

Linux Kernel and Driver Advanced



Amer Ather



Module 1-1

Linux Kernel Concepts

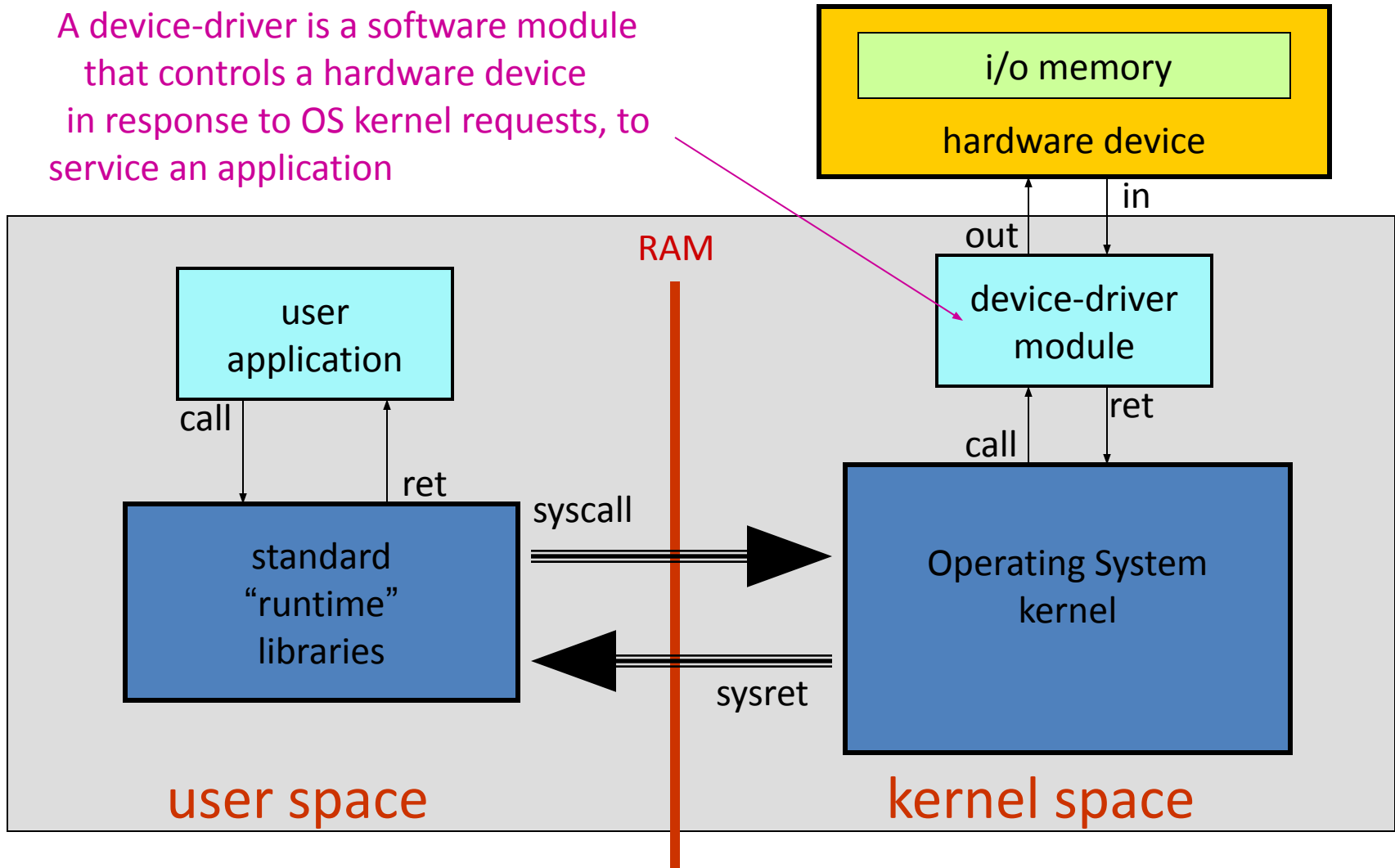
Device Driver or Kernel Module

What is a device driver or kernel module:

- A device driver is a collection of subroutines within the kernel that constitutes the software interface to an I/O device.
- When the kernel recognizes that a particular action is required from the device, it calls the appropriate driver routine, which passes control from the user process to the driver routine.
- A kernel module can be linked (loaded) and unlinked (unloaded) to the running kernel as needed or referenced
- Device Driver can also be statically linked into the kernel, thus cannot be unloaded
- Manages the low-level I/O operations of a hardware that are written with standard interfaces.
- Requires intimate knowledge of hardware registers and specification.
- Can emulate a device that exists only in software, such as RAM disks, buses, and pseudo-terminals

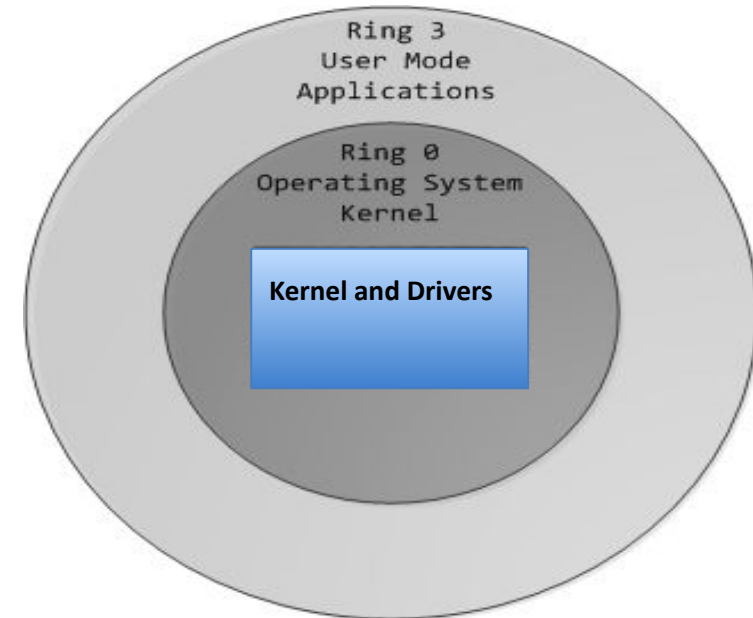
High Level View of Linux Device Driver

A device-driver is a software module that controls a hardware device in response to OS kernel requests, to service an application



