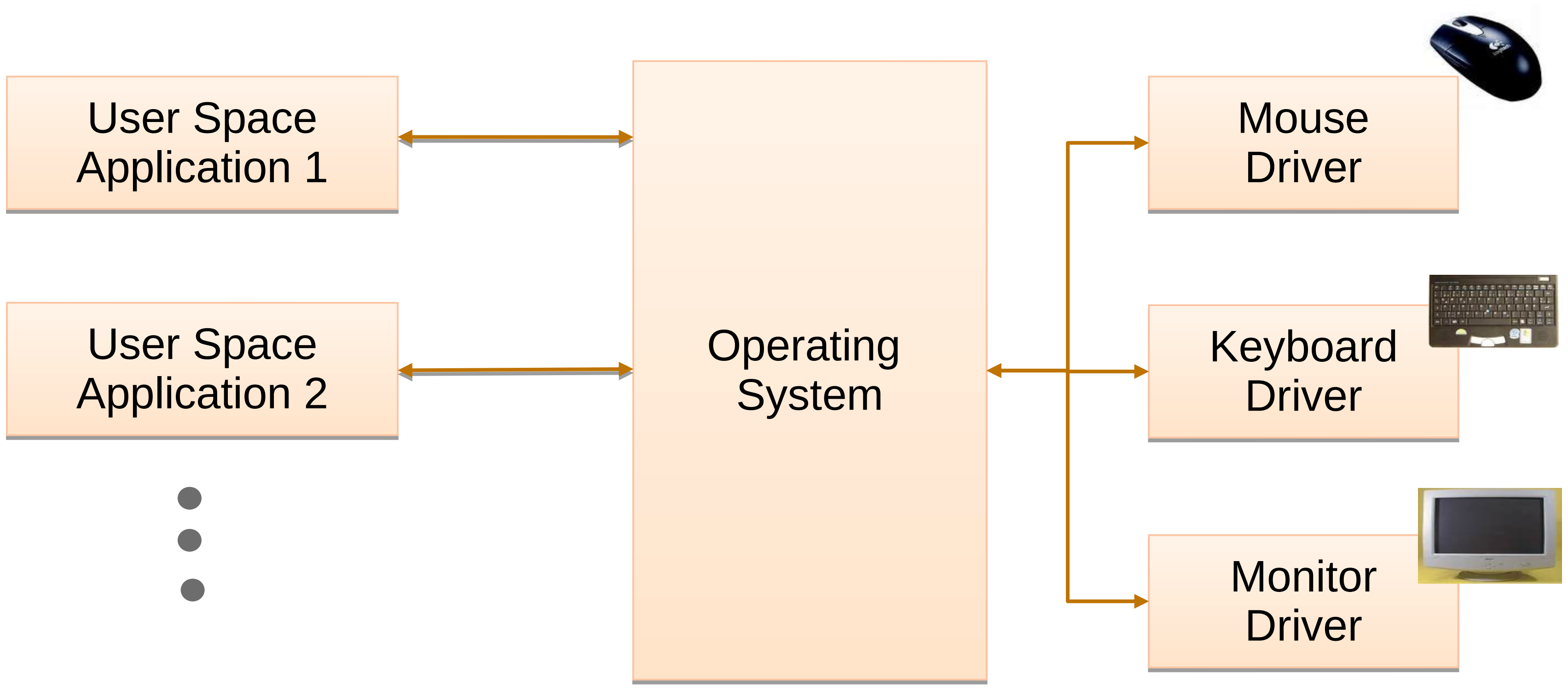
Role of a Device Driver

- Provide mechanisms and not policies
- Policy free drivers have support for
  - Synchronous and Asynchronous Operation
  - RE-ENTRANCY
  - Ability to exploit full capabilities of the hardware

Driver Classifications

- Character Drivers
  - Accessed as a stream of bytes
  - Implements open, read, write and close system calls
  - Accessed through file system nodes (Device Nodes) such as /dev/tty1, /dev/ttyS0
- Block Drivers
  - Data access is in blocks (512 bytes or more)
  - Accessed as nodes in /dev directory
- Network Drivers
  - Transactions are managed through interfaces – may be hardware or software
  - Deal with the transaction in the form of packets of information, though lower level transactions are in byte streams

Some questions that have to be answered…



Important Questions

1. How does the user space application(s) gain access to the device driver?
2. How does the kernel know if a module is a device driver or kernel module?
3. How does the kernel know the type/class of driver (Char, Block, Network)?
4. How does the kernel know the capabilities/functionalities provided by the driver?
5. How does the hardware interact with a device driver?
6. How does the kernel handle problems of synchronization, hardware timing and concurrency in the driver?

Character Drivers

Character Drivers

- Major & Minor Numbers - Device Numbers
- Necessary Data Structures
- Device Registration
- Device Operations

Devices and Device Files in the user space

- Devices are referred to by names in the filesystem
- These names are called device files or simply nodes in the filesystem.
- They find their existence in the /dev directory of the filesystem.
- They provide an interface for a user space process to access a kernel space device

Major and Minor Numbers

- On issuing an ls -al on the /dev directory, the following is listed

```
crw-rw----  1 root uucp    33,   3     Mar 20  2005 cux3
crw-rw----  1 root uucp    33,   4     Mar 20  2005 cux4
brw-r-----  1 root disk    12,   1     Mar 20  2005 dos_cd1
brw-rw----  1 root disk    14,   0     Mar 20  2005 dos_hda
lrwxrwxrwx 1 root root      8          Oct  4 15:38 dmdsp0 -> /dev/dsp
```

Device Number

Major Number

Minor Number

‘**c**’ at the beginning of the file attributes specifies that the device is a character device.

