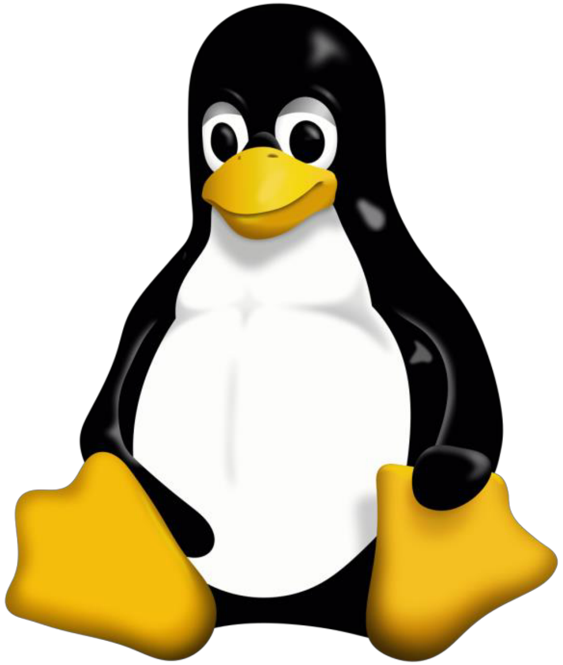


# Linux Character Device Driver

*Sunbeam Infotech*

# Char device operations

- Char device driver provide char device operations and register with the system.
- Driver device operations can be do one of the following:
    - Overwrite default function
    - Provide additional functionality
    - Minimal placeholder
    - NULL – use default/no implementation
- These operations are specified as callback functions in struct file_operations and associated with struct cdev. All functions have pre-defined prototype and purpose.
    - int (*open)(struct inode *, struct file *);
    - int (*release)(struct inode *,struct file *);
    - ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
    - ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
    - loff_t (*llseek)(struct file *, loff_t, int);
    - long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    - …