CPU Rings, Privilege and Protection

- Linux is a “protected-mode” operating system. IO devices normally are not directly accessible
- Kernel and driver execute in the highest level (aka. supervisor mode), where everything is allowed.
- Application execute in the lowest level (aka. user mode) where CPU and OS. restrict what user-mode program can do. Direct access to hardware and memory allocation is regulated by the kernel
- At any given time, x86 CPU is running in a specific privilege level (implemented as protection rings), which determines what code can and cannot do.
- Kernel and user code execute in different address space (called, Kernel or User address space), use different stack (called, user and kernel stack) and have their own memory mapping



x86 Segment Protection

- Linux kernel use a **flat address space** where user-mode segments can reach the entire linear address space (32-bit or 64-bit).
- Memory protection is done via paging unit instead of privilege protection supported by Intel segments.
 - Each page of memory (contiguous chunk of memory, 4KB in x86) is described by PTE (page table entry), which has two fields related to protection: **supervisor** flag and **read/write** flag.
- The supervisor flag is the primary x86 memory protection mechanism used by Linux kernel. When it is **ON**, page cannot be accessed from ring 3 (user space).
- While read/write flag is used for marking pages read only.
 - When a process is loaded, executable code pages are marked as read only, thereby catching pointer errors if a program attempts to write to these pages.
 - This flag also implements copy on write (**COW**) when a process is forked in Unix.

NOTE: privileges has nothing to do with user level program running as root.

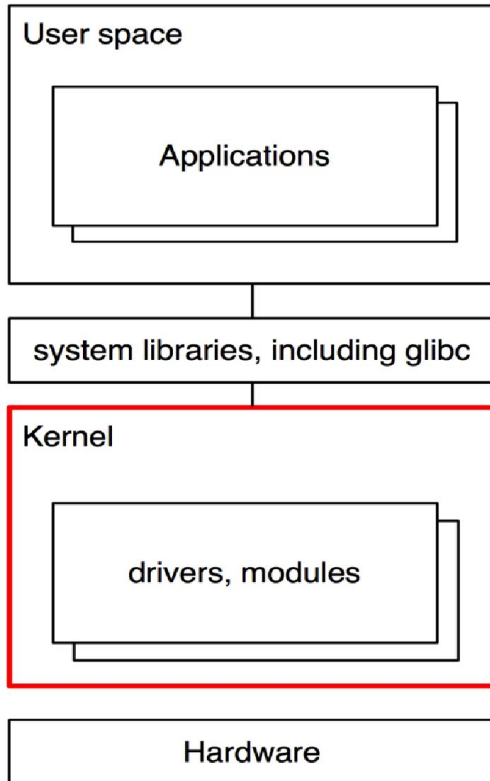
Switching CPU Modes

System Call

- To perform privileged operation, such as memory allocation, packet transfer, device IO, interrupt handling, new process creation etc.. CPU switches between privilege levels.
- When application executes a system call, it results in x86 **sysenter** instructions to be executed. CPU privilege level (CPL) is changed to CPL 0 and that causes new value to be loaded into the x86 registers (CS, EIP, SS, ESP) that now point to kernel code and stack segments.
 - **sysenter** instruction also facilitates copying system call number and arguments for system call handler
- Application blocks on system call until requested operation is completed by kernel on application behalf. Kernel issues **iret**, uses it to return from interrupt and system calls, that causes **sysexit** instruction to be executed, thus leaving ring 0 and resuming execution of user code with CPL 3.


*NOTE: When System calls return negative error code and it set errno: example: ENOMEM. System call is executed in the context of invoking process, called **process context***

Linux Environment



- Application to Kernel interface is well defined.
- Installing/compiling a new kernel does not change system libraries, such as glibc, the interface that nearly all application use
- The sysctl infrastructure to control kernel setting also stays unchanged when you add/compile a new kernel
- Only Device drivers and other kernel modules are tightly coupled with the kernel and will usually have to be recompiled with a new kernel

Driver Context

- Refers to execution environment of the driver
- Limits the operations that driver can perform depending on how driver code is invoked.
 - **Process context:** 
 - When invoked as a result of system call (synchronous fashion). It is different than user space where process is executing its own code outside of kernel.
 - **Kernel context:**
 - When invoked by some kernel subsystem and has no relationship with currently running task or process.
 - Example: Driver routine can be invoked by some kernel thread that has no relation to the current user thread.
 - **Interrupt context:**
 - When invoked to service an interrupt from the device
 - It is a restricted form of kernel context.
 - No access to user space is allowed, considering there is no associated process (*current* is not valid)
 - Driver interrupt handler runs in response to interrupt.
 - Interrupt handler runs with their interrupt line disabled, thus speed of driver's interrupt routine is crucial.

Device Files

- Device files in Linux/Unix provide abstraction for user programs to interface with device drives the same way as normal files by reading or writing the `/dev` files.
- System calls: `read`, `write`, `open`, `poll`, `close`, are mapped into driver entry points (aka. file operation methods)
 - Device nodes look like files and application `open()` and `close()` them like normal files: Example: `/dev/cdrom`, `/dev/console`, `/dev/hda`
- Unlike regular files, device files (device nodes) are special files with **major** and **minor** numbers . When application writes data into it, instead of going into a file system cache, the data is passed to device driver assigned to this major number.
- Device major and minor number links the device node (file) to driver.
- All device files with the same major number are serviced by the same driver code.

