System Programming and Computer Architecture

> ❗ This and many more summaries can be found on https://n.ethz.ch/~dcamenisch. Feel free to leave a comment in the document if you spot any mistakes! As always no guarantees for completeness or correctness are made.

▼ 0. Table of Content

1. Introduction

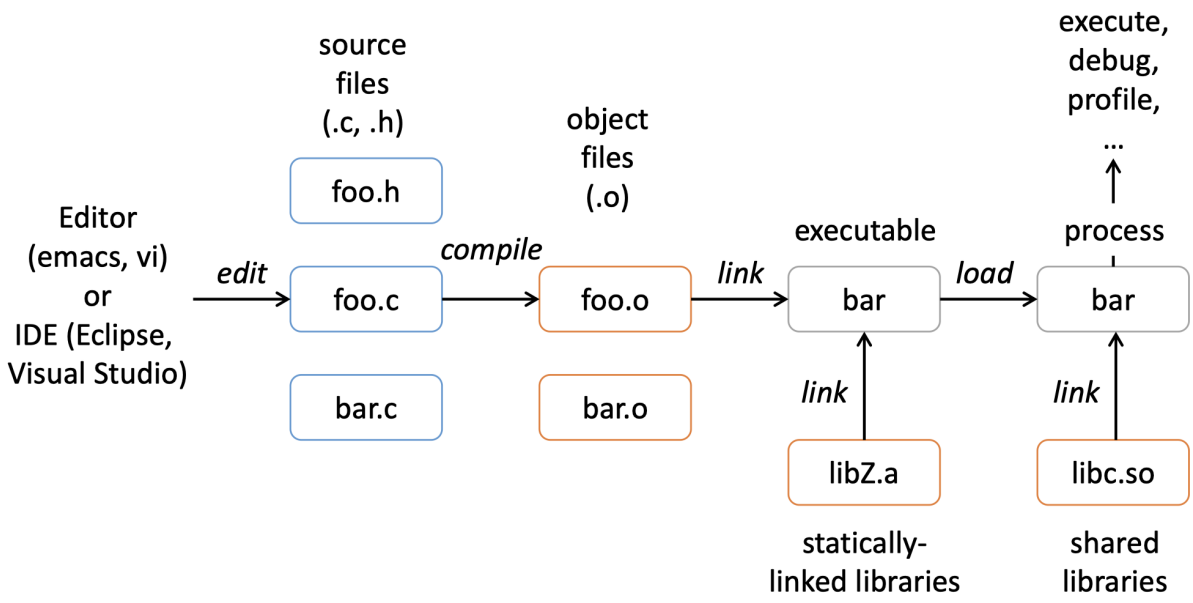
1.3 Motivation - Five realities

- **Reality #1** - `int` and `float` are not numbers.
- **Reality #2** - You've got to know assembly.
- **Reality #3** - Memory matters. RAM is not a realistic abstraction.
- **Reality #4** - There's much more to performance than asymptotic complexity. Constant factors matter too!
- **Reality #5** - Computers don't just execute programs. Programs don't just calculate values.

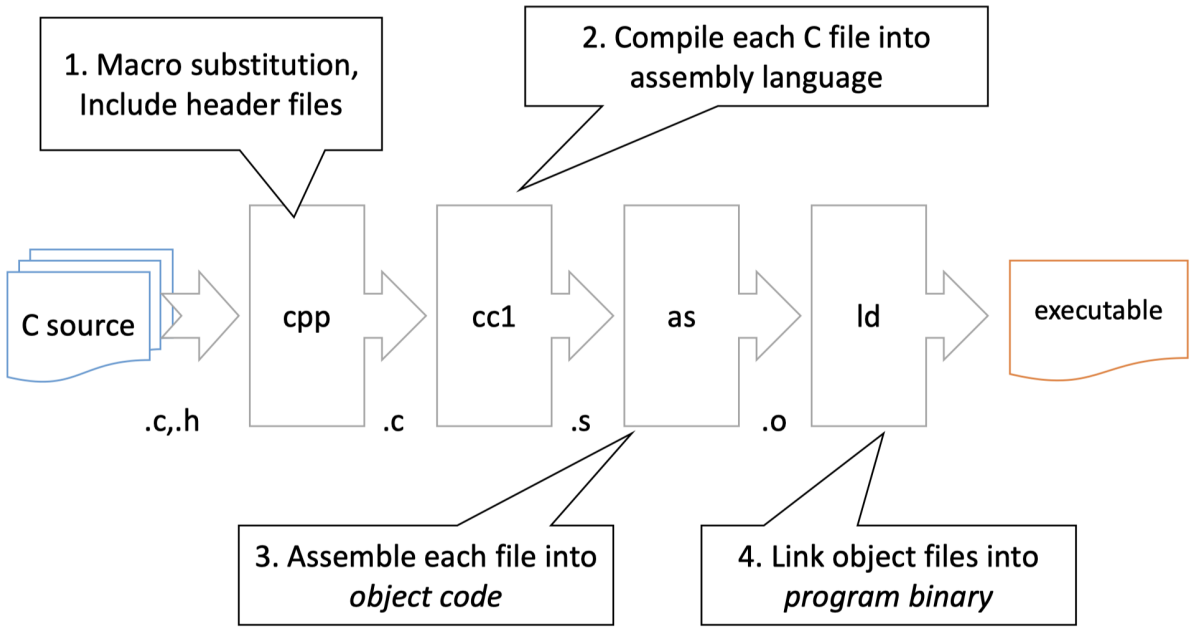
2. Introduction to C

2.1 History and Toolchain

Workflow



GNU gcc Toolchain



2.2 Control flow in C

Similar to Java or other programming languages, C has Conditionals, Loops and Functions. What might be new is, that C has Jumps:

```
break;        // behaves similar to Java, but no Label
continue;     // behaves similar to Java, but no Label
goto <Label>  // resumes code execution at the line after the Label
```

