50004 - Operating Systems - Lecture 13

Oliver Killane

25/11/21

Lecture Recording

Lecture recording is available here

Linux API

I/O Classes

Character	unstructured	Files and devices
Block	structured	Devices
Pipes	message	Interprocess communication
Socket	message	Network Interface

Sockets

- Allow for bidirectional communication between processes.
- Can be local (e.g processes communicating on the same system), or across the network (e.g connecting to a server) (pipes use machine specific file descriptors, so cannot go over network).
- There are two types (**TCP**, **UDP**)

User Space API

```c
/* Create a file with set permissions. */
fd = create(filename, permission);

/* Open a file/device with mode:
 * 0 - read
 * 1 - write
 * 2 - read/write
 */
fd = open(filename, mode);

/* close a file/device */
close(fd);

/* Attempt to read nbytes from a file/device with descriptor fd to buffer. */
nbytesread = read(fd, buffer, numbytes);

/* Attempt to write nbytes to a file/device with descriptor fd from buffer. */
nbyteswritten = write(fd, buffer, nbytes);

/* Create a pipe */
int fd[2];    /* fd[0] for reading, fd[1] for writing */
pipe(fd);

/* Duplicate the file descriptor. (Uses lowest available) */
newfd = dup(oldfd);

/* Duplicate, using newfd as the new file descriptor. */
newfd = dup2(oldfd, newfd);

/* Issue control command to a device, terminos is an array of control characters */
```

```
ioctl(fd, operation, &terminos);

/* Creates a new special file from a character or block device. */
fd = mknod(filename, permission, dev);
```

File Descriptors

- File descriptors are integers, refering to a file opened by the process.
- Process context sensitive (same descriptor, different process → different file)

Each process has 3 descriptors when starting:
0 Standard Input (**stdin**)
1 Standard Output (**stdout**)
2 Standard Error (**stderr**)
By default these point to the terminal which spawned the process.

Blocking & Asynchronous I/O

- **Blocking** process suspended until operation complete
  The I/O call only returns once completed. As a result from the program's perspective this is instantaneous.

  Process making call could be blocked for a long time, we can use threads to get around this (while one blocked, others can run) however this is complex.

- **Non-Blocking** I/O call returns as much as is available.
  Return immediately (sometimes with data, sometimes with none). By making many calls we can get some data (e.g constantly read until all data is read).

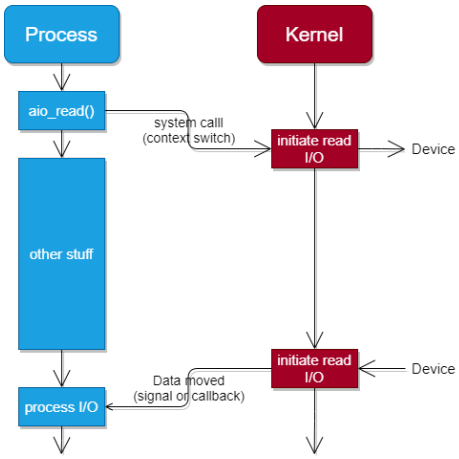
  The provides an application-level polling for I/O (Can constantly request reads from stdin, operating only if there is data at that moment).
  e.g to respond to keystrokes immediately for a terminal game poll stdin by non-blocking reads.

  Turn on using the **fcntl** system call (for sending commands to manage file descriptors, e.g make a file descriptor non-blocking for read/write).

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

- **Asynchronous**



- Process runs in parallel with the I/O operations (Non-blocking)
- When operations is complete the process is notified (callback function called, or process signal sent).
- Process can check/wait for I/O completion
- Flexible & Efficient (non-blocking but does not require lots of polling)
- More complex code (e.g futures/promises)
- Potentially less secure if buffers for recieving data are mismanaged.