Standard Interfaces to Devices Or Driver Classes

- **Block Devices:** *e.g.* disk drives, tape drives, DVD-ROM
 - Access blocks of data
 - Handling data as asynchronous chunks.
 - Commands include `open()`, `read()`, `write()`, `seek()`
 - Raw I/O or file-system access, but typically accessed via file system cache
 - Memory-mapped file access possible
- **Character Devices:** *e.g.* keyboards, mice, serial ports, some USB devices
 - Single characters at a time
 - perform IO on continuous flow of bytes, called byte stream
 - Abstraction for devices that are accessed sequentially, character-by-character.
 - Commands include `get()`, `put()`. No seeking allowed!
 - Typically accessed by user program directly via device abstraction
- **Network Devices:** *e.g.* Ethernet, Wireless, Bluetooth
 - Different enough from block/character to have own interface
 - Identified by means of name (`eth0`), used as abstraction to Kernel network stack.
 - No device file (`/dev/eth0`) in file system namespace
 - Application uses `socket` interface to interface with Network stack
 - support `select()` (implemented as `poll()`) methods

Driver Entry Points or File operation Methods

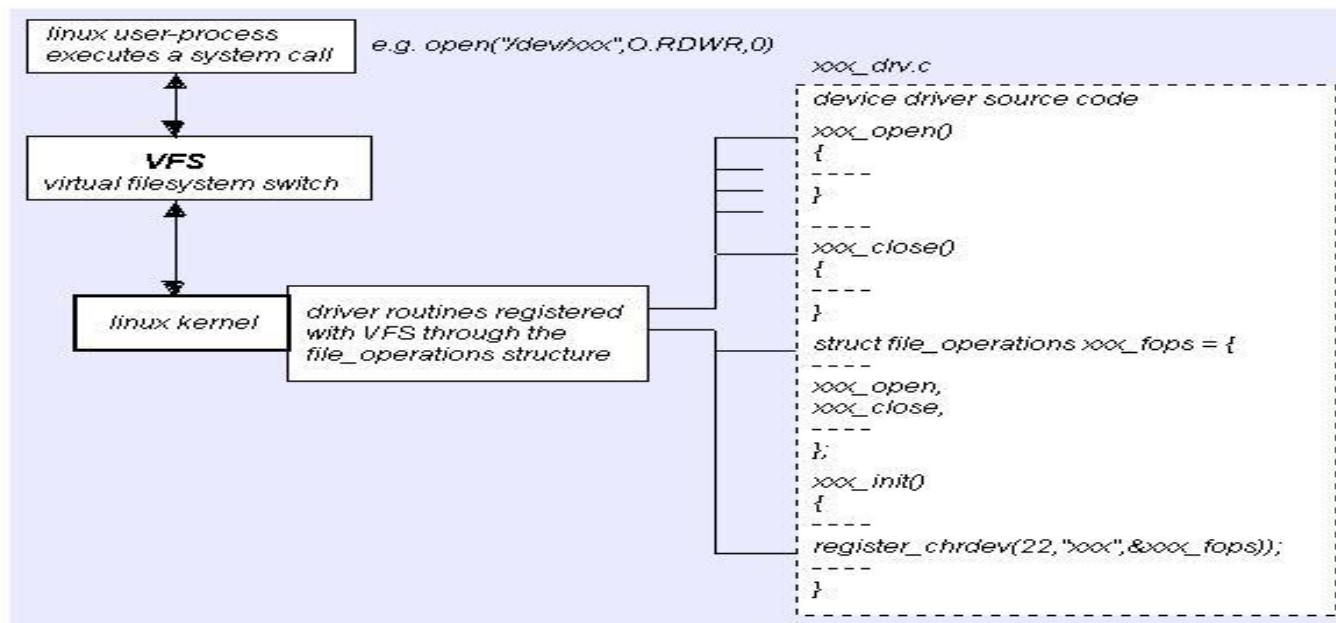
- A function within a device driver such as open(), read(), poll() etc..
- A set of routines that communicate with a hardware device and provide uniform interface to the kernel to invoke when device access is needed
- A self-contained part of kernel code that can be added to, or removed from kernel dynamically whenever driver is loaded or unloaded.
- Can be called by an external entity (application or kernel subsystem) to get access to a driver or to operate a device
- Provide user program access to a particular device via /device (/dev/) interface

Driver Entry Points or File Operation Methods

Name	Description
<p>module_init(my_init_func)</p> <p>static int __init my_init_func</p> <p>__init: function is only used during initialization</p> <p>__initdata : data used only during initialization</p> <p>See <linux/init.h></p>	<p>load, link, probe and initialize driver and device. Required, otherwise, insmod will fail to load the module. module_init() adds a special section in the module object code stating where the module initialization function to be found.</p>
<p>module_exit(my_clean_func)</p> <p>static void __exit my_clean_func</p> <p>__exit: function is called only at module unload</p> <p>__exitdata: data used during module unload</p> <p>For statically linked kernel modules and kernel configured to disallow unloading, __exit function is discarded</p>	<p>Do the reverse of init() function. Required, otherwise, rmmod will fail to unload the module. unregister the driver, unallocate memory or reset the device . If you fail, make sure to clean up the mess in this routine.</p>
xxx_open()	Invoked by open() system call in user space
xxx_close(), xxx_release()	when the device is closed
xxx_read(), xxx_write	When device is read or written
xxx_ioctl()	For passing control information to driver
xxx_mmap()	For mapping device memory in user space
xxx_poll()	Event notification

Virtual File System (VFS)

- Standard interface between a device driver and the rest of the kernel
- Application invokes file operation method using a common Virtual File system (VFS) interface that hides the complexity and implementation details
- VFS uses common set of methods and uses object oriented approach when deal with file system objects
 - VFS maps the generic `vfs_read` or `vfs_write` to driver specific routines: `foo_read`, `foo_write`
- Driver entry points are registered with the Virtual File system by updating ***file_operations*** structure



File Operations Table

- Contains function pointers to various VFS functions, such as read, write, open etc..
- file_operation structure is defined in <linux/fs.h>
- Number of operations are available:
 - Driver do *not* need to define all of them

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char *,
                    size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *,
                    size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
...
...
}
```

file operation methods

- This is how driver maps its entry point to file operation methods
- `file_operation` structure contains the address of the driver's entry points.
- Driver functions are, typically, mapped directly into system calls that user program invokes when it opens the associated device node (`/dev/foo`).

`read()` -> `my_read()`

`open()` -> `my_open()`

```
struct file_operations mydrv_fops = {  
    owner:  THIS_MODULE,  
    open:   my_open,  
    release: my_release,  
    read:   my_read,  
    write:  my_write,  
    llseek: my_llseek,  
    ioctl:  my_ioctl,  
    ...};
```


Driver Open function (example)

- Open entry point is invoked when the device node (/dev/foo) is opened by user program, typically, used for device initialization, etc.

