Abstract
Systems Programming
and
Computer Architecture

Roman Böhringer, February 2019

Table of contents

1 Introduction

Five realities we face in computer science are:

1. **int's and float's are not numbers:** $x^2 \geq 0$ is true for floats, but not for all ints (overflow). $(x + y) + z = x + (y + z)$ is true for signed and unsigned ints, but not for floats. Therefore, not all "usual" mathematical properties can be assumed.
2. **You've got to know assembly:** Understanding assembly is key to machine-level execution model, especially for behavior of programs in presence of bugs, tuning program performance, implementing system software and creating / fighting malware.
3. **Memory matters. RAM is not a realisitic abstraction:** Memory is not unbounded, performance is not uniform (caches / virtual memory) and memory is typed (different kinds of memory behave differently).
4. **There's much more to performance than asymptotic complexity:** Constant factors matter, even exact operation count does not predict performance.
5. **Computers don't just execute programs, programs don't just calculate values:** They need to get data in and out and they communicate over networks

1.1 Byte-ordering

In big-endian format, the most significant byte (the byte containing the most significant bit) is stored first i.e. has the lowest address. In little-endian format, the least significant byte is stored first. This is illustrated here for the value "01020304" (decimal 16.909.060):

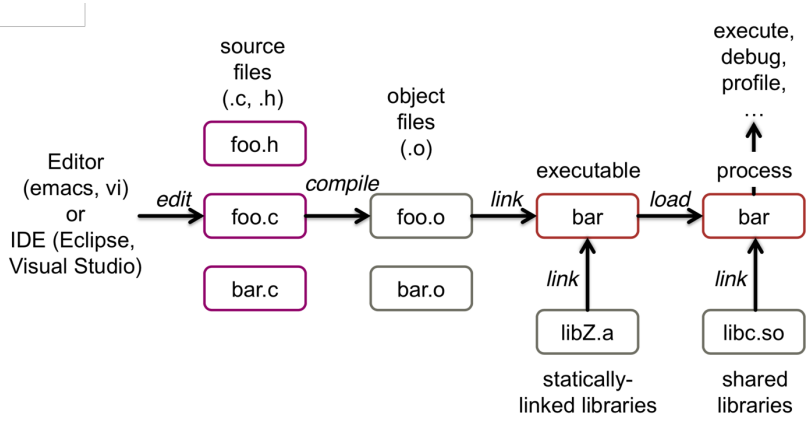
Adresse	Big Endian	Little Endian
10000	01	04
10001	02	03
10002	03	02
10003	04	01

2 The C Programming Language

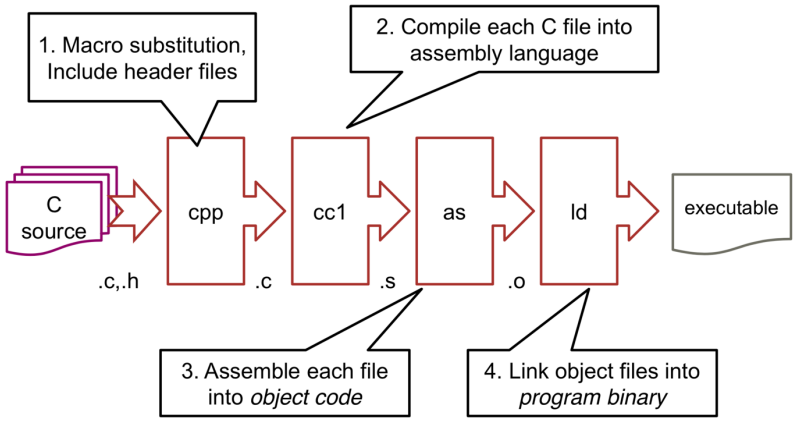
The C Programming Language was developed from 1969-1972 by Dennis Ritchie. It was highly influenced by the DEC PDP-11 architecture and is portable across many architectures. There are many standards, among others K&R C, ANSI C, C99 and C11. C has a powerful macro preprocessor (cpp). There are no objects, classes, features, methods or interfaces and the types are mostly just what the hardware provides (but there are type constructors to build structured types). There are no exceptions, the convention is to use integer return codes. Furthermore, there is no automatic memory management, heap structures are explicitly created and freed. Pointers provide direct access to memory addresses and are weakly typed by what they point to.

2.1 Workflow / Toolchain

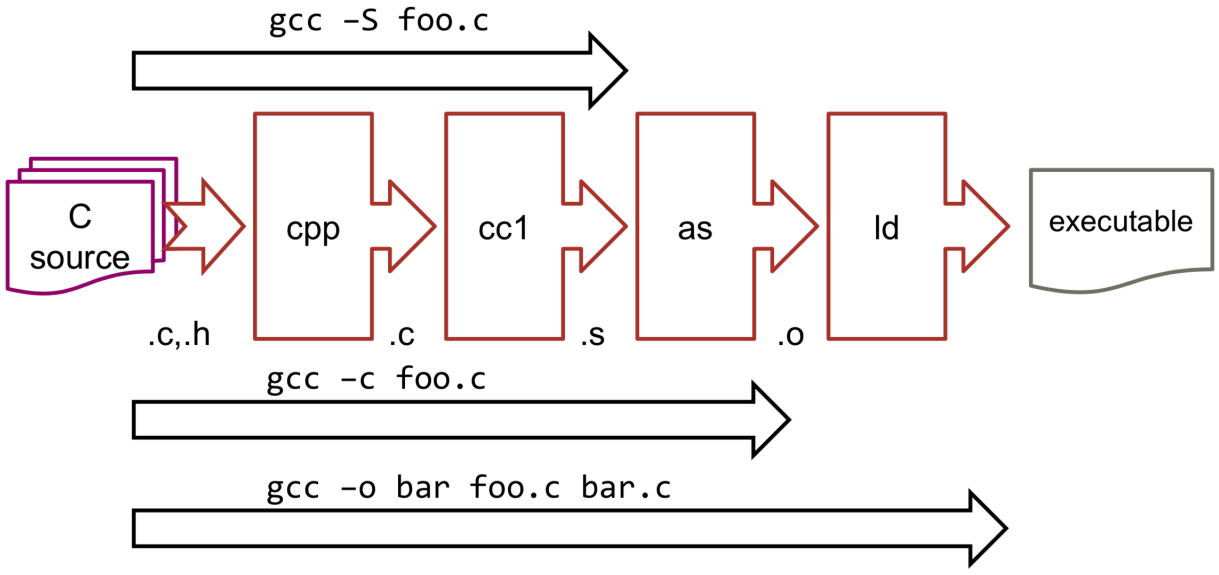
The workflow from writing to executing C programs looks like this:



The GNU gcc toolchain is a toolchain that automates many of these steps:



But by providing the right parameters, it is also possible to get intermediate results:



2.1.1 The C Preprocessor

The C preprocessor replaces `#include <file1.h>` and `#include "file2.h"` statements by the content of the file (where <> is used for system headers and "" for own headers). With `#define`, one can define tokens to replace in the source code. The preprocessor also supports conditionals (`#if … #else … #endif`) where the expression must be literals and macros only.

2.2 Control Flow in C

The following control flow statements are like in Java, C# or C++:

```
if (Expression) Statement_when_true
        else  Statement_when_false
```

```
switch (Expression) {
        case Constant_1 :  Statement;
break;
        case Constant_2 :  Statement;
break;
        …
        case Constant_n:  Statement;  break;
        default:  Statement;  break;
}
```

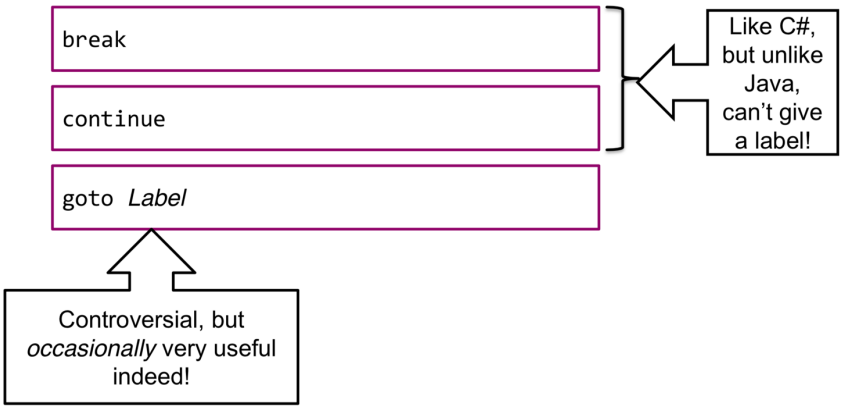
```
return (Expression)
```

```
for (Initial;  Test;  Increment)  Statement
```

```
while (Expression)  Statement
```

```
do  Statement  while (Expression)
```

The following control flow statements differ from Java:



`printf` is used for basic I/O, it is possible to pass an arbitrary amount of arguments to it (but they must match the format). The most important prinft format specifiers are:

%c	character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number

%f	floating-point number
%i	integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign
\%	print a percent sign

2.2.1 Goto

There are very few reasons to use `goto`. It can be used for early termination of nested loops (where one would use `break(label)` in Java) or for nested cleanup code.

2.2.2 setjmp / longjmp

`int setjmp(jmp_buf env)` sets up the local jmp_buf buffer and initializes it for the jump. This routine saves the program's calling environment in the environment buffer specified by the `env` argument for later use by `longjmp`. If the return is from a direct invocation, `setjmp` returns 0. If the return is from a call to `longjmp`, `setjmp` returns a nonzero value.

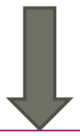
`void longjmp(jmp_buf env, int value)` restores the context of the environment buffer env that was saved by invocation of the `setjmp` routine in the same invocation of the program. The value specified by value is passed from `longjmp` to `setjmp`. After `longjmp` is completed, program execution continues as if the corresponding invocation of `setjmp` had just returned. If the value passed to `longjmp` is 0, `setjmp` will behave as if it had returned 1; otherwise, it will behave as if it had returned value.

2.3 Basic types in C

Declarations are like Java or C#. Inside a block, the scope is just the block. When using the `static` keyword, the value persists between calls. Outside a block, the scope is the entire program. With the `static` keyword, the scope is limited to the file (compilation unit).

Sizes of types differ per architecture, typical values are:

- Types and sizes:



C data type	Typical 32-bit	ia32	Intel 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	8	8	8
long double	8	10/12	10/16

Integers are signed by default. C99 introduced `stdint.h` that provides normalized types, e.g. `int32_t` or `uint32_t` (signed / unsigned 32-bit integer).

Historically, boolean values are just integers, where zero is false and anything non-zero is true. Negation (`!`) turns zero into non-zero and vice-versa. C99 introduced a new `bool` type via `stdbool.h`

Any statement in C is also an expression, therefore code like this is possible:

```
int rc;
if (rc = call_some_fn()) {
   fprintf(stderr, “Failed with return code %d\n”, rc);
   exit(1);
}
// Carry on: call succeeded.
```

There is also a type called `void` that has no value. It is used for untyped pointers (`void *`) and declaring functions with no return value.

2.4 Operators

Operators are similar to Java:

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
&	Left-to-right
^	Left-to-right
|	Left-to-right
&&	Left-to-right
||	Left-to-right
?:	Right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Right-to-left
,	Left-to-right

Decreasing precedence

- `()` is a function call
- `->` means *struct pointer indirection*

- Unary `+, -, *`
- `*` here is *pointer indirection*

- Ternary if-else operator

- Assignment operators

In many imperative languages, `x = foo();` is an assignment statement, in C it is an expression where the value is the value being assigned.

