0 Contents

1 C

1.1 fixed size types

- int8_t, int16_t, int32_t, int64_t
  fixed integers with bit count
- intmax_t max size int on platform
- intptr_t signed integer with the size of an address on this platform
- uint8_t, uintptr_t unsigned versions
- size_t
  this is used in containers, the reson for this is that *this has the max size that for example an array can be.*
  This is unsigned!

1.2 Addition of pointers

If you try to add or subtract 2 pointers to get the amount of sizeof(t) difference, then you can only do this with the exact same type, something like signed int and unsigned int will not work!

```
int32_t *y = 100;
int32_t *x = 120;
ptrdiff_t z = x - y; // z == 5
uint32_t *u = 120;
ptrdiff_t p = u - y;  // Error: Different ptr types
```

1.3 Index Operator on Pointers

You can index on pointers like an array, this can be used to get elements on any object.
Note that you have to manually make sure to stay within the bounds of that object, as otherwise you will have *undefined behavior*.

```
int32_t x = 0;
int32_t *y = &x;
y[0] = 0x42;      // same as: x = 0x42;
(&x)[0] = 0x42;   // same
0[&x] = 0x42;     // same
100[200] = 0x42;  // Error: no address
```

1.4 Padding

When you mix and match different types of different sizes inside of a struct, then the compiler will include padding based on the bigger type:

```
struct  {
char c;     // Offset 0
int32_t x;  // Offset 4 --> Padding
char d;     // Offset 8
} t;        // sizeof t == 12
# structure matters!!
struct  {
char c;     // Offset 0
char d;     // Offset 1
int32_t x;  // Offset 2 --> Padding
} t;        // sizeof t == 6
```

1.5 Forwards Declaration

```
struct Folder;
// Forward-Deklaration
struct File {
struct Folder *parent;
// OK: all pointer types
//
have same size
char name[256];
// OK: fixed size array
};
// --> Type complete
struct Folder {
struct File * file[256]; // OK: fixed size array
};
// --> Type complete
```

2 Operating System APIs

2.1 Basic features of an Operating system

- abstraction and portability
  - define generic APIs that work on all (as many as possible) devices
  - define abstractions that we don't care about -> how are files stored on the disk?
- Isolation and Resource Management
  - Isolate each usecase from each other -> posx
  - Runtime (make it blazingly fast)
  - Memory Management
  - Secondary Storage handling
- Security

2.1.1 Limitations of Portability

While the operating system can define standards, there are often things that we as developers need to consider, for example, while the operating system can define how a user will interact with the keyboard etc, if said device *doesn't have this input*, then your application will not work... Eg. An application meant for touch on the desktop might work, but not properly, and the OS can't really help there.

2.1.2 Limits of Isolation

Often, you want some form of interoperability, or you are basically forced to use that.
For example an application might want the focus of the keyboard, but then a popup appears.
If the application continues taking the focus, then the application now breaks the user experience.

2.2 Processor Privilege

Modern operating systems define a range of instructions that only the kernel is allowed to perform.
This is done to protect the operating system from attacks that might be exploitable via these privlieges.
In this case you run in *user mode*, this is also why anti-cheats are often running in kernel mode,
in this mode, the anti-cheat can access any memory all the time for whatever reason the anti-cheat would like to do so.
Should an application break the rule of user-space, the operating system will be notified and can then kill or otherwise restrict the application.

2.3 Kernel

2.3.1 MicroKernel

Microkernel is the idea that only the *absolutely necessary operations need to be in the kernel*, this means that often,
things like drivers run in the userspace, eg. Radv would be in userspace.
Features:

- Reliability
  less code is easier to maintain, which means a more stable operating system.
- Analysability
  less code is easier to bisect, meaning that bug hunting is easier.

- Performance
  Because drivers are now in a lower priority environment, they can no longer directly access hardware.
  This means that you will have a significant performance hit, which is also the reason that *linux is not a microkernel!*.

In the real world, there is no *real microkernel*, they usually add the necessary functionality of drivers and leave it at that.

2.3.2 Monolithic Kernel

Monolithic kernels have all the base functionality included. This means that you will not need to supply basic functionality to the kernel, just to get a functional operating system.