```
struct foo { void *buf; /* ... */ };
static int my_open(struct inode *inode, struct file *file)
{
    struct foo *f;

    f = kmalloc(sizeof (struct foo_module), GFP_KERNEL);
    if (!f)
        return -ENOMEM;
    f->buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
    if (!f->buf) {
        kfree(f);
        return -ENOMEM;
    }
    file->private_data = f;    // Provide access from other entry points
    memset(f->buf, 0, PAGE_SIZE);
    return 0;
}
```

A Release Function

- Called on final close
- Cleans up reference to open device

```
static int my_release(struct inode *inode, struct file *file)
{
    struct foo *f;
    f = file->private_data;
    kfree(f->buf);
    kfree(f);
    return 0;
}
```

Getting Data To and From the User

- Driver should take caution when accessing user space memory.
- User buffer that is passed to driver for data exchange may not be mapped (page faulted) to physical address. Also, the address mapped may have been swapped out or may not be even valid.
- Kernel and drivers virtual addresses are always mapped to physical addresses in advance. User virtual addresses are mapped at the time of page fault (when virtual address is accessed), called demand paging
- Thus when accessing user space memory, driver should check if the address is valid and mapped. Otherwise, it may hang the driver code, and may eventually the whole system.

A pointer to user space should never be dereferenced without first checking.

Getting Data To and From the User (cont.)

To safely exchange data between user space and driver, consider using routines that performs necessary checks by calling ***access_ok()*** to verify user space address before accessing.

int copy_to_user(void *dst, void *src, size_t size)

- copies size bytes from src in kernel-space to dst in user-space

int copy_from_user(void *dst, void *src, size_t size)

- copies size bytes from src in user-space to dst in kernel-space

Both return zero on success or number of bytes not copied, In case of error driver should return -EFAULT

get_user(lvalue, ptr) and ***put_user(expr, ptr)*** are used to get and put single values (such as int, char, or long) from/to userspace.

Example: Getting Data To and From The User

Driver Read Method:

```
static ssize_t my_read(struct file *file, char __user *ubuf, size_t
    size, loff_t *off)
{
    struct foo f;
    f = file->private_data;
    char *kbuf = f->buf;
    int bytes;
    if (bytes = copy_to_user(ubuf, kbuf + *off, size))
        return -EFAULT;
    *off += bytes;

    return bytes;
}
```

The File Structure

- Address of File structure is passed as argument to file operation methods such as: open, read, write, close etc..
- File structure represents an **open file** in the kernel.
- struct file contains information about the associated /dev node that is opened by the user program.

Important fields in file struct:

- **private_data** field in the file structure is used to keep the device state information. It is used by drivers as a placeholder to conveniently correlate information from inside other driver methods
- **f_flags**: Flags passed when file is opened.
 - Examples: O_RDONLY, O_NONBLOCK, O_SYNC
- **f_pos**: To find the current reading/writing position in the file. lseek method modifies it to change the file position.
- **f_op** field in the file structure is a pointer to *file_operation* structure. It can be used by driver to implement different behaviors under the same major number by testing minor numbers

Example: open associated with major number 1 (/dev/null, /dev/zero, and so on) substitutes the operations in filp->f_op depending on the minor number being opened thus allows several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of “method overriding” in object-oriented programming.

How to use private_data field in file structure

Private_data field in file structure is typically used by drivers for sharing state information among various driver's methods.

Driver Open Method :

```
static int my_open(struct inode *, struct file *file)
{
    char *kbuf = kmalloc(10 * PAGE_SIZE, GFP_KERNEL);
    file-> private_data = kbuf;
    return 0;
}
```

Driver Write Method:

```
static ssize_t my_write(struct file *file, const char __user *ubuf, size_t
    size, loff_t *off)
{
    char *kbuf = file->private_data;
    int bytes;
    if (bytes = copy_from_user(kbuf + *off, ubuf, size))
        return -EFAULT;
    *off += bytes;
    return bytes;
}
```

The inode Structure

- Address of inode struct is also passed as argument to file operation methods
- Unlike a file structure that represents an open file in the kernel, an inode structure represents a file on the disk (/dev/foo).
- There can be multiple open files (file structure) pointing to the same inode structure
- Inode struct contains a pointer to device file (`dev_t *i_rdev`). It represents the device number that driver can use to obtain major and minor number via macros:
`unsigned int iminor(struct inode *inode);`
`unsigned int imajor(struct inode *inode);`
- Also, a macro can be used to return a **`dev_t`** data type from the major and minor numbers
`dev_t MKDEV(unsigned int major, unsigned int minor)`
- A macro that extracts the major/minor number from the device `dev_t`
`int MAJOR(dev_t); int MINOR(dev_t);`
- **`cdev`** field in inode represents char devices (`struct cdev *i_cdev`) when inode refers to char device file.

ioctl method

- Driver method unique to a device or device class.
- It is a catch all for anything else. You can literally pass anything
- Excessive use of ioctl is discouraged considering it is difficult to maintain.
- You can write a full functional driver by implementing an ioctl method only (Not recommended). Requirement is to have open and close methods, anything else is optional and ioctl can do everything
- Allows you to send commands outside of normal file commands like: read and write etc..
- Mostly used to implement device specific actions and control device behavior.

ioctl method

- When implementing ioctl method in the driver, one must choose a unique number (called magic number) that correspond to the command sent to the driver. This is needed to avoid conflict between two drivers using the same number.
- Conflict can occur due to:
 - Two device node having the same major number
 - Application may send a right command to the wrong device if it is talking to more than one device. This may result in device hang.
- Data Direction can be specified with command (No enforcement):
 - IOC_NONE – No data exchange
 - IOC_READ - Reading Data from the driver buffer
 - IOC_WRITE – Writing Data to driver buffer
 - IOC_READ|IOC_WRITE : Both direction
- Size parameter tells the amount of data (16 KB max) exchanged between user and kernel space, using **arg** as a pointer (3rd argument to ioctl method). Again no enforcement by kernel, but helps detect user errors.
 - When data structure is passed, sizeof() is applied to it.
 - Example: MY_IOCTL = _IOWR('k', 1, struct my_data_structure);

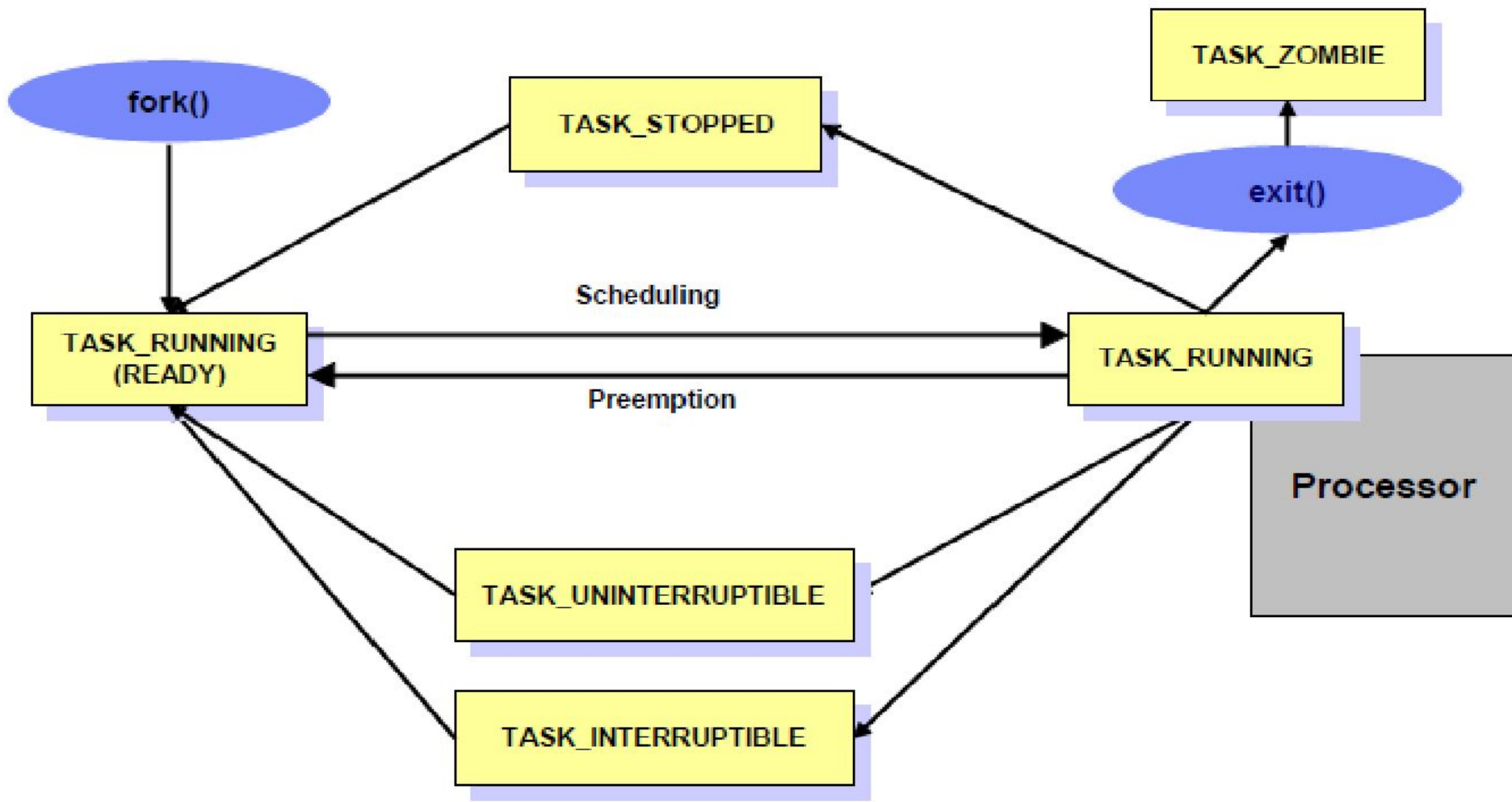
If data structure is dynamically allocated then sizeof() will return the size of pointer instead – Be Careful!

ioctl - example

```
#define MAGIC 'z'
#define GET_DELAY    _IOR(MAGIC, 1, int *)
#define SET_DELAY    _IOW(MAGIC, 2, int *)
#define INITIAL_SECS HZ/2
static unsigned long secs2hello = INITIAL_SECS;

