Systems Programming and Computer Architecture

Robin Bacher, Janis Hutz
https://github.com/janishutz/eth-summaries

January 22, 2026

TITLE PAGE COMING SOON

"*If you are using CMake to solve the exercises... First off, sorry that you like CMake*"

- Timothy Roscoe, 2025

HS2025, ETHZ

Summary of the Lectures and Lecture Slides

Quotes

"An LLM is a lossy index over human statements"

- Professor Buhmann, Date unknown

"If you are using CMake to solve the exercises... First off, sorry that you like CMake"

"You can't have a refrigerator behave like multiple refrigerators"

"Why is C++ called C++ and not ++C? It's like you don't get any value and then it's incremented, which is true"

- Timothy Roscoe, 2025

Contents

1 Introduction

This summary tries to summarize everything that is important to know for this course. It aims to be a full replacement for the slides, but as with all my summaries, there may be missing or incorrect information in here, so use at your own risk. You have been warned!

The summary does *not* follow the order the lecture does. This is to make related information appear more closely to each other than they have in the lecture and the summary assumes you have already seen the concepts in the lectures or elsewhere (or are willing to be thrown in the deep end).

The target semester for this summary is HS2025, so there might have been changes in your year. If there are changes and you'd like to update this summary, please open a pull request in the summary's repo at

`https://github.com/janishutz/eth-summaries`

2 x86 Assembly

Definition: *(Architecture)* Also known as ISA (Instruction Set Architecture) is "The parts of a processor design that one needs to understand to write assembly code". It includes for example the definition of instructions (and their options) and what registers are available. Notable examples are `x86`, `RISC-V` (this one is open-source!), MIPS, ARM, etc

`x86`-Assembly files usually use the file extension `.s` or `.asm`

Definition: *(Microarchitecture)* The implementation of the ISA. It defines the actual hardware layout and how the individual instructions are actually implemented and thus also defines things such as core frequency, cache layout and more.

Thus, the ISA is more or less precisely on the boundary of the software/hardware interface.

Definition: Complex Instruction Set (CISC):

- Stack oriented instruction set: Uses it to pass arguments, save program counter and features explicit push and pop instructions for the stack.
- Arithmetic instructions can access memory
- Condition codes set side effect of arithmetic and logical instructions.
- Design Philosophy: Add new instructions for typical tasks.

Definition: Reduced Instruction Set (RISC):

- Fewer, simpler instructions, commonly with fixed-size encoding. As a result, we might need more to get a given task done. On the other hand, we can execute them with small and fast hardware
- Register-oriented instruction set with many more registers that are used for arguments, return pointers, temporaries, etc.
- Load-Store architecture, i.e. only load and store instructions can access memory
- Thus: No Condition codes

What to choose? Both have advantages that the other has as disadvantage: Compiling for CISC is usually easier and usually results in smaller code size. For RISC however, compiler optimization can give a huge performance uplift and it can run fast with even a simple chip design.

Today, the choices are made based on outside constraints usually. For desktops and servers, there is enough compute to make anything run fast. For embedded systems though, the reduced complexity of RISC makes more sense, but for how long still?

What matters most today are non-technical factors such as existance of code for one ISA or licensing costs (and of course, Geopolitics)

Example A `C` function that simply adds 2 arguments might be compiled (unoptimized) to this:

`File: code-examples/01_asm/01_sum.s`

```
1  sum:  # Label
2          endbr64                # Indirect Branch target (No effect on code's logic)
3          pushq %rbp             # Preserve caller stack frame
4          movq  %rsp, %rbp       # Set up new stack frame
5          movl  %edi, -20(%rbp)  # Arg1 register -> stack
6          movl  %esi, -24(%rbp)  # Arg2 register -> stack
7          movl  -20(%rbp), %edx  # stack[Arg1] -> gp register
8          movl  -24(%rbp), %eax  # stack[Arg2] -> gp register
9          addl  %edx, %eax       # Add Arg1 + Arg2 -> gp register
10         movl  %eax, -4(%rbp)   # result -> stack
11         movl  -4(%rbp), %eax   # stack[result] -> return register
12         popq  %rbp             # Restore caller stack frame
13         ret                    # jump back to caller
```

2.1 The syntax

There are two common styles: AT&T syntax (common on UNIX) and Intel syntax (common on Windows)

The state that is visible to us is:

- PC (Program Counter) that contains the address of the next instruction
- Register file that contains the most used program data
- Condition codes that store status information about most recent arithmetic operation and are used for conditional branching

To view what `C` code looks like in assembly, we can use `gcc -O0 -S code.c`, which produces `code.s` which contains assembly code.

2.1.1 Registers

`x86` assembly is a bit particular with register naming (register names all start in %). The initial 16-bit version of `x86` had the following registers (sub registers are registers that can be used to access the high (`h` suffix) or low (`l` suffix) half of the register. Only registers ending in `x` feature these sub registers. They, as well as `%si` and `%di` are general purpose):

Name	Sub-registers	Description
%ax	%ah, %al	accumulate
%cx	%ch, %cl	counter
%dx	%dh, %dl	data
%bx	%bh, %bl	base
%si	-	Source index
%di	-	Destination index
%sp	-	Stack pointer
%bp	-	Base pointer
%ip	-	Instruction pointer
%sr	-	Status (flags)

When the architecture was extended to 32-bit, all registers previously available were retained and a 32 bit version of each was introduced with the prefix `e`. In other words, any 16 bit code would still work as previously, as e.g. the `%ax` register was simply now the lower 16 bits of the `%eax` register.

The same happened again when extending to 64-bit, only this time the `r` prefix was used. So, the register `$eax` was now the lower 32 bits of `%rax`.

Additionally, the following registers are also available, with `X` to be substituted with 8 through 15: `%rX` and the lower 32 bits `%rXd`

2.1.2 Instructions

Instructions usually have a 3 letter `mnemonic` with a one letter postfix that indicates the number of bytes. The following postfixes are available: `b` (byte, 1 byte), `w` (word, 2 bytes), `l` (long word, 4 bytes) and `q` (quad, 8 bytes).

The following options can be passed for source and destination: Registers,

Immediates To use a constant value (aka Immediate) in an instruction, we prefix the number with $ (following number is decimal). To use hex, we can use `$0x`, etc.

Memory addresses To treat a register as a memory address, use parenthesis, e.g. `(%rax)` interprets the value of `%rax` as a memory address. The instruction will then read the number of bytes, as specified by the postfix of the instruction.

The full syntax for memory address modes is `D(Rb, Ri, S)`, where

- `D`: Displacement (constant offset), can be 0, 1, 2 or 4 bytes (not bits, if you are confused as I was)
- `Rb`: Base register (to which offsets, etc are added). Can be any of the 16 integer registers
- `Ri`: Index register: Any, except for `%rsp` (and `%rbp` is also rarely used)
- `S`: Scale factor (1, 2, 4 or 8, to correct offsets)

The computation that happens is the following: `Mem[ Reg[Rb] + S * Reg[Ri] + D ]`. Using the `lea src, dest` instruction, we can get the address computed into the dest register. Can be abused for similar arithmetic expressions.

2.2 Data types

Assembly supports the following integer types (where GAS stands for GNU Assembly). If they are signed or unsigned does not matter (as we have seen), so it's up to you to interpret them as one or the other

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long

These integer types are also used for pointer addresses. Assembly also supports floating point numbers. They are stored and operated on in floating point registers.

Intel	GAS	Bytes	C
single	s	4	float
double	l	8	double
extended	t	16	long double

Assembly does not support any aggregate types (such as arrays, structs, etc) natively. You can however (obviously) make your own. In the following section we will cover how C datatypes are compiled into assembly.

2.2.1 Arrays

Arrays of type T and length L are allocated as a contiguous region of memory with size `L * sizeof(T)` bytes. We then also store a reference / identifier A to the array (i.e. similar to variable name in C), that holds the address of the first element of the array and can then be used in conjunction with "assembly pointer arithmetic".

Array loops that are written as for-loops in code are usually transformed into do-while loops by the compiler to save one condition check in the beginning, except of course, it might be possible that the loop is never executed.

2.2.2 Structures

We again allocate a contiguous region of memory. Only now, the number of bytes required isn't as straightforward to compute anymore, but still relatively simple: We simply sum up the sizes of all members and that will be our required sizes, so for the n members x_i of struct `my_struct`, we have $\texttt{sizeof(my\_struct)} = \sum_{i=0}^{n-1} \texttt{sizeof}(x_i)$.

However, the size of a struct may be different to fulfill alignment requirements set forth by the ISA or operating system. This could mean that the struct takes `n * K` bytes, where K is the alignment of the largest element

For alignment on x86-64 we have:

- 1 byte (no restrictions)
- 2 bytes (LSB must be 0)
- 4 bytes (2 LSB must be 00)
- 8 bytes (3 LSB must be 000)
- 16 bytes (4 LSB must be 0000)

Another issue is accessing members. The solution to this is however easy and efficient, as at compile time, the offsets are pre-determined and compiled into the setter and/or getter code for the struct.

2.2.3 Nested / Multidimensional arrays, Struct arrays

All of these arrays have similar underlying concepts in the way they are allocated, yet all are a bit different

Common ideas Each of the array's elements are allocated in contiguous regions of memory, with the elemnts also in contiguous regions of memory. (Imagine it as lining up all elements on a band, i.e. as going through the array in a nested loop and printing all the elements into a single line.) The size of the array is determined by `n * sizeof(T)`, where T is the type of the elements of the array (or outer array). This is what is different for the lot (as well as accessing elements):

Nested array T is another array. We thus have a recursive definition, where `sizeof(T)` resolves to `n * sizeof(T1)`, etc. Accessing element i, j, k is handled as follows: $o = i * \texttt{sizeof(T)} + j * \texttt{sizeof(T1)} + k * \texttt{sizeof(T2)}$, with T1 and T2 the types of the nested arrays

Struct arrays T is a struct.

2.2.4 Multi-Level arrays

In comparison to multidimensional arrays, we have arrays of pointers that contain either more arrays of pointers, (normal) arrays or pointers to other data types. The size of such a Multi-Level array is determined by: `n * sizeof(ptr)`, where `sizeof(ptr)` is the platform-specific size of a pointer and `n` is the number of elements in the array

To do an access, we need to do two (or more) memory reads, which we can again do using address computations.

The benefit of these kinds of arrays is that we can store arbitrary data types together in an array, giving us more flexibility.

2.2.5 Unions

Since unions can hold any of the elements listed (but only one at a time), we allocate based on the size of the largest element.

2.3 Operations

Assembly operations include performing arithmetic or logic functions on registers or memory data, transferring data between memory and registers and transferring control (conditional or unconditional jumps).
Note that `move` instructions *cannot* move data directly from memory to memory.

The following instruction formats are commonly used:

File: `code-examples/01_asm/00_operations.s`

```
main:
        movq %rax, %rdx # rax is src, rdx is dest
        cmp  %rax, %rdx # rax is src2, rdx is src1
        jmp  func       # func is label
        notq %rax       # rax is dest (and src)
        ret  # No argument
```

2.3.1 Arithmetic Operations

Arithmetic / logic operations need a size postfix (replace `X` below with `b` (1B), `w` (2B), `l` (4B) or `q` (8B)).

Mnemonic	Format	Computation
addX	Src, Dest	Dest ← Dest + Src
subX	Src, Dest	Dest ← Dest - Src
imulX	Src, Dest	Dest ← Dest * Src
salX	Src, Dest	Dest ← Dest << Src
sarX	Src, Dest	Dest ← Dest >> Src (arithmetic)
shrX	Src, Dest	Dest ← Dest >> Src (logical)
xorX	Src, Dest	Dest ← Dest ^ Src
andX	Src, Dest	Dest ← Dest & Src
orX	Src, Dest	Dest ← Dest | Src
incX	Dest	Dest ← Dest + 1
decX	Dest	Dest ← Dest - 1
negX	Dest	Dest ← -Dest
notX	Dest	Dest ← ~Dest

2.3.2 Condition Codes

Any arithmetic operation (that is truly part of the arithmetic operations group, so not including `lea` for example) implicitly sets the ***condition codes***. The following condition codes were covered in the lecture (operation: `t = a + b`):

- `CF` (Carry Flag): Set if carry out from MSB (unsigned overflow)
- `ZF` (Zero Flag): Set if `t == 0`
- `SF` (Sign Flag): Set if `(a - b) < 0` (for signed)
- `OF` (Overflow Flag): Set if two's complement overflow (i.e. `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`)

Explicit computation In the below explanations, we always assume `src2 = b` and `src1 = a`

To explicitly compute them, we can use the `cmpX src2, src1` instruction (with X again any of the size postfixes), that essentially computes $(a - b)$ without setting a destination register. When we execute that instruction, `CF` is set if `a < b` (unsigned), `ZF` is set if `a == b`, `SF` is set if `a < b` (signed) and `OF` is set as above, where `t = a - b`.