Security example: Asynchronous read, process sends pointer to buffer to write to. Process then does other stuff, frees buffer, buffer used for something else later. On callback data is written to freed buffer (used after free).

```
1  #include <aio.h>
2
3  int main(int argc, char **argv) {
4      int fd, ret;
5      struct aiocb my_aiocb;
6
7      fd = open("myfile", O_RDONLY );
8
9      /* Allocate buffer for aio request */
10     my_aiocb.aio_buf = malloc(BUFSIZE + 1);
11
12     /* Create aio control structure */
13     my_aiocb.aio_fildes = fd;
14     my_aiocb.aio_nbytes = BUFSIZE;
15     my_aiocb.aio_offset = 0;
16
17     /* Start read request */
18     ret = aio_read(&my_aiocb);
19
20     /* Wait until request is complete (we could also register a signal, or
21      * thread callback instead & do useful work here)
22      */
23     while (aio_error(&my_aiocb) == EINPROGRESS);
24
25         /* Check result from read */
26         if ((ret = aio_return(&my_iocb)) > 0)
27             printf("Successfully read %s", my_aiocb.aio_buf)
28         else
```

```
29              printf("Failed to read :-(")
30  }
```

Responsibilities

- **Computing track, sector & head for disk read** Device Driver (or hardware)
  Requires information about disk layout, in modern HDDs the disk controller is used at only the hardware knows locations of bad blocks.

  (Bad Block - Area that is no longer functioning properly/reliable - physicall damage or corruption)

- **Maintain cache of recently used blocks** Device Independent OS layer
  Useful across many types of block I/O and can be reused/shared between different devices

- **Write Commands to Drive Registers** Interrupt Handler
  If time-critical can be performed in an interrupt routine, if it requires more time (or is device specific) can be done by device driver.

- **Checking user is allowed to use a device** Device Independent OS layer
  Applicable to many different types of devices, OS can enforce policy uniformly over all devices.

- **Converting integers to ASCII for printing** User-Level I/O Layer
  Data representations managed by a library users link against (e.g for pretty printing), this gives user programs lots of flexibility about how to convert.

  Aslo more performant as requires no kernel involvement (which would require context switches).

Disk Management

History

1956 - IBM 305 RAMAC (First Commercial Hard Disk)

- 4.4MB
- $1.5m^2$
- $160,000

$\longrightarrow$

2005 - Toshiba 0.85" disk (Smallest ever)

- 4GB
- $< $300

Disk capacity has grown exponentially, through access speeds have not kept pace.

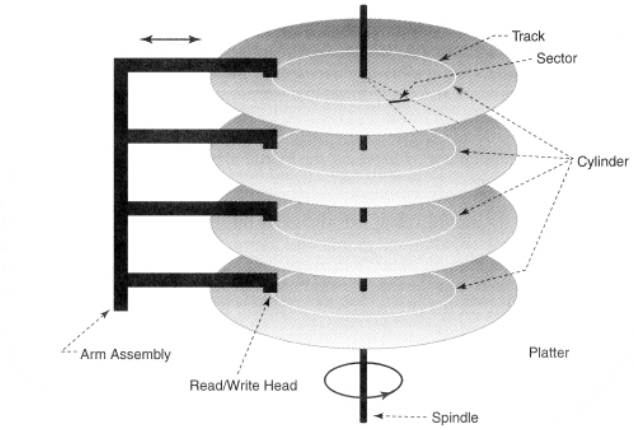
Recently this has been driven by demand from cloud providers & services (e.g Youtube, microsoft 365, facebook etc).

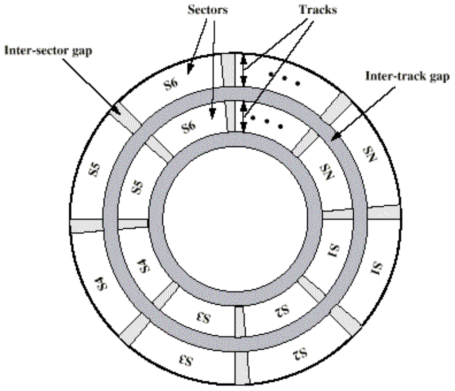
HAMR

Heat Assisted Magnetic Recording. To ensure high rpecision, even when working at very high speeds, the platter is such that it can only be written to when heated.