2.3 Basiy types in C

Declarations work like any similar language, most of the base types are the same, but there are some differences. `Booleans` where introduced really late, normaly one uses a `short` where 0 means `false` and any other value is `true`. Further any statement in C is also an expression, hence we can have something like this:

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

Lastly C has a `void` type that has no value. It is used for untyped pointers and to declare functions without return value.

2.5 Arrays in C

Arrays work similar to other languages, but one has to be carefull, **the C compiler does not check for array bounds!** If we do not initialize an array, we can not be sure what values are stored in it.

```
int a[3] = {3, 7, 9} // declaring and initializing an array of 3 int
```

Strings

C has no real string type. Instead, strings are arrays of char's terminated with null `'\0'` . Therefore the following two expressions are the same:

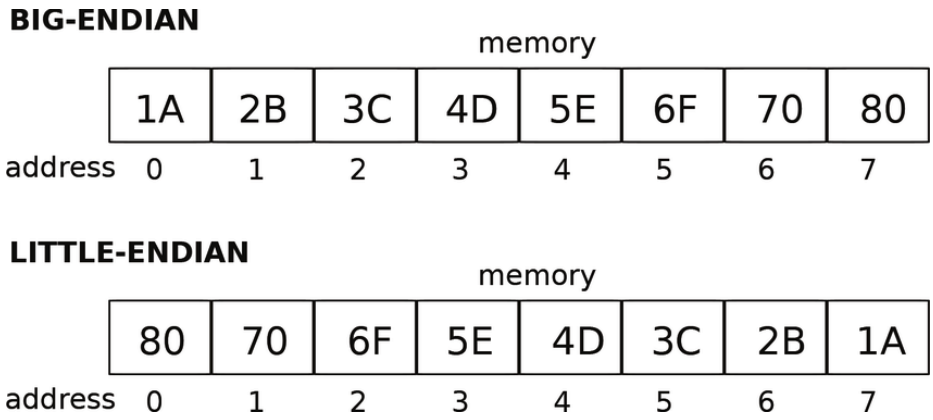
```
char str[6] = {'h','e','l','l','o','\0'};
char str[6] = "hello";
```

We generally use string libraries to manipulate strings.

3. Representing Integers in C

3.1 Recap: Encodings and operators

First we remind ourself of endianess from DDCA. **0x1A2B3C4D5E6F7080**:



Integers

Constants are by default considered to be signed integers. If a number is declared with a "U" suffix it's considered unsigned.

When mixing unsigned and signed in a single expression, **signed values are implicitly cast to unsigned numbers**!

When shifting, we differentiate between arithmetic shift and logical shift. While the logical shift fills with 0's, the arithmetic shift copies the shifted bit.

3.2 Integer ranges

For a w-bit integer, we can have the following range:

- $0...2^w - 1$ for unsigned
- $-2^{w-1}...2^{w-1} - 1$ for signed (two's complement)

Sign extensions

Given a w-bit signed integer x, convert it to a $w + k$-bit integer with the same value.

- We make k copies of the sign bit and add them to the begining of x

3.3 Integer addition and subtraction in C

Negation

Recall the following holds for 2's complement: $\sim x + 1 == -x$

Furthermore we observe, that: $\sim x + x == -1$

3.4 Integer multiplication in C

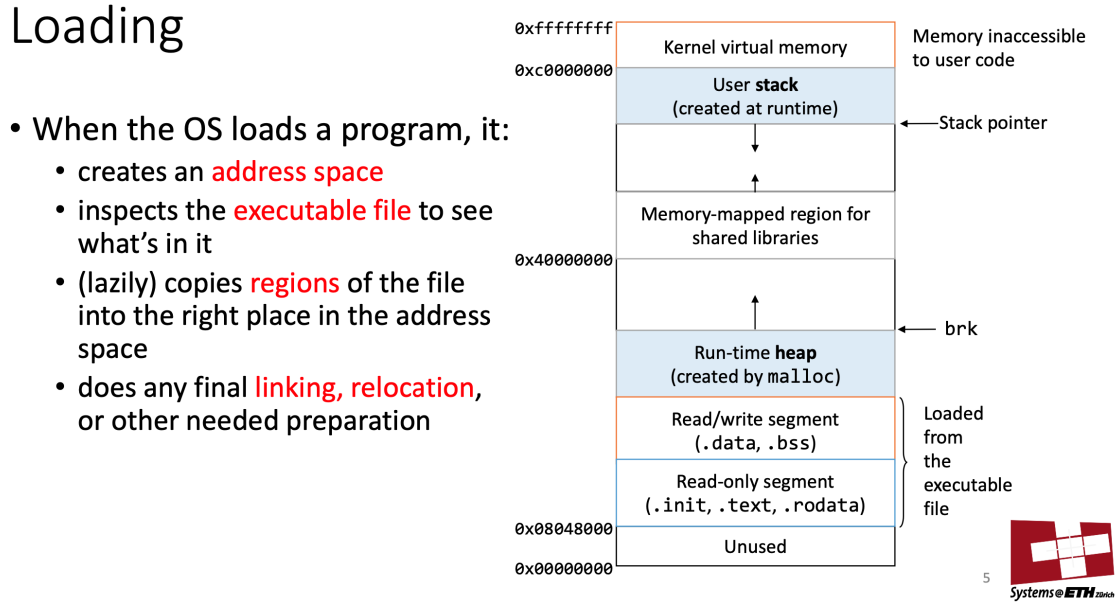
We notice, that we would need to keep expanding the word size with each product we compute, since the product of two w-bit numbers can have up to $2w$ bits.