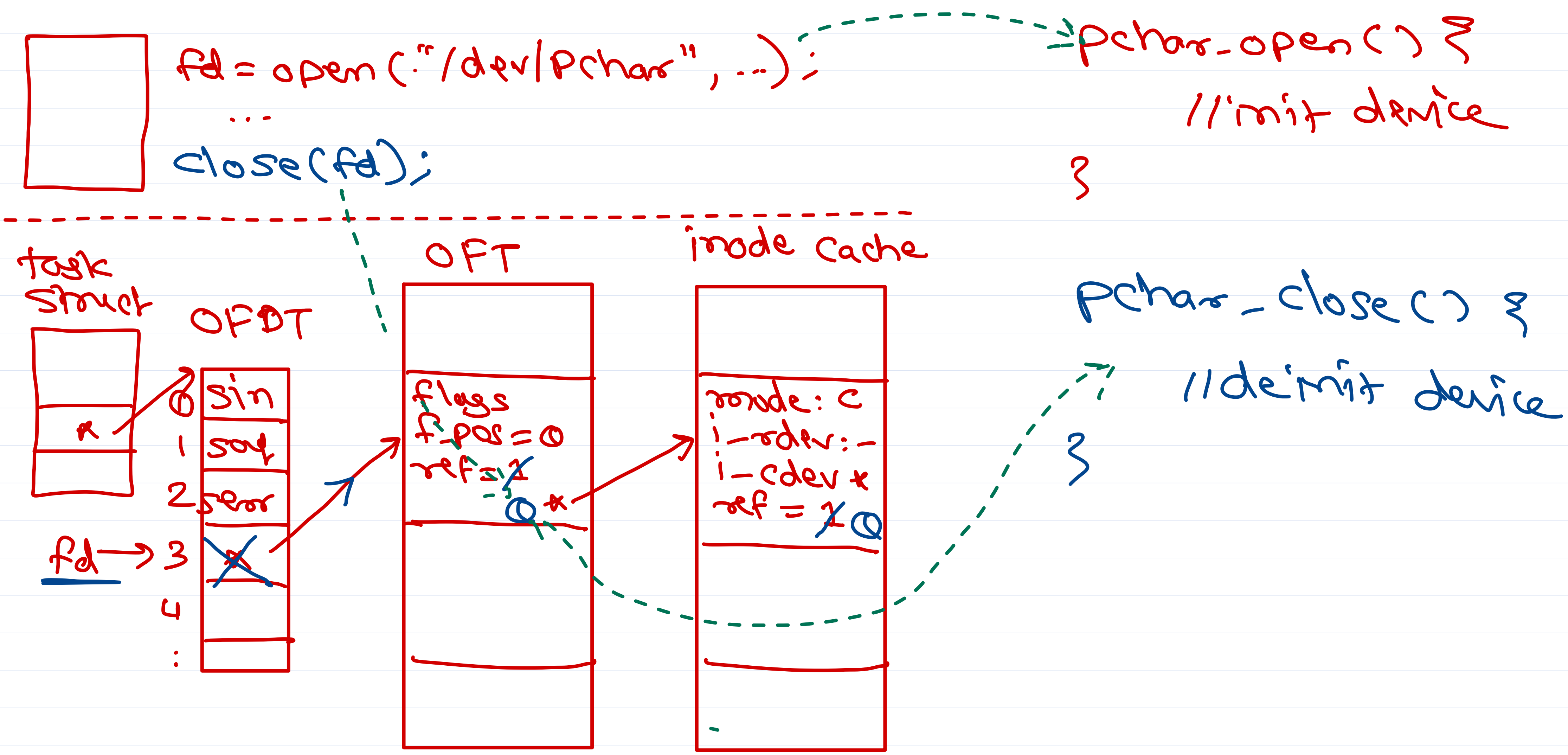
# open() operation

- Every driver/file system must have this function directly or indirectly.
- This opens a device and makes it ready for IO.
- The typical implementation contains initialization of hardware device, registering its IRQs, allocating internal buffers, etc.
- If unimplemented, kernel always give default function and never fails. Neither it notifies driver about this.
- In pseudo char device driver, there is no physical device involved and hence this function is kept empty.

for each call to open() syscall from user space, the open() device op is executed once.

# release() operation

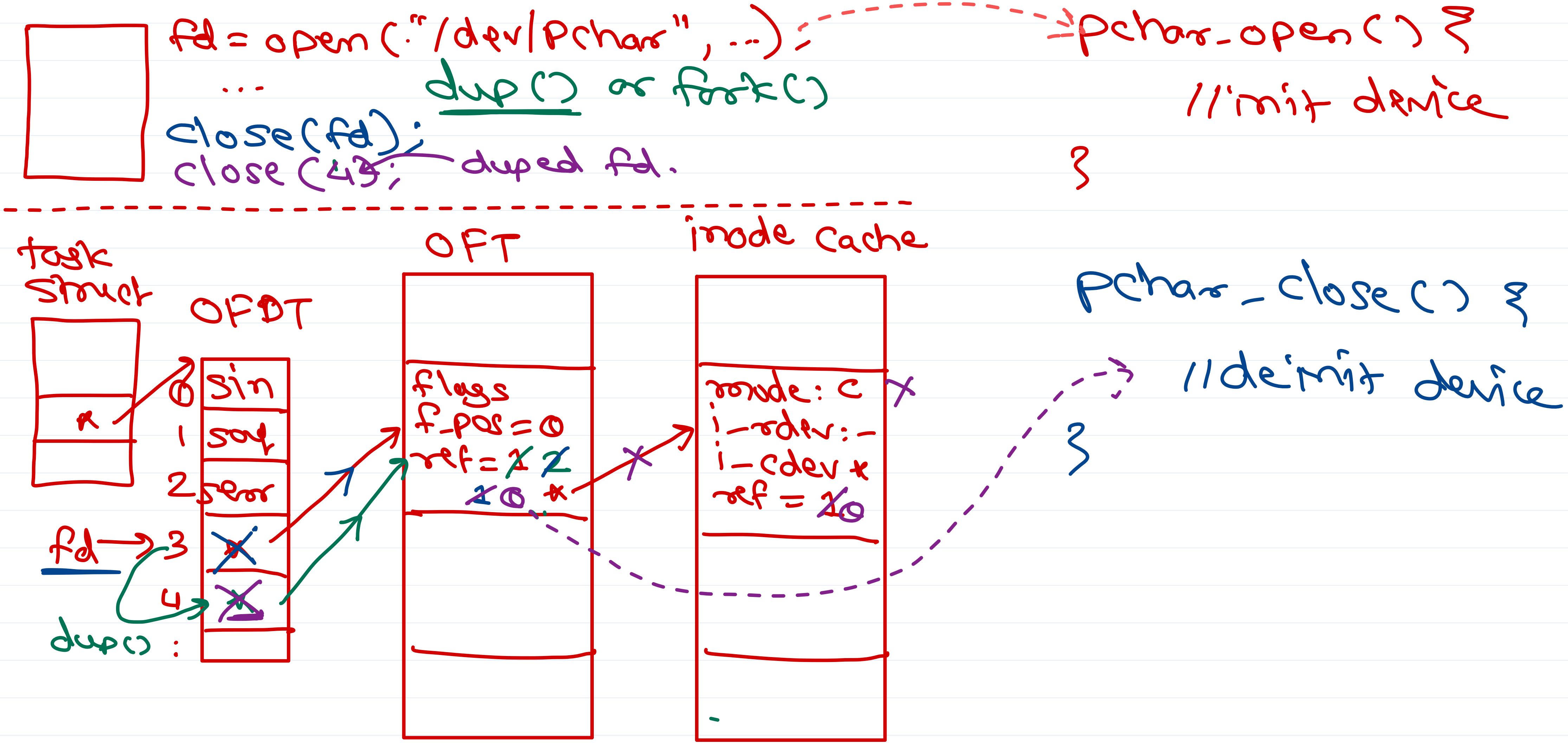
- Closes the device, decrement reference count & releases memory synchronously.
- The typical implementation release all the resources allocated in open() operation.
- If unimplemented, these things are done by the kernel in its default function.
- Not every close() syscall causes release() operation to be invoked
  - When dup() or fork() is called, no new struct file is created; rather only increments reference count in it.
  - Each call to close() decrements the reference count.
  - Only if count drops to 0, the release() operation is invoked.
  - This ensures that only one release() is called per open().
- In pseudo char device driver, there is no physical device involved and hence this function is kept empty.



```
fd = open("/dev/pchar", ...);
...
close(fd);
```

```
pchar_open() {
    //init device
}

pchar_close() {
    //deinit device
}
```

