50004 - Operating Systems - Lecture 12

Oliver Killane

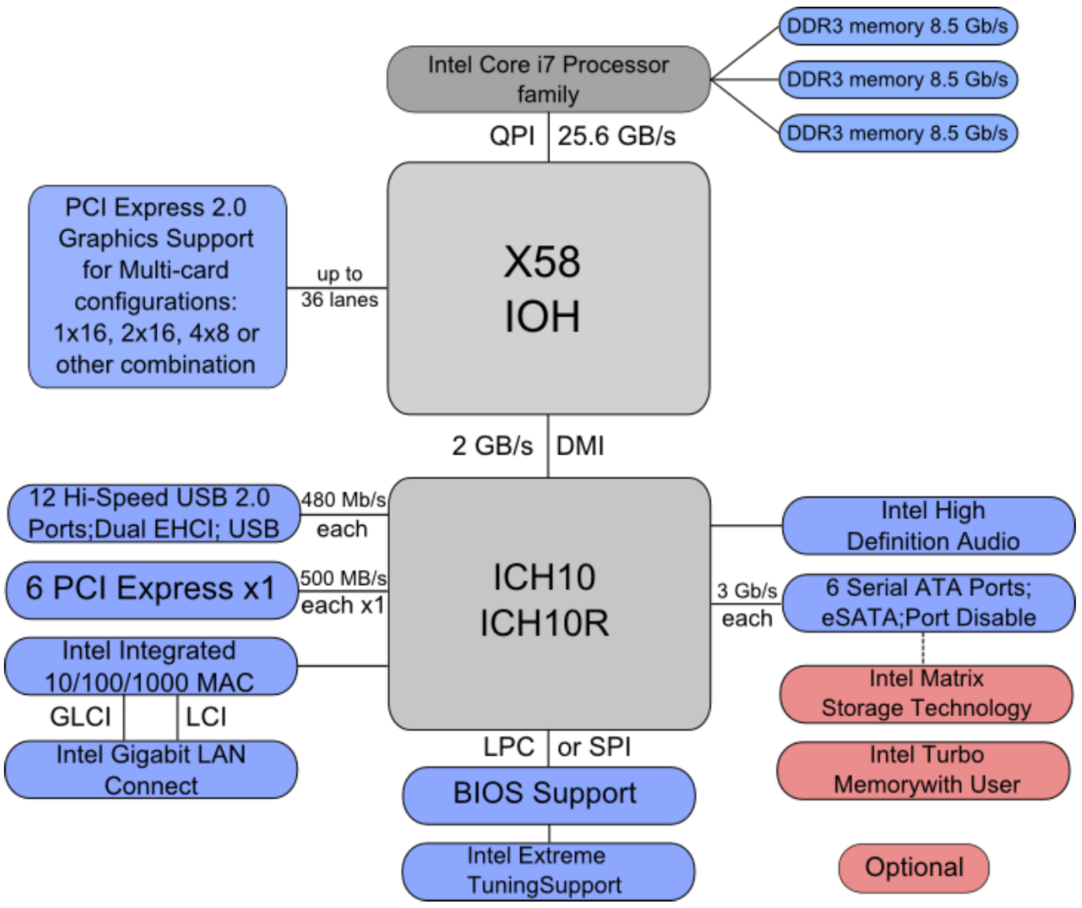
25/11/21

Lecture Recording

Lecture recording is available here

Device Management

Intel Example



Intel Core i7 Processor family
DDR3 memory 8.5 Gb/s
DDR3 memory 8.5 Gb/s
DDR3 memory 8.5 Gb/s
QPI
25.6 GB/s
PCI Express 2.0 Graphics Support for Multi-card configurations: 1x16, 2x16, 4x8 or other combination
up to 36 lanes
X58 IOH
2 GB/s
DMI
12 Hi-Speed USB 2.0 Ports;Dual EHCI; USB
480 Mb/s each
6 PCI Express x1
500 MB/s each x1
Intel Integrated 10/100/1000 MAC
GLCI
LCI
Intel Gigabit LAN Connect
ICH10 ICH10R
Intel High Definition Audio
3 Gb/s each
6 Serial ATA Ports; eSATA;Port Disable
Intel Matrix Storage Technology
Intel Turbo Memorywith User
LPC
or SPI
BIOS Support
Intel Extreme TuningSupport
Optional

North & South Bridge

Here the **X58** connects the CPU to high-speed peripherals through the high bandwidth **QuickPath Interconnect** (**QPI**). This is the **North Bridge** which connects the CPU to peripherals directly.