3.5 Integer multiplication and division using shifts

- u << k gives $u \cdot 2^k$ (both signed and unsigned)
- u >> k gives $\lfloor \frac{u}{2^k} \rfloor$ (both signed and unsigned)
  - The rounding is wrong on signed number if $u < 0$. It rounds down instead of "towards 0"
  - We therefore compute the division for signed negative integers the following way: $\lfloor \frac{u+2^k-1}{2^k} \rfloor \rightarrow$ in C: (u + (1 << k) - 1) >> k

4. C Pointers

The OS gives each process and address space, which contains process virtual memory. For example, on a 64 bit machine, one can host 2^{64} bytes of virtual memory.



4.1 Recap: the stack

The stack is allocated in Frames containing local variables, return information and temporary space. Furthermore they are required to manage the space allocated when the procedure is entered ("Set-up" code) and the space deallocated when returning ("finish" code). Remember **the stack grows from top → bottom**.

4.2 Pointers in C

In C, you can produce the virtual address where the value of a variable `x` is stored with using `&x`. Furthermore you can use `%p` in a `printf()` statement to print it out. For example:

```
#include  <stdio.h>

int main(int argc, char **argv) {
  int x;
  int a[2];

  printf("x is at %p \\n", &x);
  printf("a[0] is at %p \\n", &a[0]);
  return 0;
}
```

Pointers are variables that store memory addresses. They are declared as follows:

```
int main(int argc, char *argv[]) {
  int x = 42;
  int *p = &x;

  return 0;
}
```

We can also **dereference** a pointer, i.e. access the memory referred to by a pointer;

```
int main(int argc, char *argv[]) {
  int x = 42;
  int *p = &x;

  int y = *p;
  *p = 99   //x is now 99

  return 0;
}
```

We can also have **double pointers:**

- `int x = 0`;
- `int *p = &x`;
- `int **dp = &p`;

We notice that each time we start the same program, we get different addresse for the same variables! This is called randomized address space layout. To visualize what is going on in the virtual memory, we use so called box and arrow diagrams.

A pointer pointing to `NULL` is guaranteed to be invalid. In C on Linux, `NULL` referes to the memory address 0x0000000000000000. Any attempt to dereference `NULL` will result in a segmentation fault.

4.5 Arrays and pointers

Reminder: arrays are not pointers!

- An *array* is a collection of homogeneous data elements stored at contiguous memory addresses.
- A *pointer* is a variable that stores a memory address

An array name in an expression is treated as a pointer to the first element of the array, except when:

- The array is an operand of size of

```
int a[10];
assert(sizeof(a) == 10*sizeof(int));
assert(sizeof(&a[0]) == sizeof(int *));
```

- The array's address is taken with '&'

```
int a[10];
assert(&a == a);
```

- The array is a string literal initializer

```
char a[] = "Hello!";
char *b = "Hello!";
```

In fact, `A[i]` is always rewritten as `*(A+i)` in the compiler.

4.6 Passing by reference

In general, C passes function arguments **by value.** The callee gets a *copy* of the argument! If a calle modifies an argument, the caller's copy isn't modified. We can solve this by passing the values **by reference** (by pointers).

```
#include <stdio.h>

void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(int argc, char **argv) {
  int a = 42, b = -7;

  swap(&a, &b);
  printf("a: %d, b: &d\\n", a, b);
  return 0;
}
```

5. Dynamic Memory Allocation

So far we have seen two ways of memory allocation:

- *statically*: Globally defined variable, allocated when the program is loaded, deallocated when program exits
- *automatically*: Variables defined in functions, allocated when function is called, deallocated when function returns (allocated on the stack)

But 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
- is allocated and returned by a function as a result whose size is not known to the caller

Dynamic memory allocation is the *heart* of C!

5.1. The C memory API

The function `malloc()` allocates a block of memory of the given size. It returns a `void` pointer to the first byte of that memory (and `NULL` if the memory cannot be allocated). You should assume **the memory initially contains garbage**.

```
long *arr = (long *)malloc(10*sizeof(long));
if(arr == NULL) {
  return ERRCODE;
}
arr[0] = 5L;
```

The `calloc()` function works in the same way as `malloc()` does, **but zeroes the memory**. It is therefore slightly slower, but also less error-prone and more readable.

```
long *arr = (long *)calloc(10, sizeof(long));
if(arr == NULL) {
  return ERRCODE;
}
arr[0] = 5L;
```

To deallocate the memory, we can use the function `free()`. Remark: It's good practice to `NULL` the pointer after freeing.

```
long *arr = (long *)calloc(10, sizeof(long));
if(arr == NULL) {
  return ERRCODE;
}
//do something...
free(arr);
arr = NULL;
```