- Performance
  Since drivers have direct access to hardware, this means they can run faster!
- Security
  Since drivers have direct access to hardware, this means that misconfigured or malicious hardware, can easily infiltrate the kernel
- Reliablity
  More code means more possible bugs, and in the kernel this is worse than in the usermode.

2.3.3 Unikernel

This is a kernel that is made for one specific application, which means *it is an application!*.

- Performance
- Seurity
- Reliablity
- Only one Usecase

2.3.4 Running an instruction in Kernel Mode

When you want to run an instruction in the kernel mode, then you need to do a *syscall*.
The processor will then switch into kernel mode (if the os has given the privelege) and run the instruction in kernel mode.

2.3.5 Syscall (SVC on ARM)

There is only one function to run something in kernel mode, this means that we have to use *codes* instead.
Eg. a syscall with code 60 would be the exit code for a program. -> plox kill me
NOTE: Syscall also doesn't take arguments, therefore you need to place the arguments in registers.
This is exactly why you had to place all these things into registers, when you wanted to print a simple "hello world" in assembly.
The implication: output and input are kernel mode!!

2.4 ABI vs API

*A*pplication *B*inary *I*nterface
- concrete interfaces
- calling conventions
- projection of datastructures

*A*pplication *P*rogramming *I*nterface
- abstract interfaces
- platform/OS independent aspects

2.4.1 ABI in Linux

Calling Conventions for syscall are different for different linux kernels!
This means that you need to compile applications for each kernel!
To counter problems that will appear with this, there is a standard called *Linux Standard Base*, which defines a set of conventions to use.

2.4.2 API in Linux

The proper solution is to use APIs instead, which can be done with languages such as C (and tomorrow Rust).
This means that you *no longer use syscall*, you instead use *C functions*, which work on every kernel, not just on one.

2.5 POSIX

In general, every OS has its own ABI and API.
The unix API has been developed alongside the C API, this lead to the ISO standard.
However, at some point there were multiple standards, which meant the compatability was wrecked again.
Instead, the POSIX standard API was defined, which meant that if you wrote your program POSIX compliant, then it will run on any POSIX OS.

2.5.1 POSIX Conformity

- MacOS: since version 10.5
- Linux, not certified, but somewhat POSIX conform
  Bad: not everything is standard, but we all know that sometimes you either go your own way, or nothing happens -> matrix
- Windows: no :)
- BSD: yes

2.6 Man Pages

Man pages provide information about a POSIX system, it *is made of 9 parts:*

1. Executable Programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually fond in /dev)
5. File formats and conventions, e.g. /etc/passwd
6. Games lol
7. Miscellaneous (including macro packages and convenstions)
8. System administration commands (usually only for root)
9. Kernel routines (not standard)

2.7 Shell

- Made of an ouput and input stream.
- many different shells, bash, dash, zsh, fish, nu
- doesn't need special rights or prerequisites
- Made to call OS functions via text

2.7.1 Arguments in shell

- All arguments are considered strings -> its just an IO stream
- Spaces usually seperate the arguments
- "\" usually used to escape characters -> like space

These arguments are then passed to C or C++ program in the array called *(char **argv)*, and the *(int argc)* variable is the count of arguments.
The first entry in argv is the programname

2.7.2 Env Vars

- Key-Value pair
- Example: MOZ_ENABLE_WAYLAND=1
- can be set for the shell in .zshrc/.bashrc etc
- can be set for environment in xdg-config/environment.d, /etc/environment or .profile, /etc/profile
- Order is root first then local, if variable is set twice, then you will overwrite it!