# release() called when OFT entry's ref count become zero - dup() case



```
fd = open("/dev/pchar", ...);
...
close(fd);
close(4);   duped fd.
```

dup() or fork()

```
pchar_open() {
    //init device
}

pchar_close() {
    //deinit device
}
```

# read() operation

- Reads the device and transfers data from device to user-space.
- If kept NULL, read() syscall (from user-space) will fail.
- pseudo char device driver read() implementation:
  - Get how many bytes of data is left in buffer.
  - Decide number of bytes to be copied in user space buffer (min of available and user buffer length).
  - If bytes to read is 0, then obviously it means no data left in buffer (EOD).
  - Copy bytes from device buffer to user buffer using copy_to_user() from current file position.
  - copy_to_user() returns number of bytes not copied. Calculate number of bytes successfully copied.
  - Modify the file position.
  - Return number of bytes successfully read.

# write() operation

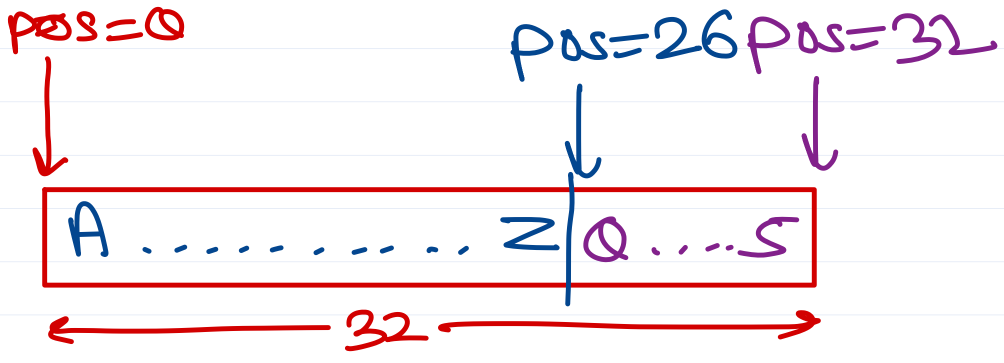
- Writes the device and transfers data from user-space to device.
- If kept NULL, write() syscall (from user-space) will fail.
- pseudo char device driver write() implementation:
  - Get how many bytes of empty space left in buffer.
  - Decide number of bytes to be copied from user space buffer (min of empty space and user buffer length).
  - If bytes to write is 0, then it means no space left in buffer (ENOSPC).
  - Copy bytes from user buffer to device buffer using copy_from_user() from current file position onwards.
  - copy_from_user() returns number of bytes not copied. Calculate number of bytes successfully copied.
  - Modify the file position.
  - Return number of bytes successfully written.