static int hello_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg) {
    switch(cmd){
        case GET_DELAY:
            if (!arg)        return -EINVAL;        /* Null pointer */
            if (copy_to_user((long *)arg, &secs2hello, sizeof(long))) return -EFAULT;
            printk(KERN_INFO " value is:%ld\n",secs2hello);
            return 0;
        case SET_DELAY:
            if (!arg)        return -EINVAL;
            if (copy_from_user(&secs2hello, (long *) arg, sizeof (long))) return -EFAULT;
            printk(KERN_INFO "set to: %ld\n", secs2hello);
            return 0;
            break;
        default:            /* unknown command */
            return -ENOTTY;
    }
}
```

Process States



Process States

A process (or thread) can be in any of the following process states. ps reports process/thread states:

\$ ps -e -m -o pid,tid, **state**, command

PS (states)	Kernel task (states)	Detail
R	TASK_RUNNING	Process is either in the CPU runq or running
D	TASK_UNINTERRUPTED	Process cannot wakeup from sleep via kill signal unless condition it is waiting on happens
S	TASK_KILLABLE	Wake up only to a fatal signal (linux 2.6.25)
S	TASK_INTERRUPTIBLE	Process is waiting for IO, Network event, Lock etc. Kill signal can wake up the process
T	TASK_STOPPED	Process stopped execution due to signal or debugger
Z	Zombie	Process exited but now waiting for parent to reap status

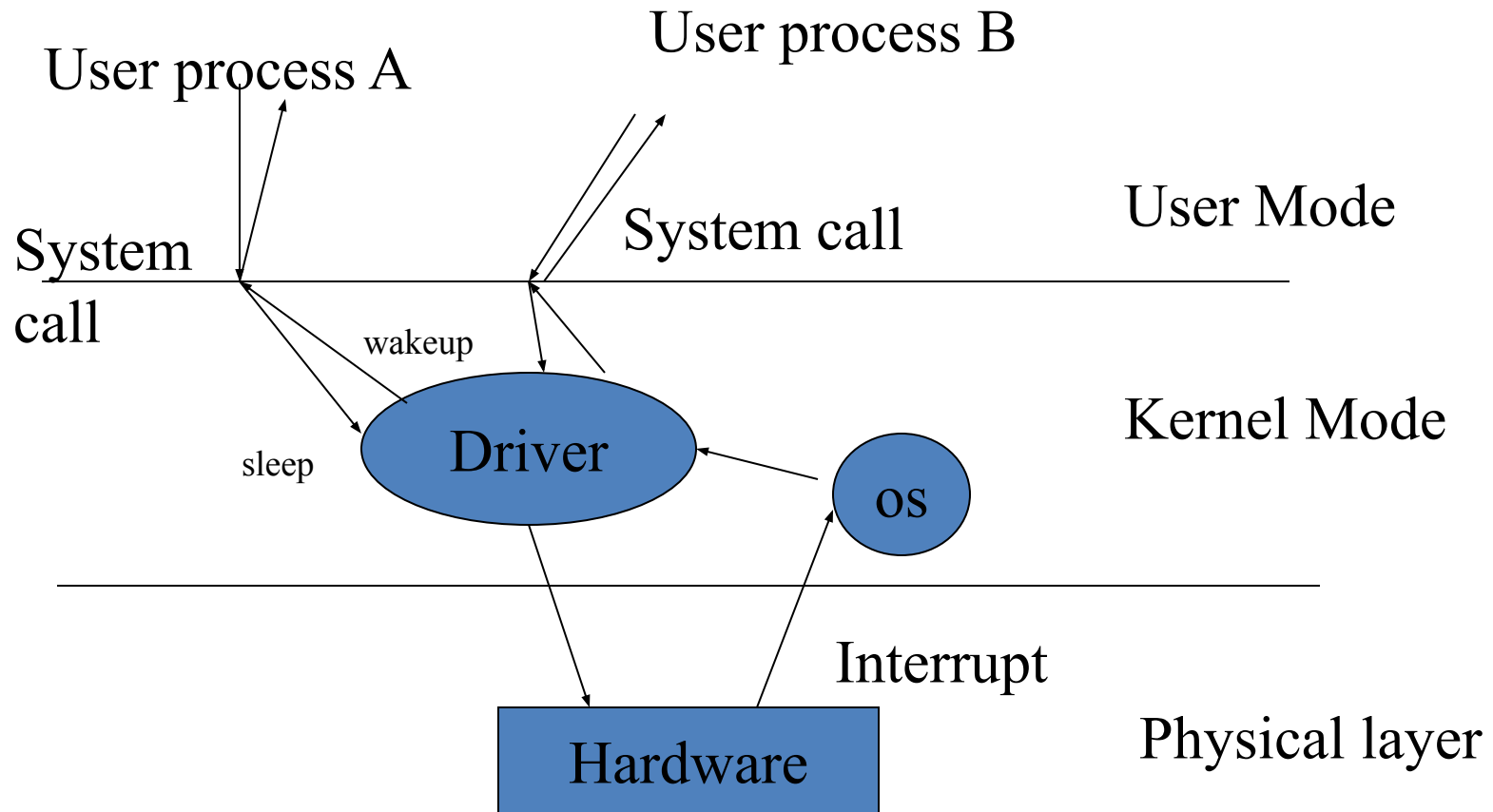
Sleeping

- Processes can go to sleep, suspend execution and allow other processes to run, until some event occurs that wake them up
 - For example a consumer might sleep until the producer creates more data, at which time the producer would wake the consumer up or vice versa.
- Driver can put the process (task) to sleep by changing its state from TASK_RUNNING to TASK_INTERRUPTIBLE, and then call schedule() to relinquish the CPU:
 - set_current_state(TASK_INTERRUPTIBLE);*
 - schedule();*
- CFS Scheduler selects next task to run with state TASK_RUNNING
 - It can be the current task, if you forget to change the state with *set_current_state()* and no other task is runnable
 - If no other runnable tasks, selects the idle task

Waking Up

- Driver wakes up a sleeping process when the resource becomes available
- Driver can trigger a wake up from interrupt service routine (ISR) after receiving the data from the device or from some other driver method signalling data or resource availability
- Task state is set to `TASK_RUNNING` so it will be picked up by the CFS scheduler on the next `schedule()`
- To find out process information that issued a system call, driver can use *current* that points to *task_structure* of running process

REQUEST FLOW

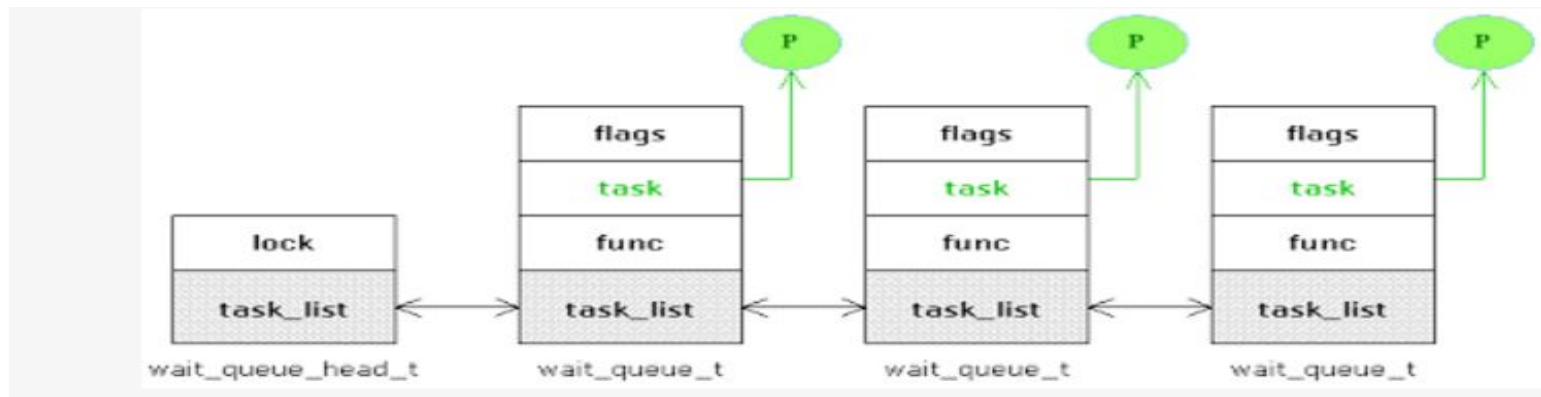


Wait Queue

- Wait queues hold tasks that need to wait for an event or a system resource. When process blocks, it is removed from the cpu run queue and placed on a wait queue
- Task status is changed to either: **TASK_INTERRUPTABLE**, **TASK_UNINTERRUPTABLE** or **TASK_KILLABLE** and inserted into per event wait queues.
- Task in a wait queue go to sleep until they are woken up by another thread or driver interrupt handler routine or any routine responsible for detecting the event.
- Interruptible wait functions return 0 if they return due to wake up or **-ERESTARTSYS** if return due to signal
- When task on a given wait queue are woken up, it first verifies the condition. It either resumes sleep (if condition is not true) or removes itself from the wait queue and sets task state to **TASK_RUNNING**
- When an event occur, all tasks are woken up unless an exclusive sleep function is invoked, **wait_event_interruptible_exclusive()**;
- Exclusive sleep and wake up are used in cases where exclusive access to a resource is required. It avoids the condition called, **thundering herd** (all sleepers are woken up, but only one of them gets the resource and other goes back to sleep)

Wait Queue

- Wait queues are doubly linked lists of **wait_queue_t** structures.
- Wait queue is of type **wait_queue_head_t** structure that can be initialized statically **DECLARE_WAIT_QUEUE_HEAD(wq)** or dynamically **init_waitqueue_head(&wq)**
- Each wait_queue structure holds pointer to task structure of the blocked process, initialized by calling **DECLARE_WAITQUEUE(wait, current)**.
- To block task on event or add to wait queue, call **add_wait_queue()**
- To remove task from a wait queue, call **remove_wait_queue()**.
- **wait_event_[interruptible|uninterruptible|killable](wq, condition)** are wrapper functions that deal with inserting/removing tasks into/from the wait queue, checking condition at wake up and changing status of the task to TASK_RUNNING
- **Wake_up_[interruptible|uninterruptible|killable](wq)** are functions for waking up a task. All tasks block on the event become runnable and scheduled on to the cpu according to the priority.
- **wait_event_timeout(wq, condition, timeout)** and **wait_event_interruptible()_timeout()** can be used instead if the driver wants to use a wait queue for some event that should occur within a timeout (specified in jiffies).



Wait Queue Example