‘**b**’ indicates that the device is a block device.

‘**l**’ indicates a link to a device file present at an other location.

Major and Minor Numbers

- Major Number
    - The major number identifies the driver associated with the device
    - Multiple drivers may share major numbers
        - For example, a major number may identify a driver for a hard disk
- Minor Number
    - Used by the kernel to determine exactly which device is being referred
    - The minor numbers could serve as an index into a list of devices for the related driver
        - For example, a minor number may be used to refer to one of the 4 hard disks present in the system
- Both the major number and minor number together are called **Device Numbers**

Device Numbers – Kernel Side

- Within the kernel, the dev_t type is used to hold device numbers.
- dev_t is a 32 bit quantity with 12 bits set aside for the major number and 20 bits set aside for the minor number
- To create a device number, use

```
dev_t MKDEV(int major, int minor)
```

- To obtain the major and minor parts of a dev_t, macros come in handy

```
int MAJOR(dev_t dev)
int MINOR(dev_t dev)
```

- Headers

```
#include <linux/kdev_t.h>
#include <linux/types.h>
```

Allocating Device Numbers in the Kernel

- To obtain a device number to work with,
  - *int **register_chrdev_region** (dev_t first, unsigned int count, char *name);*
- To dynamically allocate a device number by the kernel, use the function
  - *int **alloc_chrdev_region** (dev_t *dev, unsigned int firstminor, unsigned int count, char *name);*

– Returns 0 on successful allocation, Negative number if not allocated

Allocating Device Numbers in the Kernel

- Device numbers are freed with
  - *void* ***unregister_chrdev_region*** *(dev_t first, unsigned int count)*
- Registering the device number in the init_module
- Un-register the device number in the cleanup_module
- Header
  - #include <linux/fs.h>

User Space Access to Device Numbers

- Applications in the user space can refer to a device by creating a device file or node in the /dev/ directory of the filesystem.
- This is possible by assigning a name to a device number using the shell command
  - **mknod** *name_device device_type major minor*

    eg: mknod /dev/MyCharDevice c 253 15

    ‘**c**’ specifies that the device is a character device,

    253 represents the major number and

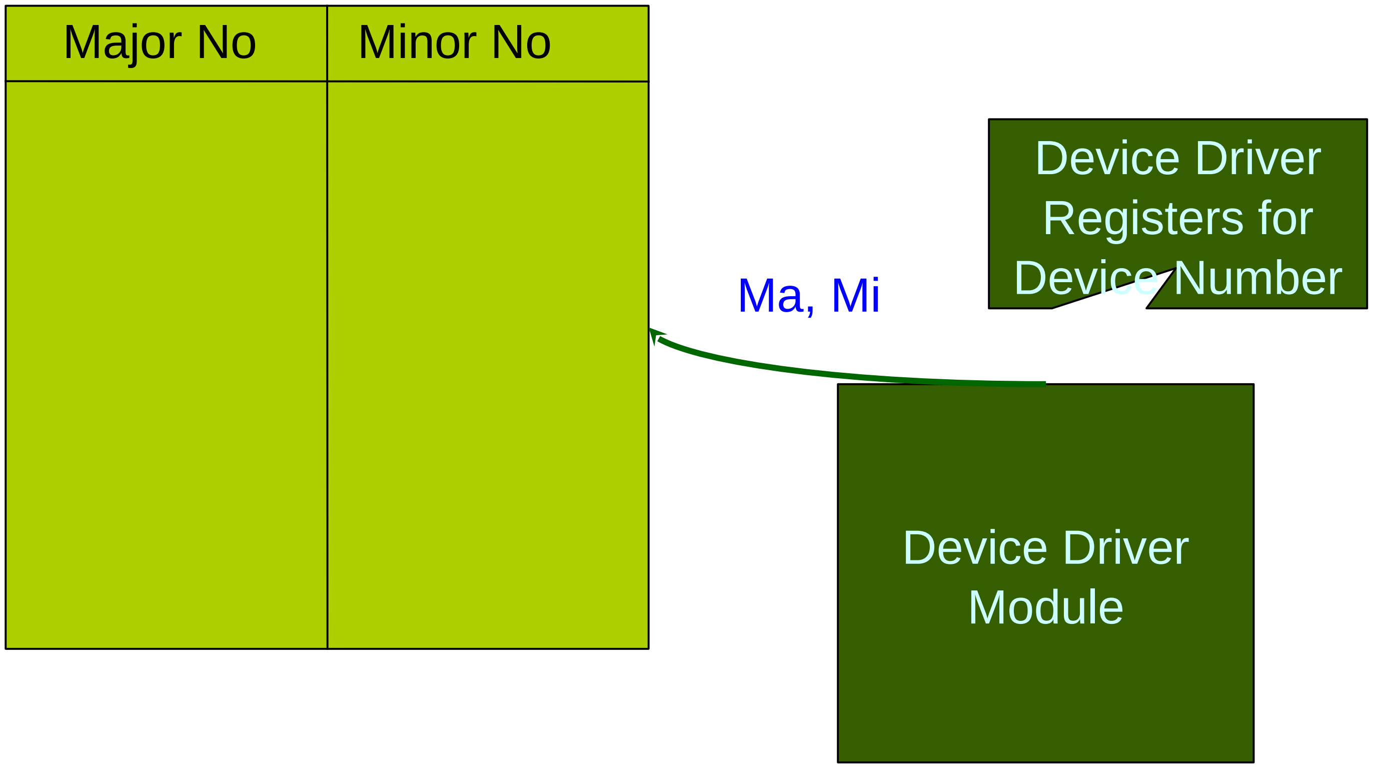
    15 represents the minor number

    MyCharDevice refers to the name associated with the driver at user space.