Another instruction that is used is `testX src2, src1` (X again a size postfix), and acts like computing `a & b` and where `ZF` is set if `a & b == 0` and `SF` is set if `a & b < 0`.

Zeroing register We can use a move instruction, but that is less efficient than using `xorl reg, reg`, where `reg` is the 32-bit version of the reg we want to zero.

Reading condition codes To read condition codes, we can use the `setC` instructions, where the `C` is to be substituted by an element of table 1

2.3.3 Jumping

To jump, use `jmp <label>` (unconditional jump) or the `jC` instructions, with `C` from table 1

setX	Condition	Description
e	ZF	Equal / Zero
ne	~ZF	Not Equal / Not Zero
s	SF	Negative
ns	~SF	Nonnegative
g	~(SF^OF)&~ZF	Greater (signed)
ge	~(SF^OF)	Greater or equal (signed)
l	SF^OF	Less (signed)
le	(SF^OF)|ZF	Less or equal (signed)
a	~CF&~ZF	Above (unsigned)
b	CF	Below (unsigned)

Table 1: Condition code postfixes for jump and set instructions

Conditional Moves

Similar to `jC`, the same postfixes can be applied to `cmovC`, for example:

```
cmpl    %eax, %edx
cmovle  %edx, %eax
```

Will move `%edx` into `%eax`, only if `%edx` is less or equal (`le`) `%eax`.

This can be used to, for example, compile ternary expressions from `C` to Assembly. However, this requires evaluating both possible expressions, which may lead to a (significant) performance overhead.

2.4 Control Flow

Control flow structures from `C` like `if/else` or `for` are compiled into assembly mainly using jumps and conditional `move`.

By the nature of Assembly and thanks to compilers optimizing aggressively, there is no *single* definitive translation of the `C` control structures: The compiler may translate it very differently depending on the context of the program.

2.4.1 Conditional statements

A function using an `if/else` construct to choose the maximum of 2 numbers might compile like this:

File: `code-examples/01_asm/02_max.s`

```
max:
    cmpl %esi, %edi     # Set condition flags
    jle .IF             # Conditional jump if %edi <= %esi
    movl %edi, %edx     # %edi -> return register
    jmp .ELSE
.IF:
    movl %esi, %edx     # %esi -> return register
.ELSE:
    ret
```

A function computing the absolute difference $|x-y|$ using an `if/else` construct, might use a conditional `move` instead:

File: `code-examples/01_asm/03_absdiff.s`

```
absdiff:
    movl %edi, %eax
    subl %esi, %eax     # arg2 - arg1 -> eax
    movl %esi, %edx
    subl %edi, %edx     # arg1 - arg2 -> edx
    cmpl %esi, %edi     # Set condition flags
    cmovle %edx, %eax   # edx -> eax, only if eax <= edx
    ret
```

2.4.2 While Loops

A recursive factorial function using a `do while` loop may be compiled like this:

File: `code-examples/01_asm/04_factorial.s`

```
factorial:
    movl  $1, %eax      # Setup
.AGAIN:
    imull %edi, %eax    # %eax *= %edi
    subl  $1, %edi      # %edi--
    cmpl  $1, %edi      # 1 < %edi ?
    jg    .AGAIN        # go back, if yes
    ret
```

The same function, using a `while` loop instead may lead to this:

File: code-examples/01_asm/05_factorial.s

```
1 factorial:
2     jmp .COMP           # Check condition
3 .LOOP:
4     imull %edx, %eax    # %eax *= %edx
5     decl %edx           # %edx--
6 .COMP:
7     cmpl $1, %edx       # 1 < %edx ?
8     jg .LOOP            # if yes, go to loop
9     ret
```

2.4.3 For Loops & Switch

`for` loops follow the same idea as `while` loops, albeit with a few more jumps.

`switch` statements are implemented differently depending on size and case values: Sparse switch statements are compiled as decision trees, whereas large switch statements may become *jump tables*.

These jump tables are usually stored in the `.rodata` section with 8-byte alignment. The jump uses offsets:

`jmp *.LABEL(, %rsi, 8)`

and we jump to the effective address of `.LABEL + rsi * 8`

2.5 The Stack

The Stack is the main way to dynamically allocate memory in Assembly, i.e. for temporary values. This process is completely manual: Allocating/Deallocating memory on the stack is entirely explicit. The stack grows *downwards*: Addresses decrease as the stack grows. The Stack pointer `%rsp` points to the current top of the stack.

Using the Stack `pushX`, `popX` exist for $x = 1, 2, 4, 8$, each corresponding to a size prefix that is set with `X`

Stack push `pushX src`: Fetch operand at `src`, decrement `%rsp` by x, then writes the operand at address of `%rsp`

Stack pop `popX dest`: Fetch operand at address of `%rsp`, increment `%rsp` by x, then writes the operand into `dest`

So intuitively, `pushX` and `popX` do exactly what is expected.

Procedure call / return Use `call LABEL`. This pushes the return label to the stack and jumps to `LABEL`. After this instruction, we also may use the `pushX` instruction to store further registers. Just remember to pop in the correct order with the correct size again!

The `ret` instruction is the return instruction and it will jump back to the caller and execution will continue there.

2.5.1 Calling Conventions

Even though Assembly will never stop you from using registers in any way, System V ABI specifies a few conventions, standardizing especially how functions are allowed to use registers.

Caller/Callee The callee is the function that is called and the caller is the code / function that calls the function.

- `%rax` and `%eax` are caller saved (usually used as return)
- `%rdi, %rsi, %rdx, %rcx` are caller saved (usually used for arguments)
- `%rsp` should not be modified manually
- `%rbp` is callee saved and used as frame pointer: usually set to `%rsp` at start of procedure and used to access frame elements (should always point to the start of the frame during function)

Register Conventions

Name	Description
`%rax`	Return value, #variable args
`%rbx`	Base pointer, Callee saved
`%rcx`	Argument 4
`%rdx`	Argument 3 (and return 2)
`%rsi`	Argument 2
`%rdi`	Argument 1
`%rsp`	Stack pointer
`%rbp`	Frame pointer, Callee saved

Name	Description
`%r8`	Argument 5
`%r9`	Argument 6
`%r10`	Static chain pointer
`%r11`	Temporary
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	GOT pointer, callee saved

If we have more than 6 arguments to be passed, we can use the stack for this. If we can do all accesses to the stack relative to the stack pointer, we do not need to update `%rbp` and not even `%rbx`, or we can use it for other purposes.

Manual stack management We can also allocate the entire stack frame directly by incrementing `%rsp` to the final position and then store data relative to it. To deallocate a stack frame, simply increment the stack pointer.

2.5.2 Examples

The stack is commonly used for recursive functions. A recursive factorial function might compile like this:

Note how `%rbx` is saved on the stack, since `rbx` is callee-saved by convention.
`%eax` is used directly, since it is caller-saved by convention.

File: code-examples/01_asm/06_factorial.s

```
factorial:
    pushq %rbx              # Preserve frame pointer
    movl %edi, %ebx
    movl $1, %eax
    cmpl $1, %edi
    jle .QUIT               # Base case reached: quit
    leal -1(%rdi), %edi     # Prepare args for next function call
    call factorial
    imull %ebx, %eax        # Use result of function call
.QUIT:
    popq %rbx               # Restore frame pointer
    ret
```

A more complex example, passing addresses as arguments:
This function swaps 2 array elements (using a `swap` function) and adds the first value to an accumulator.

File: code-examples/01_asm/07_swap_and_sum.s

```
swap_and_sum:
    movq    %rbx, -16(%rsp)     # Save %rbx
    movslq  %esi,%rbx           # Save i    (and extend)
    movq    %r12, -8(%rsp)      # Save %r12
    movq    %rdi, %r12          # Save a
    leaq    (%rdi,%rbx,8), %rdi # & a[i]   -> %rdi (arg 1)
    subq    $16, %rsp           # Allocate stack frame
    leaq    8(%rdi), %rsi       # & a[i+1] -> %rsi (arg 2)
    call    swap
    movq    (%r12,%rbx,8), %rax # a[i]
    addq    %rax, sum(%rip)     # sum += a[i]
    movq    (%rsp), %rbx        # Restore %rbx
    movq    8(%rsp), %r12       # Restore %r12
    addq    $16, %rsp           # Deallocate stack frame
    ret
```

2.6 Unorthodox Control Flow

In C, the `setjmp.h` header file can be included, which gives us access to `setjmp` and `longjmp`.

To use them, we first need to declare a `jmp_buf` somewhere, usually as a static variable.

The `setjmp( jmp_buf env )` function stores the current stack / environment in the `jmp_buf` and returns 0.

The `longjmp( jmp_buf env, int val )` function causes a second return, which returns `val`, to the `setjmp` invocation and jumps back to that place.

File: code-examples/01_asm/08_unorthodox-controlflow.c

```c
#include <setjmp.h>
#include <stdio.h>

static jmp_buf buf;

void second( void ) {
    printf( "second\n" );
    longjmp( buf, 1 );
}

void first( void ) {
    second();
    printf( "first\n" ); // Never executed
}

int main() {
    if ( !setjmp( buf ) ) // returns 0 initially
        first();
    else
        printf( "main\n" ); // 1 is returned when longjmp is executed
    return 0;
}
```

What the above code outputs is: `second` followed by `main`.

They are implemented in Assembly as follows. Nothing really surprising for the implementation there. The assembly code is from the Musl C library

File: code-examples/01_asm/09_setjmp-longjmp.s

```
# Copyright 2011-2012 Nicholas J. Kain, licensed under standard MIT license
setjmp:
        mov %rbx, (%rdi)
        mov %rbp, 8(%rdi)
        mov %r12, 16(%rdi)
        mov %r13, 24(%rdi)
        mov %r14, 32(%rdi)
        mov %r15, 40(%rdi)
        lea 8(%rsp), %rdx
        mov %rdx, 48(%rdi)
        mov (%rsp), %rdx
        mov %rdx, 56(%rdi)
        xor %eax, %eax
        ret

longjmp:
        xor %eax, %eax
        cmp $1, %esi        # CF = val ? 0 : 1
        adc %esi, %eax      # eax = val + !val
        mov (%rdi), %rbx    # rdi is the jmp_buf, restore regs from it
        mov 8(%rdi), %rbp
        mov 16(%rdi), %r12
        mov 24(%rdi), %r13
        mov 32(%rdi), %r14
        mov 40(%rdi), %r15
        mov 48(%rdi), %rsp
        jmp *56(%rdi)       # goto saved address without altering rsp
```

2.7 Coroutines

Coroutines are functions that call each other when they are done or they need new data to work on. An example is a decompresser that calls a parser when it has finished compressing parts of the file and that parser then again calls the decompresser when it has finished parsing.

We can implement that either by rewriting the functions into a single function, which often is a bit clumsy. A way around this is to use ***Continuations***, where the first function saves its state and the context is switched to the other function. That function can then load its state and continue where it left off, runs until it finishes its task, then saves its state and the context switches back to the original function.

File: code-examples/01_asm/10_coroutine.h

```h
#include <setjmp.h>
typedef void( co_start_fn )( void * );

struct coroutine {
        void *stack;        // The call stack
        jmp_buf env;        // The saved context
        co_start_fn *start; // Function to call
        void *arg;          // Argument to the function
};
struct coroutine *co_new( co_start_fn *start, void *ctxt );
void co_free( struct coroutine *self );
void *co_switchto( struct coroutine *next );
void co_init( void );
```

As you can see, the `setjmp.h` functions are the foundation of all concurrent programming.

File: code-examples/01_asm/10_coroutine.c

```c
#include "10_coroutine.h"
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>
#define CORO_STACK_SIZE 128
static struct coroutine *cur_co;
static struct coroutine *main_co;

void co_init() {
    main_co = (struct coroutine *) calloc( 1, sizeof( struct coroutine ) );
    cur_co = main_co;
    co_switchto( main_co );
}

void *co_switchto( struct coroutine *next, void *arg ) {
    if ( setjmp( cur_co->env ) == 0 ) {
        cur_co = next;
        cur_co->arg = arg;
        longjmp( cur_co->env, 1 );
    }
    return cur_co->arg;
}

static void start_cl( void ) {
    ( cur_co->start )( cur_co->arg );
    co_switchto( main_co );

    printf( "Error: returned from coroutine start closure.\n" );
    exit( -1 );
}

struct coroutine *co_new( co_start_fn *start, void *ctxt ) {
    struct coroutine *co = (struct coroutine *) calloc( 1, sizeof( struct coroutine ) );
    co->stack = calloc( 1, CORO_STACK_SIZE + 16 );
    co->start = start;
    co->arg = ctxt;
    setjmp( co->env );
    co->env[ 0 ].__jmpbuf[ 6 ] = ( (uint64_t) ( co->stack ) + CORO_STACK_SIZE ); // Machine specific
    co->env[ 0 ].__jmpbuf[ 7 ] = ( (uint64_t) ( start_cl ) );                    // Machine specific
}
```

3 The C Programming Language

I can clearly C why you'd want to use C. Already sorry in advance for all the bad C jokes that are going to be part of this section

C is a compiled, low-level programming language, lacking many features modern high-level programming languages offer, like Object Oriented programming, true Functional Programming (like Haskell implements), Garbage Collection, complex abstract datatypes and vectors, just to name a few. (It is possible to replicate these using Preprocessor macros, more on this later).