To use variables in C: getenv, putenv, setenv, unsetenv
Technically, there is a variable called *environ*, which is an array of null pointers to strings which is 0 terminated.
However, this variable *is not defined!!*
getenv

```
//definition
char * getenv (const char * key)

char *value = getenv("PATH");
// value = "/home/ost/bin:/home/ost/.local/bin"
// returns nullptr -> 0 if variable not set
```

setenv

```
// definition
int setenv(const char *key, const char *value, int overwrite);

int ret = setenv("HOME", "/usr/home", 1);
// returns code 0 if ok, error code otherwise
// sets variable with value, overwrite if not 0
// error if variable doesn't exist!
```

unsetenv

```
// definition
int unsetenv(const char *key);

int ret = unsetenv("HOME");
// returns 0 if ok, error code otherwise
// removes the env variable
```

putenv

```
//definition
int putenv (char * kvp)

int ret = putenv("HOME=/usr/home");
// adds env variable pair
// returns 0 if ok, error code otherwise
// if variable already exists -> overwrite
```

In general, use env vars as a flag to configure things, not as a config that you *need* to configure.
For other operating systems this is done via other management solutions like windows -> registry.... haah get shit on windoof users

3 Filesystems

Essentially an API that makes sure that applications do not need to understand/know how the hardware works

3.1 Files

Files have 2 parts:
- Data
  Sequence of bytes that represent the file
- Metadata
  visible for users: size, date, owner, filetype hint
  not visible for users: place on hardware, connection of blocks on hardware

3.1.1 FileTypes

- ending after . -> .pdf
- File-endings have pretty much no point, they are just for the user
- File-endings are used as a hint to open said file with a specific program
- Type can be deducted via *magic numbers* inside of the file

General Advice:
- Data is trash unless proven otherwise
- Validate ALL data
- General advice: when decoding a file, continuously check if the file is really a pdf, or whatever you expect, test it to be.
  A few lines may not be enough to prove that it really is of said type.

3.2 Directories

- Essentially a file with a special type
- Each directory other than root directory has exactly one parent -> tree
- root directory is often / -> penguinOS

3.2.1 Special directories

- . -> reference on self
- .. -> parent reference
  for root this would just be itself..
- $PWD -> working directory. getcwd in C
- chdir / fchdir -> cd in C

Example for getting the current working directory in C:

```
int main (int argc, char** argv) {
  char *wd = malloc (PATH_MAX);
  getcwd (wd, PATH_MAX);
  printf ("Current WD is %s", wd);
  free (wd);
  return 0;
}
```

3.2.2 Paths

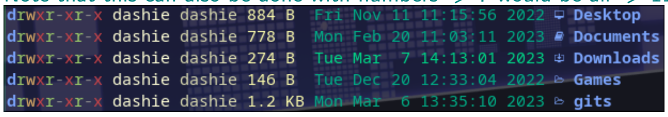
- absolute path /home/dashie/../dashie/.zshrc
  The reason for the .. in the middle is that a canonical path is an absolute path, but without either .. or . in the middle.
- relative path ../ai-app/ai-app.tex
- canonical path /home/dashie/.zshrc
  can be received with "realpath".

3.2.3 Max Path

POSIX systems can have different max path lengths, these are defined in "limits.h".
Macros:
- NAME_MAX max length of filename (exclusive 0 termination)
- PATH_MAX max length of path (inclusive 0 termination)
- _POSIX_NAME_MAX minimal value of NAME_MAX according to posix
- _POSIX_PATH_MAX minimal value of PATH_MAX according to posix

3.2.4 Rights

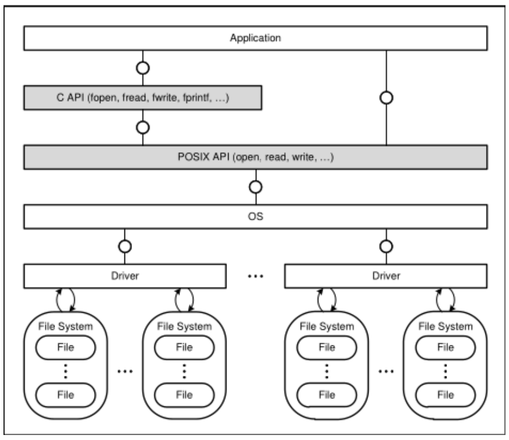
There are 3 permission categories, each having 3 groups, *read-write-execute* with *owner-group-other*
Technically there is one more bit, the sticky bit, but this is not really used anymore.
110-110-110 -> owner and group can do all, other can't do anything
Note that this can also be done with numbers -> 7 would be all -> 111



```
drwxr-xr-x dashie dashie 884 B  Fri Nov 11 11:15:56 2022 Desktop
drwxr-xr-x dashie dashie 778 B  Mon Feb 20 11:03:11 2023 Documents
drwxr-xr-x dashie dashie 274 B  Tue Mar  7 14:13:01 2023 Downloads
drwxr-xr-x dashie dashie 146 B  Tue Dec 20 12:33:04 2022 Games
drwxr-xr-x dashie dashie 1.2 KB Mon Mar  6 13:35:10 2023 gits
```