`i++` returns the current value for `i` and increases `i` (post-increment) whereas `++i` returns `i + 1` (pre-increment).

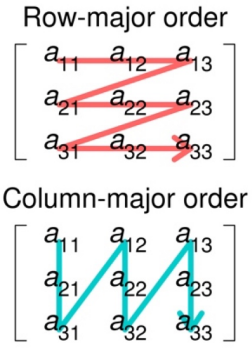
Most C types can be cast to another type, the bit-representation does (usually) not change! For example:

```
unsigned int ui = 0xDEADBEEF;
signed int i = (signed int) ui;
```

2.5 Arrays

An array is a finite vector of variables where each variable has the same type. For an N-element array `a`, the first element is `a[0]` and the last is `a[N-1]`. The compiler in C does not check array bounds!

Arrays in C are stored in row-major order:



Arrays can be initialized when they are defined:

```
int a[3] = {3, 7, 9};
```

```
int a[3][3] = {
        { 1, 2, 3},
        { 4, 5, 6},
        { 7, 8, 9},
};
```

2.5.1 Strings

C has no real string type, strings are array of char's terminated with null byte ('`\0`'). So the following two lines are identical:

```
char str[6] = {'h','e','l','l','o','\0'};
char str[6] = "hello";
```

There are lots of library functions to manipulate strings, e.g. `char* strcpy (char* dest, const char* src)`.

2.6 Representing integers

`signed` integers are usually interpreted in two's complement representation, whereas `unsigned` integers are interpreted as normal binary integers.

For all "integral" data types (`long, int, short, char, unsigned`), the bit-level operations `&` (AND), | (OR), ~ (NOT), ^ (XOR) are available in C. They view arguments as bit vectors and the operators are applied bitwise. Bit vectors are sometimes used to represent subsets (where $a_j = 1$ if $j \in A$), then the operators represent Intersection (`&`), Union (|), Symmetric difference (^) or Complement (~).

In contrast, the logic operations `&&` (Logic AND), || (Logic OR), `!` (Logic negation) view 0 as `false` and anything nonzero as `true`. They use early termination (i.e. not the whole argument is checked if it's already `false` / `true` for sure). Therefore, `p && *p` is often used to avoid null pointer access.

A left shift `x << y` shifts the bit-vector `x` left by `y` positions and fills with 0's on the right. A logical right shift `x >> y` shifts the bit-vector `x` right by `y` positions, throws away extra bits on the right and fills with 0's on the left. In contrast, an arithmetic right shift replicates the most significant bit on the left. Unsigned values are always right shifted logically, whereas signed values are usually shifted arithmetically. The behavior is undefined if shift amount is smaller than zero or greater than the word size.

Constants are considered to be signed integers by default, they are unsigned if they have "U" as suffix (e.g. `14U`). Implicit casting occurs via assignments and procedure calls and also in comparison operations.

When an integer is sign-extended, the most significant bit is copied to the new bits for signed integers.

Integers are added according to modular addition, therefore they form an Abelian group (closed under addition, commutative, associative, 0 as additive identity and every element has an additive inverse).

2.6.1 Two's Complement Representation

In two's complement representation, the first bit is interpreted as -2^{n-1}. To convert a negative number, the bits of its positive representation (in normal binary notation) are inverted and the value of 1 is added. Addition and subtraction are very simple for two's complement numbers, the methods are identical to ones' complement (for subtraction, the number is negated first according to the method described above).

2.6.2 Integer multiplication

Like addition, multiplication is implemented as modular arithmetic (i.e. the higher order bits are ignored). Therefore, unsigned multiplication with addition forms a commutative ring (closed under multiplication, commutative, associative, 1 is the multiplicative identity, multiplication distributes over addition). Signed multiplication is a ring as well, but ordering properties aren't obeyed (e.g. $TMax + 1 == TMin$).

Power-of-2 multiply can be formed with shifts, `u << k` gives $u * 2^k$ (signed and unsigned). Because most machines shift and add faster than multiplying, instead of e.g. $u * 24$, a compiler will often generate `u << 5 - u << 3`.

For unsigned integers, `u >> k` gives $\lfloor u/2^k \rfloor$. For signed integers, we get the same result for positive integers, but it rounds in the wrong direction when $u < 0$ (we would want round toward 0). The correct result is given by $u + (2^k - 1) \gg k$ for $u < 0$.

2.7 Floating Point

In 1985, IEE754 was established as uniform standard for floating point arithmetic. A floating point number consists of a sign bit, a mantissa / significant and an exponent. C guarantees two levels, `float` for single precision (4 byte) and double for `double` precision (8 byte). Casting between `int`, `float` and `double` changes bit representation according to the following rules:

- `double`/`float` → `int`: Truncates fractional part (like rounding towards zero). Undefined when out of range / NaN: Generally sets to TMin
- `int` → `double`: Exact conversion as long as `int` has ≤ 53 bit word size.
- `int` → `float`: Will round according to rounding mode.

Floating point values are normalized, when exp≠000..0 and exp≠111..1. In these cases:

The exponent is coded as biased value, E=Exp-Bias. The bias is $2^{e-1} - 1$, where e is the number of exponent bits. The significand / mantissa is coded with implied leading 1. It is minimum when 000..0 (M is then 1.0) and maximum when 111..1 (M is then 2.0 – ε).

Floating point values are denormalized, when exp=000..0. The exponent value is then E=-Bias + 1, therefore $-2^{e-1} + 2$ (e.g. -126 for `float`). But the significand / mantissa is interpreted with a leading 0. exp=0 and frac=0 is therefore 0 (there is actually +0 and -0, which can be used for limits) and exp=0 and frac≠0 is for numbers very close to 0.0 (equispaced).

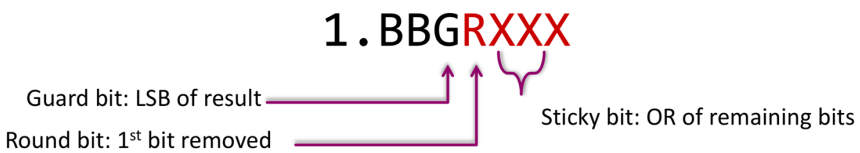
Furthermore, there are some special values (that occur when exp=111..1):

- exp=111..1, frac=000..0: Infinity (i.e. positive / negative overflow)
- exp=111..1, frac≠000..0: Not-a-Number (NaN), e.g. sqrt(-1), infinity – infinity, etc…

To compare floating pointer numbers, one can use unsigned integer comparison, but the sign bit has to be compared first.

There are three steps when creating a floating point number:

1. **Normalize to have leading 1:** Set binary point so that numbers have the form 1.xxxx
2. **Round to fit within fraction:** We use the following terminology:



   The round up conditions are then Round=1, Sticky=1 (which implies > 0.5) or Guard=1, Round=1 and Sticky=0, e.g.:

19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001

3. **Postnormalize to deal with effects of rounding:** When rounding has caused overflow, we need to shift right once and increment the exponent.

Floating point addition is not associative (because of rounding and overflow), infinity and NaN has no additive inverse, the other elements have.

Multiplication isn't associative as well and multiplication does not distribute over addition.

2.7.1 SSE Floating Point

SSE are SIMD (single-instruction, multiple data) vector instructions. They introduce new data types, registers and operations. SSE3 registers are 128 bit and can be filled with different combinations of data types (e.g. 16 bytes, 8 * 2 bytes, etc…)

2.8 Pointers

In C, `&x` produces the virtual address where the value of x is stored. A pointer is a variable that contains a memory address (it points to somewhere in the process' virtual address space). A pointer is declared as `int *p` or `int* p` (the location of the star doesn't matter). A pointer is dereferenced with `*p`. `NULL` is a guaranteed-to-be-invalid memory location.

Pointers are typed and pointer arithmetic follows pointer type. For example, if we add 1 to an int pointer, the pointer value is increased by `sizeof(int)`, therefore normally 4. If we add 1 to a char pointer, the pointer value is increased by `sizeof(char)`, normally 1.

An array name in an expression is treated as a pointer to the first element of the array. `A[i]` and `*(A + i)` is therefore the same. They are different in some special cases:

1. When the array's address is taken with `&`: `&A` creates a pointer of the type `T(*)[size]`, i.e. a pointer to an array, not to a single element. If we increment the pointer, it'll add the size of the entire array.
2. The array is a string literal initializer: When a string literal is allocated like `char *p = "test";` the string literal is stored in the read-only data section. In contrast, when he's allocated like `char a[] = "value";` the literal is stored in the data section and is therefore modifiable.
3. The array is an operand of `sizeof()`: sizeof(a) will give the size of the whole array.

Array names as function parameters are pointers, therefore `int fun(int *arr)` and `int fun(int arr[])` are identical!

C passes function arguments by value, the callee gets a copy of the argument. Pointers can be used to pass by reference.

`int *p[13]` or `int *(p[13])` declares p as an array[13] of pointers to int whereas `int (*p)[13]` declares p as pointer to an array[13] of int.

2.8.1 Function pointers

`int (*func)(int *, char);` declares a pointer to a function that takes two arguments, a pointer to `int` and a `char` and returns an `int`.

2.9 Dynamic memory allocation

Global variables are statically allocated (i.e. allocated when program is loaded and deallocated when program exits; because the compiler puts them into the data section) whereas stack variables are automatically allocated on the stack. Often we want memory that persists across multiple function calls but not for the whole lifetime of the program, is too big to fit on the stack and / or is allocated and returned by a function as a result whose size is not known to the caller. Therefore, we have dynamically allocated memory where the program explicitly requests a new block of memory and it persists until code explicitly deallocates it (manual memory management) or the garbage collector collects it (automatic memory management). C requires manual memory management.

`malloc()` allocates a block of memory of the given size and returns a pointer to the first byte of that memory (or `NULL` if the memory cannot be allocated). The memory is not zeroed and can contain garbage. `void *calloc(size_t nm, size_t sz);` allocates `nm * sz` bytes and zeroes the memory. `free(void *)` releases memory at the pointer. It must point to the first byte of malloc'ed memory. It is a good practice to `NULL` the pointer after freeing (to prevent further use). `realloc(void *ptr, size_t size)` changes the size of the block. As with `free()`, the pointer must point to the first byte of a malloc'ed block. It might copy the data to a new location, therefore it is necessary to always use the new address returned.

A memory leak happens when code fails to deallocate memory that will no longer be used.

2.9.1 Implementing Dynamic Memory Allocation

The virtual memory hardware and the kernel allocate pages but application objects are typically smaller An allocator manages objects within pages. A "block" is in this context a contiguous

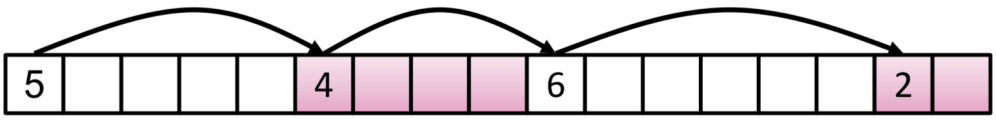
range of bytes of any size. Applications can issue arbitrary sequence of `malloc()` and `free()` requests where free() requests must be to a `malloc()`'d block. Allocators have to deal with the following:

- They can't control the number or size of allocated blocks
- They must respond immediately to `malloc()` requests.
- They must allocate blocks from free memory.
- They must align blocks so they satisfy all alignment requirements (data is aligned so that a single word access doesn't need 2 separate memory accesses or – in the worst case – 2 separate page accesses)
- Can manipulate and modify only free memory
- Can't move the allocated blocks once they are `malloc()`'d

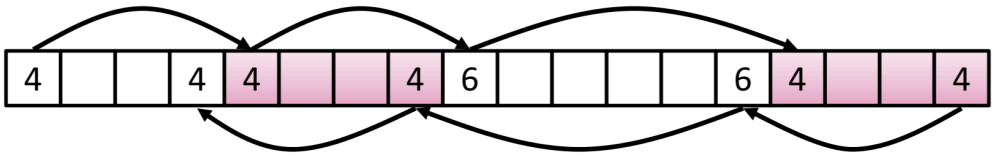
Internal fragmentation occurs when payload < block size and is caused by overhead of maintaining heap data structures, padding for alignment purposes and explicit policy decisions (e.g. to return a big block to satisfy a small request). External fragmentation occurs when there is enough aggregate heap memory, but no single free block is large enough.