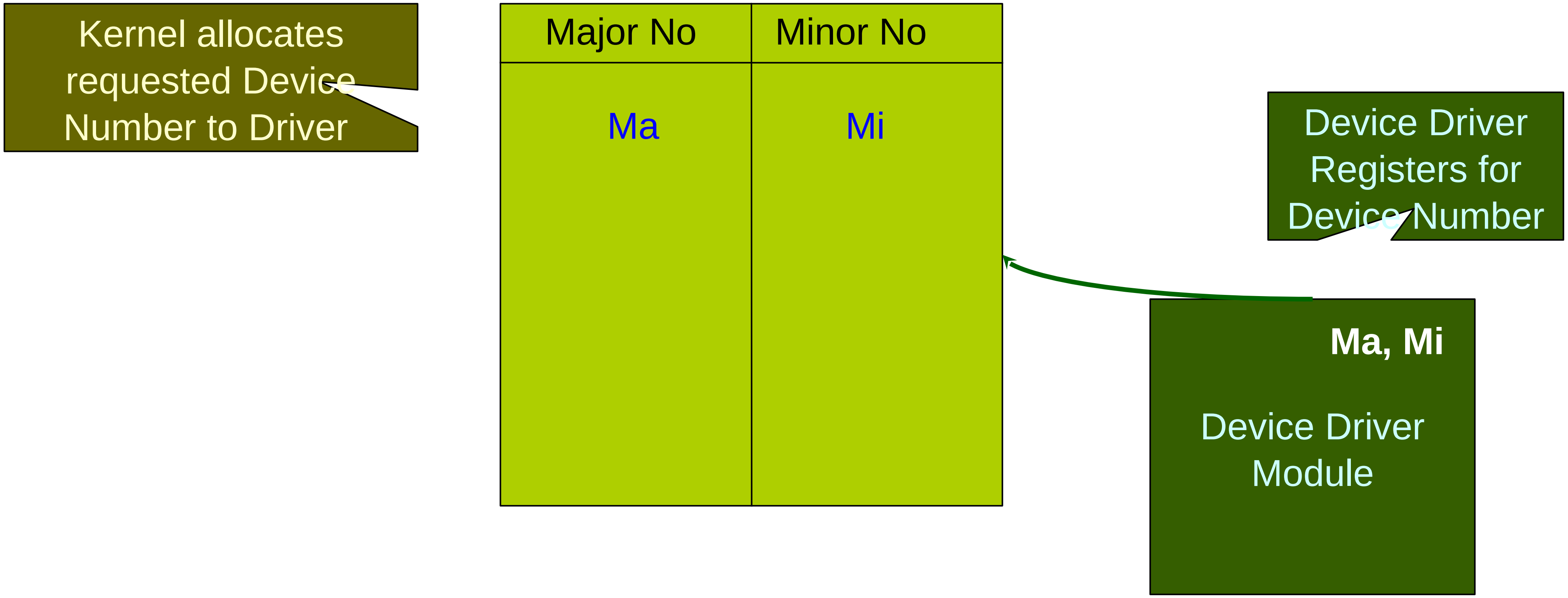
Device Numbers

Major No	Minor No

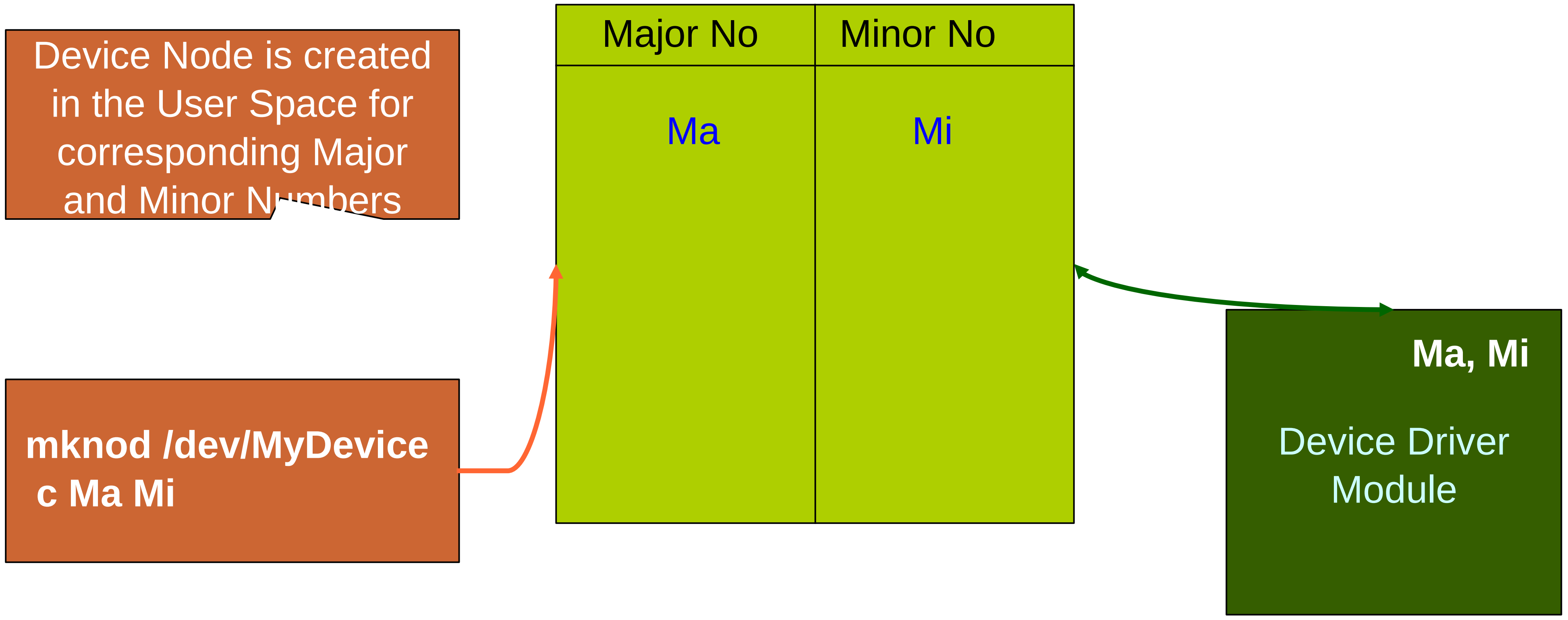
Device Numbers



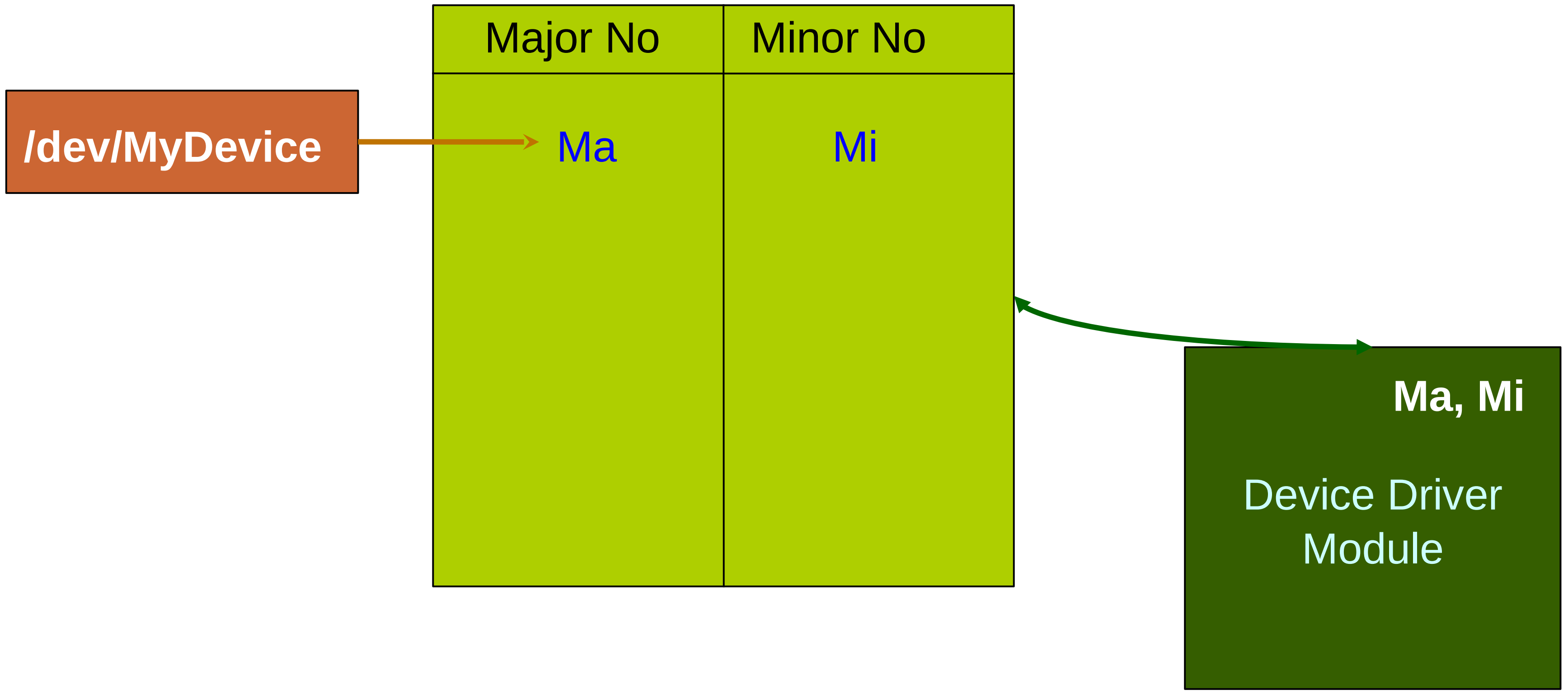
Device Numbers


Kernel allocates requested Device Number to Driver
Major No
Minor No
Ma
Mi
Device Driver Registers for Device Number
Ma, Mi
Device Driver Module

