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