On the other hand, it offers low-level hardware access, the ability to directly integrate assembly code into the `.c` files, as well as bit level data manipulation and extensive memory management options, again just to name a few.

This of course leads to C performing excellently and there are many programming languages whose compiler doesn't directly produce machine code or assembly, but instead optimized C code that is then compiled into machine code using a C compiler. This has a number of benefits, most notably that C compilers can produce very efficient assembly, as lots of effort is put into the C compilers by the hardware manufacturers.

There are many great C tutorials out there, a simple one (as for many other languages too) can be found here

3.1 Basics

C uses a very similar syntax as many other programming languages, like Java, JavaScript and many more... to be precise, it is *them* that use the C syntax, not the other way around. So:

File: code-examples/00_c/00_basics/00_intro.c

```c
// This is a line comment
/* this is a block comment */
#include "01_func.h" // Relative import

int i = 0; // This allocates an integer on the stack

int main( int argc, char *argv[] ) {
    // This is the function body of a function (here the main function)
    // which serves as the entrypoint to the program in C and has arguments
    printf( "Argc: %d\n", argc );    // Number of arguments passed, always >= 1
                                     // (first argument is the executable name)
    for ( int i = 0; i < argc; i++ ) // For loop just like any other sane programming language
        printf( "Arg %d: %s\n", i, argv[ i ] ); // Outputs the i-th argument from CLI

    get_user_input_int( "Select a number" ); // Function calls as in any other language
    return 0;                                // Return a POSIX exit code
}
```

In C we are referring to the implementation of a function as a ***(function) definition*** (correspondingly, *variable definition*, if the variable is initialized) and to the definition of the function signature (or variables, without initializing them) as the ***(function) declaration*** (or, correspondingly, *variable declaration*).

C code is usuallt split into the source files, ending in `.c` (where the local functions and variables are declared, as well as all function definitions) and the header files, ending in `.h`, usually sharing the filename of the source file, where the external declarations are defined. By convention, no definition of functions are in the `.h` files, and neither variables, but there is nothing preventing you from putting them there.

File: code-examples/00_c/00_basics/01_func.h

```c
#include <stdio.h> // Import from system path
                   // (like library imports in other languages)

int get_user_input_int( char prompt[] );
```

3.1.1 Control Flow

Many of the control-flow structures of `C` can be found in the below code snippet. A note of caution when using goto: It is almost never a good idea (can lead to unexpected behaviour, is hard to maintain, etc). Where it however is very handy is for error recovery (and cleanup functions) and early termination of multiple loops (jumping out of a loop). So, for example, if you have to run multiple functions to set something up and one of them fails, you can jump to a label and have all cleanup code execute that you have specified there. And because the labels are (as in Assembly) simply skipped over during execution, you can make very nice cleanup code. We can also use `continue` and `break` statements similarly to `Java`, they do not however accept labels. (Reminder: `continue` skips the loop body and goes to the next iteration)

File: code-examples/00_c/00_basics/01_func.c

```c
#include "01_func.h"
#include <stdio.h>

int get_user_input_int( char prompt[] ) {
    int input_data;
    printf( "%s", prompt );           // Always wrap strings like this for printf
    scanf( "%d", &input_data );       // Get user input from CLI
    int input_data_copy = input_data; // Value copied

    // If statements just like any other language
    if ( input_data )
        printf( "Not 0" );
    else
        printf( "Input is zero" );

    // Switch statements just like in any other language
    switch ( input_data ) {
        case 5:
            printf( "You win!" );
            break; // Doesn't fall through
        case 6:
            printf( "You were close" ); // Falls through
        default:
            printf( "No win" ); // Case for any not covered input
    }

    while ( input_data > 1 ) {
        input_data -= 1;
        printf( "Hello World\n" );
    }

    // Inversed while loop (executes at least once)
    do {
        input_data -= 1;
        printf( "Bye World\n" );
        if ( input_data_copy == 0 )
            goto this_is_a_label;
    } while ( input_data_copy > 1 );

this_is_a_label:
    printf( "Jumped to label" );
    return 0;
}
```

3.1.2 Declarations

We have already seen a few examples for how C handles declarations. In concept they are similar (and scoping works the same) to most other C-like programming languages, including Java.

File: code-examples/00_c/00_basics/02_declarations.c

```c
int my_int;              // Allocates memory on the stack.
                         // Variable is global (read / writable by entire program)
static int my_local_int; // only available locally (in this file)
extern const char *var;  // Defined in some other file
const int MY_CONST = 10; // constant (immutable), convention: SCREAM_CASE

enum { ONE, TWO } num; // Enum. ONE will get value 0, TWO has value 1

enum { O = 2, T = 1 } n; // Enum with values specified

// Structs are like classes, but contain no logic
struct MyStruct {
        int el1;
        int el2;
};

// Like structs, but can only hold one of the values!
union MyUnion {
        int ival;
        float fval;
        char *sval;
};

int fun( int j ) {
    static int i = 0;              // Persists across calls of fun
    short my_var = 1;              // Block scoped (deallocated when going out of scope)
    int my_var_dbl = (int) my_var; // Explicit casting (works between almost all types)
    return i;
}

int main( int argc, char *argv[] ) {
    if ( ( my_local_int = fun( 10 ) ) ) {
        // Every c statement is also an expression, i.e. you can do the above!
    }
    struct MyStruct test;            // Allocate memory on stack for struct
    struct MyStruct *test_p = &test; // Pointer to memory where test resides
    struct MyStruct test2;
    union MyUnion my_uval; // Work exactly like structs for access
    test.el1 = 1;          // Direct element access
    test_p->el2 = 2;       // Via pointer
    test2 = test;          // Copies the struct
    return 0;
}
```

A peculiarity of C is that the bit-count is not defined by the language, but rather the hardware it is compiled for.

C data type	typical 32-bit	ia32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	4	8	8
long double	8	10/12	16

Table 2: Comparison of byte-sizes for each datatype on different architectures

Type format Be however aware that this table uses the LP64 format for the x86-64 sizes and this is the format all UNIX-Systems use (i.e. Linux, BSD, Darwin (the Mac Kernel)). 64 bit Windows however uses LLP64, i.e. `int` and `long` have the same size (32) and `long long` and pointers are 64 bit.

Integers By default, integers in C are `signed`, to declare an unsigned integer, use `unsigned int`. Since it is hard and annoying to remember the number of bytes that are in each data type, C99 has introduced the extended integer types, which can be imported from `stdint.h` and are of form `int<bit count>_t` and `uint<bit count>_t`, where we substitute the `<bit count>` with the number of bits (have to correspond to a valid type of course).

Booleans Another notable difference of C compared to other languages is that C doesn't natively have a `boolean` type, by convention a `short` is used to represent it, where any non-zero value means `true` and `0` means `false`. Since boolean types are quite handy, the `!` syntax for negation turns any non-zero value of any integer type into zero and vice-versa. C99 has added support for a bool type via `stdbool.h`, which however is still an integer.

Implicit casts Notably, C doesn't have a very rigid type system and lower bit-count types are implicitly cast to higher bit-count data types, i.e. if you add a `short` and an `int`, the `short` is cast to `short` (bits 16-31 are set to 0) and the two are added. Explicit casting between almost all types is also supported. Some will force a change of bit representation, but most won't (notably, when casting to and from `float`-like types, minus to `void`)

Expressions Every C statement is also an expression, see above code block for example.

Void The `void` type has ***no*** value and is used for untyped pointers and declaring functions with no return value

Structs Are like classes in OOP, but they contain no logic. We can assign copy a struct by assignment and they behave just like everything else in C when used as an argument for functions in that they are passed by value and not by reference. You can of course pass it also by reference (like any other data type) by setting the argument to type `struct mystruct * name` and then calling the function using `func(&test)` assuming `test` is the name of your struct

Typedef To define a custom type using `typedef <type it represents> <name of the new type>`.

You may also use `typedef` on structs using `typedef struct <struct tag> <name of the new alias>`, you can thus instead of e.g. `struct list_el my_list;` write `list my_list;`, if you have used `typedef struct list_el list;` before. It is even possible to do this:

```c
typedef struct list_el {
    unsigned long val;
    struct list_el *next;
} list_el;

struct list_el my_list;
list_el my_other_list;
```

Namespaces C has a few different namespaces, i.e. you can have the one of the same name in each namespace (i.e. you can have `struct a`, `int a`, etc). The following namespaces were covered:

- Label names (used for `goto`)
- Tags (for `struct`, `union` and `enum`)
- Member names one namespace for each `struct`, `union` and `enum`
- Everything else mostly (types, variable names, etc, including typedef)

3.1.3 Operators

The list of operators in C is similar to the one of Java, etc. In Table 3, you can see an overview of the operators, sorted by precedence in descending order. You may notice that the & and * operators appear twice. The higher precedence occurrence is the address operator and dereference, respectively, and the lower precedence is bitwise and and multiplication, respectively.

Very low precedence belongs to boolean operators && and ||, as well as the ternary operator and assignment operators

Operator	Associativity
`() [] -> .`	Left-to-right
`! ~ ++ -- + - * & (type) sizeof`	Right-to-left
`* / %`	Left-to-right
`+ -`	Left-to-right
`<< >>`	Left-to-right
`< <= >= >`	Left-to-right
`== !=`	Left-to-right
`&` (logical and)	Left-to-right
`^` (logical xor)	Left-to-right
`|` (logical or)	Left-to-right
`&&` (boolean and)	Left-to-right
`||` (boolean or)	Left-to-right
`? :` (ternary)	Right-to-left
`= += -= *= /= %= &= ^=—= <<= >>=`	Right-to-left
`,`	Left-to-right

Table 3: C operators ordered in descending order by precedence

Associativity

- Left-to-right: $A + B + C \mapsto (A + B) + C$
- Right-to-left: `A += B += C` $\mapsto$ `(A += B) += C`

As it should be, boolean and, as well as boolean or support early termination.

The ternary operator works as in other programming languages `result = expr ? res_true : res_false;`

As previously touched on, every statement is also an expression, i.e. the following works

```
printf("%s", x = foo(y)); // prints output of foo(y) and x has that value
```

Pre-increment (++i, new value returned) and post-increment (i++, old value returned) are also supported by C.

C has an assert statement, but do not use it for error handling. The basic syntax is `assert( expr );`

3.1.4 Arrays

C compiler does not do any array bound checks! Thus, always check array bounds. Unlike some other programming languages, arrays are ***not*** dynamic length.

The below snippet includes already some pointer arithmetic tricks. The variable `data` is a pointer to the first element of the array.

File: `code-examples/00_c/00_basics/03_arrays.c`

```c
#include <stdint.h>
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int data[ 10 ];                  // Initialize array of 10 integers
    data[ 5 ] = 5;                   // element 5 is now 5
    *data = 10;                      // element 0 is now 5
    printf( "%d\n", data[ 0 ] );     // print element 0 (prints 10)
    printf( "%d\n", *data );         // equivalent as above
    printf( "%d\n", data[ 5 ] );     // print element 5 (prints 5)
    printf( "%d\n", *( data + 5 ) ); // equivalent as above
    int multidim[ 5 ][ 5 ];          // 2-dimensional array
                                     // We can iterate over it using two for-loops
    int init_array[ 2 ][ 2 ] = {
        {1, 2},
        {3, 4}
    };                       // We can initialize an array like this
    int empty_arr[ 4 ] = {}; // Initialized to 0
    return 0;
}
```

3.1.5 Strings

C doesn't have a `string` data type, but rather, strings are represented (when using `ASCII`) as `char` arrays, with length of the array $n+1$ (where n is the number of characters of the string). The extra element is the termination character, called the `null character`, denoted `\0`. To determine the actual length of the string (as it may be padded), we can use `strnlen(str, maxlen)` from `string.h`

File: `code-examples/00_c/00_basics/04_strings.c`

```c
#include <stdio.h>
#include <string.h>

int main( int argc, char *argv[] ) {
    char hello[ 6 ] = "hello";                          // Using double quotes
    char world[ 6 ] = { 'w', 'o', 'r', 'l', 'd', '\0' }; // As array

    char src[ 12 ], dest[ 12 ];
    strncpy( src, "ETHZ", 12 );     // Copy strings (extra elements will be set to \0)
    strncpy( dest, src, 12 );       // Copy strings (last arg is first n chars to copy)
    if ( strncmp( src, dest, 12 ) ) // Compare two strings. Returns 1 if src > dest
        printf( "Hello World" );
    strncat( dest, " is in ZH", 12 ); // Concatenate strings
    return 0;
}
```

3.1.6 Integers in C

As a reminder, integers are encoded as follows in big endian notation, with x_i being the i-th bit and w being the number of bits used to represent the number:

- ***Unsigned***: $\sum_{i=0}^{w-1} x_i \cdot 2^i$
- ***Signed***: $-x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-1} x_i \cdot 2^i$ (two's complement notation, with x_{w-1} being the sign-bit)

The minimum number representable is 0 and -2^{w-1}, respectively, whereas the maximum number representable is $2^w - 1$ and $2^{w-1} - 1$. `limits.h` defines constants for the minimum and maximum values of different types, e.g. `ULONG_MAX` or `LONG_MAX` and `LONG_MIN`

We can use the shift operators to multiply and divide by two. Shift operations are usually *much* cheaper than multiplication and division. Left shift (`u << k` in `C`) always fills with zeros and throws away the extra bits on the left (equivalent to multiplication by 2^k), whereas right shift (`u >> k` in `C`) is implementation-defined, either arithmetic (fill with most significant bit, division by 2^k. This however rounds incorrectly, see below) or logical shift (fill with zeros, unsigned division by 2^k).