With the function `realloc()` we can resize a memory allocation. `realloc()` must point to the first byte of the malloc'ed block. The old pointer passed in is now longer valid, one has to use the pointer returned by `realloc()`.

```
long *arr;
if(!(arr = (long *)malloc(10*sizeof(long)))) { //checks if pointer is NULL
  return ERRCODE;
}
//do something...
if(!(arr = (long *)realloc(arr, 20*sizeof(long)))) { //checks if pointer is NULL
  return ERRCODE;
}
```

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

5.3 Structured Data

A `struct` is a C type that contains a set of fields. It's a bit like a *class* but contains no methods or constructors. Instances of `struct` can be allocated on a stack or heap.

```
// New strctured data type called "struct Point"
struct Point {
  int x;
  int y;
};

struct Point origin = {0,0};
```

We use "`.`" to refer to fields in a struct and "`->`" to refer to fields through a pointer to a struct.

```
struct Point {int x, y;};

int main(int argc, char *argv[]) {
  struct Point p1 = {0,0};
  struct Point *p1_ptr = &p1;

  p1.x = 1;
  p1_ptr->y = 2;
  return 0;
}
```

Copy by assignment works for struct, i.e. one can assign the value of a struct from a struct of the same type, which copies the entire contents.

Unions

Unions are like a struct, but holds only one of a set of alternative values. They are accessed like a struct.

```
union u {
  int ival;
  float fval;
  char *sval;
};

union u my_uval;
```

5.4 Type definitions

`typedef` introduces a new type definition. Using `typedef` can make code a lot easier to read, especially if you are working with complicated types.

```
typedef unsigned uint32_t;
uint32_t ui;

typedef struct skbuf skbuf_t;
skbuf_t *sptr;
```

5.5 Dynamic data structures

Following an example of a **generic linked list:**

```
struct node {
  void *element;
  struct node *next;
};

struct node *Push(struct node *head, void *e) {
  struct node *n = (struct node *)malloc(sizeof(struct node));

  assert(n != Null);
  n->element = e;
  n->next = head;

  return n;
}
```

We use a `void` pointer to make it generic, meaning we have to always convert to `void *` before pushing, and cast it back from `void *` when accessing.

6. Wrapping up C (for now)

6.1 The C Preprocessor

Macro definitions: In C we have token-based macro substitution.

```
#define FOO BAZ
#define BAR(x) (x+3)
#define QUX
```

`FOO` will be replaced with `BAZ` and `BAR(4)` is replaced with `BAR(4 + 3)` (a string, *4* is not replaced with *7*). Notice that `QUX` would get replaced with `1`.

Macros can be large

```
#define SKIP_SPACES(p, limit)
  do { char *lim = (limit);
    while(p < lim) {
      if(*p++ != ' ') { p--; break; }
    }
  } while(0)
```

Be carefull, using larger marcos can introduce problems with null statements.

6.2 Modularity

Declaration vs. Definitions

- A declaration says something exists, somewhere
- A definition says what it is

C deals with so-called ***compilation units*** which is a C file, plus everything it includes. Declarations can therefore be annotated with:

- *extern*: Definition is somewhere else, either in this compilation unit or another
- *static*: definition is in this compilation unit, and can't be seen outside of it

Modularity in C

A **module** is a self-contained piece of a larger program. The modules *interface* consists of:

- externally visible:
  - functions to be invoked
  - typedefs and perhaps global variables
  - cpp macros
- internal functions, types, global variables
  - that clients should not look at

We implement those modules using **C header files**. For example, the module *foo* has the interface *foo.h*. Clients of *foo* can use it with `#include "foo.h"`. The header file includes no definitions, but only external declarations. The implementation is typically done in *foo.c*, which also includes *foo.h,* which contains no external declarations, but only definitions and internal declarations.

Example: linked lists again

ll.h

```
// Note that the definition of the struct is in the header file!
struct node {
  void *element;
  struct node *next;
};

extern struct node *Push(struct node *head, void *element);
```

ll.c

```
#include "ll.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct node *Push(struct node *head, void *element) {
  //implementation here
}
```

ll_test.c

```
#include "ll.h"

int main(int argc, char **argv) {
  struct node *list = NULL;
  char *hi = "hello";
  char *bye = "goodybe";

  list = Push(list, (void *) hi);
  list = Push(list, (void *) bye);
  return 0;
}
```

6.3 Function pointers

In C we can write something like this:

```
int (*func)(int *, char);
```

Here, `func` is a pointer to a function which takes two arguments, a pointer to an `int` and a `char`, and returns an int. As with all types it can be used with `typedef`.

6.4 Assertions

Assertions are of the following form:

```
assert( <scalar expression> );
```

At run time, the expression is evaluated. If the assertion evaluates to *true*, nothing happens, else the code is aborted and the assertion failure is printed in the console. Assertions are makros.

6.6 setjmp() and longjmp()

`setjmp(env)` saves the current stack state / environment in `env` and returns 0.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

`longjmp(env, val)` causes another return to the points saved by `env`. This new return returns `val`.

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

6.7 Coroutines

Coroutines are general control structures where flow control is cooperatively passed between two different routines without returning. In C, coroutines can be implemented with the help of `setjmp()` and `longjmp()`. For further details on how the implementation exactly works, look at the slides.