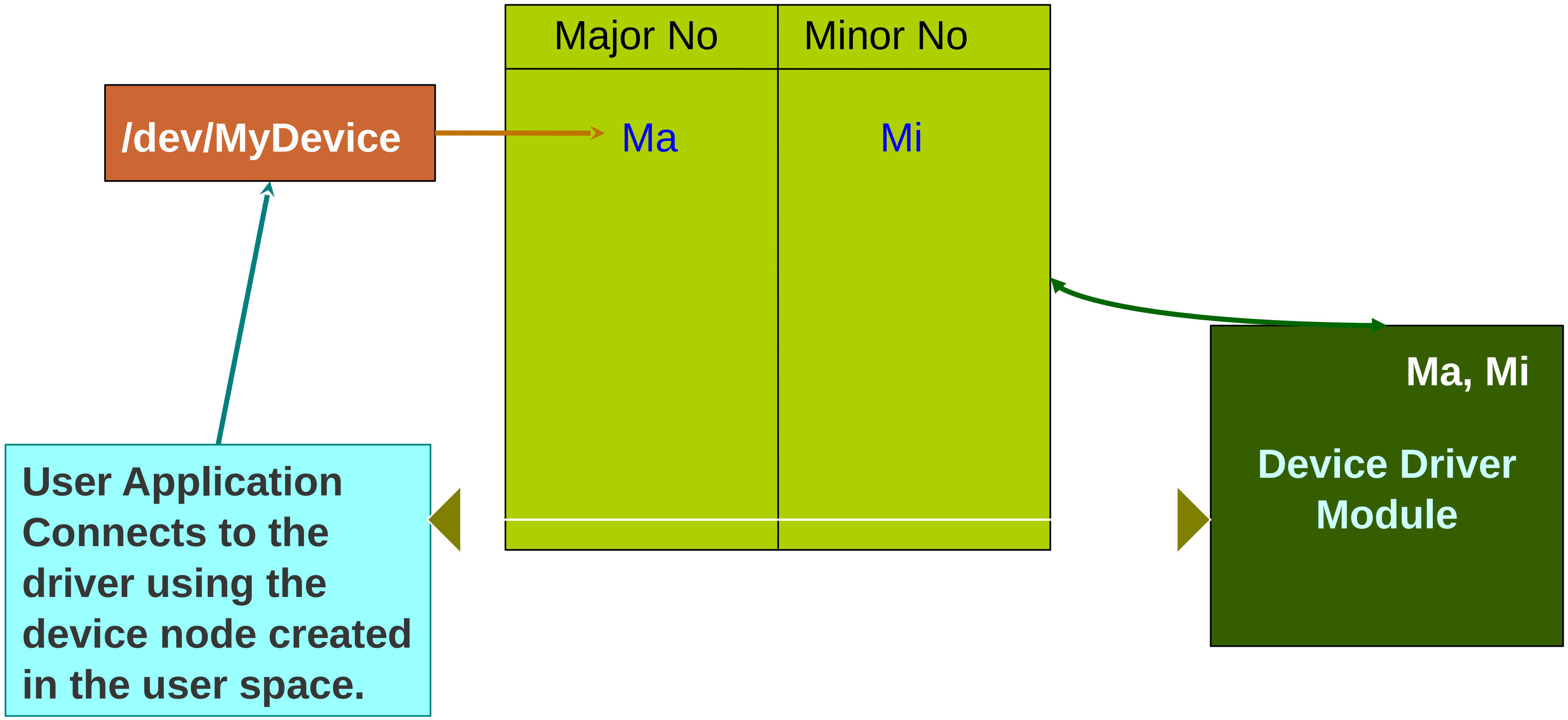
Device Numbers


Device Node is created in the User Space for corresponding Major and Minor Numbers
mknod /dev/MyDevice c Ma Mi
Major No
Minor No
Ma
Mi
Ma, Mi
Device Driver Module

Device Numbers



Device Numbers


Major No
Minor No
/dev/MyDevice
Ma
Mi
Ma, Mi
Device Driver Module
User Application Connects to the driver using the device node created in the user space.

Our Questions that we began with

Important Questions

1. How does the user space application(s) gain access to the device driver?
    1. By Opening the driver using the device file, created in the user space using the utility **mknod**
    2. In the user space, the device file, accessed through its name is more fundamental, rather than major-minor numbers
    3. The device file connects the user space to the device numbers registered in the kernel space
2. How does the kernel know if a module is a device driver or kernel module?
    1. The driver registers for a device number in the kernel space as part of its implementation
    2. Only devices register for a device number, telling the kernel that the module is a device driver
    3. Registering for a device number happens in the init section of the module, while unregistering is to be done in the exit section
3. How does the kernel know the type/class of driver (Char, Block, Network)?
4. How does the kernel know the capabilities/functionalities provided by the driver?
5. How does the hardware interact with a device driver?
6. How does the kernel handle problems of synchronization, hardware timing and concurrency in the driver?

Important Data Structures

Struct **File_Operations**
{
    Open
    Read
    Write
    Close
    IOCTL
}

Struct **File**
{

*file_operations
            mode_t
f_mode
            loff_t f_pos
            .....
}

Struct **Inode**
{
            *Device
Number
            *Char Device
      ....
      .....