Signed division using arithmetic right shifts has the issue of incorrect rounding when number is < 0. Instead, we represent $s/2^k = s + (2^k - 1)$ >> k for $s < 0$ and $s/2^k = s >> k$ for $s > 0$

In expressions, signed values are implicitly cast to unsigned

This can lead to all sorts of nasty exploits (e.g. provide -1 as the argument to `memcpy` and watch it burn, this was an actual exploit in FreeBSD)

Addition & Subtraction

A nice property of the two's complement notation is that addition and subtraction works exactly the same as in normal notation, due to over- and underflow. This also obviously means that it implements modular arithmetic, i.e.

$$\texttt{Add}_w(u, v) = u + v \mod 2^w \quad \text{and} \quad \texttt{Sub}_w(u, v) = u - v \mod 2^w$$

Multiplication & Division

Unsigned multiplication with addition forms a commutative ring. Again, it is doing modular arithmetic and

$$\texttt{UMult}_w(u, v) = u \cdot v \mod 2^w$$

3.1.7 Pointers

On loading of a program, the OS creates the virtual address space for the process, inspects the executable and loads the data to the right places in the address space, before other preparations like final linking and relocation are done.

Stack-based languages (supporting recursion) allocate stack in frames that contain local variables, return information and temporary space. When a procedure is entered, a stack frame is allocated and executes any necessary setup code (like moving the stack pointer, see later). When a procedure returns, the stack frame is deallocated and any necessary cleanup code is executed, before execution of the previous frame continues.

In C a pointer is a variable whose value is the memory address of another variable

Of note is that if you simply declare a pointer using `type * p;` you will get different memory addresses every time. The (Linux)-Kernel randomizes the address space to prevent some common exploits.

File: code-examples/00_c/00_basics/05_pointers.c

```c
#include "01_func.h" // See a few pages up for declarations
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void a_function( int ( *func )( char * ), char prompt[] ) {
    ( *func )( prompt ); // Call function with arguments
}

int main( int argc, char *argv[] ) {
    int x = 0;
    int *p = &x;          // Get x's memory address
    printf( "%p\n", p );  // Print the address of x
    printf( "%d\n", *p ); // Dereference pointer (get contents of memory location)
    *p = 10;              // Dereference assign
    int **dbl_p = &p;     // Double pointer (pointer to pointer to value)
    int *null_p = NULL;   // Create NULL pointer
    *null_p = 1;          // Segmentation fault due to null pointer dereference

    // pointer arithmetic
    int arr[ 3 ] = { 2, 3, 4 };
    char c_arr[ 3 ] = { 'A', 'B', 'C' };
    int *arr_p = &arr[ 1 ];
    char *c_arr_p = &c_arr[ 1 ];
    c_arr_p += 1; // Now points to c_arr[2]
    arr_p -= 1;   // Now points to arr[0]

    char *arr_p_c = (char *) arr_p; // Cast to char pointer (points to first byte of arr[0])
    printf( "%d", *( arr_p - 5 ) ); // No boundary checks (can access any memory)
    assert( arr == &( arr[ 0 ] ) ); // Evaluates to true
    int new_arr[ 3 ] = arr;         // Compile time error (cannot use other array as
      ↪ initializer)
    int *new_arr_p = &arr[ 0 ];     // This works

    a_function( &get_user_input_int, c_arr );

    return EXIT_SUCCESS;
}
```

Some pointer arithmetic has already appeared in section 3.1.4, but same kind of content with better explanation can be found here

Pointer Arithmetic Note that when doing pointer arithmetic, adding 1 will move the pointer by `sizeof(type)` bits.

You may use pointer arithmetic on whatever pointer you'd like (as long as it's not a null pointer). This means, you *can* make an array wherever in memory you'd like. The issue is just that you are likely to overwrite something, and that something might be something critical (like a stack pointer), thus you will get ***undefined*** behaviour! (This is by the way a common concept in `C`, if something isn't easy to make more flexible (example for `malloc`, if you pass a pointer to memory that is not the start of the `malloc`'d section, you get undefined behaviour), in the docs mention that one gets undefined behaviour if you do not do as it says so... RTFM!)

As already seen in the section arrays (section 3.1.4), we can use pointer arithmetic for accessing array elements. The array name is treated as a pointer to the first element of the array, except when:

- it is operand of `sizeof` (return value is $n \cdot$ `sizeof(type)` with n the number of elements)
- its address is taken (then `&a == a`)
- it is a string literal initializer. If we modify a pointer `char *b = "String";` to string literal in code, the `"String"` is stored in the code segment and if we modify the pointer, we get undefined behaviour

Fun fact: `A[i]` is always rewritten `*(A + i)` by compiler.

Function arguments Another important aspect is passing by value or by reference. You can pass every data type by reference, you can not however pass an array by value (as an array is treated as a pointer, see above).

Body-less loops

```
1 int x = 0;
2 while ( x++ < 10 ); // This is (of course) not a useful snippet, but shows the concept
```

Function pointers A function can be passed as an argument to another function using the typical address syntax with the `&` symbol is annotated as argument using `type (* name)(type arg1, ...)` and is called using `(*func)(arg1, ...)`.

3.2 The C preprocessor

To have gcc stop compiliation after running through cpp, the `C preprocessor`, use `gcc -E <file name>`.

Imports in `C` are handled by the preprocessor, that for each `#include <file1.h>`, the preprocessor simply copies the contents of the file recursively into one file.

Depending on if we use `#include <file1.h>` or `#include "file1.h"` the preprocessor will search for the file either in the system headers or in the project directory. Be wary of including files twice, as the preprocessor will recursively include all files (i.e. it will include files from the files we included)

The `C` preprocessor gives us what are called `preprocessor macros`, which have the format `#define NAME SUBSTITUTION`.

File: code-examples/00_c/01_preprocessor/00_macros.c

```c
#include <stdio.h>
#include <stdlib.h>
#define FOO      BAZ
#define BAR( x ) ( x + 3 )
#define SKIP_SPACES( p )                                        \
    do {                                                        \
        while ( p > 0 ) { p--; }                                \
    } while ( 0 )
#define COMMAND( c ) { #c, c##_command } // Produces { "<val(c)>", "<val(c)>_command" }

#ifdef FOO // If macro is defined, ifndef for if not defined
    #define COURSE "SPCA"
#else
    #define COURSE "Systems Programming and Computer Architecture"
#endif

#if 1
    #define OUT HELLO // if statement
#endif

int main( int argc, char *argv[] ) {
    int i = 10;
    SKIP_SPACES( i );

    printf( "%s", COURSE );

    return EXIT_SUCCESS;
}
```

To avoid issues with semicolons at the end of preprocessor macros that wrap statements that cannot end in semicolons, we can use a concept called semicolon swallowing. For that, we wrap the statements in a `do ... while(0)` loop, which is removed by the compiler on compile, also taking with it the semicolon.

There are also a number of predefined macros:

- `__FILE__`: Filename of processed file
- `__LINE__`: Line number of this usage of macro
- `__DATE__`: Date of processing
- `__TIME__`: Time of processing
- `__STDC__`: Set if ANSI Standard `C` compiler is used
- `__STDC_VERSION__`: The version of Standard `C` being compiled
- ... many more

In headers, we typically use `#ifndef __FILENAME_H_` followed by a `#define __FILENAME_H_` or the like to check if the header was already included before

3.3 Memory

In comparison to most other languages, C does not feature automatic memory management, but instead gives us full, manual control over memory. This of course has both advantages and disadvantages.

File: code-examples/00_c/02_memory/00_memory.c

```c
#include <stdlib.h>

int main( int argc, char *argv[] ) {
    long *arr = (long *) malloc( 10 * sizeof( long ) ); // Allocate on heap
    if ( arr == NULL )                                  // Check if successful
        return EXIT_FAILURE;
    arr[ 0 ] = 5;

    long *arr2;
    if ( ( arr2 = (long *) calloc( 10, sizeof( long ) ) ) == NULL )
        return EXIT_FAILURE; // Same as above, but fewer lines and memory zeroed

    // Reallocate memory (to change size). Always use new pointer and do check!
    if ( ( arr2 = (long *) realloc( arr2, 15 * sizeof( long ) ) ) == NULL )
        return EXIT_FAILURE;

    free( arr );  // Deallocate the memory
    arr = NULL;   // Best practice: NULL pointer
    free( arr2 ); // *Can* omit NULLing pointer because end

    return EXIT_SUCCESS;
}
```

Notably, the argument `size_t sz` for `malloc`, `calloc` and `realloc` is an `unsigned` integer of some size and differs depending on hardware and software platforms.

`malloc` keeps track of which blocks are allocated. If you give `free` a pointer that isn't the start of the memory region previously `malloc`'d, you get undefined behaviour.

Memory corruption There are many ways to corrupt memory in C. The below code shows off a few of them:

File: code-examples/00_c/02_memory/01_mem-corruption.c

```c
#include <stdlib.h>

int main( int argc, char **argv ) {
    int a[ 2 ];
    int *b = malloc( 2 * sizeof( int ) ), *c;
    a[ 2 ] = 5;          // assign past the end of an array
    b[ 0 ] += 2;         // assume malloc zeroes out memory
    c = b + 3;           // mess up your pointer arithmetic
    free( &( a[ 0 ] ) ); // pass pointer to free() that wasn't malloc'ed
    free( b );
    free( b );  // double-free the same block
    b[ 0 ] = 5; // use a free()'d pointer
    // any many more!
    return 0;
}
```

Memory leaks If we allocate memory, but never free it, we use more and more memory (old memory is inaccessible)

Dynamic data structures We build it using structs that have a pointer to another struct inside them. We have to allocate memory for each element and then add the pointer to another struct. For a generic dynamic data structure, make the element a `void` pointer. This in general is the concept used for functions operating on any data type.

3.3.1 Dynamic Memory Allocation

Memory allocated with `malloc` is typically 8- or 16-byte aligned.

Explicit vs. Implicit In explicit memory management, the application does both the allocation *and* deallocation memory, whereas in implicit memory management, the application allocates the memory, but usually a *Garbage Collector* (GC) frees it.

For some languages, like Rust, one would assume that it does implicit allocation, but Rust is a language using explicit management, it's just that the *compiler* and not the programmer decides when to allocate and when to deallocate.

Assumption in this course: Memory is ***word*** addressed (= 8 Bytes on 64-bit platform).

Goals The allocation should have the highest possible throughput and at the same time the best (i.e. lowest) possible memory utilization. This however is usually conflicting, so we have to balance the two.

Definition: ***Aggregate payload*** P_k: All `malloc`'d stuff minus all `free`'d stuff

Definition: ***Current heap size*** H_k: Monotonically non-decreasing. Grows when `sbrk` system call is issued.

Definition: ***Peak memory utilization*** $U_k = (\max_{i<k} P_i)/H_k$

A bit problem for the `free` function is to know how much memory to free without knowing the size of the to be freed block. This is just one of many other implementation issues:

(1) How much memory to free? → Headers
(2) How do we keep track of the free blocks? I.e. where and how large are they? → Free lists
(3) What do we do with the extra space of a block when allocating a smaller block? → Coalescing
(4) How do we pick a block? → Placement policies
(5) How do we reinsert a freed block into the heap? → When to coalesce

This all leads to an issue known as ***fragmentation***

Definition: ***Internal Fragmentation***: If for a given block the payload (i.e. the requested size) is smaller than the block size. This depends on the pattern of previous requests and is thus easy to measure

Definition: ***External Fragmentation***: There is enough aggregate heap memory, but there isn't a single large enough free block available This depends on the pattern of future requests and is thus hard to measure.

Header : Stores size of block and is usually placed in the word that preceeds the allocated block (standard method).

Free lists

M1 ***Implicit list*** using length: Links all blocks and uses a low-order bit to indicate free / allocated, as for aligned blocks, a / some low-order bit(s) are always 0).
M2 ***Explicit list*** among free blocks using a pointer in the first (and possibly second) word of the block
M3 ***Segregated free list***: different free lists for different size classes
M4 ***Blocks sorted by size***: Using a balanced tree with pointers within each free block and the length is the key

Definition: ***Coalescing*** Connecting two (or more) (free) blocks to form a larger (free) block.