The **ICH10** (I/O Controller Hub) supports lower speed peripherals, and connects to the CPU through the **North Bridge**.

I/O Device Management Objectives

- **Fair Access to Shared Devicess** Prevent processes hogging resources
- **Exploit Parallelism**
  Can use devices in parallel (e.g send packets using network card while writing to disk), and some devices have parallelism themselves.
- **Provide uniform & Simple view of I/O**
  Abstract devices away from processes (e.g filesystem, not disk). And use uniform interfaces (when new devices added, programs do not need to change to utilise).
  - Uniform naming and error handling.
  - Hide complexity of device handling.

Device Independence

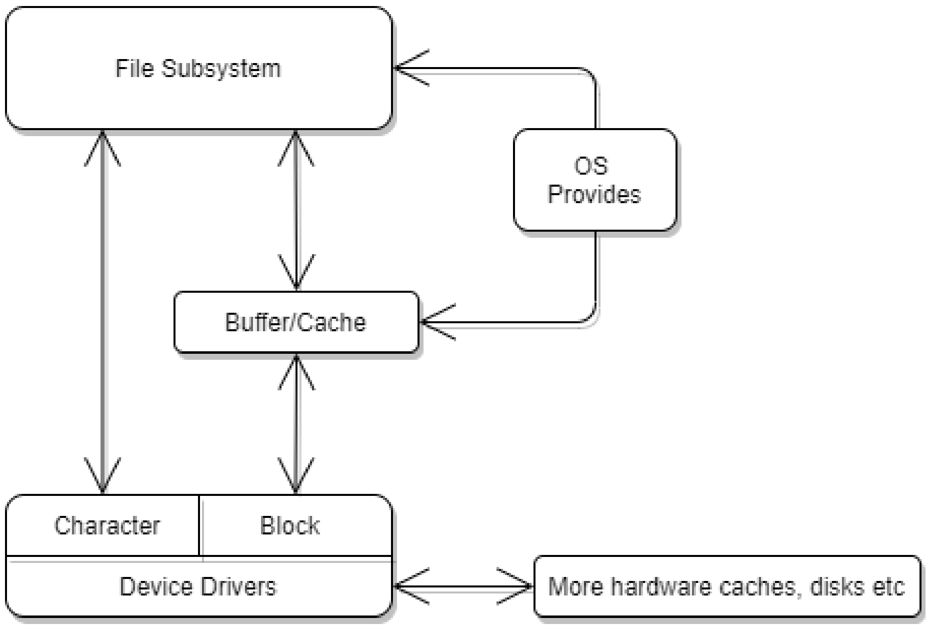
Have the device be independent from its type (e.g terminal, disk, dvd drive) and which instance (e.g disk 1, 2, 3).

E.g can use the same interface for all disks connected to a system. Can read data from dvd drive, disk, terminal in a single interface (sometimes we split into classes when differences are large enough).

Device Variations

- **Unit of Data Transfer** Character (bytes) or block
- **Supported Operations** e.g read, write, seek
- **Synchronous or asynchronous** e.g network card (send request, then get result back sometime), disk (read and block until the data is read)
- **Speed differences** e.g NVMe SSD vs Tape Drive
- **Shareable** e.g disks are shareable, printers are not (can print one at a time)
- **Types of error conditions** e.g Disk errors, vs GPU temperature warnings

Character vs Block Device



Note the type (character/block) depends on the type of device.