return 26

```
-write(fd, "A...Z", 26);
-write(fd, "0...9", 10);
-write(fd, "DESD", 4);
```

return 6

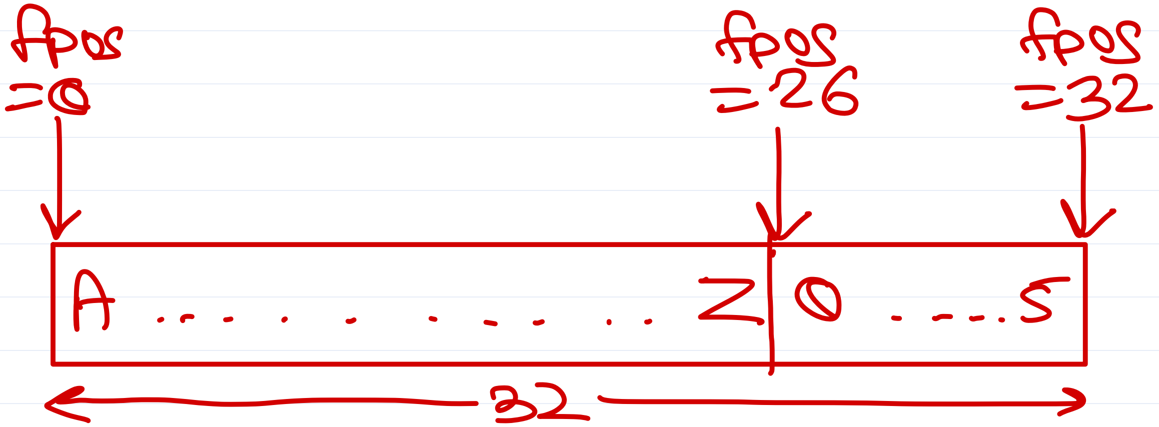
```
pchar_write(pfile, ubuf, bufsize, poffset) {
    avail = SIZE - *poffset;
    towrite = avail < bufsize ?
                        avail : bufsize;
  nbytes= to-write
        -copy_from_user(
                buffer + *poffset,
                ubuf, towrite);

      *poffset += nbytes;

      return nbytes;
}
```



buffer

```
read(fd, buf, 26);
read(fd, buf, 10);
read(fd, buf, 4);
```

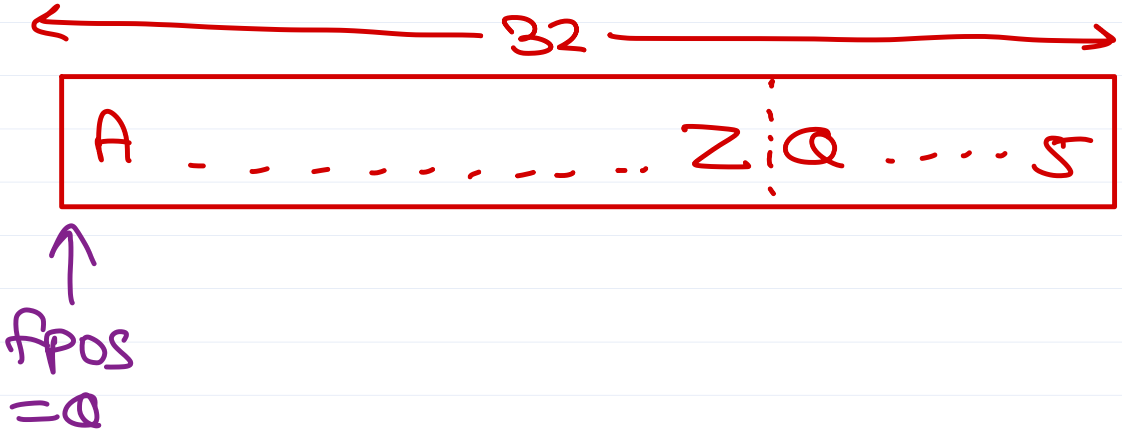
```
pchar_read(pfile, ubuf, bufsize, poffset)
{
    avail = SIZE - *poffset;
    if (avail == 0) {
        return 0; // end of dev buffer
    }
    toread = MIN(avail, bufsize);
    nbytes = toread -
        copy_to_user(ubuf, buffer +
            *poffset, toread);
    *poffset = *poffset + nbytes;
    return nbytes;
}
```