We can do this efficiently in one direction with just a header, however in both directions requires what are referred to as `boundary tags`. They are simply headers on both sides of the block and this allows us to traverse backwards.

We can do coalescing in constant time, by looking at the previous block's footer and next block's header to check if they are free or not.

- If the previous block is free, we can coalesce it by updating its header to include the length of the to be freed block and the middle two words. Same update has to happen to the to be freed block's footer.
- If the next block is free, update its footer to the size of the to be freed block plus the free block's size plus the two words in the middle. Do the same update to the to be freed block's header.

If both blocks are free, then of course we can do this step in one go for both.

Using the headers or boundary tags is just one option to do it and it can be optimized.

Implicit Free List If we use the size that is stored in the header, we know where the next block is going to be already. To know if a block is allocated, we use a low-order bit. This is possible, sincee if the blocks are aligned, then some of the low-order bits are always 0.

Allocating Blocks: We traverse the list, and choose depending on the placement policy: `first fit`, `next fit` or `best fit` policies. When a block is picked, we might want to split it by adding a header to the remaining part.
Freeing Blocks: We can simply clear the allocated flag, which might lead to fragmentation. Thus: Coalesce the freed blocks.

This is the simplest approach, the price we pay is that allocating is linear w.r.t *all blocks* managed by the allocator.

Explicit Free List To improve on this, here we maintain pointers to the next and preferably previous free block(s). This means that all free blocks form a (doubly) linked list, and we don't traverse already allocated blocks.
Since the free blocks aren't used by the program, the list pointers can be stored inside the free blocks.

Allocating Blocks: We remove the element from the list by updating the pointers and we can again use the `first fit`, `next fit` or `best fit` policies.
Freeing Blocks: Requires updating the list pointers, which depends on the list ordering. If boundary tags are used for coalescing, we also have to clear the allocated bit in the footer and header.

The list ordering may be LIFO (last-in-first-out) or address-ordered. LIFO (with `first fit`) allows constant time freeing, while address-ordered provides better memory utilization (linear time freeing).

To prevent fragmentation, we want to use coalescing again, which can be implemented using boundary tags (again leads to linear time coelescing, if the list is FIFO). This time, the pointers need to be updated too. How this works (and how fast) depends on the list ordering.

Segregated Free List An issue with the previous approaches is finding a *large enough* block, which is why finding a fit remains in linear time. Here, we keep separate lists for different *size classes* to make this faster.
For each *size class* a separate free list is maintained. For example, the classes might be:

$$\{1\}, \{2\}, \{3,4\}, \{5-8\}, \ \dots \ , \{1025-2048\}, \{2049-4096\}, \{4097-\infty\} \qquad \text{(Partitions by Powers of 2)}$$

Allocating Blocks: We first check the list for the requested size class, and if no fitting block is found, move up to the next list. If we have reached the last list and there is no suitable block, request new memory using `sbrk()`.
Freeing Blocks: Requires adding the newly freed block into the respective free list, potentially after coalescing.

This leads to an increased throughput and better memory utilization as compared to the previous two. Additionally, this structure opens up many optimization options.

The way splitting, coalescing, etc. is implemented varies a lot by implementation.
For example, we might choose to only coalesce if the length of a certain list has reached a certain threshold.
Similarly, if a list of some size class is empty, we might either request new memory and create a new list, or split a block from a higher size class.

Some options used are: *Simple Segregated Storage*, *Segregated Fits*, *Buddy System*

Fun fact : the GNU `malloc` package uses a Segregated Free List, with *Segregated fits*.

Placement policies

- ***First fit*** Search the list from the beginning, pick first free block that fits. This will usually cause "splinters" at the beginning of the list and can take linear time in the total number of blocks (allocated and free)
- ***Next fit*** Like first fit, but start at point the previous search finished at. This should be faster, however leads to worse fragmentation.
- ***Best fit*** Searches the list and chooses the *best* free block that fits and has fewest bytes left over. Leads to lower fragmentation, but is slower than first fit.

3.3.2 Garbage Collection

The memory manager must somehow be able to tell what memory can be freed. In general, we cannot know if memory is going to be used or not, except if there exists no pointer to it anymore. Garbage collectors use graphs to track pointer availability. In other words, a block is reachable if there exists a path from a root node to it.

An easy GC algorithm is called ***Mark and Sweep***. It has an extra bit in the header called the *mark bit* and can be built on top of malloc/free. The concept is to use malloc until we "run out of space" and to then run these steps:

- ***Mark***: Starts at each root node and sets a mark bit on each reachable block.
- ***Sweep***: Scan all blocks and free all blocks that are unmarked.

3.3.3 Common pitfalls

- Dereferencing bad pointers (e.g. passing an `int` to a function expecting a pointer)
- Reading uninitialized memory (memory allocated with `malloc` should be considered garbage)
- Overwriting memory (if you mess up pointer arithmetic or don't do boundary checks)
- Referencing nonexistent variables (variables go out of scope on function returns, except `static`)
- Freeing blocks multiple times (can corrupt the heap)
- Referencing freed blocks (always `NULL` pointers after using `free()`)
- Failing to free blocks (memory leak incoming and make sure to free the ENTIRE data structure!)

Some of these bugs (especially bad references) can usually be found using a debugger.

Substitute `malloc` with a `malloc` that has extra checking code (like `UToronto CSRI malloc` to detect memory leaks)

Another option is using `valgrind` (a memory debugger). Or, simply don't bother with `C` and use `Rust`.

3.4 Variadic functions

Variadic functions take a variable number of arguments and use the `...` syntax in C. A notable example of such a function is `printf`

File: code-examples/00_c/03_others/00_variadic.c

```c
#include <stdarg.h> // Variadic function utilities
#include <stdio.h>

void print_int( unsigned int num_ints, char *msg, ... ) {
    va_list ap; // keeps track of current argument, similar (in concept) to iterator
    va_start( ap, msg ); // Initialize the iterator based on last fixed arg
    for ( int i = 0; i < num_ints; i++ )
        printf( "int %d = %d\n", i, va_arg( ap, int ) ); // Returns next arg cast to int
    va_end( ap ); // Free up iterator
}
```

3.5 Code Vulnerabilities

A brief interjection on some code vulnerabilities.

System-level protections

- **Compiler-inserted checks** on functions
- **Randomized stack offsets**: Allocate *random* amount on stack before running the program
- **Nonexecutable segments**: Memory needs a special *execute* permission

3.5.1 Buffer overflow

Buffer overflows are a method for code injection on vulnerable code with specific buffer size-checking deficiencies. There are 2 ways to do this:

1. Change a function call or return address
2. Push malicious assembly onto the stack

For example, consider this code:

File: `code-examples/00_c/05_vulnerabilities/01_buffer_overflow_echo.c`

```c
#include <stdio.h>

void echo() {
    char buf[4];    // Limited size
    gets(buf);      // Assumes size matches, does not check!
    puts(buf);
}

int main()
{
    printf("Type a string:");   // No size check enforced!
    echo();
    return 0;
}
```

This is a problem, since `echo` may be compiled to something similar to this:

File: `02_buffer_overflow_echo_asm.s`

```asm
echo:
    subq $24, %rsp      # Allocate stack space for buf
    movq %rsp, %rdi
    call gets
    movq %rsp, %rdi
    call puts
    addq $24, %rsp
    ret
```

Since `buf` is on the stack, and there is no size-enforcement when writing to `buf`, malicious input can write *before* `%rsp`, since the Stack grows downwards. This means stack memory that the program is intending to use again can be modified.

However, inserting exectuable assembly like this usually does not work, since the stack may not be executable due to missing system permission.

The vulnerability above could be fixed by using `fgets(buf, 4, stdin)` instead, which checks the size.

Heap overflow On the heap, buffer overflows work differently, as the heap contains no return addresses. However, the heap stores function pointers, which can be modified. Further, sophisticated attacks can use buffer overflow to potentialy modify pointers in dynamically allocated memory.

3.5.2 Return-oriented Programming

Return-oriented Programming is a more sophisticated exploit, which does not rely on injecting any new code.

The key idea is: Overwrite return addresses and jump to *specific* machine instruction sequences *already present* in process memory.

3.6 Floating Point

Floating point numbers are a representation of real numbers.

Though there are many ways to accomplish this, *IEEE Standard 754* is used practically everywhere, also in x86. This standard is a little more complicated than fractional binary numbers, but has a few numeric advantages, especially for representing very large (very small) numbers.

`float.exposed` is an excellent website to understand floating point by example.

3.6.1 Fractional Binary Numbers

We can represent any real number (with a finite decimal representation) as:

$$d = \sum_{i=-n}^{m} 10^i \cdot d_i \qquad \underbrace{d_m d_{m-1} \cdots d_1 d_0 \,.\, d_{-1} d_{-2} \cdots d_{-(n-1)} d_{-n}}_{d_i \text{ is the } i\text{-th digit of } d \text{ (neg. indices indicate decimals)}}$$

We can use the same idea for Base 2 as well:

$$b = \sum_{i=-n}^{m} 2^i \cdot b_i \qquad b_m b_{m-1} \cdots b_1 b_0 \,.\, b_{-1} b_{-2} \cdots b_{-(n-1)} b_{-n}$$

To get an intuition for this representation, looking at some examples is helpful:

A few observations:

1. Shifting the dot right: Division by 2
2. Shifting the dot left: Multiply by 2
3. Numbers of the form $0.111\ldots$ are just below 1.0
4. Some numbers representable in finite Base 10 are infinite in Base 2, e.g. $\frac{1}{5} = 0.20_{10}$

Binary	Fraction	Decimal
0.0	$\frac{0}{2}$	0.0
0.01	$\frac{1}{4}$	0.25
0.010	$\frac{2}{8}$	0.25
0.0011	$\frac{3}{16}$	0.1875
0.00110	$\frac{6}{32}$	0.1875
0.001101	$\frac{13}{64}$	0.203125
0.0011010	$\frac{26}{128}$	0.203125
0.00110101	$\frac{51}{256}$	0.19921875

A major issue with this representation is that very large (respectively very small) numbers require a large representation. E.g $a_{10} = 5 \cdot 2^{100}$ has the representation $a_2 = 101\underbrace{000000000000000\ldots}_{100 \text{ Zeros}}$. Floating Point is designed to address this.

3.6.2 Floating Point Representation

Floating point numbers instead use the representation:

$$a = \underbrace{(-1)^s}_{\text{Sign}} \cdot \underbrace{M}_{\text{Mantissa}} \cdot \underbrace{2^E}_{\text{Exponent}}$$

Single precision and Double precision floating point numbers store the 3 parameters in separate bit fields s, e, m:

Single Precision:

31: Sign	$30-23$: Exponent	$22-0$: Mantissa

Bias: 127, Exponent range: $[-126, 127]$

Double Precision:

63: Sign	$62-52$: Exponent	$51-0$: Mantissa

Bias: 1023, Exponent range: $[-1022, 1023]$

Most of the extra precision in 64b floating point numbers is associated to the mantissa. Note how double precision is necessary to represent all 32b signed Integers, and not all 64b signed Integers can be represented in either format.

The way these bitfields are interpretd *differs* based on the exponent field e:

1. **Normalized Values**: Exponent bit field e is neither all 1s nor all 0s.
   In this case, E is read in *biased* form: $E = e - b$. The bias is $b = 2^{k-1} - 1$, where k is the amount of bits reserved for e. This produces the exponent ranges $E \in [-(b-1), b]$.
   The mantissa field m is interpreted as $M = 0.m_{n-1} \dots m_1 m_0 + 1$, where n is the amount of bits reserved for m
2. **Denormalized Values**: Exponent bit field e is all 0s.
   In this case, E is read in *biased* form $E = 1 - b$. (Instead of $E = e - b$)
   The mantissa field m is interpreted as $M = 0.m_{n-1} \dots m_1 m_0$ (without adding 1)
3. **Special Values**: Exponent bit field e is all 1s.
   $m = 0$ represents infinitiy, which is signed using s.
   $m \neq 0$ is `NaN`, regardless of what is in m or s.

Why is the Bias chosen this way? It allows smooth transitions between normalized and denormalized values.

3.6.3 Properties

The advantage of having denormalized values is that 0 can be represented as the bit-field with all 0s. Further, this enforces equidistant points for values close to 0, whereas normalized values increase in distance as they move further from 0.

Example 8b Floating Point table to visualize the different cases.