Block	Maximize throughput	Disks, network cards etc
Character	Minimize latency	Keyboard, terminal etc

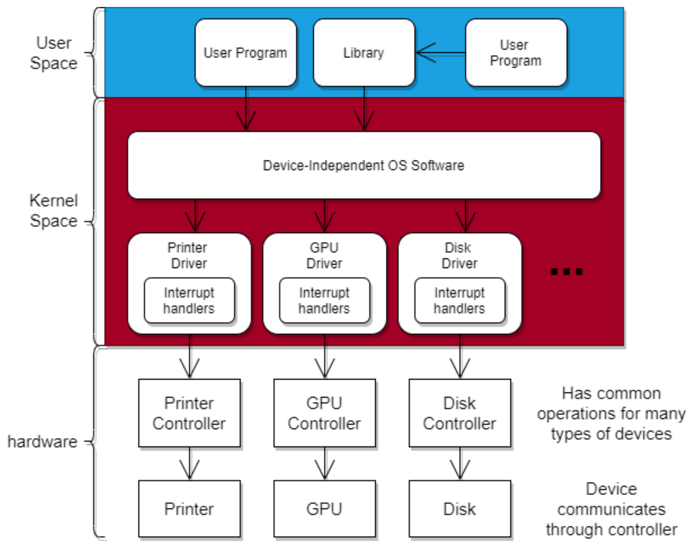
Character Devices:

1	mem	Device file representing the physical memory of the os.
2	pty	Pesudoterminal (bidirectional communication channel).
180	usb	USB devices.

Block Devices:

1	ramdisk	access ram disk in raw mode.
2	fd	Floppy Disk.
7	loop	Loop device, maps data blocks to a file in the filesystem, or another block device.

I/O Layers



Interrupt Handler

An interrupt is a signal sent from a device to the CPU to inform it that the device needs attending to (e.g device connecting, finished reading, error has occured etc.).

Drivers register handlers to deal with types of interrupts, when the CPU receives an interrupt, it runs the relevant handler.

- For block devices, on transfer completion signal device handler.
- For character devices, when character transfered, process next character.

For modern systems (e.g nvme storage) a hybrid approach makes use of the following:

- Polling (queue of requests and responses), very fast but uses CPU time (analogous to spin locks)
- Wait for interrupt (better for longer waits - another process can be scheduled)

Device Driver

Handles a type of device, can control multiple devices of the same type.

- Implements block read/write
- Access device registers (writing control information to a device)
- Initiating Operations (start device e.g at boot)
- Scheduling requests (if a device is shared, placing in order, often for performance - order hard disk operations to do the least disk traversal).
- Handle Errors

Device Independent OS Layer

Use standard interfaces for drivers of device types:

- Simplifies OS design.
- Interface to write new drivers to.
- No OS changes required to support new drivers, just support interfaces.

This layer also provides **device independence**:

- **Map Logical to Physical Devices** (Naming and Switching)
  Can map one logical device to many physical, or vice versa (e.g disk RAIDs).
- **Request Validation against device**
  Check device & driver is working correctly.
- **Allocation**
  Determine which processes can access which logical devices.

  Control **dedicated allocation** (process exclusive access to a device)
- **Buffering**
  As previously mentioned - for performance & block size independence.
- **Error Reporting**

User-Level I/O Interface

System call interface to allow user programs to interact (often through other 3rd party libraries).

- basic I/O operations (**close**, **read**, **write**, **seek**)
- Sets up parameters (device independent)
- Can be synchronous (blocking) or asynchronous (non-blocking)

Unix files

Unix accesses virtual devices as files. This allows access to devices using the normal standard input/output calls. For example the common:

File Descriptor	Name
0	Standard Input
1	Standard Output
2	Standard Error

There are also files such as **/dev/kdb** for keyboard access.

Device Allocation

- **Dedicated Device** (e.g DVD writer, terminal, printer)
  - One process gets exclusive access to a device.
  - Typically allocated for long periods of time (e.g minutes - hours: printing, or controlling a terminal display output)
  - If another process tries to access, it fails (potentially adding to a queue of open requests).
  - Only allocated to authorised processes (e.g don't want malicious processes blocking access to a resource).

- **Shared Device** (e.g disks, window terminals etc)
  OS can provide systems for sharing, e.g file system accessible by all processes makes use of the disk.
- **Spooled Device** (e.g printer, dvd writer - many other dedicated devices)
  A shared pool. A **daemon** process has dedicated access, processes send requests/jobs to the **daemon**.
  - Provides sharing on non-sharable resources.
  - Reduces I/O time resulting in greater throughput.