```
DECLARE_WAIT_QUEUE_HEAD(wq);
/**
DECLARE_WAITQUEUE(wait, current); // initialize the wait struct to point to "current" task structure
add_wait_queue(&wq, &wait);        // insert the wait struct to wait queue wq.
for (;;) {
    set_current_state(TASK_INTERRUPTIBLE);
    if (condition)                  // check condition when wake up
        break;
    schedule();                     // relinquish the cpu
    if (signal_pending (current))
        return -ERESTARTSYS;
}
set_current_state(TASK_RUNNING); // if condition is true
remove_wait_queue(&wq, &wait);

} */
```

You can replace the whole block inside `/**... */` with

```
wait_event_interruptable(wq, condition);
if (signal_pending (current))
    return - ERESTARTSYS;
```

....

Task is woken up by calling `wake_up_interruptable()` and passing a matching waitq as argument `wake_up_interruptable(&wq)`.

poll, select and epoll

- Applications that deal with multiple input and output streams (file descriptors) use `poll()`.
- `poll()` is implemented via multiple wait queues. `poll()` system call sleeps until event occurs on one of the wait queues that wakes it up
- Normally process blocks on `poll()` until the event occurs.
- User application supply number of file descriptors to `select()` and request driver to notify state change: data is ready for read or space available in buffer to write etc..
- When the `poll` call is returned, the ***poll_table*** structure is de-allocated, and all wait queue entries previously added to `poll_table` are removed. User program requires to reissue `poll()` if it wants to continue to poll for events on file descriptors.
- Kernel invokes driver's poll file operation method by passing *poll_table struct*.
- ***poll_table*** structure is implemented in kernel as a table of wait queues owned by driver. ***poll_table*** has elements of type ***poll_table_entry*** structure. Each ***poll_table_entry*** structure has:
 - file struct
 - wait_queue_head_t pointer
 - wait queue.

Application Code - User application using poll() instead of read() - example

```
#define EXPIRE 1000          /*poll time out */
#define BUFSIZE 40
int main(void) {
    struct pollfd fds[1];
    int rfd;
    ssize_t ret;
    int total_bytes = BUFSIZE;
    char rbuf[BUFSIZE];

    rfd = open("/dev/poll_dev", O_RDONLY);
    fds[0].fd = rfd;
    fds[0].events = POLLIN;
    while (1) {
        ret = poll(fds, 1, EXPIRE * 1000);          /*blocking for event on fd for 100 seconds */
        if (ret == -1)
            { perror ("poll"); return -1; }
        if (!ret) {
            printf("%d seconds elapsed, \n", EXPIRE); return 0;
        }
        if (fds[0].revents & POLLIN){ printf("fd is readable\n");
        ...
        } // keep calling poll after servicing events.
    }
    return 0;
}
```

Driver poll method –example:

```
static DECLARE_WAIT_QUEUE_HEAD(drv_wait);
```

```
static unsigned int drv_poll (struct file *file, poll_table *wait)
```

```
{
```

```
/*Driver calls poll_wait() to place the task into one of the wait queue owned by driver. There  
can be multiple wait queues own by driver*/
```