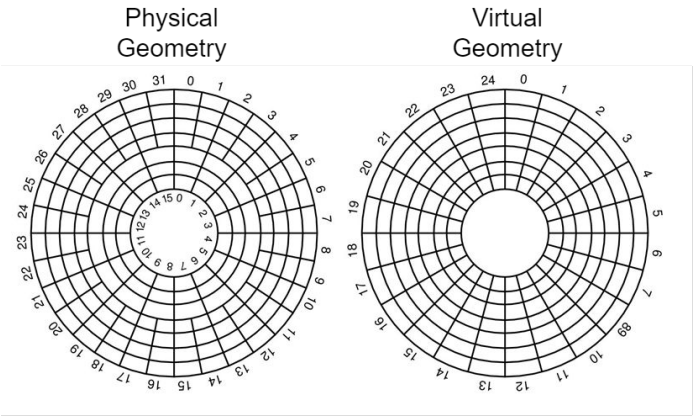
A laser shines just ahead of the read/write head, heating areas that will be written too. Cold areas ignore any reading/writing.

Disk Layout





Sector Layout



- Surface split into tracks (concentric circles)
- Track split into equal size sectors (outer tracks have more sectors)
- Physical address is (cylinder, surface, sector), but this is hidden from the OS.

Modern systems use **logical sector addressing/logical block addresses (LBA)**:

- Sector Numbered consecutively $0..n$

- Makes disk management much easier (just get sector number & offset in sector)
- Helps to work around BIOS limitations (e.g IBM PC BIOS could only address 8GB (6 bits sector, 4 head, 14 cylinder), this is too small).

Capacity

Some sources will use 1000, others 1024 as base for KB, MB, GB, TB. Ensure you are consistent with which base you use during exams.

Disk Formatting

Low Level Format

- **Preamble** Identify the type of block
- **Data** For storage
- **ECC** Error Correction Codes (to correct small errors in read/write)

Some techniques are also used such as **interleaving** (for older systems, sequential data spread apart such that CPU has time to process a sector, before the next one is read) and cylinder skew (cylinder skew of n means when the head is at sector x on one surface, on the next its at sector $x + n$).

High Level Format

- **Boot Block** A block (usually at start of disk) that is dedicated to starting the system.
- **Free Block List** Stores which blocks are currently in use, and which are free to allocate.
- **Root Directory** Start of the file system, all subdirectries emenate from here.
- **Empty File System**

Example Question

A disk controller with enough memory can perform **read-ahead**, reading blocks before the CPU has requested them.

Should it also do **write-behind** (writing to controller memory, informing CPU it is done, only to write back later)

Answer: Not in general - as disk controller memory is still volatile, so in a power loss the data would be lost. (Method around is a small battery large enough to write back in case of a power failure).

Disk Delay

Sector Size	521 bytes	
Seek time	adjacent cylinder (<1ms), avg (8ms)	(Move to correct track)
Latency/Rotation time	$4ms$	(Spin platter to get to beginning of block)
Transfer time	$> 100MB/s$	(Spin platter over block being read)

Seek time is $2-3$x larger than latency time

Disk Scheduling

- Minimise seek/latency times
- Order pending requests to take advantage of head position

Disk Performance

Where:

- b - bytes to transfer
- N - bytes per track
- r - rotation speed in r/s (rotations per second)

- **Seek Time** t_{seek}
- **Transfer Time** $t_{transfer} = \dfrac{b}{r \times N}$
  At most need half a revolution, so $\dfrac{1}{r}$ is seconds per revolution.
- **Total Access Time** $t_{access} = t_{seek} + \dfrac{1}{2 \times r} + \dfrac{b}{r \times N}$
  Seek + latency time (to get to start worst case rotate half) + How much to rotate while reading (e.g reading x bytes from track size $2x$ requires a half rotation)