Buffered vs Unbuffered I/O

- **Buffered**

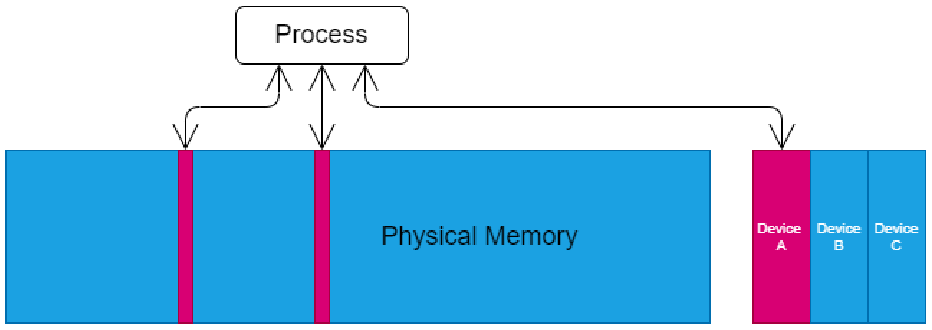
Output	User data transferred to OS output buffer. Process continues and only suspends once buffer is full.
Input	OS reads ahead. Reads taken from buffer, process blocks when buffer empty.

  - Smooths peaks in I/O traffic (allows for limited load balancing).
  - Can allow for different data transfer unit sizes between devices (e.g buffer contains blocks).
- **Unbuffered**
  - Data Transfered directly between device and user space.
  - Each read/write causes physical I/O (device does something, not just accessing a hidden buffer).
  - Device handler used for every transfer.
  - High switching overhead (e.g every read requires the driver to take over, and to do some physical action).

Device Drivers

Memory Mapped I/O

Device can be addressed as a special memory location.



Hence we can use the virtual memory setup to restrict access (e.g set supervisor bit to prevent user access).

I/O

- **Programmed I/O** (simple but inefficient)
  Wait for device (spin), then continue execution.
- **Interrupt Driven** (large overhead, good for long expected waits)
  Hardware sends an interrupt when operation completed, can do other work while waiting.
- **Direct Memory Access (DMA)** (requires hardware, but reduces CPU intervention)
  A DMA controller (often in the device) waits for the device to respond, then once the full result is available, places this directly into memory.

Linux

Loadable Kernel Module (LKM)

Device drivers are loadable modules that are loadeed and linked dynamically with the running kernel.

This requires binary compatability (module must be specific to kernel version).

Linux uses **Kmod**

- Kernel subsystem that manages modules without user intervention.
- Determines modules dependencies.
- Loads modules on demand (e.g load driver for network card when it is connected to the system).

Basic LKM module

```c
/* Used for all initialisation code. */
int init_module (void)
{
    ...
}

/* Used for clean shutdown */
void cleanup_module (void)
{
    ...
}
```

Kernel can open file, look for symbol table (generated by compiler), and call the corresponding functions.

```bash

# insmod loads a module to the kernel
# 'sudo' as this operations is restricted to the root user (access to all
# commands and files).
sudo insmod some_module.o
```

IO Management

/dev /sys /proc

Linux (in UNIX fashion) uses files to represent many drivers, services & methods to collect system information. Some of the main directories for this are:

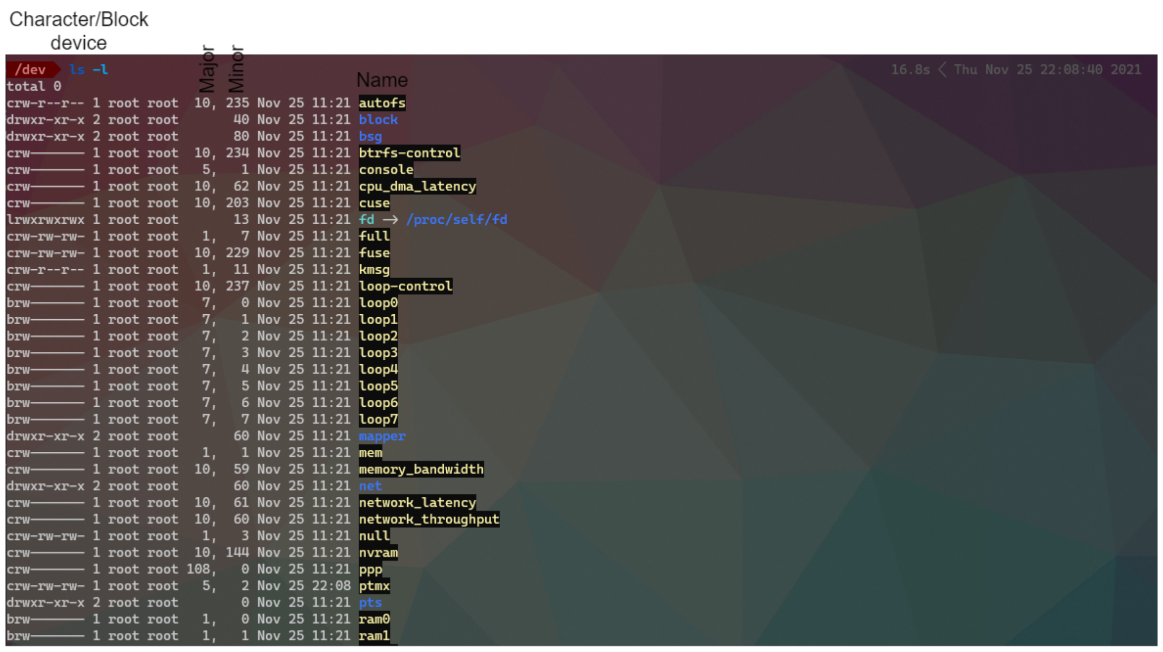
- **/dev** device file directory
  Contains files for devices (virtual included).
- **/proc** virtual filesystem for processes
  Contains information on running processes, each represented by files of size 0. Each numbered directory is actually the **pid** of a running process.

  Other files can be read to get system information such as:

/proc/meminfo	Information on the memory system including free pages, kernel stack pages etc.
/proc/interrupts	Number of interrupts received for categories.
/proc/stat	System status information (e.g number of processes).
/proc/version	OS version information.

- **/sys**
  A filesystem like view of the kernel and its configuration/settings.

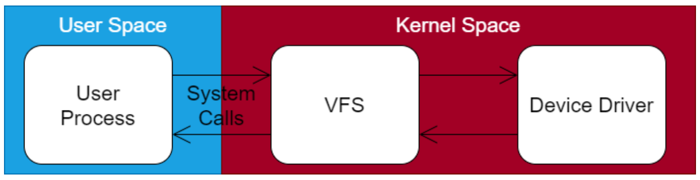
- **Device Classes** Group similar types of devices (similar function, performance needs)
- **Identification Numbering** $< Major >:< Minor >$
  Major Determines which diver is controlling. (e.g IDE Disk Drive, Serial port etc.)
  Minor Distinguishes devices of the same class. (e.g Drive Number)
  Examples from kernel documentation are here.
- **Special Files**
  Most devices are represented by device/special files in the **/dev** directory.



```
/dev  ls -l
total 0
crw-r--r-- 1 root root  10, 235 Nov 25 11:21 autofs
drwxr-xr-x 2 root root       40 Nov 25 11:21 block
drwxr-xr-x 2 root root       80 Nov 25 11:21 bsg
crw------- 1 root root  10, 234 Nov 25 11:21 btrfs-control
crw------- 1 root root   5,   1 Nov 25 11:21 console
crw------- 1 root root  10,  62 Nov 25 11:21 cpu_dma_latency
crw------- 1 root root  10, 203 Nov 25 11:21 cuse
lrwxrwxrwx 1 root root       13 Nov 25 11:21 fd -> /proc/self/fd
crw-rw-rw- 1 root root   1,   7 Nov 25 11:21 full
crw-rw-rw- 1 root root  10, 229 Nov 25 11:21 fuse
crw-r--r-- 1 root root   1,  11 Nov 25 11:21 kmsg
crw------- 1 root root  10, 237 Nov 25 11:21 loop-control
brw------- 1 root root   7,   0 Nov 25 11:21 loop0
brw------- 1 root root   7,   1 Nov 25 11:21 loop1
brw------- 1 root root   7,   2 Nov 25 11:21 loop2
brw------- 1 root root   7,   3 Nov 25 11:21 loop3
brw------- 1 root root   7,   4 Nov 25 11:21 loop4
brw------- 1 root root   7,   5 Nov 25 11:21 loop5
brw------- 1 root root   7,   6 Nov 25 11:21 loop6
brw------- 1 root root   7,   7 Nov 25 11:21 loop7
drwxr-xr-x 2 root root       60 Nov 25 11:21 mapper
crw------- 1 root root   1,   1 Nov 25 11:21 mem
crw------- 1 root root  10,  59 Nov 25 11:21 memory_bandwidth
drwxr-xr-x 2 root root       60 Nov 25 11:21 net
crw------- 1 root root  10,  61 Nov 25 11:21 network_latency
crw------- 1 root root  10,  60 Nov 25 11:21 network_throughput
crw-rw-rw- 1 root root   1,   3 Nov 25 11:21 null
crw------- 1 root root  10, 144 Nov 25 11:21 nvram
crw------- 1 root root 108,   0 Nov 25 11:21 ppp
crw-rw-rw- 1 root root   5,   2 Nov 25 22:08 ptmx
drwxr-xr-x 2 root root        0 Nov 25 11:21 pts
brw------- 1 root root   1,   0 Nov 25 11:21 ram0
brw------- 1 root root   1,   1 Nov 25 11:21 ram1
```