These rights are also stored as Macros in POSIX -> "sys/stat.h"
These can be chained with | -> S_IRWXU | S_IRGRP

4 File API



The main difference with the POSIX API to the C API, is that the POSIX API gives us raw data, without any interpretation of the data, while the C API supports more specific things such as sockets, decoding etc.

4.1 POSIX File API

General Info:
- All file functions are declared in <unistd.h> and <fcntl.h>
- error codes can be checked with "errno"
- raw data
- should only be used for binary data, not for anything that needs interpretation

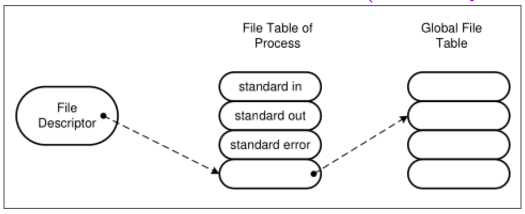
4.1.1 Usage of errno

```
if (chdir("docs") < 0) { // type is int
  if (errno == EACCESS) { // EACCESS defined in the function documentation
    printf ("Error: %s\n", strerror (errno));
    // or you can use perror
    perror ("Error"); // this makes use of the standard error stream
  }
}
```

Note, not all function set this flag, and should be used immediately, as other function will overwrite it.
Codes for the error are directly defined in the function documentation.
Returns the address of a string, which describes the code in text.
perror is the same as strerror but with a special error stream.

4.1.2 File-Descriptor

- valid within a process
- indexed in a table of all open files in a process
- Process file table is indexed in the global table of all open files
- Receives data in order to identify physical file (correct hardware with correct driver)
- State defined: knows current offset (offset of byte that will be read next)



In each process open file index for a process there are 3 predefined file descriptors

```
// STDIN_FILENO = 0  -> standard input
// STDOUT_FILENO = 1 -> standard output
// STDERR_FILENO = 2 -> standard error
```

4.1.3 Opening files with POSIX API

```
int open (char *path, int flags, ...)
```

- O_RDONLY: Read only
- O_RDWR: read and write
- O_CREAT: create file if not exists, needs another parameter for access rights
- O_APPEND: set offset to end of file before each write access
- O_TRUNC: set length of file to 0

4.1.4 Close files with POSIX API

```
int close(int fd)
```

deallocates file descriptor fd, which can now be used by other functions.
returns 0 if ok, and -1 for error

4.1.5 Usage of open and close

```
int fd = open("filename.file", O\_RDONLY);
if (fd < 0) {
  // le error handling
}
// do something with file
close(df);
```

4.1.6 Read data with POSIX API

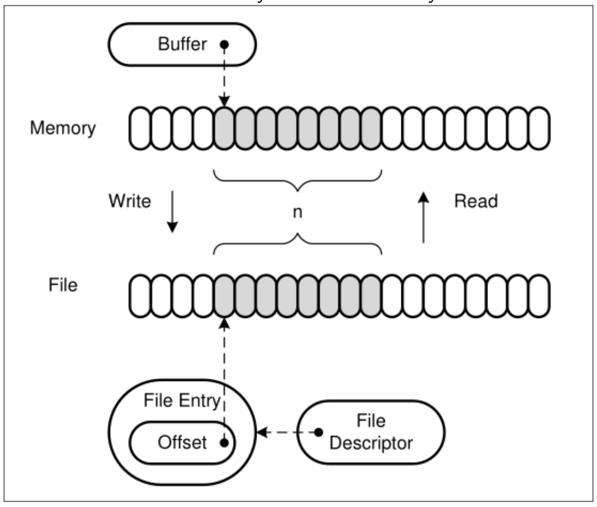
```
ssize_t read (int fd, void *buffer, size_t n)
// ssize_t is a signed size_t
```