8b precision Floating Point: $\underbrace{0}_{s}\ \underbrace{0000}_{c}\ \underbrace{000}_{m}$

Case	s	e	m	E	Value
Denormalized	0	0000	000	-6	0
	0	0000	001	-6	$\frac{1}{8} \cdot \frac{1}{64} = \frac{1}{512}$
	0	0000	010	-6	$\frac{2}{8} \cdot \frac{1}{64} = \frac{2}{512}$
			$\vdots$		$\vdots$
	0	0000	110	-6	$\frac{6}{8} \cdot \frac{1}{64} = \frac{6}{512}$
	0	0000	111	-6	$\frac{7}{8} \cdot \frac{1}{64} = \frac{7}{512}$
Normalized	0	0001	000	-6	$\frac{8}{8} \cdot \frac{1}{64} = \frac{8}{512}$
	0	0001	001	-6	$\frac{9}{8} \cdot \frac{1}{64} = \frac{9}{512}$
			$\vdots$		$\vdots$
	0	0110	110	-1	$\frac{14}{8} \cdot \frac{1}{2} = \frac{14}{16}$
	0	0110	111	-1	$\frac{15}{8} \cdot \frac{1}{2} = \frac{15}{16}$
	0	0111	000	0	$\frac{8}{8} \cdot 1 = 1$
	0	0111	001	0	$\frac{9}{8} \cdot 1 = \frac{9}{8}$
	0	0111	010	0	$\frac{10}{8} \cdot 1 = \frac{10}{8}$
			$\vdots$		$\vdots$
	0	1110	110	7	$\frac{14}{8} \cdot 128 = 224$
	0	1110	111	7	$\frac{15}{8} \cdot 128 = 240$
Special	0	1111	000	n/a	∞

3.6.4 Rounding

The basic idea of Floating Point operations is:

1. Compute exact result
2. Round, so it fits the desired precision

IEEE Standard 754 specifies 4 rounding modes: *Towards Zero, Round Down, Round Up, Nearest Even*.

The default used is *Nearest Even*[1], which rounds up/down depending on which number is closer, like regular rounding, but picks the nearest even number if it's exactly in the middle.

Rounding can be defined using 3 different bits from the *exact* number: G, R, S

$$a = 1.BB\dots BB\underbrace{G}_{\text{Guard}}\underbrace{R}_{\text{Round}}\underbrace{XX\dots XX}_{\text{Sticky}}$$

1. **Guard Bit** G is the least significant bit of the (rounded) result
2. **Round Bit** R is the 1st bit cut off after rounding
3. **Sticky Bit** S is the logical OR of all remaining cut off bits.

Based on these bits the rounding can be decided:

$$R \wedge S \implies \text{Round up} \qquad G \wedge R \wedge \neg S \implies \text{Round to even}$$

Example Rounding 8b precise results to 8b precision floating point (4b mantissa):

Value	Fraction	GRS	Incr?	Rounded
128	1.000|0000	000	N	1.000
13	1.101|0000	100	N	1.101
17	1.000|1000	010	N	1.000
19	1.001|1000	110	Y	1.010
138	1.000|1010	011	Y	1.001
63	1.111|1100	111	Y	10.000

Post-Normalization: Rounding may cause overflow. In this case: Shift right once and increment exponent.

[1] Changing the rounding mode is usually hard to do without using Assembly.

3.6.5 Operations

Multiplication is straightforward, all 3 parameters can be operated on separately:

$$(-1)^{s_1} M_1 \cdot 2^{E_1} \cdot (-1)^{s_2} M_2 \cdot 2^{E_2} \quad = \quad (-1)^{s_1 \oplus s_2} (M_1 \cdot M_2) 2^{E_1 + E_2}$$

Post-Normalization:

1. If $M \geq 2$, shift M right and increment E
2. If E out of range, overflow (set to ∞)
3. Round M to fit desired precision.

Addition is more complicated: (Assumption: $E_1 \geq E_2$)

$$(-1)^{s_1} M_1 \cdot 2^{E_1} + (-1)^{s_2} M_2 \cdot 2^{E_2} \quad = \quad (-1)^{s'} M' \cdot 2^{E_1}$$

s', M' are the result of a signed align & add.
This means $(-1)^{s_1} M_1$ is shifted left by $E_1 - E_2$, and then $(-1)^{s_2} M_2$ is added.

Post-Normalization:

1. if $M \geq 2$, shift M right, increment E
2. if $M \leq 1$, shift M left k, decrement E by k
3. Overflow E if out of range (set to ∞)
4. Round M to desired precision

3.6.6 Mathematical Properties

Floating point is *almost* an Abelian Group.

- **Closed** under Addition, Multiplication (But may generate `NaN`, $\pm\infty$)
- **Commutative**
- **Not Associative** (Overflow & Rounding)
- $0, 1$ **are Identity**
- **Additive Inverse** (Except $\pm\infty$ and `NaN`)
- **Monotonicity** (Except for $\pm\infty$ and `NaN`)
- **Not Distributive** (Overflow & Rounding)

3.6.7 Floating Point in C

C99 guarantees `float` and `double`, and `long double` is usually interpreted as quadruple precision.

Casting/Conversion between Integer types and `float`, `double` *changes* the bit representation in most cases (e.g. 0 stays the same)

4 The gcc toolchain

The `GNU Compiler Collection`, or short (which is also it's executable name) `gcc`, is a `C` compiler toolchain that is commonly used to compile `C` code on UNIX platforms.

It includes all the necessary tools to compile a `C` program:

- A preprocessor, called `cpp`
- The actual `C` compiler, called `cc` (though on the slides it states it is called `cc1`, but at least in the Arch Linux package, `cc1` is not a thing)
- An `x86` assembler, called `as`
- A static linker, called `ld`

However, the individual parts are usually not called individually, but using the toolchain command `gcc` (and that is usually again abstracted away using CMake or Make, which is in turn commonly called via a build system like Meson to automatically build packages for distribution).

`gcc` has (as of `GCC 15.2.1 20260103` on Arch Linux) about 1000 CLI arguments that can be passed. Below is a list of the most important flags that can be passed, as discussed in the lectures:

Flag	Description
`-E`	Stop after the preprocessor (output is a `.i` file)
`-S`	Stop after the compiler (output is assembly in `.s` file)
`-c`	Stop after the assembler (output is `.o` file)
`-o`	Specify the executable name
`-DNDEBUG`	Removes all assert statements
`-OX`	Optimization level where `X` can be one of `0, 1, 2, 3`
`-g`	Compile with debugging information
`-Wall`	Enable common warnings
`-Wextra`	Enable more warnings
`-Werror`	Makes all warnings errors
`-march=XXX`	Optimize for the architecture (can be e.g. `native`, `x86-64-v4`, ...)
`-fno-tree-vectorize`	Do not vectorize code (`-O3` commonly enables vectorization)

Table 4: Command line flags for GCC

4.1 Compiler optimizations

While the compiler can do quite a bit to speed up code, it can't rework the core logic, as it has to guarantee that the executable does do what was specified in the code.

So, it is really important to not only consider asymptotic runtime (as `100n` and `5n` are both $\mathcal{O}(n)$, but oviously the latter is 20 times faster). We thus need to optimize the algorithms, data representations, loops, etc and for that, we need to properly understand how programs are compiled, executed and how the hadware works.

When using `gcc`, it is usually a good idea to compile a final build with the `-O2` or `-O3` flags.

The `-march` flag was already mentioned in table 4 and can be used if you want to go above and beyond, as it will optimize for the specific hardware. The values that can be passed to `-march` are listed here and even include a specific CPU microarchitecture. For example, to compile for Intel Alderlake (12000 series), you can specify `-march=alderlake`

To understand what you need to optimize, you need to understand what the compiler is good at:

- Register allocation
- Scheduling (i.e. code selection and ordering)
- Dead code elimination
- Eliminating minor (!) inefficiencies

and what it is not good at:

- Improving Asymptotic efficiency (compiler can't turn BubbleSort into e.g. QuickSort)
- Improving the constant factor (if your implementation is slow, it likely won't magically become faster, though some bad practices can be eliminated)
- Overcoming other optimization blockers such as memory aliasing and procedure side-effects

Code motion is a compiler technique, where it moves certain computations out of loops that always produce the same result. However: Always remember that the compiler will be ***conservative***, i.e. it will always err on the side of caution.

Strength reduction is a compiler technique, where e.g. sequences of products are turned into cheaper additions in each iteration. An example is that if you have an operation such as `n * i` in a loop, the compiler might replace that with a variable `ni` that is incremented by `n` in each iteration. Similarly, it might replace `16 * x`, or even worse still, `x / 16` with `x << 4` or `x >> 4`, respectively

Common sub-expressions can be extracted into pre-computations and then only use cheaper operations on the individual steps. A good example is if you are using similar multiplications that then only require one addition or subtraction to get to a result close by.

4.1.1 Optimization blockers

A sure-fire way to make your code slow is by using a large number of procedure calls. They are among the slowest operations in `C`. And, the compiler cannot safely extract the function in a for loop like this:

```c
int i;
for (i = 0; i < strlen(s); i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
        s[i] -= ('A' - 'a');
    }
}
```

The compiler can't safely remove `strlen(s)` from the loop, as it may have side-effects, i.e. may modify other program content other than simply returning a value. Thus, only ever call functions in the loop condition when you need the side-effects and otherwise, pre-compute it and simply use a variable to check against. You can declare a function *side-effect free* using `__attribute__((pure))` in the function declaration. The compiler may then extract `strlen(s)` from the loop.

Another common blocker is memory aliasing. This happens when two pointers point to the same address and of course, since we can do pointer arithmetic, it is very easy to do that in `C`. The easiest way to prevent this from happening is to use local variables where possible, such that they do not need to be passed in using a pointer.

Normally the compiler assumes there can be another pointer that accesses the memory pointed to by this pointer. If you use the `restrict` keyword on the variable (i.e. in a function declaration, we have `void test(double restrict *a)`), the compiler will assume that for the lifetime of this pointer, there are no other pointers that will be used to access the memory to which it points.

Another technique to improve throughput for something like matrix multiplications is to do it in blocks due to the way caching works. Since the compiler doesn't *understand* your code, it can't do this for you (as it assumes associativity of the operation)

4.2 Linking

Linking is the final step in the compilation pipeline: separately compiled object files are combined into an executable.

The advantages of using Linkers are clear:

1. **Separate Compilation**: Changing one source file requires only recompiling that file.
2. **Space Optimization**: Executable code only contains functions (e.g. from libraries) that are actually used.

4.2.1 Symbol Resolution

The first step during Linking is Symbol Resolution.

In the context of Linking, all variables and functions are considered *Symbols*. Compilers store all symbol definitions in a *Symbol Table*. The linker associates symbol references with *exactly one* definition.

Definition: Symbol types

- **Global Symbols** can be referenced by other modules (e.g. `non-static` in `C`)
- **External Symbols** are referenced globals defined elsewhere
- **Local Symbols** are defined and referenced exclusively in one module (e.g. `static` in `C`)

Note: Local linker symbols and local program variables are *not* the same.

Definition: Symbol strength

Duplicate symbols either lead to linking errors (`-fno-common`, the default) or compile (`-fcommon`)

- **Strong Symbols** are procedure names and initialized globals
- **Weak Symbols** are uninitialized globals (on `-fcommon`)

in `C`, function symbols can explicitly be declared weak using:

```
#pragma weak func
__attribute__((weak)) __ func()
```

Duplicate Handling The linker uses these definitions to handle duplicates:

1. Given multiple strong symbols are illegal
2. Given a strong symbol and multiple weak symbols, pick the strong symbol
3. Given multiple weak symbols, choose an *arbitrary* one

4.2.2 Relocation

The second step during Linking is Relocation.

Code and data sections of separate sources are combined, and symbols are relocated from relative locations (in `.o` files) to absolute locations (in `.exe` files)

Command line order matters for this, since the Linker will scan `.o` and `.a` files in this order. In general, libraries should therefore be linked *last*.

4.2.3 Packaging Libraries

Using just the Linker, there are only 2 inconvenient ways to package libraries:

1. All functions into 1 file $\mapsto$ linking unnecessarily big objects.
2. One function per file $\mapsto$ Requires linking a lot of files, annoying for programmer.

Static Libraries solve this: The linker looks for functions inside the static library, and only links matching archive *members* into the executable. However, these come with issues too:

1. Duplication in stored executables (e.g. `libc.a` functions)
2. Duplication in running executables
3. Any fix in a library requires importing applications to explicitly relink

Shared Libraries solve this: These are linked at load-time or during run-time. Another advantage is that *multiple* processes can use the same shared library simultaneously. This is how, for example, `libc` is packaged.