Important Data Structures

- File Operations Structure
  - Defines the operations/services/functionalities/capabilities/methods provided by the driver
  - It is a structure of function pointers to each operation/service provided
  - The operations include some functionalities such as open, read, write, close, ioctl etc.,

```
struct file_operations {
        .owner = THIS_MODULE,
        .read  = MycharDev_read,
        .write = MyCharDev_write,
        .open = MyCharDev_open,
        .release = MyCharDev_close,
}
```

  - Drivers should define an instance of File Operations structure and initialize the members of the structure to functions being provided by the driver
- Header: #include <linux/fs.h>

Important Data Structures

- File
  - This structure represents an open file. It is used to control the operations of the driver.
  - It is **created by the kernel** and passed to the driver whenever a user makes a call to the open system call
  - It exists until the last close function is executed.
  - After all instances of the file are closed, the kernel releases the data structure
  - It is passed to all functions that operate on the file
  - It contains a pointer to the file operations structure created in the driver

Important Data Structures

- File: Some important fields in this structure
  - **mode_t f_mode** -Identifies the file as either readable or writable
  - **loff_t f_pos** -The current reading or writing position
  - **struct file_operations *f_op** -The operations associated with a file. The kernel assigns the pointer as a part of its implementation of open and then reads it when it needs to dispatch any operations.
  - **void *private_data** -this pointer can be used to point to allocated data, but it must be freed before the file structure is destroyed

Important Data Structures

- Inode Structure
  - The **inode** structure is internally used by the kernel to represent files. It is entirely different from the file structure
  - There can be many **file** structures associated with a file but they all point to a single inode structure for that file
  - The inode structure holds two important fields with respect to drivers
    - ***dev_t i_rdev:*** *Used to hold the actual device number*
    - ***struct cdev *i_cdev****: Kernels internal structure that represents char devices*
  - Access to i_rdev should be performed through access functions

Device Registration

- The kernel uses structures of type ***"struct cdev"*** to represent character devices internally
- It is initialized in one of the two methods

  *struct cdev *my_cdev = cdev_alloc();*

  my_cdev->ops = &myfops; (or)

  *void cdev_init(struct cdev *cdev, struct file_operations *fops);*

- Once the structure has been set up, the kernel is notified with the call to

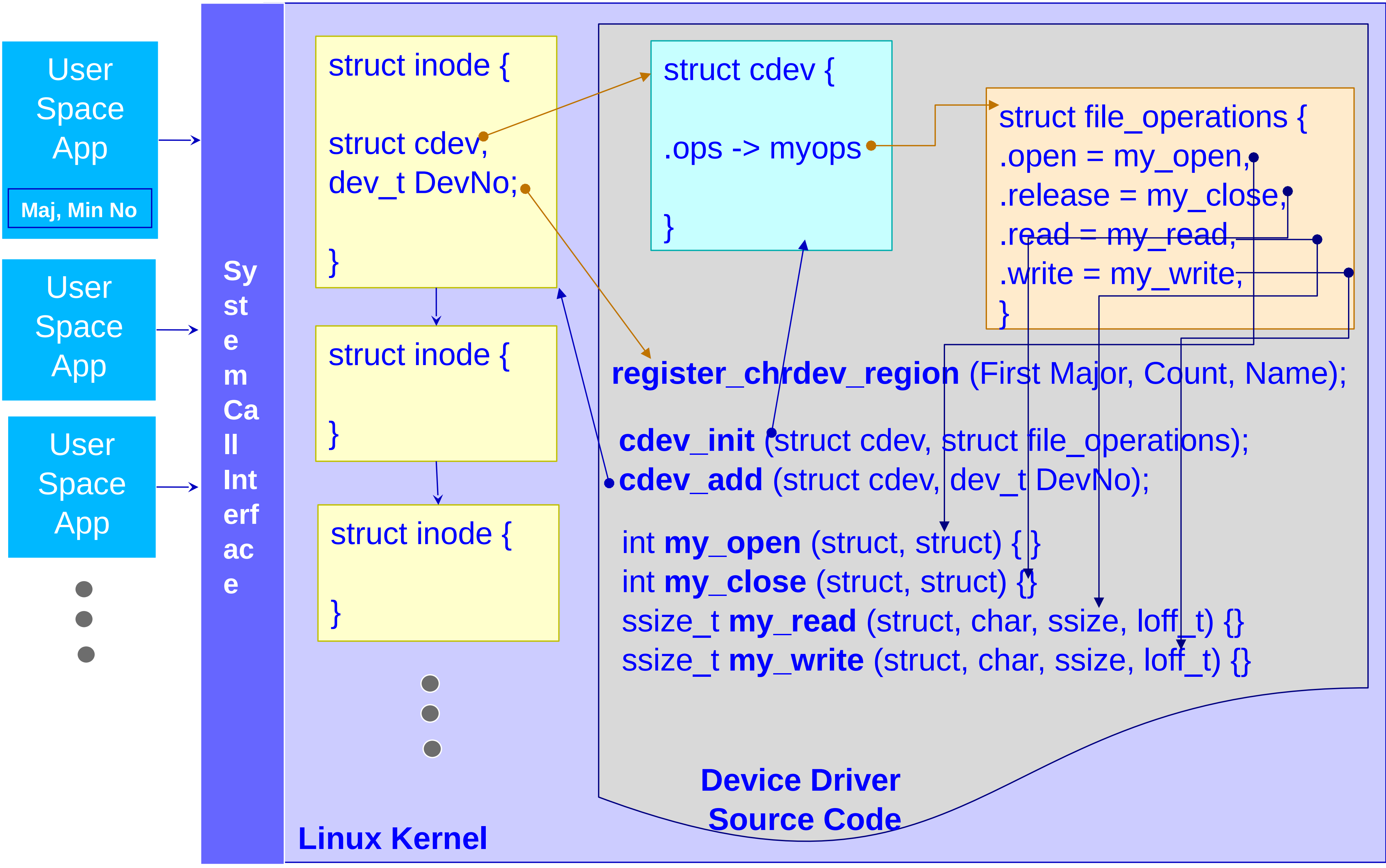
  *int cdev_add(struct cdev *dev, dev_t num, unsigned int count);*