- tries to copy the next n(parameter) bytes to current offset from fd to the buffer
- returns count of read bytes or -1 when error
- blocks thread until: n bytes are copied, error occurs, end of file has been reached
- increments offset of fd by the amount of read bytes

4.1.7 Write data with POSIX API

```
ssize_t write (int fd, void *buffer, size_t n)
// ssize_t is a signed size_t
```

- tries to copy the n bytes from the buffer to the offset on fd
- returns count of written bytes or -1 on error
- blocks thread until n bytes are written, error occurs, or the end of file has been reached
- increments offset of fd by the amount of bytes written



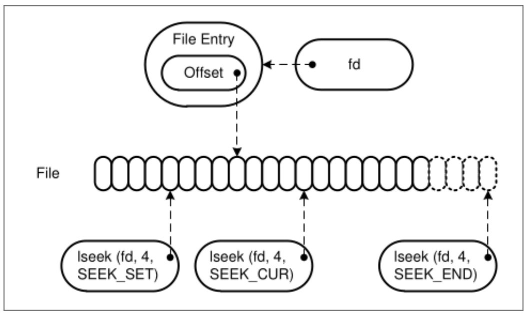
4.1.8 Jump in a file with POSIX API

```
off_t lseek (int fd, off_t offset, int origin)
```

- parameter offset that should be the new offset of fd
- origin = SEEK_SET for start of file
- origin = SEEK_CUR for current offset
- origin = SEEK_END for end of file
- returns new offset or -1 on error

Example usage:

```
lseek (fd, 0, SEEK_CUR) return current offset
lseek (fd, 0, SEEK_END) return size of file
lseek (fd, n, SEEK_END) go beyond end of file, which will make write put 0s in this space if called.
```



4.1.9 pread and pwrite in POSIX API

```
ssize_t pread (int fd, void *buffer, size_t n, off_t offset)
ssize_t pwrite (int fd, void *buffer, size_t n, off_t offset)
```

These are alternatives to write and read, that do not change the offset, however, this means that you will need to define where we are currently!

4.1.10 windoof proprietary paths

- \instead of / because fuck you
- \also needs to be escaped if you want to write it
- root directory per disk instead of per system
- C is the default disk, A and B were reserved for floppy disks
- functions in windoof:
  - open -> CreateFile
  - read -> ReadFile
  - write -> WriteFile
  - lseek -> SetFilePointer
  - close -> CloseHandle

4.1.11 Example for reading and writing data in POSIX API

```
#define N 32
char buf[N];
char spath[PATH_MAX];
char dpath[PATH_MAX];

/* get paths from somewhere */

int src = open(spath, O_RDONLY);
int dst = open(dpath, O_WRONLY | O_CREAT, S_IRWXU);
ssize_t read_bytes = read(src, buf, N);
write(dst, buf, read_bytes);
close(src);
close(dst);
```

4.2 C Stream API

Idea: Operating systems do things differently, even something as simple as a newline is handled differently, so we need an API that can translate this to the correct symbol:

```
// Windows: \r \n = 13d 10d = 0Dh 0Ah
// Linux: \n = 10d = 0Ah
// Mac OS: \r = 13d = 0Dh (before Mac OSX, now just like penguinOS)
```

- OS independent
- stream-based: symbol-oriented
- can be buffered or unbuffered
  dependent on the implementation, transparent for applications
- normally buffered for files
  independently transfers data-blocks between files and buffers
- Has a file Position indicator
  - for buffered streams: defines position in buffer
  - for unbuffered streams: is the offset in the file-descriptor

4.2.1 FILE datastructure

- has information about a stream
- should not be used directly, instead only per pointers that are created via the C-API
- should not be copied, pointer can be used as ID by the API
- three predefined standard-streams
  - FILE *stdin
  - FILE *stdout
  - FILE *stderr

4.2.2 open file with C-API

```
FILE * fopen (char const *path, char const *mode)
// flags
// "r": like O_RDONLY
// "w": like O_WRONLY | O_CREAT | O_TRUNC
// "a": like O_WRONLY | O_CREAT | O_APPEND
// "r+": like O_RDWR
// "w+": like O_RDWR | O_CREAT | O_TRUNC
// "a+": like O_RDWR | O_CREAT | O_APPEND
```