7. Implementing dynamic memory allocation

Explicit vs implicit memory allocators

- Explicit: application allocates and frees space (`malloc()` and `free()` in C)
- Implicit: application allocates, but does not free (Freeing is done by Garbage Collector)

7.1 The problem

Constraints

Applications:

- Can issue arbitrary sequence of `malloc()` and `free()`
- `free()` requests must be to a `malloc()`'d block

Allocators:

- Can't control the number or size of allocated blocks
- Must respond immediately to `malloc()` requests (can't reorder or buffer requests)
- Must allocate blocks from free memory
- Must align blocks so they satisfy all alignment requirements (8 byte alignment for GNU *malloc*)
- Can manipulate and modify only free memory
- Can't move the allocated blocks once they are `malloc()`'d

Performance goal: peak memory utilization

Given some sequence of *malloc* and *free* requests $R_0,\ R_1, ...,\ R_k, ...,\ R_{n-1}$.

Def: Aggregate payload P_k

- `malloc(p)` results in a payload of p bytes
- after request R_k has completed, the **aggregate payload** P_k is the sum of currently allocated payloads, i.e. all `malloc()`'d stuff minus all `free()`'d stuff

Def: Current heap size H_k

- assume H_k is monotonically nondecreasing (reminder: it grows when allocator uses `sbrk()`)

Def: Peak memory utilization after k requests

- $U_k = (\max_{i<k} P_i)$

Implementation issues

- How to know how much memory is being `free()`'d when it's given only a pointer and no length?
- How to keep track of the free blocks?
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?
- How to pick a block to use for allocation - many might fit?
- How to reinsert a `free()`'d block into the heap?

Challenge: fragmentation

Internal fragmentation

For a given block, internal fragmentation occurs if the payload is smaller than the block size. This is caused by:

- overhead of maintaining heap data structure
- padding for alignment purposes
- explicit policy decisions

External fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough.

Knowing how much to free

The standard method is to keep the length of a block in the word preceding the block (called header or header field). This requires an extra word for every allocated block.

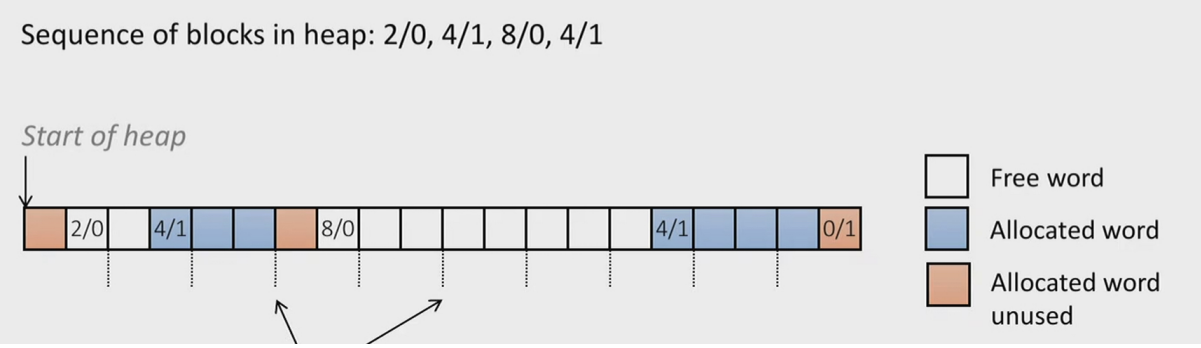
In the following sections, we'll get to know different methods of how to keep track of free blocks.

7.2 Implicit free lists

For each block we only need to know the length of it and whether it's allocated or if it's free. We could store this information in two words, which would be wasteful.

If the blocks are aligned, some low-order address bits are always 0, we therefore can use the lowest-order bit as a flag to indicate if the block is allocated (1) or if its free (0). This flag bit needs to masked out when reading the size of the block.

Example



For this heap with have a **16 byte (2 word) alignment.** This means, that each block must start at a multiple of 16 bytes. The first word, which stores the size and the allocated-flag therefore starts one word before the alignment.

Implicit list: finding a free block

First fit

Search the list from the beginning and choose the first free block that fits:

```
p = start;
while((p < end) &&      // not passed end
      ((*p & 1) ||      // already allocated
      (*p <= len))) {   // too small
  p = p + (*p & -2);    // goto next block (word addressed)
}                     // (-2 to mask of the flag bit)
```

Next fit: Like first-fit, but search list starting where previous search finished.

Best fit: Search the list, choose the best free block: fits, with fewest bytes left over

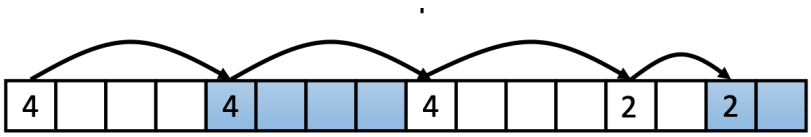
Implicit list: allocating in a free block

Since the space to be allocated might be smaller than the free space, we might want to **split** the block into two/multiple blocks:

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if(newsize < oldsize) {
    *(p+newsize) = oldsize - newsize;   // set length in remaining part of block
  }
}
```

Implicit list: freeing a block

The simplest implementation is to only clear the "allocated" flag. But this can lead to false fragmentation.



Here we could not allocate a block of size 5, even though we have enough space.

7.3 Coalescing

Implicit list: coalescing

To overcome the previously mentioned problem of two blocks not being joined when freeing one, we need to **join (coalesce)** the *free*'d block with the next or previous blocks, if they are free. We can do this the following way:

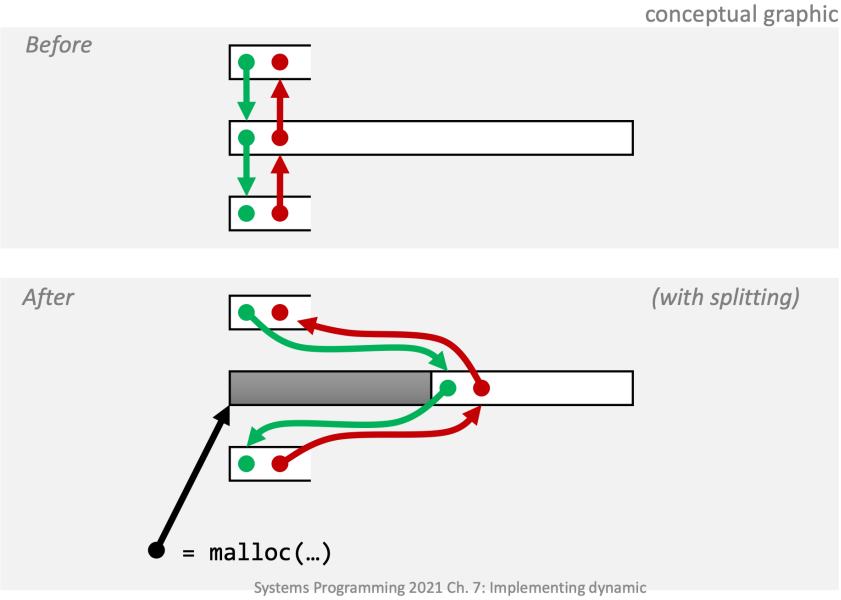
```
void free_block(ptr p) {
  *p = *p & -2;                 // clear allocated flag
  next = p + *p;                // find the next block
  if((*next & 1) == 0) {
    *p = *p + *next;            // add to this block if not allocated
  }
}
```

This helps us with the next block, but what if we need to coalesce with the previous block? For this we introduce bidirectional coalescing. Now we need to replicate the header at the end of the block (footer).

7.4 Explicit free lists

The idea is to maintain a list of only the *free* blocks, not *all* blocks. The next free block could be anywhere, so we need to store forward and backward pointers, not just sizes. Our blocks are now similar to a linked list, we still have header and footer, but additionally the second / third block are pointers to the next / previous free block.

Allocating from explicit free lists



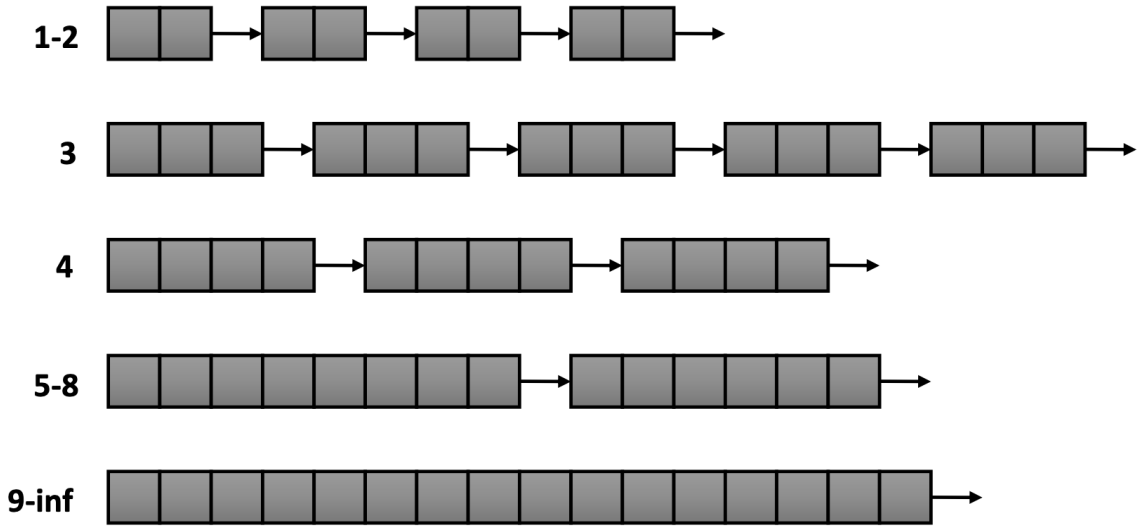
For insertion we can either use LIFO, or we can use address-ordering, both have their own advantages.

Explicit list: summary

Allocating a block is done in linear time in number of free blocks instead of all blocks (as it is in implicit lists). This is much faster when most of the memory is full. It is slightly more complicated to allocate and free blocks since we need to splice blocks in and out of the list.

7.5 Segregated free lists

Segregated free lists (**seglist**) are based on the idea of explicit free lists, with the difference that we have different free lists for different size classes.

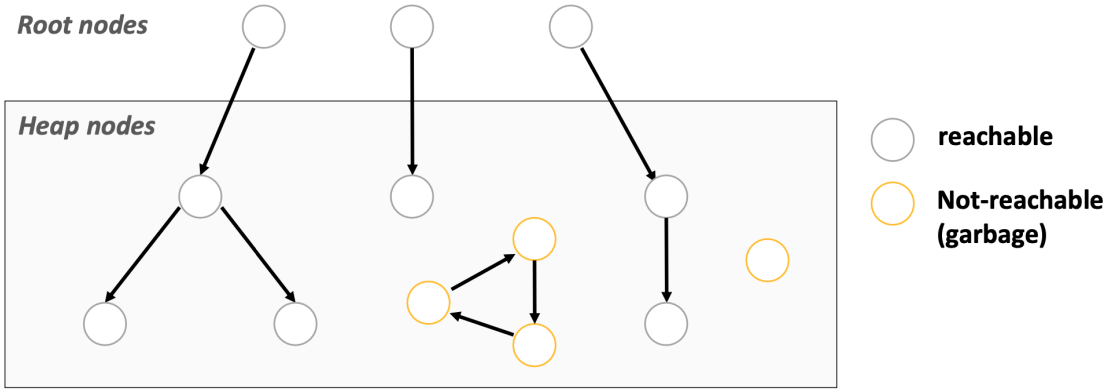


We often have separate classes for each small size, for larger sizes we often have one class for each power-of-two size. Allocating is pretty straight forward, we simply look for a block with size larger than what we want. Freeing is also simple, we first coalesce and then place the block on the appropriate list.

7.6 Garbage collection

Garbage collection automaticly reclaims heap-allocated storage that isn't used anymore - the application doesn't have to free memory. There are multiple ways of doing this, we only looked at mark and sweep collecting.

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)



When we run out of space, we run our garbage collector:

- ***Mark***: Start at roots and set mark bit on each reachable block
- ***Sweep:*** Scan all blocks and free blocks that are not marked

7.7 Memory pitfalls

These are some of the most common mistakes, that are made working with memory allocation.

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks → Memory leaks

8. Basic x86 Architecture

8.1 What is an instructions set architecture?

The **architecture** are the parts of a processor design that one needs to understand to write assembly code (i.e. instruction set, registers, etc.). The **microarchitecture** is the implementation of the architecture (i.e. cache sizes, core frequency, etc.).

Complex Instruction Set (*CISC*)

- Stack-oriented instruction set
- Arithmetic instructions can access memeory
- Condition codes
- Easy for compiler
- Smaller code size

Reduced Instruction Set (*RISC*)

- Fewer, simpler instructions
- Register-oriented instruction set
- No condition codes
- Better for optimizing compilers
- Run fast with simple chip design

8.3 Basics of machine code

Compiling into assembly

Given the following piece of C code:

```
int sum(int x, int y) {
  int t = x + y;
  return t;
}
```

If we run the following command in the terminal `gcc -0 -S code.c` we can produce the file `code.s` containing the assembly code of the above written file.

```
sum:
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, -20(%rbp)
  movl %esi, -24(%rbp)
  movl -24(%rbp), %eax
  movl -20(%rbp), %edx
  addl %edx, %eax
  movl %eax, -4(%rbp)
  movl -4(%rbp), %eax
  popq %rbp
  ret
```

Object code

The **assembler** does:

- Translate `.s` into `.o`
- Binary encoding of each instructions
- Nearly-complete image of executable code

The **linker** does:

- Resolves references between files
- Combines with static run-time libraries

Machine instruction example

C Code: Add two signed integers

```
int t = x + y;
```

Assembly: Add two 4-byte integers

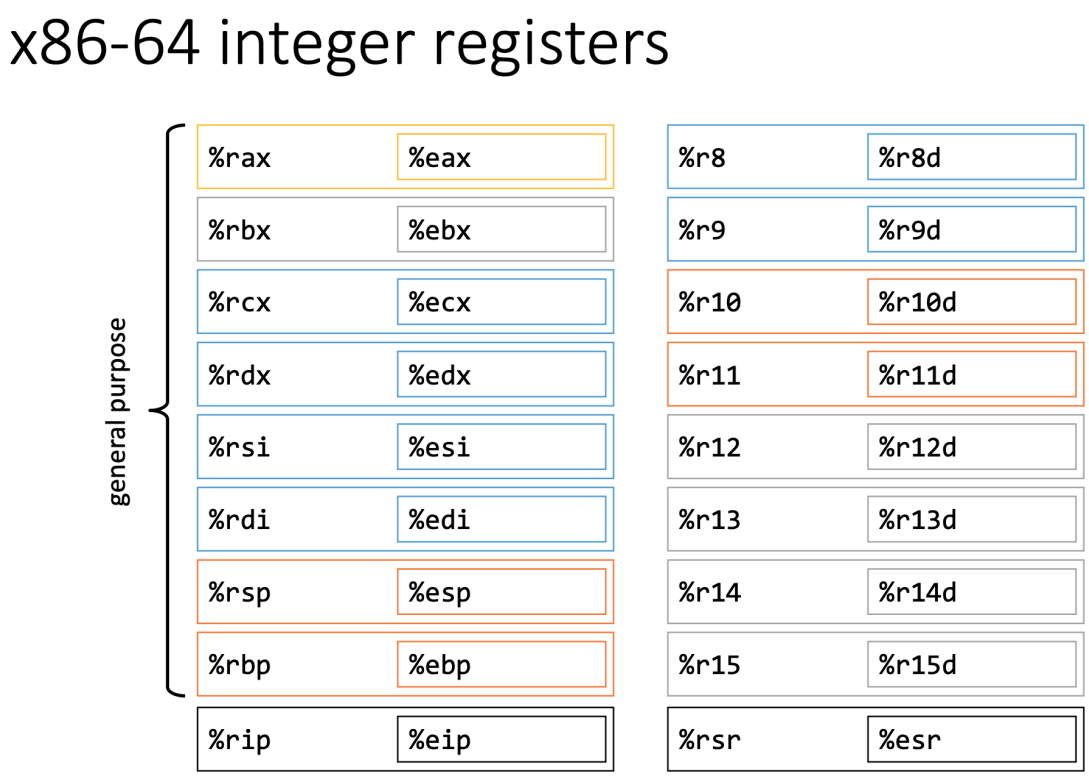
```
addl 8(%rbp), %eax
```

- Operands:
  - x: Register `%eax`
  - y: Memory `M[%rbp+8]`
  - t: Register `%eax`
- Return function value at `%eax`