- To remove a character device

  *void cdev_del(struct cdev *dev);*

- Header: #include <linux/cdev.h>

A pictorial representation of a Device Driver


User Space App
Maj, Min No
User Space App
User Space App
System Call Interface
struct inode {
struct cdev;
dev_t DevNo;
}
struct inode {
}
struct inode {
}
Linux Kernel
struct cdev {
.ops -> myops
}
struct file_operations {
.open = my_open,
.release = my_close,
.read = my_read,
.write = my_write,
}
register_chrdev_region (First Major, Count, Name);
cdev_init (struct cdev, struct file_operations);
cdev_add (struct cdev, dev_t DevNo);
int my_open (struct, struct) { }
int my_close (struct, struct) {}
ssize_t my_read (struct, char, ssize, loff_t) {}
ssize_t my_write (struct, char, ssize, loff_t) {}
Device Driver Source Code

Our Questions that we began with

Important Questions

1. How does the kernel know the capabilities/functionalities provided by the driver?
    1. The file_operations structure intimates the kernel about the capabilities/functionalities of the driver
    2. The initialized file_operations structure should be connected to the cdev structure to complete the process of intimation
2. How does the kernel know the type/class of driver (Char, Block, Network)?
    1. The cdev structure tells the kernel that the driver is implemented for character devices.
    2. Registering/Adding the initialized cdev structure intimates the kernel about the driver, as well as its functionalities
3. How does the kernel handle problems of synchronization, hardware timing and concurrency in the driver?
4. How does the hardware interact with a device driver?

The *Open* Method

- The open method is provided for a driver to perform any initialization in preparation for later operations.
  - Check for device specific errors
  - Initialize the device if it is being opened for the first time
  - Update the f_op pointer, if necessary
  - Allocate and fill any data structure to be put in