# llseek() operation

- Modify current file position of the device.
- Pseudo char device driver llseek() implementation:
  - Calculate new position depending on offset and origin.
    - If origin is SEEK_SET (0), new pos = 0 + offset. (offset is +ve)
    - If origin is SEEK_CUR (1), new pos = cur pos + offset. (offset is +ve/-ve)
    - If origin is SEEK_END (2), new pos = ~~cur pos~~ size + offset. (offset is -ve)
  - Ensure that new file position is valid. Otherwise do the necessary adjustment. or return error
  - Return the new file position.

```
pchar_lseek(pfile, offset, origin) {
    switch(origin) {
        case SEEK_SET: new pos = 0 + offset; break;
        case SEEK_END: new pos = MAX + offset; break;
        case SEEK_CUR: new pos = pfile->f_pos + offset; break;
    }
    //validate & adjust pos
    pfile->f_pos = new pos;
    return newpos;
}
```



user space lseek() calls:

ret = lseek(fd, offset, origin);

- ret
  - ↳ newpos w.r.t. start of file
  - ↳ error
- offset
  - ↳ 0
  - ↳ +ve
  - ↳ -ve
- origin
  - ↳ 0-SET
  - ↳ 1→CUR
  - ↳ 2→END

① lseek(fd, 0, SEEK_SET); → 0

② lseek(fd, 0, SEEK_END); → size

③ lseek(fd, 0, SEEK_CUR); → cur pos.

④ lseek(fd, 10, SEEK_SET); → 10

⑤ lseek(fd, -10, SEEK_END); → 22

⑥ lseek(fd, -10, SEEK_CUR);

⑦ lseek(fd, 40, SEEK_SET);
↳ return error or
take fpos to end of file.

⑧ lseek(fd, -40, SEEK_END);
↳ return error or
take fpos to start of file.

# Using kfifo in char device driver

- Instead of using fixed sized buffer, it is recommended to use kfifo for better memory utilization.
- Can be allocated statically (using DEFINE_FIFO) or dynamically (using kfifo_alloc()).
- kfifo is commonly used to store data
  - from hardware device (before sending to user space).
  - from user space (before sending to hardware device).
- Copying data from or to user space directly is possible using kfifo_from_user() and kfifo_to_user().
- If synchronization is needed, kfifo_in_spinlocked() and kfifo_out_spinlocked() can be used.

# ioctl() operation

- read(), write() are typical IO operations on the device.
- ioctl() system call: #include <sys/ioctl.h>
  - int ioctl(int fd, unsigned long cmd, ...);
- ioctl() is special ad-hoc operation that can be used for arbitrary purposes.
  - Manipulating device state directly.
  - Monitoring device state (debugging).
  - Direct hardware control operations.
- Example: handling CD-ROM using ioctl().
  - https://www.kernel.org/doc/Documentation/ioctl/cdrom.txt
  - e.g. ioctl(fd, CDROMEJECT, 0);
- Newer kernel version replace ioctl() with unlocked_ioctl() implementation.
  - long (*unlocked_ioctl)(struct file *pfile, unsigned int cmd, unsigned long param);

cdrom_test.c

```
#include <sys/ioctl.h>
int main() {
    int fd = open("/dev/sr0", O_RDONLY|
                               O_NON BLOCK);
    ioctl(fd, CDROMEJECT, 0);

    close(fd);
    return 0;
}
```

*Thank you!*

Nilesh Ghule <nilesh@sunbeaminfo.com>