Struct

A `struct` forms a compound type out of several individual variables.

- It allows to store several pieces of data in one variable
- OSs generally use them for control data structures.
- individual components of a `struct` are called members

`fork()`

This function creates a new process. The return value is the zero in the child and the process-id number of the child in the parent, or -1 upon error.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{

    pid_t pid = fork();
    switch (pid) {
        case -1:
            printf("Error. fork failed\n");
            break;
        case 0:
            printf("I am the child.\n");
            break;
        default:
            printf("I am the parent.\n");
            wait(NULL);
            printf("Child terminated.\n");
    }

    return 0;
}
```

`execve()`

This can be used to replace the currently running program with another, for example to load a new binary into the **current** address space. When a shell

creates a new process to execute some program, it will first fork itself and then invoke `execve()` within the child process to replace the shell code with the code of the program that shall be executed.

Parameter Passing : Stack

Using a stack for parameter passing has two main advantages:

1. The context of each callee (its frame) is automatically pushed onto and popped from the stack, no additional actions are needed.
2. The parameters being pushed onto the stack are accessed via the stack frame pointer, thus you can implement procedures with a variable number of parameters very easily.

Function vs. Macro

The code of a function exists only once, independent of how often the function is being called.

For macros, each call results in its repective code being inserted at the call-site.

- In C, macros are expanded in the source code by the pre-processor.
- Code with many macros results in larger code than the same code using functions.
- Runtime behaviour: function calls incur a small overhead compared to macros for jumping to and back from the called function
- Macros are untyped: The compiler does not check whether the macro arguments have a specific type.
  - This offers flexibility but is also a well-known source of errors.

Assertions

- Designed for debugging purposes and aid in finding errors during the development phase of a software.
- Usually check again consitions that should never happen
- In release builds, assertions are typically deactivated; the compiler omits the checks.

Inline Assembly

- CPU registers can be directly accessed

- Sometimes used to gain access to special CPU instructions that are generally not generated by the compiler. (Example: `sysenter`)
- The `__volatile__` statement disables all other potential optimization by the compiler, such as moving or omitting the assembly.

Threads

Stack overflow exception - occurs when a stack reaches its maximum size and cannot grow any futher.

- Might be due to configured limits or because the placement of the stacks in the VAS does not allow further growth

Dynamic Shared Libraries

- Loaded into the VAS of the process when the process starts
- Usually done by a component called 'loader/dynamic loader'
- The position at which libraries are mapped depends on what other libraries have to be loaded and if security features such as AS Layout Randomisation (ASLR) are active.

For these to work:

1. The code of the shared library must itself be functional
   - Solved by using Position Independent Code: compiler only uses relative addressing when accessing data and code
2. External code needs a way to access the exported functionality of the shared library
   - Can be solved by introducing a level of indirection
     - Each library has a dedicated Global Offset Table that will hold the addresses of imported variables
     - GOT is at a fixed known position to the code that accesses it, allowing the code to fetch the address from the GOT at runtime.
     - Calling external functions works similarly. Instead of producing code that directly calls the imported function, the compiler generates calls that will first fetch the address of the imported function from an address table.

both independent of the mapping address of the library.

Emulation using IPC

How can you perform asynchronous IPC if your OS only provides synchronous IPC mechanisms?

Use a separate proxy thread for delivering the asynchronous message via synchronous IPC.

- If the receiver is not waiting for reception, the proxy will block on the send operation.
- Original thread can continue execution, because it is not responsible for transmitting the message
- Requires one proxy thread per outstanding asynchronous message

How can you perform synchronous IPC if your OS only provides asynchronous IPC mechanisms?

```
do {
    res = async_send(msg, receiver) ;
} while ( res != MESSAGE SENT ) ;
```

- The sender must still be notified by the receiver when the message is received.

Futex

Combine the advantages of (user-mode) spinlocks (no kernel entries necessary) and mutexes (no busy waiting).

Before a thread blocks on the mutex and thus needs to enter the kernel, it first spins a certain time in user-mode, trying to acquire the spinlock. This way, the futex tries to avoid the costly blocking wait in the kernel.

Segmentation

MMU has 2 registers:

1. One that contains the physcial address of the segment table
2. One that specifies the size of the table