```
poll_wait(file, &drv_wait, wait);
```

```
/** Availability of data
```

```
A driver file operation method routine may call wake_up_interruptable(&drv_wait) to wake  
up the drv_poll(). It may also be called by driver's interrupt routine
```

```
*/
```



```
/* When Driver has data for application to read, it calls */
```

```
If (data_is_available()) return (POLLIN | POLLRDNORM);
```

```
...
```

```
/* When Driver is ready to accept data, it calls */
```

```
if (data_can_be_written()) return (POLLOUT | POLLWRNORM);
```

```
return 0;
```

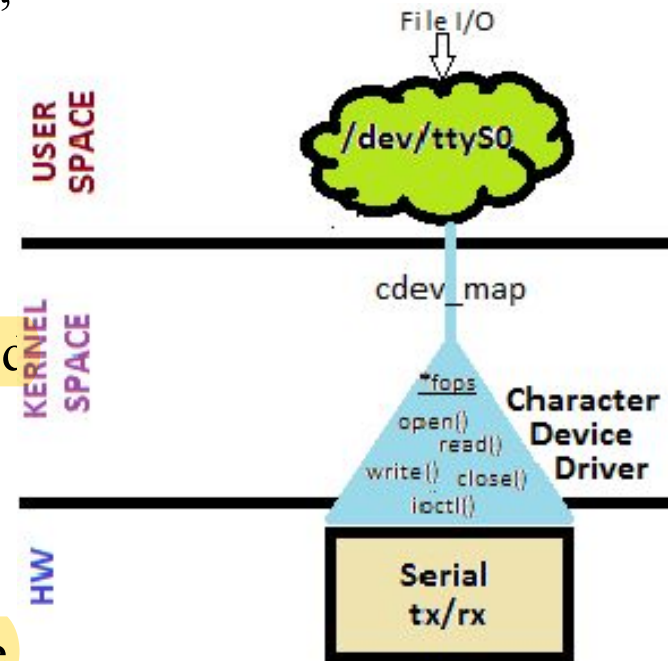
```
}
```

Character Drivers

- Simple to write!
- Offers byte-oriented interface. Usually can be accessed sequentially
- Catch all for many other types of devices that don't fit into any other class. Devices that can be considered as files
 - monitor, keyboard serial and parallel ports are example of character devices
- Cannot “seek” forward or backward through the data the way you can do with block device
- Data transfers without a specific device address. In contrast, block device drivers can address specific location on the device

Character Drivers

- struct ***file_operations*** is used to implement read, write, poll, open methods of char driver.
- struct ***cdev*** structure encapsulates the file_operations structure and some other important driver information (major/minor)
- After a call to ***cdev_add()***, your char device is ready and file operation methods can be invoked via system call
- cdev is embedded in the inode struct. inode struct is used to represent a file where as cdev structure represents char devices. So cdev field in the inode struct is a pointer to char device file.



cdev_init(): initializes the cdev structure with defined file operation methods

cdev_add(): add a char device to system. Device is ready to be used. file operation methods can be called

cdev_del: remove a char device from the system

Character Drivers

Device Node Management

- Both device node (/dev) management and major numbers assignment have been changed from static to dynamic model. It is part of Linux Device Model
- Thus, you can assign a major number statically (old) or dynamically (preferred):
 - Static:** if you always want the same major number, then use MKDEV and register it via `register_chrdev()`

```
dev = MKDEV (DEV_MAJOR, DEV_MINOR);  
ret = register_chrdev(dev, "mydev", &mydev_fops);
```
 - Dynamic:** Request dynamic allocation of device major number with one minor

```
alloc_chrdev_region(&dev, 0, 1, "mydev");  
int major = MAJOR(dev);
```
- Device nodes are also created and removed dynamically at driver load and unload time. To create nodes in /dev and /sys kernel sends uevents to udev daemon for creating /dev nodes:
 - Driver calls `class_create()` to create a sysfs entry for the device
 - Driver calls `device_create()` to generates a uevent that udevd listens and creates a matching device node

```
struct class *my_dev_class  
my_dev_class = class_create(THIS_MODULE, "mydev");  
device_create(my_dev_class, NULL, dev, NULL, "mydev", 0);
```

Character Drivers

init and exit functions

- **init** routine is responsible for initializing the device and registering or linking the device to the rest of the kernel.
- **exit** routine role is to undo things done in init routine. It releases any resources module has allocated and deregisters the driver from the appropriate kernel subsystems.
- In order for module to be loaded and unloaded, driver should have init and exit functions:

```
int __init xxx_init (void);
```

```
static void __exit xxx_cleanup (void);
```

- For kernel to find the driver init and exit functions, driver should call the routines below with its init and exit routines as arguments.

```
module_init(xxx_init);
```

```
module_exit(xxx_cleanup);
```

These macros adds a special section to the module's object code stating where the initialization function is to be found.

Character Drivers

init and exit functions

init function:

- Allocate cdev structure and then connect driver's entry points (file operation methods) to cdev struct.

```
my_cdev = cdev_alloc();  
cdev_init(my_cdev, mydev_fops);
```

- Owner field of file operation structure is set to the address of this module. This helps kernel to keep track of open or release count of the char device

```
my_cdev->owner = THIS_MODULE;
```

- Connect driver's major number to cdev structure

```
cdev_add(&my_cdev, dev, 1);
```

exit function:

- Release the memory associated with cdev and release the major number:

```
cdev_del(&my_cdev);  
unregister_chrdev_region(MAJOR(dev), 1);
```

- Destroy the sys class and remove the device nodes

```
device_destroy(mydev_class, MKDEV(MAJOR(device_number), 1));  
class_destroy(mydev_class);
```

Character Drivers – open/release

- When the device node is opened by application, kernel invokes driver's `open()` routine.
- One can simply trigger open method of a device by doing `cat`:

```
# cat /dev/my_dev/0
```
- Driver open routines takes inode and file pointer as argument:

```
int mydev_open(struct inode *inode, struct file *file)
```
- Driver normally use private_data field of *struct file* for keeping the state information and access that information from other entry points.
- When application calls `close()` on the file descriptor associated to the device, driver `release()` file operation method is called. Same arguments are passed