- creates FILE-Object and stream for the file
- returns pointer to created FILE-Object or 0 on error

4.2.3 close file with C-API

```
int fclose (FILE *file)
```

- calls fflush
- closes stream defined by file parameter
- removes file from memory
- returns 0 when ok, otherwise EOF

4.2.4 flush of file with C-API

```
int fflush (FILE *file)
```

- writes content to write from memory into file (if content exists)
- will automatically be called when the buffer is full or file is closed
- returns 0 when of, otherwise EOF

4.2.5 Conversion from POSIX-API to C-API

```
FILE * fdopen (int fd, char const *mode)
// like fopen, but instead of path, we use a file descriptor

int fileno (FILE *stream)
// returns file-descriptor for the stream, or -1 on error
```

4.2.6 read from file with C-API

```
int fgetc (FILE *stream)
```

- reads the next byte from stream *as unsigned char* and returns it as int
- increments the file-position indicator by 1

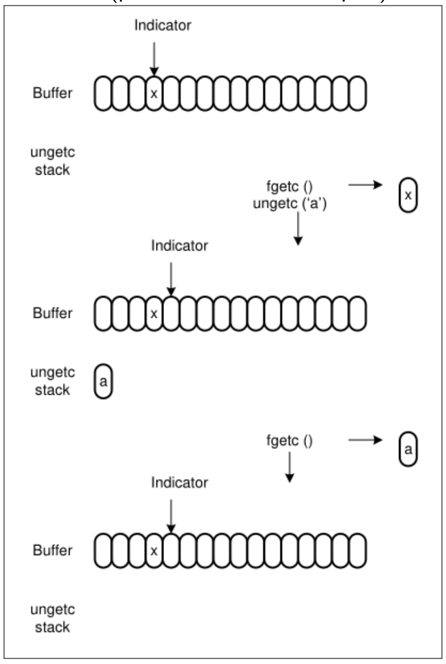
```
char * fgets (char *buf, int n, FILE *stream)
```

- reads to n-1 symbols from stream, or until newline or EOF
- adds a 0 and creates a null terminated string
- returns buffer(string) or 0 on error
- increments the file-position indicator by the amount of read symbols

4.2.7 "un"read from file with C-API

```
int ungetc (int c, FILE *stream)
```

- puts c(parameter -> file-descriptor) back to the stream into a special *unget stack*
- the next fget call will prefer the symbols in the unget stack
- no change in the file itself
- unget stack has a minimum size of 1 -> works at least once, can work multiple times depending on implementation
- returns c(parameter -> file-descriptor) or EOF on error



4.2.8 Write to file with C-API

```
int fputc(int c, FILE *stream)
```

- converts file-descriptor c in unsgined char and writes it to stream
- returns either c or EOF
- increments the file-position indicator by 1

```
int fputs (char *s, FILE *stream)
```

- writes the symbols from the string s until the 0 termination symbol into the stream
- the 0 termination will not be written
- returns EOF on error

4.2.9 End of file and Error in File C-API

```
int feof (FILE *stream)
// returns 0 when end of file has NOT been reached

int ferror (FILE *stream)
// retuns 0 when NO error occurred

// Example usage:
int return_value = fgetc (stream);
if (return_value == EOF) {
  if (feof (stream) != 0) {
    /* EOF reached */
  } else if (ferror (stream) != 0) {
    /* error occurred, check errno */
  }
}
```

4.2.10 Manipulation of file-position indicator with C-API

```c
long ftell (FILE *stream)
// returns the current file-indicator
// POSIX extension of ftello with return type off_t

int fseek (FILE *stream, long offset, int origin)
// set the file-position indicator like lseek
// POSIX extension of fseeko with off_t as type for the offset

int rewind (FILE *stream)
// reset the stream
// equivalent to fseek(stream, 0, SEEK_SET) and clear the error state
```

4.3 Ext2

4.4 Ext4

5 Processmodels

6 Communication and Synchronization

7 Programs and libraries

8 Graphical Overlays