To know how much to free, the standard method is to keep the length of a block in the word preceding the block (header field / header). To keep track of free blocks, there are multiple methods:

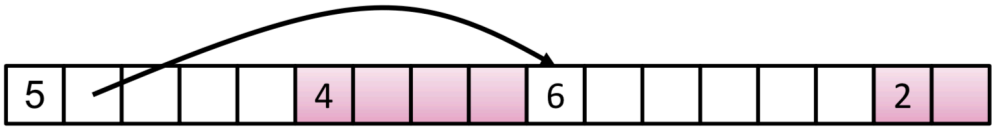
1. Implicit list using length (**implicit free list**) – links all blocks:



For each block, we need the length and the info if it is allocated. To prevent wasting two words, one can use lower order bits (that are always 0 when the blocks are aligned) to store the info if it is allocated. To find a new block, first fit (search list from beginning, choose first free block), next fit (like first fit, but search list starting where previous search finished) or best fit (search whole list, choose block with fewest bytes left over). Blocks can also be splitted to prevent internal fragementation. Splitted blocks should be coalesced (joined) with next / previous free blocks. To implement coalescing with previous block, boundary tags can be used:



2. Explicit list among the free blocks using pointers:



For the insertion policy, one can use LIFO (last-in-first-out) where a freed block is inserted at the beginning of the free list or address-ordered policy so that the freed blocks are always in address order.

3. Segregated free list (different free list for different size classes).
4. Blocks sorted by size (e.g. with a balanced tree with pointers within each free block, where the length is the key).

2.9.2 Garbage Collection

For a memory manager to know when memory can be freed, it needs to figure out that there are no more pointers to certain blocks because then they cannot be used anymore. Therefore, the memory manager must distinguish pointers from non-pointers. The “mark-and-sweep” algorithm views memory as a graph where the root nodes are locations not in the heap that contain pointers into the heap (e.g. registers, locations on the stack, global variables). The algorithm starts at roots and sets the mark bit on each reachable block. It then scans all blocks and frees blocks that are not marked.

2.10 Structs and Unions

A struct is a C type that contains a set of fields, e.g.:

```
struct Point {
      int x;
      int y;
};
struct Point origin = { 0, 0 };
```

“.” is used to refer to fields in a struct and “->” to refer to field through a pointer to a struct (`p->x` is equivalent to `(*p).x`).

When a value of a struct is assigned to another struct (of the same type), the entire contents are copied.

Unions are like a struct, but hold only one of a set of alternative values. They are accessed like a struct and there is no checking on which value is correct.

In C, typedefs can be used to avoid complex declarations. This can also be used for structs, e.g. by declaring `typedef struct list_el el_t;` and then using `el_t` when creating new `struct list_el`.

2.11 Modularity

A declaration says something exists, somewhere. A definition says what it is. For example, here is a declaration (on the top) and a definition (on the bottom) of `strncpy`:

```
char *strncpy(char *dest, const char *src, size_t n);
```

```
char *strncpy(char *dest, const char *src, size_t n)
{
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++) {
        dest[i] = src[i];
    }
    for ( ; i < n; i++) {
        dest[i] = '\0';
    }
    return dest;
}
```

Systems Programm

Declarations can be annotated with:

- **extern**: definition is somewhere ese, either in this compilation unit or another (default without explicit annotation)
- **static**: definition (that is also static) is in this compilation unit and can't be seen outside it.

C header files specify interfaces. They (should) contain only external declarations, no definitions. The implementation is typically in a .c file that also include its own header file. It is usual to have an Include-Guard (that ensures that the content of the header file only appear once) in a header file:

```
#ifndef __FILE_H_
#define __FILE_H_
... (declarations, macros)
#endif // __FILE_H_
```

2.12 Linking

2.12.1 Static Linking

A linker (`ld` on unix) links multiple separately compiled relocatable object files and produces one fully linked executable object file. This is called static linking.

Thanks to linking, programs can be written as a collection of smaller source filer, rather than one monolithic mass (**modularity**). Furthermore, when only one source file is changed, only this one needs to be recompiled and everything relinked, there's no need to recompile other source files (**time saving**) and common functions can be aggregated into a single file, yet executable

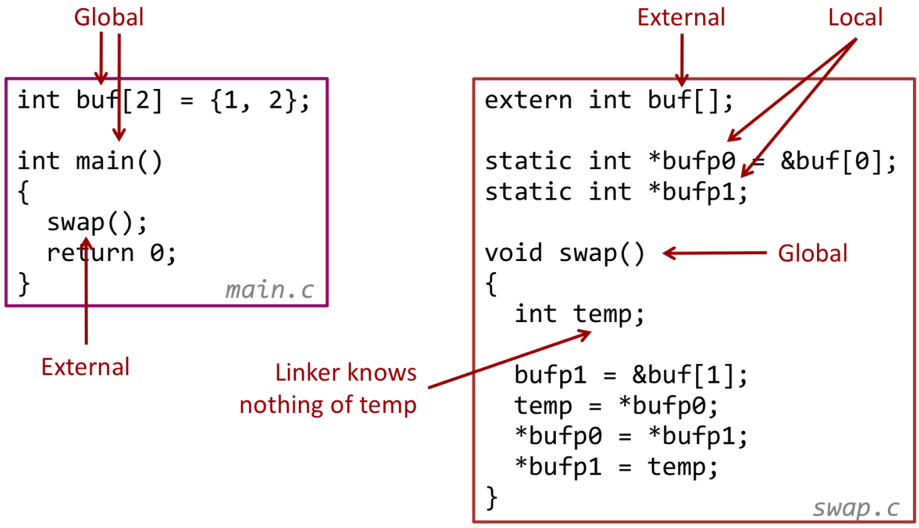
files and running memory images contain only code for the functions they actually use (**space saving**).

A linker works in multiple steps:

1. **Symbol resolution:** Programs define and reference symbols (variables and functions). Symbol definitions are stored (by the compiler) in the symbol table, where each entry includes name, type, size and location of the symbol. The linker associates each symbol reference with exactly one symbol definition.
2. **Relocation:** The linker merges separate code and data sections into single sections, relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable and updates all references to these symbols to reflect their new positions.

There are three different type of symbols:

1. **Global symbols:** Symbols defined by module m that can be referenced by other modules, e.g. non-static C functions and non-static global variables.
2. **External symbols:** Global symbols that are referenced by module m but defined by some other module.
3. **Local symbols:** Symbols that are defined and referenced exclusively by module m, e.g. C functions and variables defined with the static attribute (not local program variables, these are no symbols!)



Before relocation, the `main.o` and the `.data` section of the `swap.o` of the code above looks like this:

main.o

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   48 83 ec 08      sub    $0x8,%rsp
   4:   b8 00 00 00 00   mov    $0x0,%eax
   9:   e8 00 00 00 00   callq  e <main+0xe>
                a: R_X86_64_PC32 swap-0x4
   e:   b8 00 00 00 00   mov    $0x0,%eax
  13:   48 83 c4 08      add    $0x8,%rsp
  17:   c3               retq
```

```
Disassembly of section .data:

0000000000000000 <buf>:
   0:   01 00 00 00 02 00 00 00
```

```
Disassembly of section .data:

0000000000000000 <bufp0>:
   0:   00 00 00 00 00 00 00 00
                0: R_X86_64_64   buf

Disassembly of section .bss:

0000000000000000 <bufp1>:
   0:   00 00 00 00 00 00 00 00
```

As we can see, the address of the call in main() is incorrect and as well as the value of bufp0. After the relocation, they look like this:

```
00000000004004ed <main>:
  4004ed: 48 83 ec 08               sub    $0x8,%rsp
  4004f1: b8 00 00 00 00            mov    $0x0,%eax
  4004f6: e8 0a 00 00 00            callq  400505 <swap>
  4004fb: b8 00 00 00 00            mov    $0x0,%eax
  400500: 48 83 c4 08               add    $0x8,%rsp
  400504: c3                        retq
```

```
Disassembly of section .data:

0000000000601038 <buf>:
  601038:    01 00 00 00 02 00 00 00

0000000000601040 <bufp0>:
  601040:    38 10 60 00 00 00 00 00

Disassembly of section .bss:

0000000000601050 <bufp1>:
  601050:    00 00 00 00 00 00 00 00
```

`bufp0` now contains the address of `buf[0]` and the call the address / offset of `swap`.

Besides the distinction between symbol types, we also distinguish strong and weak symbols. Strong symbols are procedures and initialized globals. Weak symbols are uninitialized globals. The rules are:

- Multiple strong symbols are not allowed, the linker will give an error.
- Given a strong symbol and multiple weak symbols, the linker wil choose the strong symbol. References to the weak symbol resolve to the strong symbol.
- If there are multiple weak symbols, the linker will pick an arbitraty one.

This can lead to situations like this, where writes to x in p2 will overwrite y, because p2 assumes x to be 8 bytes (double) but the linker links to the 4 byte x (int) in p1:

```
int x=7;
int y=5;
int p1() {}
```

```
double x;
int p2() {}
```

Writes to x in p2 will overwrite y!
Nasty!

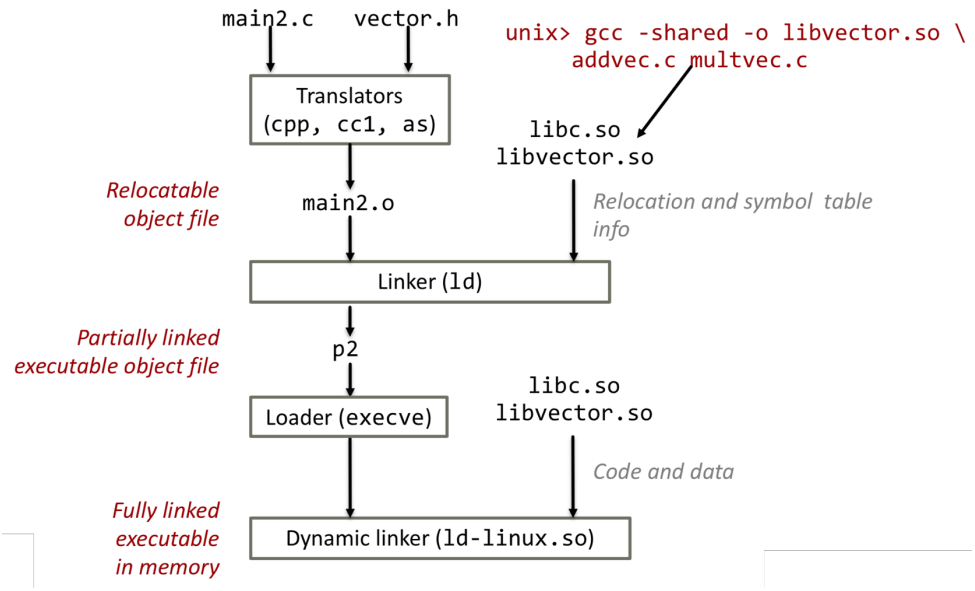
2.12.2 Static Libraries

Static libraries are used to package functions commonly used by programmers (e.g. math, I/O, memory management, etc…) together. They are archive files (`.a`) that contain related relocatable object files. The linker then tries to resolve unresolved external references by looking for the

symbols in one or more archives and if an archive member file resolves reference, links it into the executable. When the linker resolves external references, he scans .o files and .a files in the command line order. During the scan, he keeps a list of the current unresolved references and as each new .o or .a file is encountered, tries to resolve each unresolved reference in the list. If there are any entries in the unresolved list at the end of the scan, there is an error. For this reason, libraries should be at the end of the command line.

2.12.3 Shared Libraries

Static libraries have the disadvantages that they are duplicated in the stored executable and the running executable, and a minor bug fix of system libraries requires each application to explicitly relink. Shared libraries are object files that contain code and data that are loaded and linked into an application dynamically (either at load- or run-time). Load-time-linking is common for Linux and handled automatically by the dynamic linker (`ld-linux.so`), for instance `libc.so` is usually dynamically linked. Dynamic linking in unix is done by calls to the `dlopen()` interface (e.g. for high-performance web servers). Shared library routines can be shared by multiple processes. Dynamic linking at load-time looks conceptually like this:

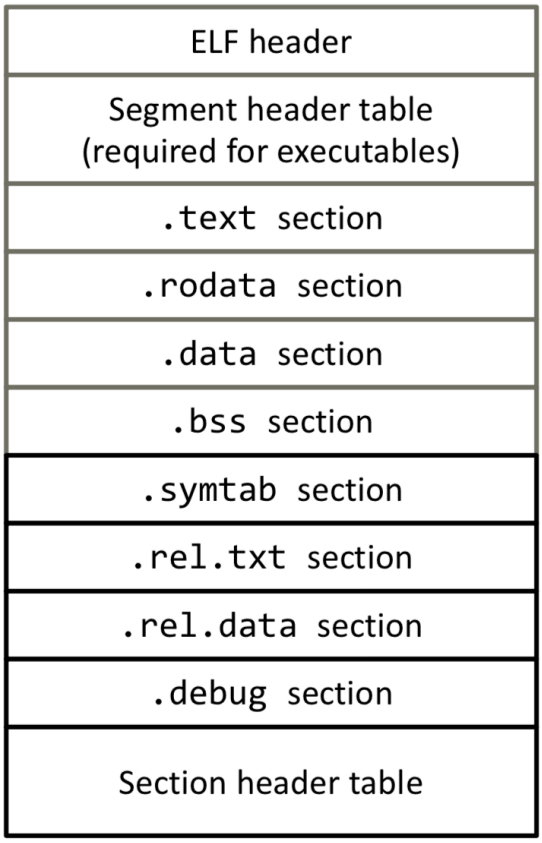


2.12.4 Executable and Linkable Format (ELF)

We differ between three different kind of object files:

1. **Relocatable object file (.o file):** Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
2. **Executable object file:** Contains code and data in a form that can be copied directly into memory and then executed.
3. **Shared object file (.so file):** Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load or run-time (called Dynamic Link Libraries, DLLs, by Windows).

ELF is the standard binary format for object files. It is one unified format for relocatable object files, executable object files and shared object files. The generic name for ELF files is ELF binaries. The file format looks like this:

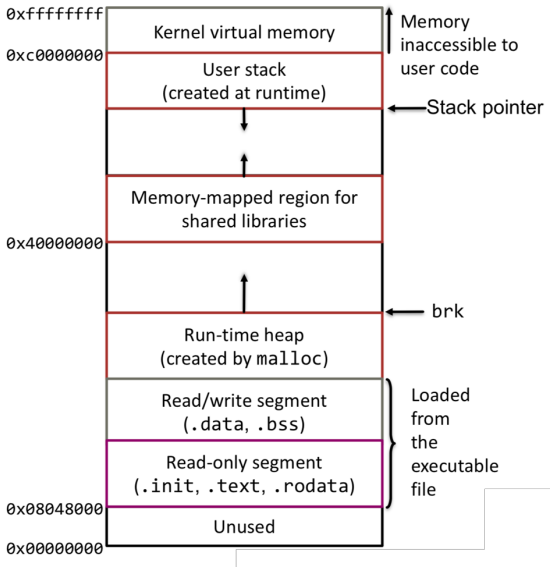


The header contains things like the word size, byte ordering, file type, etc… The `.text` section contains code, the `.rodata` section read only data like jump tables, the .data section initialized global variables and the `.bss` section (that has section header but occupies no space) uninitialized global variables. A Mnemonic for the `.bss` section is “better save space”.

The `.symtab` section contains the symbol table, the `.rel.text` section relocation info for the .text section, the `.rel.data` section relocation info for the `.data` section, the `.debug` section info for symbolic debugging (produced with `gcc -g` flag) and the section header table offsets and sizes of each section.

3 Address space

The operating system gives each process an address space. When the OS loads a program, it creates the space, inspects the executable file, (lazily) copies regions of the file into the right place in the address space and does any final linking, relocation or other needed preparation. The address space looks like this:



To make buffer overflow attacks harder, the address space layout is randomized. Linux randomizes the base of stack and shared library locations.

3.1 The Stack

Stack-based languages are languages supporting recursion. The code must therefore be reentrant (multiple simultaneous instantiations of single procedure). The state of each instantiation needs to be stored. The stack is allocated in frames (the collection of all data on the stack associated with one subprogram call is called a stack frame). A frame contains local variables, return information and temporary space. The frame is allocated when the procedure is entered and deallocated on return. The stack grows down!

4 x86 Architecture

An ISA (instruction set architecture) is the parts of a processor design that one needs to understand to write assembly code (e.g. instruction set specification, registers). The microarchitecture is the implementation of the architecture (cache sizes and core frequency). CISC stands for complex instruction set, which was the dominant style through mid-80s. Typical properties are that the instruction set is stack-oriented (the stack was used to pass arguments), arithmetic instructions can access memory and condition codes. In contrast, RISC (reduced instruction set) has fewer, simpler instructions that can be executed with small and fast hardware. The instruction set is register-oriented and only load and store instruction can access memory.

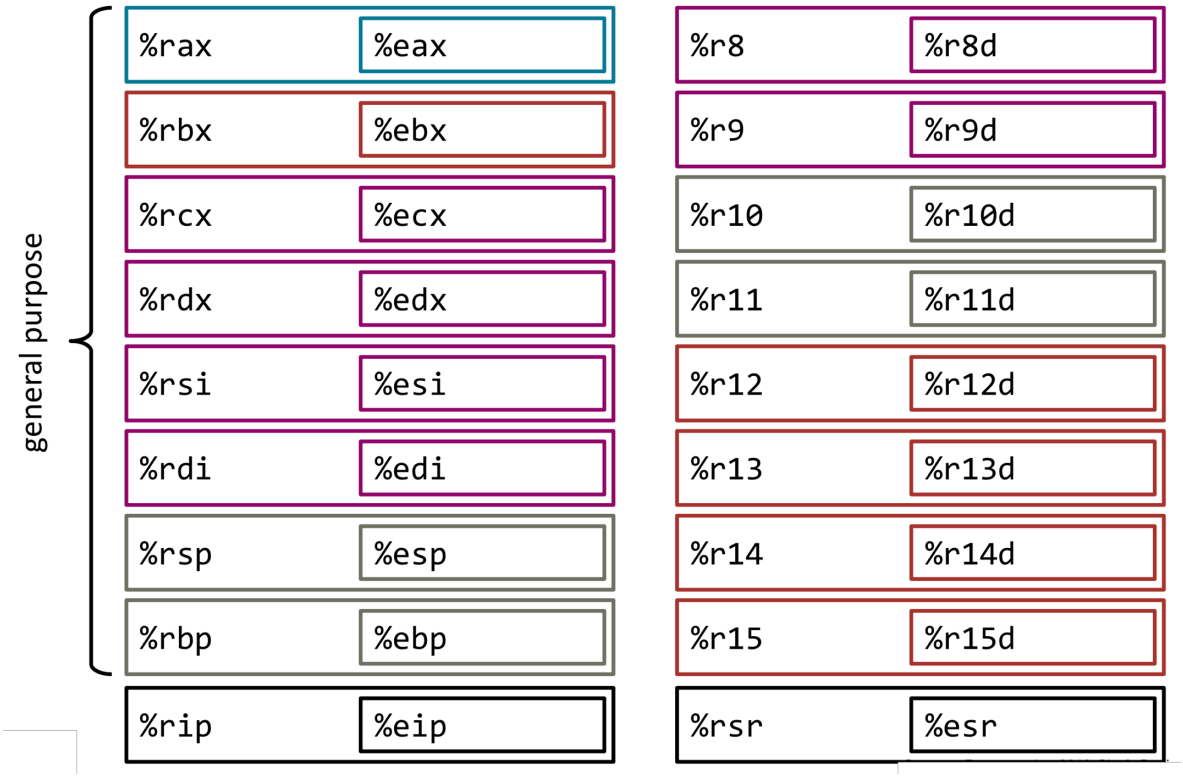
The Programmer-Visible State of a CPU is the Program Counter (PC; called “RIP” on x86-64), register file and condition codes (that store status information about most recent arithmetic operation and are used for conditional branching). Memory is a byte addressable array that contains code, user data, (some) OS data and includes the stack used to support procedures.

4.1 Assembly & Architecture

The only assembly data types are “Integer” data of 1, 2, 4 or 8 bytes (data values) and floating-point data of 4, 8 or 10 bytes. No aggregate types like arrays, structures, etc… they are just contiguously allocated bytes in memory.

Assembly code can perform arithmetic function on register or memory data, transfer data between memory and register and transfer control (unconditional jumps from / to branches, conditional branches).

The x86-64 integer registers are:



Where the name in the inner block refers to the lower 32 bit of the register. By using the lower two chars of the register name, one can refer to the lower 16 bits (or in the case of %[a-d]x even to the lower 8 bits by %[a-d]l and the 9th to 16th bit by %[a-d]h). `movb` is used to move 1-byte “byte”, `movw` for 2-byte “word”, `movl` for 4-byte “long word” and `movq` for 8-byte “quad word”. Operands can be Immediate (prefixed with $ sign), register or memory (simplest example `(%rax)`, many other address modes). Memory-memory transfer with a single instruction is not possible! The `lea src,dest` instruction sets dest to the address denoted by the expression in `src`. It can be used to compute arithmetic expressions of the form x+k*y where k=1,2,4 or 8.

The x86 condition codes are CF (carry flag), SF (sign flag), ZF (zero flag) and OF (overflow flag). They are implicitly set by arithmetic operations, e.g. for t=a+b:

- CF set if carry out from most significant bit
- ZF set if t==0
- SF set if t < 0 (as signed)
- OF set if two’s complement (signed) overflow, i.e. (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

They are not set by the lea instruction.

They can also be set explicitly by the compare instruction: `cmpl/cmpq a,b` (which sets CCs according to the result of `b – a`) and then be used in the conditional jump instruction. Mnemonic: The following conditional jump condition always "views" the arguments in the opposite order as the `cmp` instruction (in AT&T syntax!), e.g.:

```
cmpl %eax, %edx # if (edx <= eax)
jle .L2
```

4.2 Compiling C Control Flow

Conditional expressions can be translated to assembly as follows:

Goto version

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
   . . .
   goto Done;
Else:
   val = Else-Expr;
Done:
   return
```

or

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
   . . .
Done:
   return
Else:
   val = Else-Expr;
   goto Done;
```

Instead of branches, it's also possible to compute both values and use a conditional move in the end (when the branches have no side effects!).

In do-while loops, a check at the bottom of the loop is performed to check if the loop should be executed another time:

Goto version

```
Loop:
   Body
   if (Test)
      goto Loop
```

For a while loop, one can add a test to the beginning or add a new "middle" label and jump to it in the first iteration:

Goto version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Goto version

```
goto middle;
loop:
  Body
middle:
  if (Test)
    goto loop;
```

For loops can be translated to while loops and from there on be handled like a normal while loop:

For version

```
for (Init; Test; Update )
    Body
```

While version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Switch statements can be implemented as jump tables where the value is used to index into the jump table and the table contains the labels for all switch table entries to jump to. This is not practical for sparse switch statements because a table entry is needed for every case between `minVal` and `maxVal`. For sparse statements, the cases are therefore often organized as binary tree, resulting in logarithmic performance.

4.2.1 Procedure call and return

The stack grows toward lower addresses, the register `%rsp` therefore contains the lowest stack address = address of “top” element. `pushl src` decrements the stack by 4 and writes operand at address given by `%rsp` whereas `popl dst` reads operand at address `%rsp` and increments `%rsp` by 4.

`call label` pushes the return address (address of instruction beyond call) on the stack and jumps to `label`. `ret` pops the address from the stack and jumps to the address.

There are conventions on which registers are “caller save” (caller saves temporary in its frame before calling) and “callee save” (callee saves temporary in its frame before using).

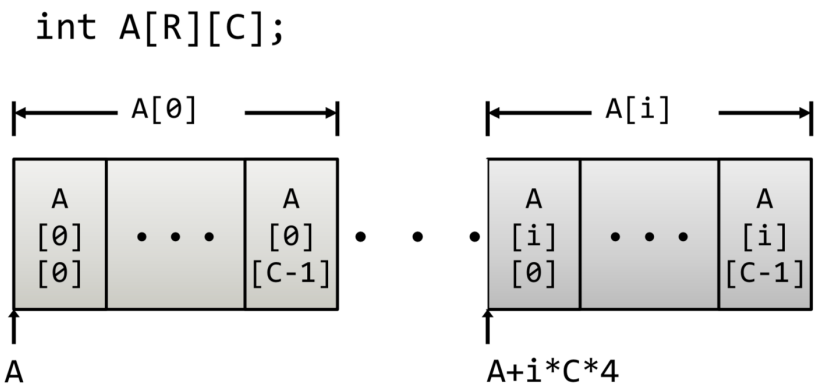
The zone below the stack pointer is called red zone. This zone can be temporarily used to delay allocation, i.e. first move objects and then decrement the stack pointer.

Tail call optimization is when a function avoids creating a new stack frame because it will just pass along a value of another called function.

4.3 Compiling C Data Structures

To access array elements, e.g. `a[i]` where `a` is an int array, one can use memory reference, e.g. `movl (%rdi,%rsi,4), %eax` where `%rdi` contains the starting address of a and `%esi` the array index.

For nested arrays, the contiguous allocation and the “row-major” ordering is guaranteed. For a nested array `A[R][C]`, the `i`th Row (that is an array of `C` elements) starts at Address A + i * (C * K).

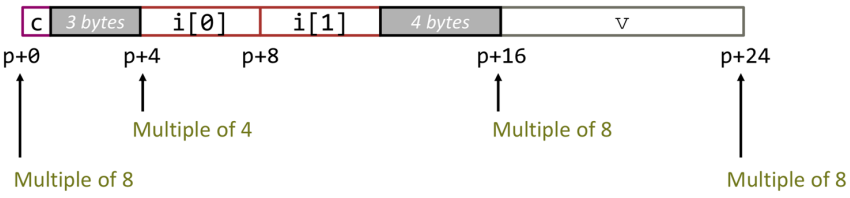


The compiler uses these calculations for accesses.

A multi level array is an array that contains pointers to other arrays (that can be anywhere in memory). Therefore, two memory reads are needed for an access (first to get pointer to row array, second to access element within array).

struct’s are contiguously allocated in memory. To ensure correct alignment of fields, the compiler inserts gaps in struct. For every primitive type of size k, the address must be a multiple of k (no restrictions for chars, multiple of two for short, of four for int, etc…). Each structure has alignment requirement K where K is the largest alignment of any element. In this example, K=8 due to the double and therefore 7 padding bytes are inserted by the compiler:

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



In an array of structures, the alignment requirements have to be satisfied for every element.

Unions are allocated according to the largest element. A union can be used to access the bit-pattern of a float by defining a union with a unsigned integer and a float field, setting the float field and then getting the unsigned integer field.

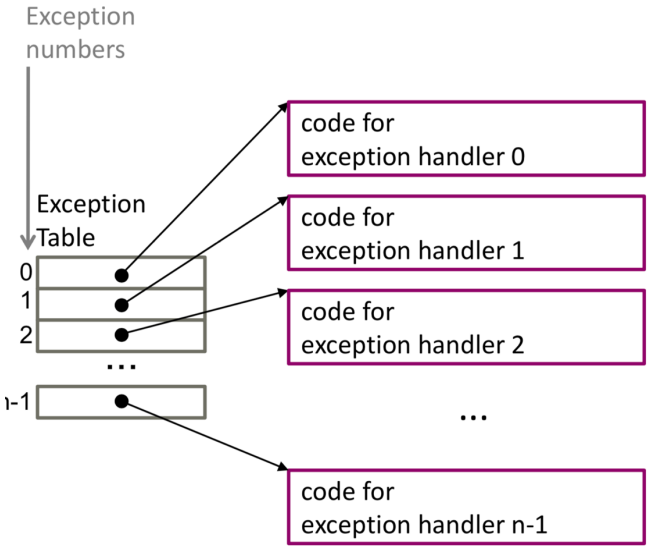
4.4 Code Vulnerabilities

A worm is a program that can run by itself and can propagate a fully working version of itself to other computers whereas a virus is code that adds itself to other programs and cannot run indenpendently.

In a stack overflow bug, a (too) small buffer is overwritten by too much data. This can cause the return address to get overwritten, therefore allowing an attacker to control code execution. An attacker can also place exploit code in the buffer and set the return address to jump to this exploit code, therefore allowing arbitrary code execution.

4.5 Exceptions

From startup to shutdown, a CPU simply reads and executes a sequence of instructions. This sequence is the CPU's control flow. To change the control flow, jumps & branches and call & return can be used. But this is insufficient for a useful system because it is difficult to react to changes in system state (arrival of data, user input, etc…). Therefore the system needs mechanisms for "exceptional control flow". An exception is a transfer of control to the OS in response to some event. Each type of event has a unique exception number k, where k is the index into exception table (interrupt vector). The handler k is called each time the exception k occurs.



Exceptions cause a switch to kernel mode.

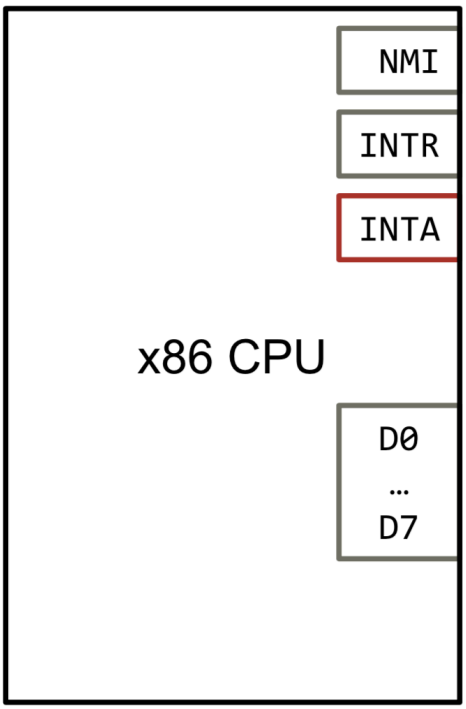
Synchronous exceptions are caused by events that occur as a result of executing an instruction and are divided into:

- **Traps:** Intentional exceptions that return control to "next" instruction, e.g. system calls or breakpoint traps
- **Faults:** Unintentional but possibly recoverable exceptions, e.g. page faults or floating point exceptions. The faulting ("current") instruction is either re-executed or the program is aborted.
- **Aborts:** Unintentional and unrecoverable errors, e.g. parity error or machine checks. The current program is aborted.

Asynchronous exceptions (interrupts) are caused by events external to the processor. They are indicated by setting the processor's interrupt pin and the handler returns to the "next" instruction. Examples are I/O interrupts (CTRL-C, arrival of a packet from the network, arrival of data from disk), hard reset interrupts (hitting the reset button) or soft reset interrupts (hitting CTRL-