During runtime, shared libraries can be loaded using `dlopen`:

File: code-examples/00_c/04_toolchain/01_dynamic_linking.c

```c
#include <stdio.h>
#include <dlfcn.h>  // contains addvec

int x[2] = {1, 2}; int y[2] = {3, 4}; int z[2];

int main(int argc, char *argv[])
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);       // Declaration
    char *error;

    handle = dlopen("./libvector.so", RTLD_LAZY);   // Load .so, makes addvec usable
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    addvec = dlsym(handle, "addvec");               // get a pointer to advec
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    addvec(x, y, z, 2);                             // Now callable like any other function
    printf("z = [%d %d]\n", z[0], z[1]);

    if (dlclose(handle) < 0) {                      // Unload shared library
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

4.3 File types

The most common file types used during compilation:

- **Source Code File** (`.c`) Uncompiled source code in `C`.
- **Relocatable Object File** (`.o`) Code & Data in a format ready for linking.
- **Shared Object File** (`.so`) Special object file, can be loaded & linked dynamically: at load or run time.
- **Exectuable File** Code & Data in a format that can be directly copied into memory & run.
- **Archive files** (`.a`) concatenate related `.o` files into one `.a`, with an index.

Alternate names On Windows, `.dll` files are used instead of `.so` and are called *Dynamic Link Libraries*.

4.3.1 Executable and Linkable Format (ELF)

The standard unified format for all object files (`.exe`, `.o`, `.so`) in use since UNIX.

Section	Content
ELF header	contains basic information: Word size, byte ordering, file type, machine type
Segment header table	page size, virtual address memory segments, segment sizes
`.text`	actual code
`.rodata`	(Read-only data) for example, jump tables
`.data`	initialized global variables
`.bss`	uninitialized global variables
`.symtab`	symbol table, procedure and static variable names, section names & locations.
`.rel.text`	relocation info for `.text`, e.g. addresses of instructions that need modifying
`.rel.data`	relocation info for `.data`, e.g. addresses of pointers that need modifying
`.debug`	info for symbolic debugging (`gcc -g`)

5 Hardware

Hardware was only touched on briefly in this course, and knowledge of a previous Computer Architecture course is assumed. This section is similarly incomplete, only focusing on the specific topics covered in the lecture.

5.1 Code Optimizations

By understanding the underlying hardware, simple code optimizations can be made to improve performance significantly.

To analyze program performance, a solid benchmark and performance metrics need to be chosen.

Common metrics are:

1. **Latency** i.e. how long does a request take?
2. **Throughput** i.e. how many requests (per second) can be processed?

For pipelined processors, throughput is more tricky to specify. Usually, the steady-state throughput is considered.

Definition: **Cycles per Element** (CPE) is used as a performance metric for operations on vectors/lists.
The execution time t can then be formulated as: $t = \text{CPE} \cdot n + \text{Overhead}$. ($n$ is the vector/list size)

Definition: **Cycles per Instruction** (CPI) is the inverse of *Instructions per Cycle*

CPI can be further divided into $\text{CPI} = \text{CPI}_{\text{Base}} + \text{CPI}_{\text{Stall}}$ assuming a pipelined processor, which may stall for many reasons: Data hazards, control hazards, memory latency ...

Definition: **Clock Cycle Time** (CCT) is the inverse of *Clock Frequency*

Using these we can define Program Execution Time: $t = n \cdot \text{CPI} \cdot \text{CCT}$ (n is the instruction count)

Definition: **Instruction Level Parallelism** (ILP) is the natural parallelism occuring through independent instructions.

Superscalar Processors capitalize on ILP by executing multiple instructions per cycle. The instructions are (usually) scheduled dynamically. This is particularly great since it requires no effort from the programmer.

Different instructions generally may take vastly different amounts of time, though this can be reduced via pipelining and multiple execution units.

Example This table is specific to an Intel Haswell CPU.

Instruction	Latency	Cycles/issue
Load/store	4	1
Int Add	1	1
Int Multiply	3	1
Int/Long Divide	3–30	3–30
FP Multiply	5	1
FP Add	3	1
FP Divide	3–15	3–15

Performance of these operations is **Latency bound** if they are sequential, **Throughput bound** if they can run parallel.

Example **Loop unrolling** can increase performance for latency bound operations. (Optimization flags *should* do this)

Unrolling can be done to an arbitrary degree, but there are diminishing returns at some point. An ideal unrolling factor must be found experimentally.

Example **Reassociation** can also improve performance for parallelizable operations, if it breaks some sequential dependency.

Example **Separate Accumulators** can improve performance for latency bound operations too, as separate load/store units may be used.

5.2 Vector Operations

Extreme performance gains beyond the results of the previous section can be gained using hardware vector registers on supported CPUs.

Example In Intel AVX2, 256b vector registers like `%ymm0`, `%ymm1` can be used to perform component-wise single/double precision FP operations.

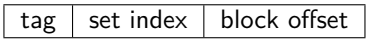
```
vaddsd  %ymm0, %ymm1, %ymm1      # Comp.-wise 32b FP add
vaddsd  %ymm0, %ymm1, %ymm1      # Comp.-wise 64b FP add
```

5.3 Caches

Processors generally improve quicker than memory speed does. Therefore, optimizing memory is necessary, and this is what Caches do.

Structure Caches can be defined using S, E, B s.t. $S \cdot E \cdot B =$ Cache Size.
$S = 2^s$ is the set count, $E = 2^e$ is the lines per set, and $B = 2^b$ is the bytecount per cache block.

Address Using the above, the address can be separated into fields which dictate the cache location:



Since we have $S = 2^s$ sets and $B = 2^b$ bytes per block, we need s bits for the set index, b bits for block offset. The remaining part (tag) is stored with the cache block and needs to match for a cache hit.

Definition: Direct-mapped i.e. $E = 1$ (1 cache line per set only).

Definition: 2-way Set-Associative i.e. $E = 2$ (2 cache lines per set).

Example The importance of caches can quickly be seen when looking at the memory hierarchy:

Cache type	What is cached?	Where is it cached?	Latency (cycles)	Managed by
Registers	4/8-byte words	CPU core	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	64-byte blocks	On-chip L1	1	Hardware
L2 cache	64-byte blocks	On-chip L2	10	Hardware
Virtual memory	4 kB page	Main memory (RAM)	100	Hardware + OS
Buffer cache	4 kB sectors	Main memory	100	OS
Network buffer cache	Parts of files	Local disk, SSD	1,000,000	SMB/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

5.3.1 Cache Addressing Schemes

The cache can see either the virtual or physical address, and the tag and index do *not* need to both use the physical/virtual address.

Indexing	Tagging	Code
Virtually Indexed	Virtually Tagged	VV
Virtually Indexed	Physically Tagged	VP
Physically Indexed	Virtually Tagged	PV
Physically Indexed	Physically Tagged	PP

5.4 Virtual Memory

Conceptually, Assembly operations treat memory as a very large contiguous array of memory: Each byte has an individual address.

