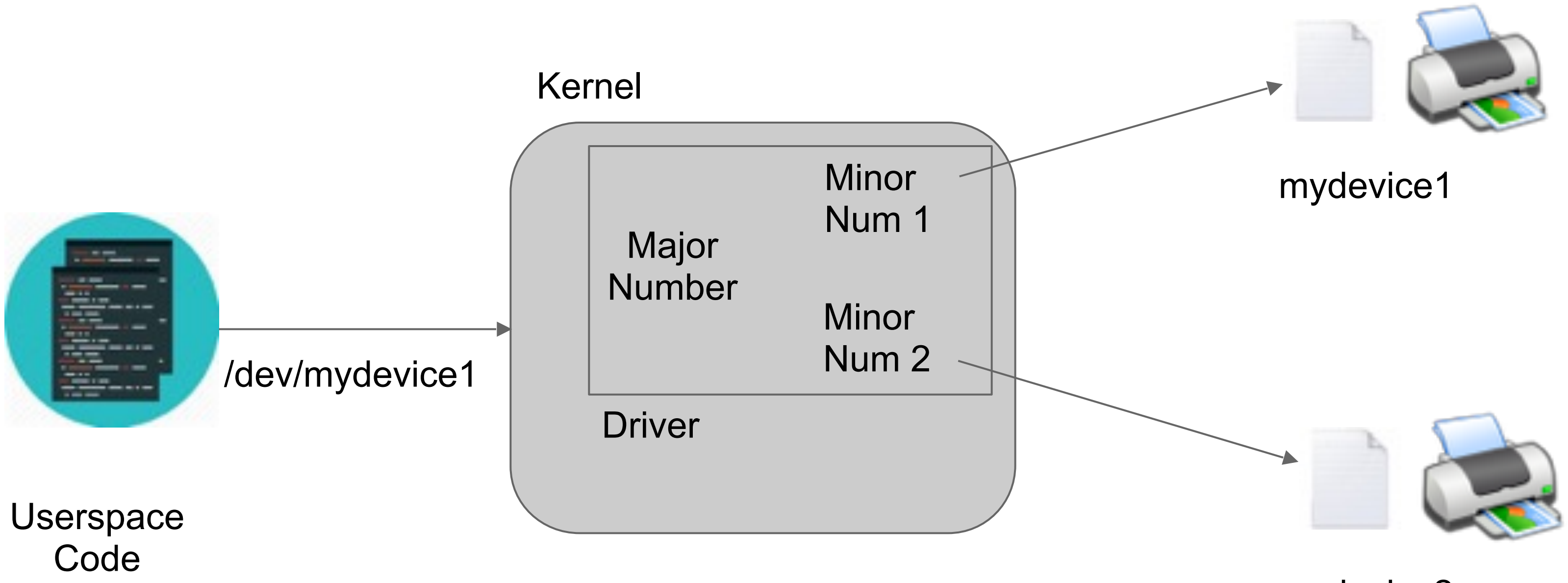
# Major and Minor Number



Kernel

Major Number

Minor Num 1

Minor Num 2

Driver

/dev/mydevice1

Userspace Code

mydevice1

mydevice2

# Step1: Write kernel_module.c

- Steps given in kernel module slides.

- Kernel module should have a file name
    - E.g. “mydevice” as given in slides

- It should define the allowed file operations on this file
    - “my_fops” in the slides contains the function pointers to the allowed file operation functions

- Driver should also request a minor number for the device
    - using MISC_DYNAMIC_MINOR
    - operating system dynamically assigns minor number to this file

- To register this device with kernel, you must call “misc_register()” function.
    - Ideal place to call “ misc_register()” function is in init_module() function as it is the first function that is called when you insert the module in the kernel.

- To unregister the device, you should call “misc_deregister()”.
    - Ideal place to call it is in cleanup_module()

# Step 2: Write user space program

- We can use regular file operations on device files:
    - Open - Called each time the device is opened from user space.
    - Read - Called when a process has already opened the file and tries to read from it.
        - Use “copy_to_user” to copy data to user space from kernel
    - Write - Called when a process tries to write into the device file.
        - Use “copy_from_user” to copy data to kernel from user space
    - Close - Called when the device is closed in user space.

# How do file ops work on character devices

- A file operation on a device file will be handled by the kernel module associated with the device.
- Call “open” system call to open “mydevice” file
- Call “read” system call to read from the “mydevice” file

```
fd = open("/dev/mydevice", O_RDWR);
```

  - opens /dev/mydevice device for read and write operation.
  - OS will call my_open() file operation handler in the kernel module which is associated with the device.
  - misc_register(&my_misc_device) instruction in my_module_init() registers the module. It creates an entry in the “/dev” directory for “mydevice” file and informs the operating system what file-operations handler functions are available for this device.

# Memory allocation/deallocation in Kernel

Memory Allocation:

kmalloc(): Allocates physically contiguous memory

```
void * kmalloc(size_t size, int flags)
```

kzalloc(): Allocates memory and sets it to zero

vmalloc(): Allocates memory that is virtually contiguous and not necessarily physically contiguous.

```
void * vmalloc(unsigned long size)
```

Memory Deallocation: kfree()

# Moving data in and out of the Kernel

- **copy_to_user()**
  - unsigned long copy_to_user (void __user * *dst*, const void * *src*, unsigned long *n*);
  - Copies data **from kernel space to user space**
  - Returns number of bytes that could not be copied. On success, this will be zero.
  - Checks that dst is writable by calling access_ok on dst with a type of VERIFY_WRITE. If it returns non-zero, copy_to_user proceeds to copy

- **copy_from_user()**
  - unsigned long copy_from_user (void * *dst*, const void __user * *src*, unsigned long *n*);
  - Copies data **from user space to kernel**
  - Returns number of bytes that could not be copied. On success, this will be zero.

**Question:** Why shouldn’t you use **memcpy** or **call by reference** to access userspace data?