ALT-DELETE on a PC). The interrupt handler receives these interrupts and is maskable to ignore or delay some interrupts. There is a interrupt vector to dispatch the interrupt to the correct handler based on priority, some interrupts are nonmaskable.

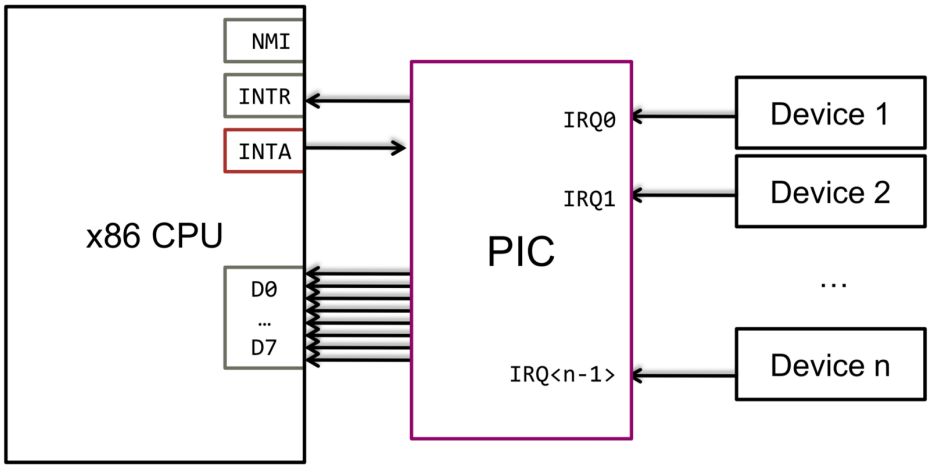
x86 has two interrupt pins, INTR for interrupt requests and NMI for non-maskable interrupts:



When a NMI is asserted, the CPU completes the current instruction and issues exception #2 (always, cannot be disabled by the processor). There's no obvious way to tell what caused the interrupt the OS (INT 0x2 handler) must poll potential sources.

Interrupt requests can be disabled by the IE status flag. If they are enabled, the current instruction is completed, the CPU acknowledges the interrupt using the INTA pin and the interrupt vector is then supplied on the data bus. The CPU issues then the exception number from the vector.

Because this can lead to "interrupt conflicts" (same vector for different devices) and multiple interrupts can be issued, there are programmable interrupt controllers (PICs) that buffer the interrupts and where the OS can pick the mapping of physical pins to vectors. Interrupts can be prioritized and individual device's interrupts be selectively masked.



5 Code Optimizations

5.1 Optimizing Compilers

Compilers are good at mapping program to machine (register allocation, code selection and ordering, dead code elimination or elimaniting minor inefficiencies). But they are not good at improving asymptotic efficiency, it's up to the programmer to select the best overall algorithm. Furthermore, there are certain "optimization blockers" (potential memory aliasing, procedure side-effects). If in doubt, the compiler is conservative. He has to operate under fundamental constraints (must not change program behavior under any possible condition) and behavior that may be obvious to the programmer can be obfuscated by language and coding styles. Most analysis is performed only within procedures (because whole-program analysis is too expensive in most cases) and most analysis is based only on static information.

Here are some optimization techniques:

1. **Code motion:** Reduce frequency with which computation is performed (sometimes also called precomputation). Often times done by the compiler. For example:

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

2. **Strength reduction:** Replace costly operation with simpler one, e.g. shift/add instead of multiply / divide. The usefulness is machine-dependent.
3. **Common subexpressions:** Reuse portions of expressions. Some operations are not associative and cannot be eliminated.

Potential optimization blockers are:

1. **Procedure calls:** If a procedure is called in a loop condition (e.g. `strlen(s)`), the compiler won't move the procedure out of the inner loop because it may have side effects and might not return the same value for given arguments.
2. **Memory aliasing:** When we write often times to the same memory location, the compiler will not extract these writes into local variables automatically because these locations may be aliased by other variables. Therefore, it can be good to introduce local variables for locations that are often written to in an inner loop.

5.2 Architecture and Optimization

Hardware can execute multiple instructions in parallel (Instruction-Level Parallelism), but performance is limited by data dependencies. Sequential dependencies in inner loops especially limit performance, because performance is then determined by the latency of the operation, not by the throughput (because before every new issuance, the previous one has to be finished).

With loop unrolling, this can be improved because multiple work can be done per iteration. For floating point, it's also important to keep the associativity in mind and considering reassociating the operations. Separate accumulators are a different form of reassociation. Unrolling and accumulating can be combined, resulting in different combinations of unrolling and accumulating factors.

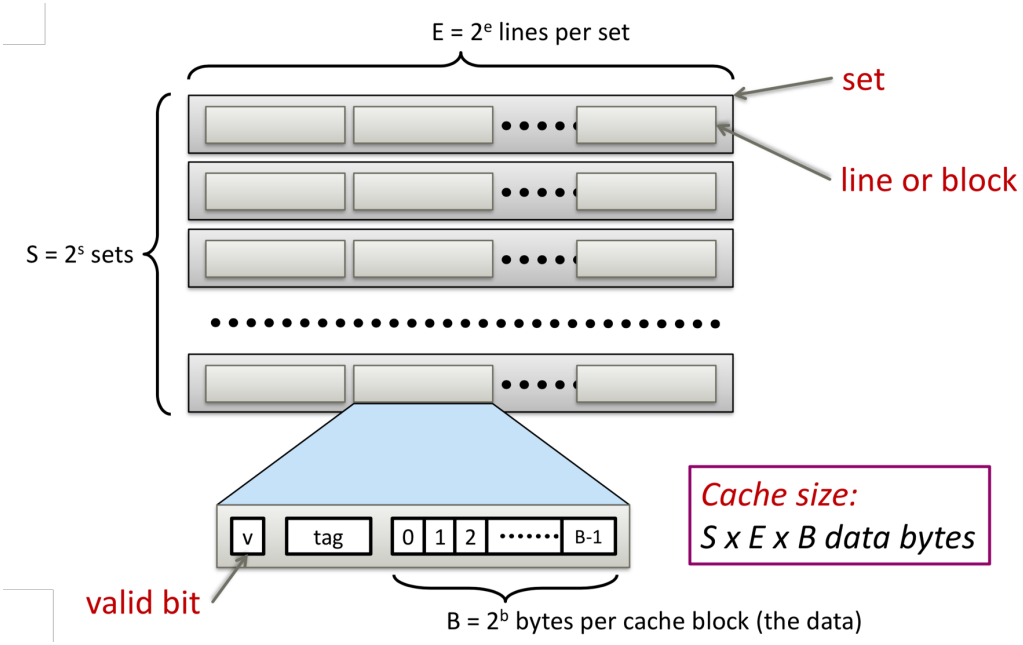
6 Caches

Caches work because of locality. Programs tend to use data and instructions with addresses near or equal to those they have used recently. We differ between temporal locality (recently referenced items are likely to be referenced in the near future) and spatial locality (items with nearby addresses tend to be referenced close together in time).

When data in a block is needed and is not in cache, there is a cache miss. The block is fetched from memory (or an upper cache) and is stored in cache. The placement policy determines where b goes and the replacement policy determines which block gets evicted. The miss rate is the fraction of memory references that are not found in cache (misses / accesses) and the hit time is the time to deliver a line in the cache to the processor. The miss penalty is the additional time required because of a miss. There are different types of cache miss:

- **Cold (compulsory) miss:** Occurs on first access to a block
- **Conflict miss:** Occurs when a slot is too small to fit all blocks
- **Capacity miss:** Occurs when the set of active cache blocks (working set) is larger than the cache
- **Coherency miss:** Occurs in multiprocessor systems

A cache consists of $S = 2^s$ sets with $E = 2^e$ lines per set and a line / block contains a valid bit, tag and $B = 2^b$ bytes per cache block:



A cache read goes to the following steps:

- Locate set
- Check if any line in set has matching tag
- If yes and the line is valid: Hit
- Locate data starting at offset

In a direct mapped cache, we have $E = 1$. If there is no match, the old line is evicted and replaced. In a multiple-way set-associative cache, the line is replaced according to the replacement policy (e.g. random, least recently used, etc…)

There are two different things to do on a write-hit. One possibility is a write-through cache (write immediately to memory) or a write back cache (defer write to memory until replacement of line, needs a dirty bit). On a write miss, a write-allocate cache will load the block into the cache and update the line in cache (common with write-back caches) and a no-write-allocate cache will write immediately to memory (seen with write-through caches).

Caches are hierarchically divided, here is a typical division:

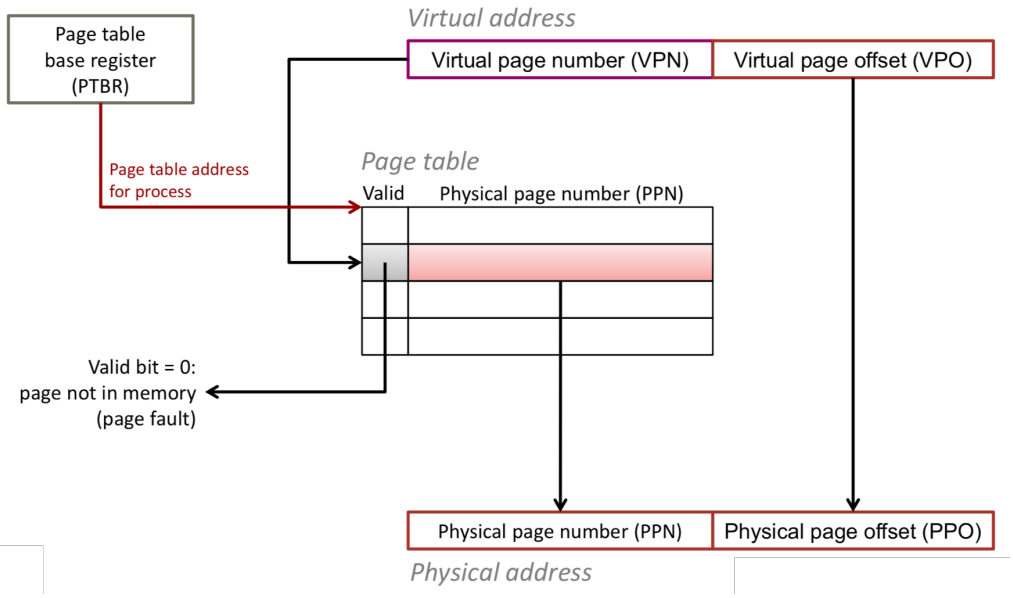
Cache type	What is cached?	Where is it cached?	Latency (cycles)	Managed by