Device Access

Device files are accessed via the **virtual file system (VFS)**.



Most drivers implement **read**, **write**, **seek** etc (much like a file), however all contain other operations which do not fit this abstraction.

Linux uses the **ioctl** (I/O Control) system call to support special tasks (e.g getting printer status, ejecting a CD drive)

```c
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);

/* for example to eject a CD-ROM */
ioctl(cddrom, CDROMEJECT, 0);
```

Character Device I/O

- Data transitted as a stream of bytes (read a byte, then another is presented)
- Represented by a char_device_struct structure.
- char_device_struct contains a pointer to a file_operations struct.

The file_operations struct:

- Maintains operations supported by the device driver.
- Stores function pointers to operations (read, write etc).

file_operations in the linux kernel (github).

```c
struct file_operations {
        struct module *owner;

        loff_t   (*llseek)         (struct file *, loff_t, int);
        ssize_t  (*read)           (struct file *, char __user *, size_t, loff_t *);
        ssize_t  (*write)          (struct file *, const char __user *, size_t, loff_t *);
        ssize_t  (*read_iter)      (struct kiocb *, struct iov_iter *);
        ssize_t  (*write_iter)     (struct kiocb *, struct iov_iter *);
        int      (*iopoll)         (struct kiocb *kiocb, struct io_comp_batch *, unsigned int flags);
        int      (*iterate)        (struct file *, struct dir_context *);
        int      (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*poll)           (struct file *, struct poll_table_struct *);
        long     (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long     (*compat_ioctl)   (struct file *, unsigned int, unsigned long);
        int      (*mmap)           (struct file *, struct vm_area_struct *);

    unsigned long mmap_supported_flags;

    int (*open)    (struct inode *, struct file *);
        int (*flush)   (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync)   (struct file *, loff_t, loff_t, int datasync);
        int (*fasync)  (int, struct file *, int);
        int (*lock)    (struct file *, int, struct file_lock *);

    ssize_t       (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);

        int     (*check_flags)(int);
        int     (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int     (*setlease)(struct file *, long, struct file_lock **, void **);
        long    (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *, loff_t, size_t, unsigned int);
        loff_t (*remap_file_range)(
        struct file *file_in, loff_t pos_in,
        struct file *file_out, loff_t pos_out,
        loff_t len, unsigned int remap_flags);

        int (*fadvise) (struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

char_device_struct in the linux kernel (github).

```c
static struct char_device_struct {
        struct char_device_struct *next;
        unsigned int major;
        unsigned int baseminor;
        int minorct;
        char name[64];
        struct cdev *cdev;              /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];

struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
} __randomize_layout;
```

Note: this is a basic example, very interesting!

Block Device I/O

- **Block I/O Subsystem**
  Consists of several layers, with modularised operations (common code ine ach layer).

  The tw main strategies to minimise time accessing block devices:

  1. caching data.
  2. Clustering I/O Operations (store up tasks and execute several at once).

  When given a task, check cache. If data not present, queue request on request queue for device.

- **Direct I/O**
  Bypass the kernel cache when accessing a device.

  Useful for databases and some other applications where caching can reduce performance/-consistency (e.g values very rarely accessed more than once, or doing caching in use level).