    filp->private_data

*int (*open)(struct inode *inode, struct file *filp)*

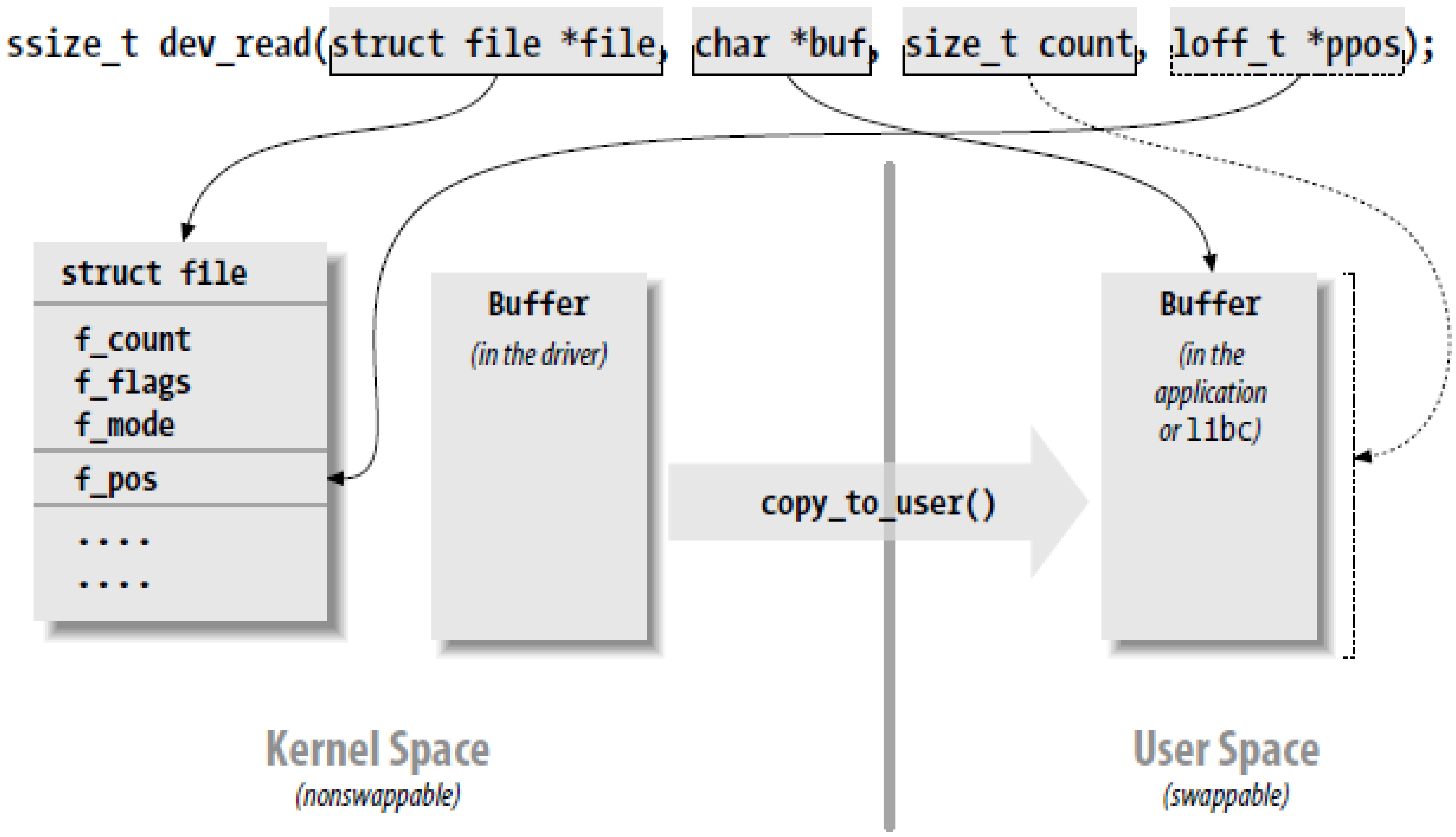
The *Release* Method

- As the name signifies, this call is used to release access to the requested device. The common functions that are performed in this method are
  - Deallocate anything that the open allocated
  - Shut down the device on last close
- *int (*release)(struct inode *inode, struct file *filp)*

The *Read* Method

- Interacts with the user space application program and implements the read functionality for the user space
  - *ssize_t read (struct file *filp, char __user *buff, size_t count, loff_t *offp)*
- If the return value equals the count, the requested number of bytes have been transferred.
- If it is positive but less than the count, only part of the data has been transferred. 0 indicates EOF. A negative value indicates error

The Read Method



The Read Method: Return Values

- Return Values as interpreted by the user application
  - If the value equals the count argument passed to the *read* system call, the requested number of bytes has been transferred. This is the optimal case.
  - If the value is positive, but smaller than count, only part of the data has been transferred.
  - If the value is 0, end-of-file was reached (and no data was read).
  - A negative value means there was an error.

Accessing the User Buffer

- Drivers should not dereference a user space pointer directly because
  - User space pointer may not be a valid location, depending on the architecture and kernel configuration
  - User space memory is paged. Could create an inconsistent access
  - User program may be buggy or malicious, allowing the program access into kernel space
- Access user space data – Header: #include <linux/uaccess.h>
  - *unsigned long copy_to_user (void __user *to, const void *from, unsigned long count);*
  - *unsigned long copy_from_user (void *to, const void __user *from, unsigned long count);*

Accessing the User Space Buffer

- Header
  - #include <linux/uaccess.h>
- Functions
  - *unsigned long copy_to_user (void __user *to, const void *from, unsigned long count);*
  - *unsigned long copy_from_user (void *to, const void __user *from, unsigned long count);*
- Return Value
  - Amount of data remaining to be copied. If 0, copy complete.
  - If pointer is invalid, or during copy if it encounters an invalid pointer, count of remaining data is returned

The *Write* Method

- Similar to the read method.
    - *ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp)*

To-Dos for Writing your Char Driver

Driver Side – Kernel Space

1. **Create a Device Number** for your driver (Maj & Min No)
2. **Register the device number** with the kernel
3. **Create your file_operations** structure and initialize it to the required functions
4. **Create the cdev structure**, allocate memory, intialize it and add to the kernel
5. **Define the functions** that the driver provides – open, read, write and close

Application Side – User Space

1. **Create a device node** for the specified Maj & Min No pair
2. **Write a user space application** that opens the device and performs read and write functionality

Assignments

1. Write a character driver with open and close functionality
   - Test the driver by writing an application that opens and closes the device driver. When open or closed, the open and close calls in the driver should be executed.
2. Write a character driver with open, read, write and close functionalities
   - Test the driver through a user application by reading data from the driver and writing data to the driver
3. Write a character driver to dynamically allocate a maj, min no pair from the kernel.
   - Test the same and conclude

Assignments

- Write a calculator driver in the kernel which performs the following
  - Create 4 Device Numbers – Each device number depicts a specific calculation operation like add, subtract, multiply and divide
  - Implement 8 methods – Read_Add, Read_Sub, Read_Mul, Read_Div, Write_Add, Write_Sub, Write_Mul and Write_Div
  - In user space, create 4 device nodes for the 4 device numbers created – /dev/AddDev, /dev/SubDev, /dev/MulDev, /dev/DivDev
  - Write 4 user applications in the user space to test the above. If /dev/AddDev application is run, it should write 2 numbers to the kernel and the kernel should add it and return the sum in the subsequent read.

References

- Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman,”*Linux Device Drivers*”,3rd Edition, O'Reilly Publications, March 2005
- http://www.senet.com.au/~cpeacock, “*Interfacing the Serial/ RS-232 port,V 5.0*”
- http://www.kernel.org
- http://en.wikipedia.org/wiki/Linux_kernel
- http://lists.ucc.gu.uwa.edu.au/pipermail/ucc/2003-June/009997.html

Thank You