Registers	4/8-byte words	CPU core	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	64-byte blocks	On-chip L1	1	Hardware
L2 cache	64-byte blocks	On-chip L2	10	Hardware
Virtual memory	4kB page	Main memory (RAM)	100	Hardware + OS
Buffer cache	4kB 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

6.1.1 Cache Optimizations

To optimize code for cache utilization, the code should have spatial (i.e. contiguous data accesses) and temporal (accesses to the same data not too far in time) locality. This can be achieved by a proper choice of algorithm and loop transformations. Blocking divides a bigger problem (e.g. a matrix) into smaller blocks and therefore improves locality.

7 Virtual Memory

Due to virtual memory, each process gets its own private memory space that is mapped to physical memory. Every byte in main memory has one physical address but has one (or more) virtual addresses. The MMU (memory management unit) is responsible for the translation of virtual addresses to physical addresses. The page table contains in its entries the physical page numbers (PPN), the translation looks like this:



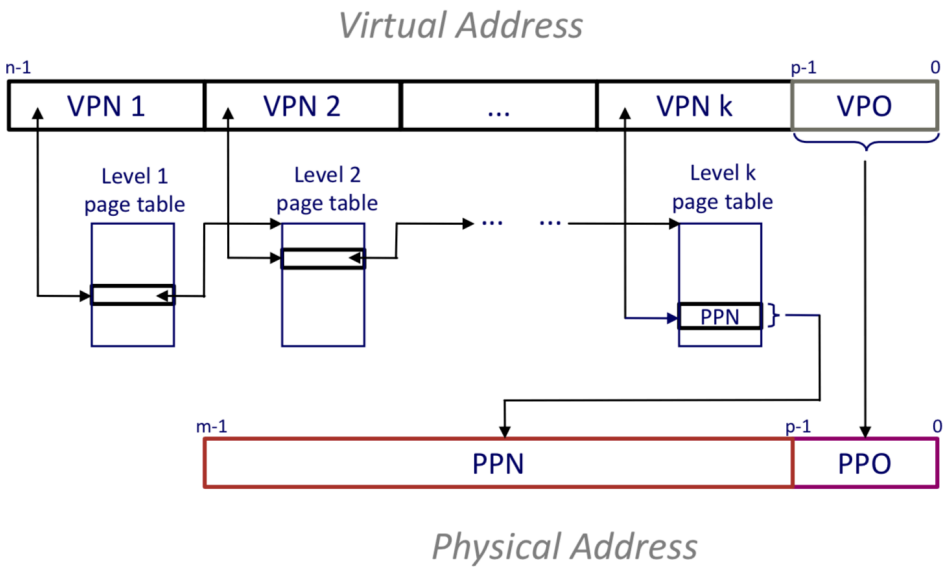
The translation lookaside buffer (TLB) is used to cache parts of the page table.

Virtual memory is mainly used for three reasons:

1. **Efficient use of limited main memory (RAM):** RAM can be used as a cache for the parts of a virtual address space, some non-cached parts are stored on disk and some (unallocated) non-cached parts are stored nowhere. Only active areas of virtual address space needs to be kept in memory.
2. **Simplifies memory management for programmers:** Each process gets the same full, private linear address space. Furthermore, code and data can be shared among processes by mapping virtual pages to the same physical pages.
3. **Isolates address spaces:** One process can't interfere with another's memory because they operate in different address spaces and a user process can't access privileged information because different sections of address spaces have different permissions.

Virtual memory deals with large page sizes and is fully associative (any virtual page can be placed in any physical page). The replacement algorithms are highly sophisticated and expensive and write-back rather than write-through is used. This is all because the disk is way slower than DRAM.

Because Linear page tables would be very big, multi-level page tables are used where different parts of the virtual address are used to index into different levels of the page table:



To speed up cache access when using virtual memory, the bits that determine the cache index are usually identical in virtual and physical address. Therefore, the processor can index into the cache while the address translation is taking place. This is called "virtually indexed, physically tagged" (VP). There are also these different options:

- **Virtually indexed, virtually tagged (VV):** Issues occurs because of homonyms (same names for different data). Potential solutions are to flush the cache on context switch, force non-overlapping address-space layout or to tag the virtual address with an address space ID (ASID).

- **Physically indexed, physically tagged (PP):** In this case, the translation must complete before the cache access can start. This is typically used off-core (L2 or L3 unified cache).

Some MMUs support large pages (or even huge pages) where the last part of the VPN is also considered part of the VPO / PPO.

8 Multiprocessing / Multicore

Multicore processors contain multiple processor cores per chip. Common are shared memory multiprocessors that have a single physical address space that is shared by all processors and communication between processors happens through shared variables in memory. The hardware typically provides cache coherence.

Coherency refers to the concept that values in caches all match each other and processors all see a coherent view of memory. Consistency deals with the order in which changes to memory are seen by different processors.

Sequential consistency refers to a consistency model where operations from a processor appear (to all others) in program order and every processor's visibility order is the same interleaving of all the program orders. Sequential consistency is violated in many architectures and different architectures implement different consistency models.

8.1 Cache Coherency with Snooping

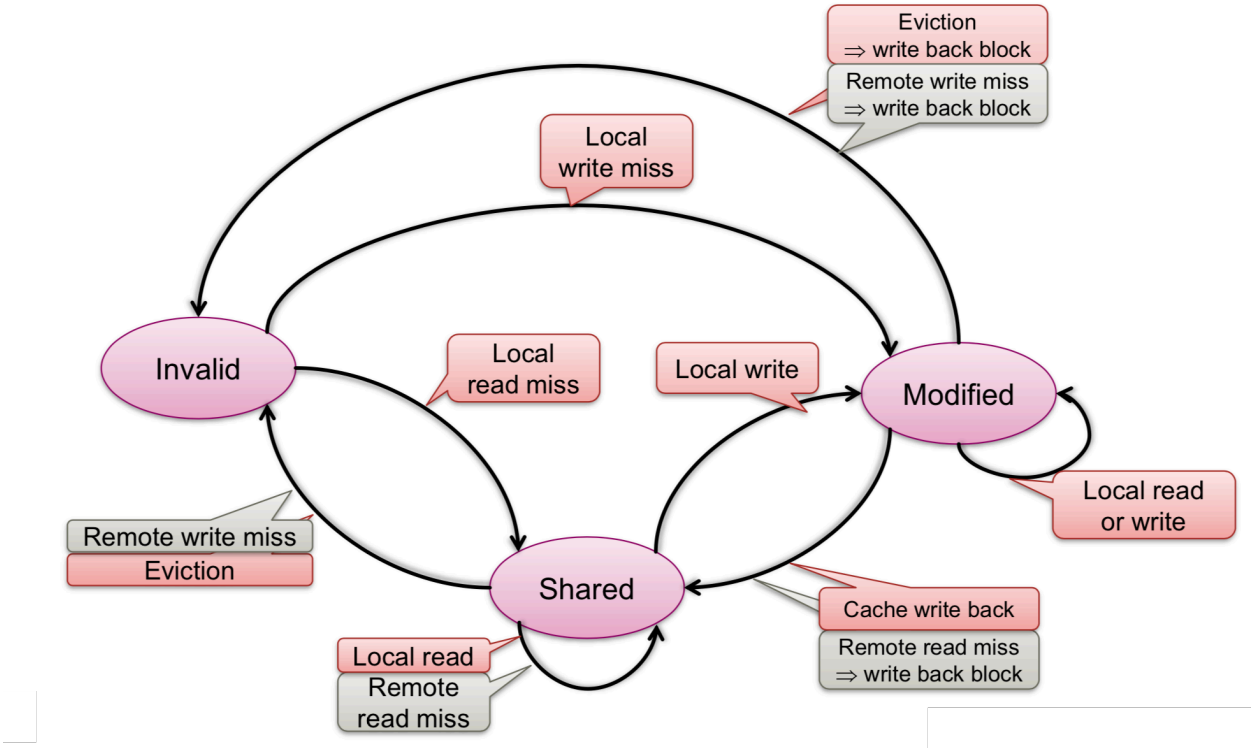
In this coherence model, the cache "snoops" on reads / writes from other processors. If a line is valid in local cache and a remote (other processor) writes to the line, the local line gets invalidated. This requires a write-through cache.

Write-back caches require a cache coherency protocol, e.g. MSI:

8.1.1 MSI

In this protocol, every line can be in one of three states: Modified, Shared or Invalid. The cache logic must respond to processor reads and writes and remote bus reads and writes. It must accordingly change cache line state and write back data (flush) if required.

The MSI state diagram (where red transitions are local transitions and green remote (snooped) transitions) looks like this:



A problem of MSI is that it's not always possible to distinguish between remote processor read and write misses. When a processor is in I state, executing a write miss, he needs to first read (allocate) the line. A different processor in M state, that observes this remote read, doesn't know if he should transition to invalid or shared.

8.1.2 MESI

The MESI protocol adds a new line state "exclusive":

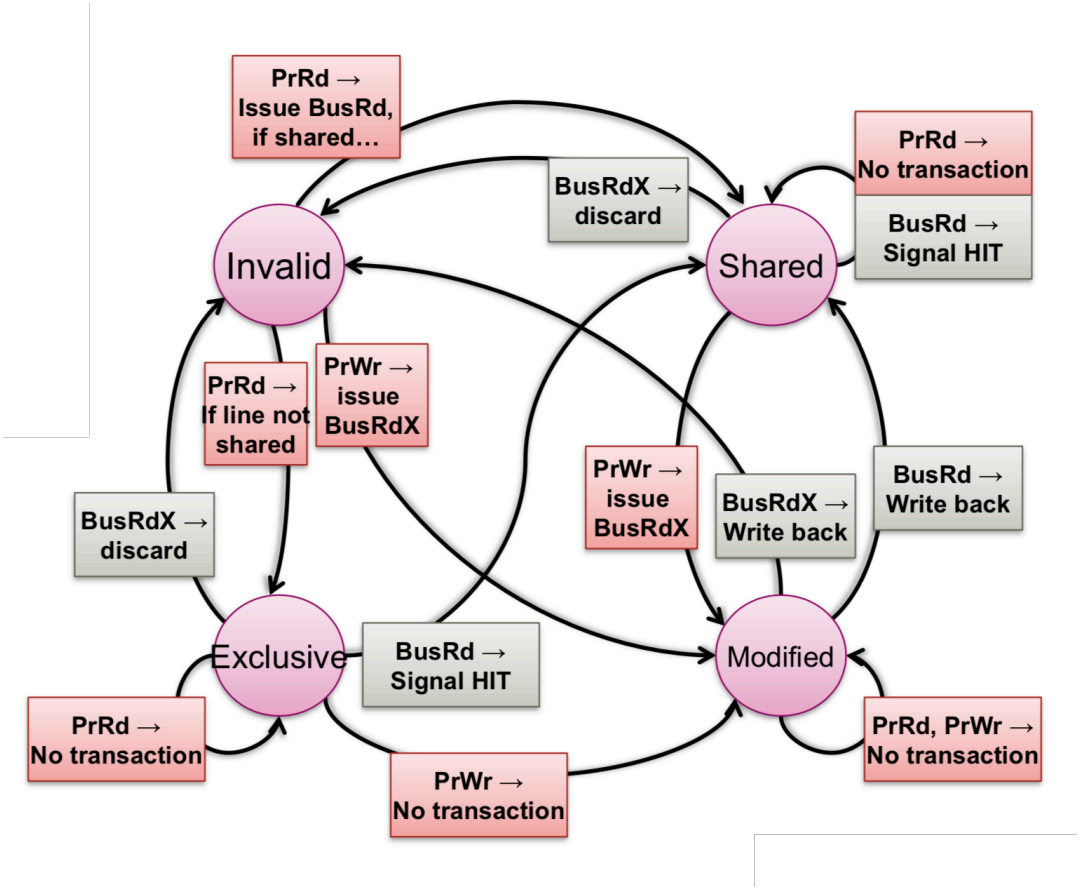
- **Modified:** This is the only copy, it's dirty
- **Exclusive:** This is the only copy, it's clean
- **Shared:** This is one of several copies, all clean
- **Invalid**

Furthermore, a new bus signal RdX "read exclusive" is added and a cache can there load into either "shared" or "exclusive" states. Also, the "HIT" signal is added to signal to a remote processor that its read hit in local cache. The invariants of both protocols are:

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

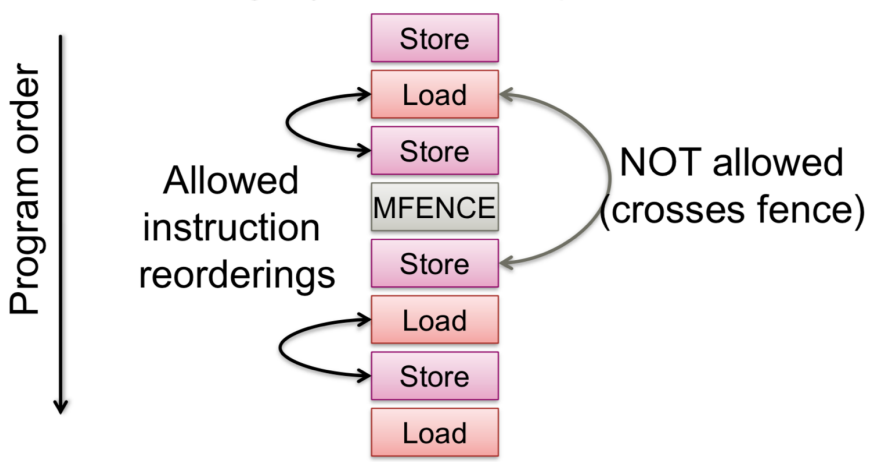
	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

The whole MESI state diagram looks like this:



8.2 Barriers and Fences

Compiler barriers prevent the compiler from reordering visible loads and stores but he may still reorder register access (private). Memory barriers prevent the CPU from reordering any loads or stores past it, e.g.:



8.3 Multicore Synchronization

Most modern hardware supports atomic instructions. We differentiate TAS (Test-and-set) and CAS (Compare-and-set). The semantics of them are:

```
boolean TAS(memref s)
atomic  if (mem[s] == 0) {
            mem[s] = 1;
            return true;
        } else
            return false;
```

```
int CAS (memref a, int old, int new)
atomic  oldval = mem[a];
        if (old == oldval)
            mem[a] = new;
        return oldval;
```

It's very easy to implement a spin lock with TAS:

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

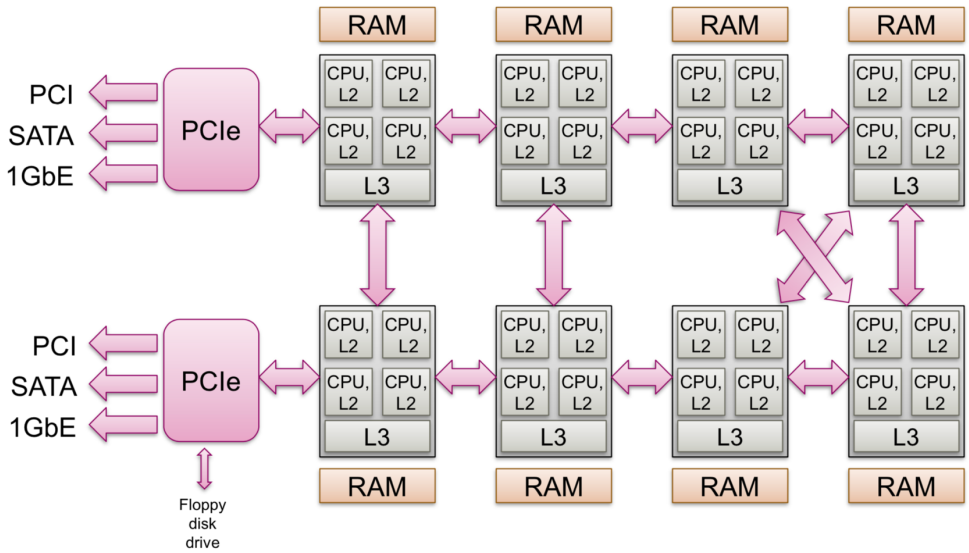
This can be further improved by a Test-and-Test-and-Set (TATAS) lock or with backoff.

8.4 SMT / Hyperthreading

To improve performance when one thread is idling (e.g. waiting for data), the processor can select instructions from threads on a per-instruction basis (fine-grained multithreading).

8.5 Non-Uniform Memory Access (NUMA)

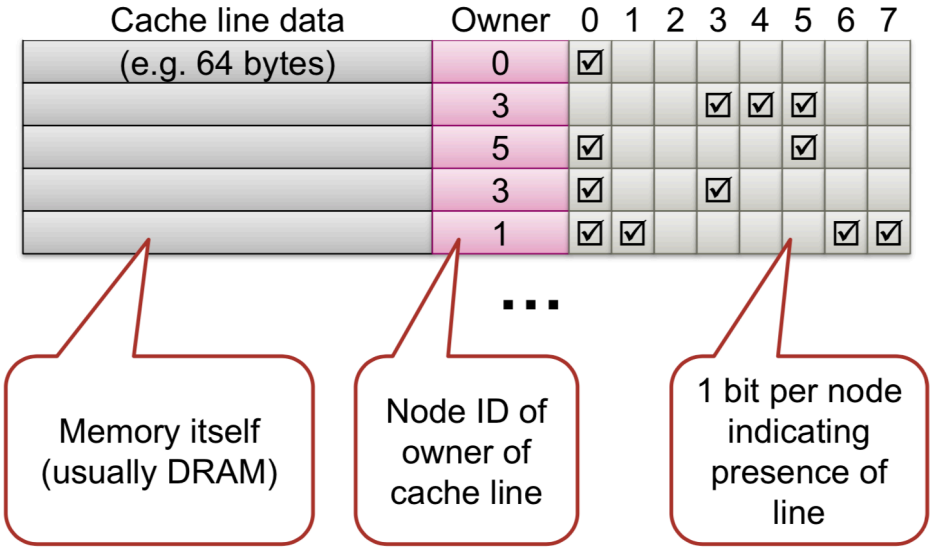
The bus in a typical CPU can be replaced by fast interconnection networks. This removes the bus bottleneck because processors have independent paths to memory. An example is the 8-socket 32 core AMD Barcelona:



With NUMA, cache coherence can't be solved by snooping on the bus anymore, because it's not a bus. There are multiple solutions:

1. **Bus emulation:** Similar to snooping, but without a shared bus (each node sends a message to all other nodes, etc…)

2. **Cache directory:** Each node's local memory is augmented with a cache directory with the necessary meta-data and the owner maintains the set of nodes that may have the line:



9 Devices

A device is a piece of hardware visible from software, that occupies some location on a bus, has a set of registers, a source of interrupts and may initiate direct memory access transfers.

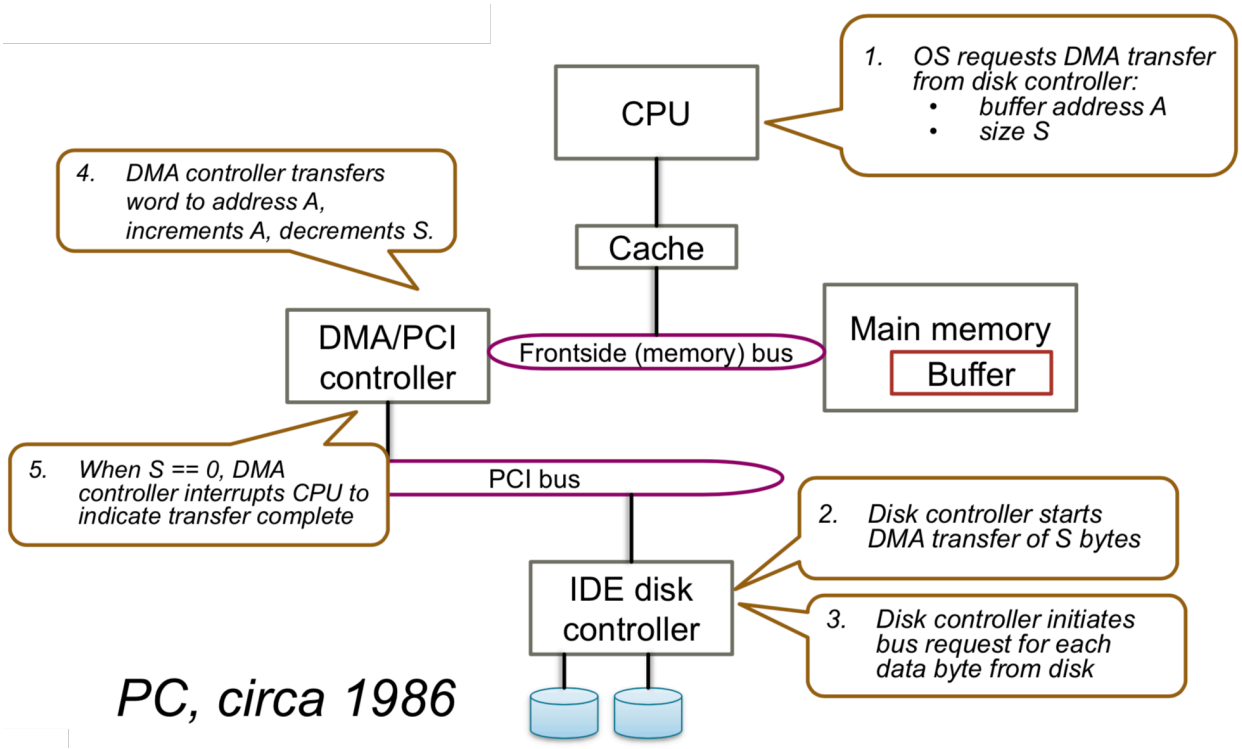
The CPU can load from device registers to obtain status information and read input data and can store to device registers to set device state and configuration, write output data and reset states. There are three possibilities to address the registers:

1. **Memory mapped:** Registers appear as memory locations; the access is done using loads / stores.
2. **"I/O instructions":** Different (16 bit) address space for older I/O devices, these days specific to x86 architecture
3. **Indirection:** An index register with an offset and a data register with the actual value is written (used to save address space)

Device registers don't behave like RAM, register contents change without writes from the CPU and writes to registers are used to trigger actions. Device register access must bypass the cache. Therefore, PTEs for device registers have a "no cache" flag in the MMU. To avoid polling all the time, interrupts are used. To avoid that the CPU must copy all the data, direct memory access (DMA) is used:

9.1 Direct Memory Access (DMA)

DMA avoids programmed I/O for lots of data. It requires a DMA controller and bypasses the CPU to transfer data directly between I/O device and memory. A very simple DMA transfer looks like this:



The key DMA advantage is that data transfer is decoupled from processing. The CPU does not need to copy data to / from the device. DMA also means that memory becomes inconsistent with CPU caches. To solve this, the CPU can map DMA buffers non-cacheable (which has a large performance impact because the CPU wants to process the data most of the time), the cache can “snoop” DMA controller bus transactions or the OS can explicitly flush/invalidate cache regions.

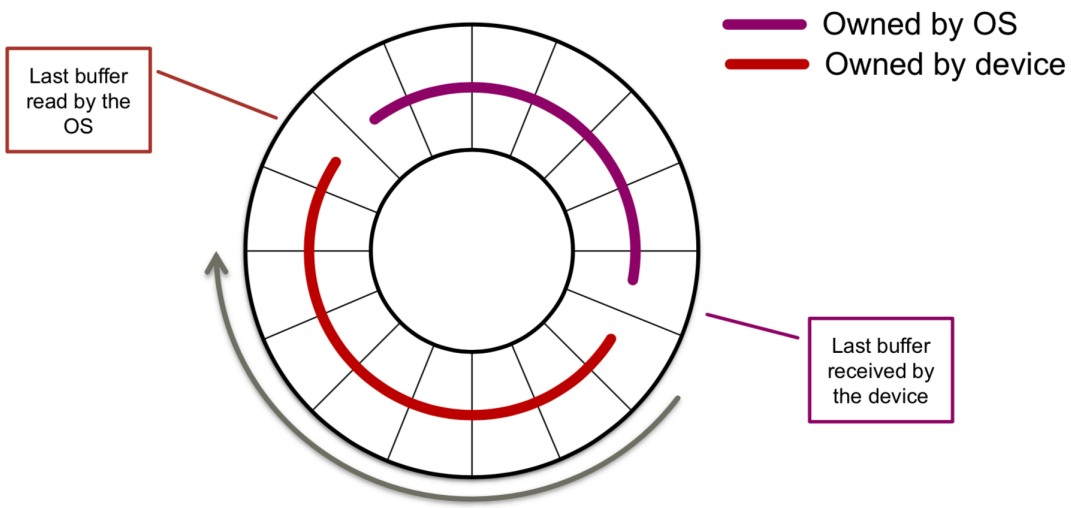
DMA addresses are physical whereas the user and OS deals with virtual addresses. The OS must therefore manually translate virtual ←→ physical addresses when programming DMA controllers which can require more than just a hardware page table. Furthermore, DMA of a single virtual address region might not be contiguous in physical address space (solved by scatter-gather DMA controllers). Newer systems provide an IOMMU.

9.2 Device Drivers

A driver and a device are both state machines. Data must be transferred between them and events to signal state transitions. Device ←→ CPU communication is done by reading / writing registers, by interrupt requests from the device or by shared memory (CPU writes to memory, DMA reads or DMA writes to memory, CPU reads). The last method is asynchronous.

9.3 Buffer / Descriptor rings

A buffer descriptor is a ring that contains pointers (descriptors) to other bits of memory. Most modern devices deal with them. They allow software more flexibility in data placement, buffers can be any size, buffers can vary dynamically and there is no need to mix data and metadata.



These rings allow the OS and device pointers to move independently around the ring with very little explicit coordination as long no pointer catches up with the other (over / underrun). When the device has no buffers for received packets, it starts discarding packets and will start copying them to memory when a buffer is free. When the CPU reads all received packets, it must wait. It normally signals the device to interrupt it when a new packet has been received.

9.4 Discoverable buses: PCI

The OS needs to know which devices exist and where the device registers are in the physical address space. One solution to it is PCI (Peripheral Component Interconnect). PCI tries to solve the problems of device discovery, address allocation, interrupt routing and intelligent DMA (“bus mastering” so that devices no longer need a DMA controller).

PCI is a tree that can have multiple bridges. PCI devices are self-describing, each device has a configuration header with a manufacturer ID, model ID, class code, version identifier, etc…