```
movl    (%rcx), %eax    # Refers to a Virtual Address
```

In truth of course, this is an abstraction for the memory hierarchy. Actual allocation is done by the compiler & OS.

The main advantages are:

- Efficient use of (limited) RAM: Keep only active areas of virtual address space in memory
- Simplifies memory management for programmers
- Isolates address spaces: Processes can't interfere with other processes

5.4.1 Address Translation

Address translation happens in a dedicated hardware component: The Memory Management Unit (MMU).

Virtual and Physical Addresses share the same structure, buth the VPN is usually far longer than the PPN, since the virtual space is far bigger. Offsets match.

Virtual:	V. Page Number	V. Page Offset

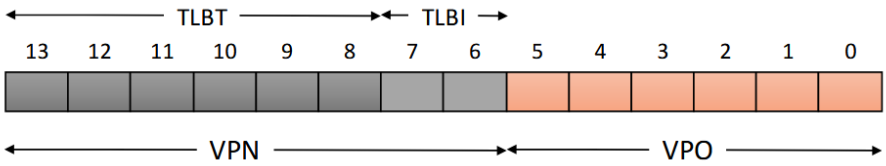
Physical:	P. Page Number	P. Page Offset

The Page Table (Located at a special Page Table Base Register (PTBR)) contains the mapping VPN $\mapsto$ PPN. Page Table Entries (PTE) are cached in the L1 cache like any other memory word.

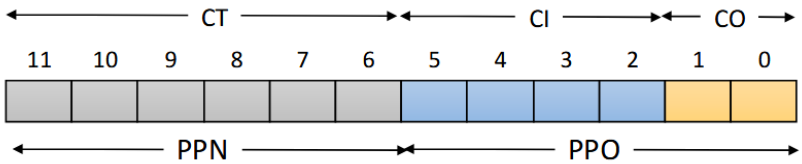
The Translation Lookaside Buffer (TLB) is a small hardware cache inside the MMU, which is faster than an L1 hit.[2]

Example We consider $N = 14$ bit virtual addresses and $M = 12$ bit physical addresses. The offset takes 6 bits.[3]

If we assume a TLB with 16 entries, and 4 way associativity, the VPN translates like this:



Similarly, if we assume a direct-mapped 16 line cache with 4 byte blocks:



Multi-Level page tables add further steps to this process: Instead of a PT we have a Page Directory Table which contains the addresses of separate Page Tables. The top of the VPN is used to index into each of these, which technically allows any depth of page tables.

5.4.2 x86 Virtual Memory

In x86-64 Virtual Addresses are 48 bits long, yielding an address space of 256TB.
Physical Addresses are 52 bits, with 40 bit PPNs, yielding a page size of $4KB$.

[2] In practice, most address translations actually hit the TLB.
[3] The images in this example are from the SPCA lecture notes for FS25.

5.5 Exceptions

Control flow is mainly dictated by program state, and manipulated using jumps, branches, calls and returns. To react to changes in system state, exceptional control flow is used instead.

Low level mechanisms

- Hardware Exceptions
- Exceptions via combination of Hardware and OS software

High level mechanisms

- Process context switch
- Signals
- Nonlocal jumps
- Language-level exceptions (e.g. Java)

Generally, on an exception, control is transferred to a handler specific to the type of exception, which investigates the situation and returns control upon success. Mostly, this is handled via a *Exception Table* which is allocated on boot. On exception, this table is indexed depending on the type of exception to locate the corresponding handler. This causes a switch to Kernel Mode.

Definition: **Exception**: A control transfer to the OS in reponse to an event

- **Synchronous**: result of executing some instruction
- **Asynchronous**: result of an event external to the processor

Type of exception	Cause	Async/Sync
Interrupt	Signal from I/O device	Async
Trap	Intentional exception	Sync
Fault	Potentially recoverable error	Sync
Abort	Nonrecoverable error	Sync

5.5.1 Synchronous Exceptions

Definition: **Trap** is an intentional exception that transfers control back to the next instruction

For example, opening a file in `C` executes a trap via a system call.

Definition: **Fault** is an unintentional, possibly recoverable exception. Either re-executes faulty instruction or aborts

For example, page faults, protection faults, floating point exceptions

Definition: **Abort** is unintentional and unrecoverable. Always aborts the program.

For example, a machine error.

5.5.2 Asynchronous Exceptions

Asynchronous Exceptions are indicated by setting the processor's (physical) interrupt pin.

For example,

- **Interrupts** are actions like network data arrival or hitting a key on the keyboard
- **Hard Reset Interrupts** are executed by hitting the system reset button
- **Soft Reset Interrupts** are caused by, for example, hitting `CTRL+ALR+DEL` (on Windows)

5.6 Multi-Core

5.6.1 Background

In the early days of computer hardware it was fairly easy to get higher performance due to the rapid advances in transistor technology. However, today, what is known as Moore's Law (i.e. that the transistor count of integrated circuits doubles every two years). However, due to power constraints and the slowing down of advances in transistor technology, the transistor count growth has slowed down quite a bit since the beginning of the century and is predicted to further stagnate as time goes on.

This leads to various issues, among others, the performance of CPUs isn't going up as quickly anymore as it used to. Additionally, due to power constraints, building faster and faster single-core CPUs is not possible and the advances in that field have slowed to a crawl.

To mitigate and offset these issues, manufacturers started to add multiple cores to parallelize operations. This however brings a whole host of new issues with it, for example, how do you make sure that no data races occur, how do you schedule, etc? These questions have mostly been answered in the course Parallel Programming, so we will not cover that here.

The only reason transistor count is still growing at a seemingly constant rate today is that manufacturers manage to cram more and more cores into a CPU. But even that has slowed down in recent years.

While in 2019 a highest core count AMD EPYC CPU (i.e. the EPYC 7742 from the ROME family) had 64 Zen 2 cores, in 2025 the highest core count EPYC CPU (i.e. the EPYC 9965 from the EPYC Turin Dense Family) had 192 Zen 5c cores, where the highest full core CPU was the EPYC 9755 (from the EPYC Turin family), which had 128 Zen 5 cores.

The way they manage this while not hitting the power wall is by making the CPUs physically larger. While a consumer Ryzen 9 9950X3D (the fastest consumer CPU at the time of writing) easily fits into the palm of even a small hand, an EPYC Turin CPU is so large that it covers most of even a big hand.

5.6.2 Limitations

The Power Wall More and more transistors need more and more power, thus leading to power delivery and dissipation becoming and issue. To compute the power dissipation, use the formula $P_{diss} = P_{dyn} + P_{leak} + P_{short}$, where $P_{dyn} = CV^2f$ (with C the capacitance, V the supply voltage and f the processor frequency) is the dynamic power, P_{leak} the leakage power (see DDCA) and P_{short} the short circuit power while switching.

At some point the chip becomes almost impossible to cool. A great example of a CPU series that suffers from this is the Intel Rocket Lake CPUs. The Intel Core i9-14900K is notoriously hot-running, using almost 300 watts for a very small chip and thus runs very hot.

Thus, to further increase performance, chip designers are trying to make the hardware more efficient, which allows them to further boost performance with extra power headroom.

The Memory Wall Between 1985 and 2005, CPU performance has increased on average by 55% a year, whereas memory throughput has only increased by roughly 10% a year. Thus, performance has more and more become limited by memory performance rather than pure CPU performance and to this day is the largest overhead in most applications.

The ILP Wall While it is possible to improve single core performance using instruction-level parallelism, this has been thoroughly exhausted and is not a feasible way to significantly improve CPU performance.

Around 2003, all of these walls were hit simultaneously, as they hit a power wall and thus could not clock the processors any higher, the memory access times were the limiting factors and ILP was almost completely exhausted, as not enough parallel instructions existed in code.

Current trends are a reduction in clock frequency in favour of more parallelism in the hardware, e.g. by providing more cores, or better caching, branch prediction, etc.

5.6.3 Coherency and Consistency

Definition: Coherency The values in cache all match each other and the processors all see a coherent view of the memory

Definition: Consistency The order in which changes are seen by different processors is consistent

Most modern system's CPU cores are caches coherent, i.e. it behaves as if all cores access a single memory array. This leads to one big advantage: It is easy to program, however is hard to implement in hardware and memory is also slower as a result.

Memory consistency on the other hand is not standardized across manufacturers. We are asking questions like what happens if several processors read and write data, which value is read by each one of them? That question is not easy to answer and there is also more than one "correct" answer. The key though is to *have* an answer.

Definition: **Program order** is the order in which a program on a processor *appears* to issue reads and writes. This only refers to local reads and writes and even on a uniprocessor (in other words, a single core processor), it does not correspond to the order in which the CPU issues them

Definition: **Visibility order** is the order in which all reads and writes are seen by one or more processors. This refers to all operations on the machine and might not be the same for all processors. Each processor then reads the value written by the last write in visibility order.

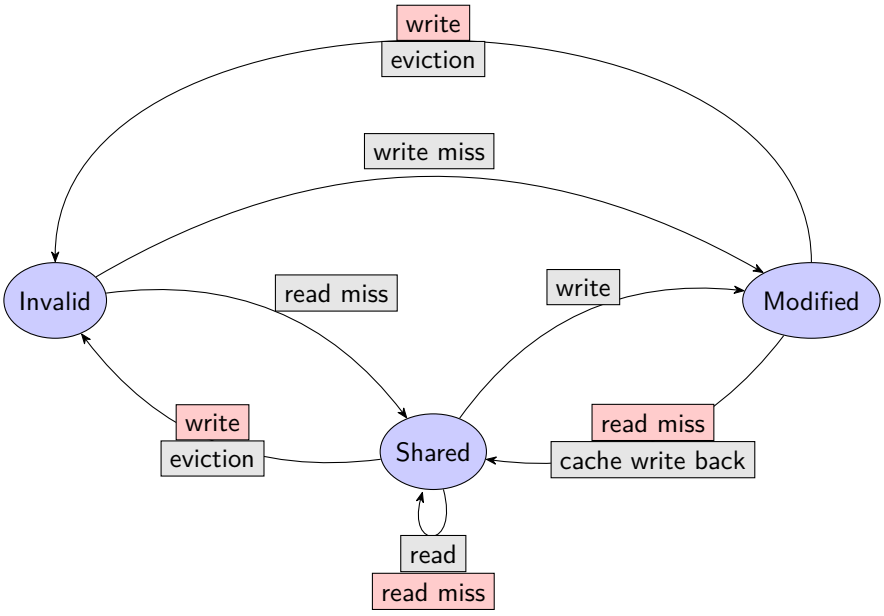
Definition: **Sequential consistency** Operations from a processor appears to all others in ***program order*** and every processor's visibility order is the same *interleaving* of all the program orders. For that to work, every processor has to issue memory operations in program order, the RAM then has to totally order all operations and the operations have to be globally atomic.

You can imagine that to work as though the each processor issues a memory operation and the memory picking a (random) processor to process the request from, which it completes fully and then chooses another processor.

Advantages include being easily understandable for programmers, it being easier to write correct code for and it makes code more easily automatically analyzable. On the other hand, it is hard to make it fast, as we cannot reorder reads or writes (which can often speed up processing) and we cannot combine writes to the same cache line.

In Hardware it can be challenging to maintain proper sequential consistency, as multiple caches could hold invalid lines. This can however be worked around in hardware.

Snoopy caches are caches that "snoop" on reads and writes from other processors and thus, if a line that is valid in the local cache is written by another processor, the local line is invalidated. A write-through cache makes life a bit easier, but it can also work with a write-back cache, if cache lines can be marked as dirty (i.e. modified). It also requires a cache coherency protocol. A simple example is the `MSI` protocol, where a line can have three states (modified, shared, invalid). It basically forms a finite state machine that looks a bit like this:



As nice as MSI is, as basically everything that is simple, it comes with issues, primarily here that it introduces unnecessary broadcasts.

MESI is an extension to the MSI protocol in which the processor gets to know that it is the only reader of a block. It has four states:

- ***Modified***: This is the only copy, but it's modified.
- ***Exclusive***: This is the only copy and it is not modified
- ***Shared***: This might be one of several copies, all clean
- ***Invalid***

When accessing cache, it signals a remote processor that it has hit the local cache. The cache can then load a block in either *shared* or *exclusive* states depending on whether or not the block is a HIT in the remote processor cache.

This finite state machine is much more complex and can be found on slide 55 in the lecture slides of lecture 20.

MOESI AMD then added an owner state, in which the line can be modified, but there exist dirty copies in other caches. It has the benefit of being more quickly readable, by using the owner's cache. This of course is only beneficial if the latency to the remote cache is lower than to main memory, which in case of AMD CPUs, it is starting with the Zen 3 architecture and thus the Vermeer series of desktop CPUs (i.e. Ryzen 5000 series), as they are the first AMD CPUs with a unified L3 cache for each CCX, compared to unified cache for only four cores.

MESIF Intel added a forward state, in which cache requests are forwarded to the most recent cache line. Again, we only benefit from this if cache latency is lower than main memory latency and thus, Alder Lake (12000 series) and later benefit from the more than previous generations. (technically, also Intel 4004 in 1971 until the first Pentium, but they are hardly relevant today)

5.6.4 Relaxing Sequential Consistency

As we have outlined, sequential consistency may not be desirable when trying to build a high-performance system. We thus may want to relax sequential consistency. A primary reason to this is out-of-order execution giving a massive speed boost, as we do not have to wait for slow memory accesses to finish and can already compute what we have ready. Luckily, there are plenty of ways to work around this. For example, we can make later writes bypass earlier writes, later reads bypass earlier writes, break write atomicity (i.e. the order is not fixed anymore) or we cannot make any ordering guarantees at all.

x86 introduces specific instructions for synchronizing, for example `lfence` (load fence), `sfence` (store fence), `mfence` (memory fence) and others. It is typical for an x86-64 processor to implement relaxed sequential consistency and this is sometimes referred to as Total Store Ordering (TSO).

As a general rule, the weaker the consistency model is, the quicker it runs and the cheaper it is in hardware. However too week a consistency model and some algorithms will simply stop working correctly.

Definition: Barriers / Fences are synonyms and are used to stop either the compiler or the CPU from reordering instructions or statements for order-critical operations.

Compiler barriers If we use `gcc`, we can use the following compiler intrinsic to stop it from reordering visible loads and stores:

```
__asm__ __volatile __ ("" ::: "memory");
```

The above intrinsic will also apply a memory barrier, which in terms of assembly code will use the `mfence` instruction. This instruction stops the CPU reordering past it, i.e. any instruction before the fence cannot happen after one behind the fence. However, any instructions past the fence are fair game and can be reordered (i.e. two instructions behind the fence can be reordered).

If we only need this for stores or loads, we can use `lfence` or `sfence`, respectively.

5.6.5 Multicore synchronization

There are two main ways to synchronize, which are:

1. ***Atomic operations*** such as `TAS`, `CAS`, etc. It does still have ordering constraints specified in the memory model
2. ***Interprocessor interrupts*** (IPIs) This invokes the interrupt handler on remote CPU, but is VERY slow (500+ cycles) and thus often avoided, except in the OS

TAS (Test-and-Set) We can only set to the memory location using TAS if said location is 0. It can thus be used for a mutex, with a simple spinlock, which is simple to implement and often the fastest if the lock isn't held for long. Since we most commonly do not read a value of `0` in the lock memory location, we can use a TATAS (Test And Test-and-Set) lock to reduce the performance overhead.

File: `code-examples/03_hw/01_tas.c`

```c
void acquire( int *lock ) {
    while ( TAS( lock ) == 1 );
}

void acquire_tatas( int *lock ) {
    do {
        while ( *lock == 1 );
    } while ( TAS( lock ) == 1 );
}

void release( int *lock ) {
    *lock = 0;
}
```

A word of caution: Do not use TAS to check if a value has changed outside a lock. It will most likely not not work in `C` and almost certainly not in `Java` or any higher level languages

CAS (Compare-and-Swap)

5.7 Devices

From a programmer's perspective a Device can be seen as:

- Hardware assessible via software
- Hardware occupying some bus location
- Hardware mapping to some set of registers
- Source of interrupts
- Source of direct memory transfers

5.7.1 Device Registers

Sometimes devices expose register: The CPU can load from these to obtain e.g. status info or inpur data. The CPU can store to device registers to e.g. set device state or write output.

Device registers can be addressed in 2 different ways:

1. **Memory Mapped**: Registers *appear* as memory locations, access via `movX`
2. **IO Instructions**: Special ISA instructions to work with devices

It's important to note: despite *appearing* as memory, device registers behave differently: State may change without CPU manipulation and writes may trigger actions. The specific way this interaction works is device-specific.

Example A very simple device driver in `C` may look like this:

File: code-examples/03_hw/00_driver.c

```c
#define UART_BASE 0x3f8                     // Registers specified by
#define UART_THR (UART_BASE + 0)            // the device
#define UART_RBR (UART_BASE + 0)
#define UART_LSR (UART_BASE + 5)

void serial_putc(char c) {
    while( (inb(UART_LSR) & 0x20)== 0);     // Wait until FIFO can hold more chars
    outb(UART_THR, c);                      // Write character to FIFO
}

char serial_getc() {
    while( (inb(UART_LSR) & 0x01) == 0);    // Wait until there is a char to read
    return inb(UART_RBR);                   // Read from the receive FIFO
}
```

Of course, a proper driver would also include error handling, initialization and wouldn't spin to wait. To avoid waiting, interrupts are usually used.

Caches For Device Registers, the cache must be bypassed. Memory mapped IO causes a lot of issues for caching:

1. Reads can't be cached (Value may change independent of CPU)
2. Write-back doesn't work exactly (Device controls when the write happens)
3. Reads and writes can't be combined into 1 cache-line

5.7.2 Direct Memory Access

Direct Memory Access (DMA) requires a dedicated DMA controller, which is generally built-in nowadays. DMA allows bypassing the CPU entirely: Data is transferred directly between IO device and memory. This is especially useful for large transfers, but not small transfers due to the induced overhead.

The key advantage is that data transfer and processing are decoupled.
The CPU never needs to deal with copying between device and memory, and the CPU cache is never polluted.

The key disadvantage is that Memory is inconsistent with the CPU cache. This is addressed in various ways:

1. CPU may mark DMA buffers as *non-cacheable*
2. Cache can *snoop* DMA bus transactions (Doesn't scale well, only for small systems)
3. OS can explicitly flush/invalidate cache regions. (Usually done by a device driver)

Another issue is that DMA addresses are *Physical*. The OS (via device drivers) must *manually* translate these to virtual addresses. Some systems also contain a dedicated component called IOMMU that deals with this.

5.7.3 Device Drivers

Device drivers are programs used by the OS to communicate with devices. In a nutshell, the driver is the only program that *directly* interacts with the device, any other program talks to the device *through* the driver (which ideally abstracts away a lot of the process).

Intuitively, both the driver and device can be thought of as state machines, which affect eachother.

Definition: **Descriptor Ring** is a type of buffer commonly used to interact with devices. The datastructure is a looped queue (ring): Device reads from the head, OS writes at the tail. The space beyond is then "owned" by the device/OS.

This can either be implemented as contiguous memory or using pointers (which is mainly what is done in practice, for flexibility). Overruns (Device has no buffers for received packets) and Underruns (CPU has read all received packets) are usually handled sensibly: i.e. the CPU waits for an interrupt or the device will simply wait.

Parallel Programming: These are producer/consumer queues! But these use messages instead of mutexes and monitors.

Remember: Rust and the like have an `unsafe` block... C's equivalent to this is

```c
int main( int argc, char *argv[] ) {
    // Unsafe code goes here
}
```

i.e. ***YOU are the one that makes C code safe!***