```
0x401046:  03 45 08
```

- 3-byte instruction
- Stored at address `0x401046`

8.4 x86 architecture



Moving data

We can move data with `movx Source, Dest` where x is in {b, w, l, q}

- `movq Source, Dest` - Move 8-byte "quad word"
- `movl Source, Dest` - Move 4-byte "long word"
- `movw Source, Dest` - Move 2-byte "word"
- `movb Source, Dest` - Move 1-byte "byte"

Furthermore we have the following **operand types:**

- *Immediate*: Constant integer data, example `$0x400` or `$-533`
- *Register*: One of 16 integer register, example `%eax` or `%r14d`
- *Memory*: 1, 2, 4, or 8 consecutive bytes from memory at address given by register, example `(%rax)`

Simple memory addressing modes:

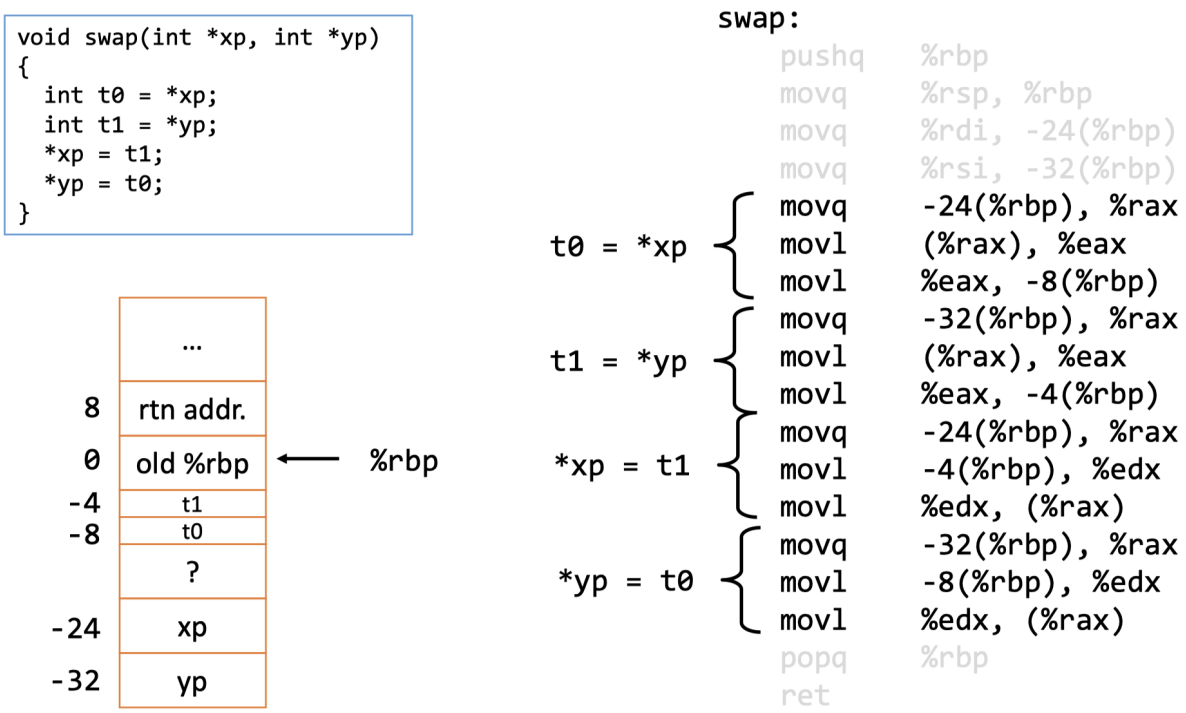
- Normal - (R) - Mem[Reg[R]]
  - `movq (%rcx), %rax`
- Displacement - D(R) - Mem[Reg[R] + D]
  - Register R specifies start of memory region and D specifies a offset
  - `movl 8(%ebp), %edx`

We can define the **complete memory addressing mode** in the most general form as follows:

`D(Rb, Ri, S)` which is somewhat equivalent to Mem[Reg[Rb] + S*Reg[Ri] + D] where

- D: Constant displacement of 1, 2, or 4 bytes
- Rb: Base register, any of 16 integer registers
- Ri: Index register, any, except for `%rsp`
- S: Scale, 1, 2, 4, or 8

Example: *swap (without optimizer)*



8.6 Condition codes

Condition codes are single bit register that are set by arithmetic operations:

- CF *Carry Flag* (for unsigned) - set if carry out from most significant bit
- ZF *Zero Flag - result = 0*
- SF *Sign Flag* (for signed) - set if negative result
- OF *Overflow Flag* (for signed) - set if two's complement overflow

These are not set by `lea` instructions.

Reading Condition Codes

With so called *SetX Instructions* we can set single bytes based on combinations of condition codes:

9. Compiling C Control Flow

9.1 `if-then-else` statements

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

Else:
  val = Else-Expr;

Done:
  return
```

9.2 `do-while` loops

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

9.3 `while` loops

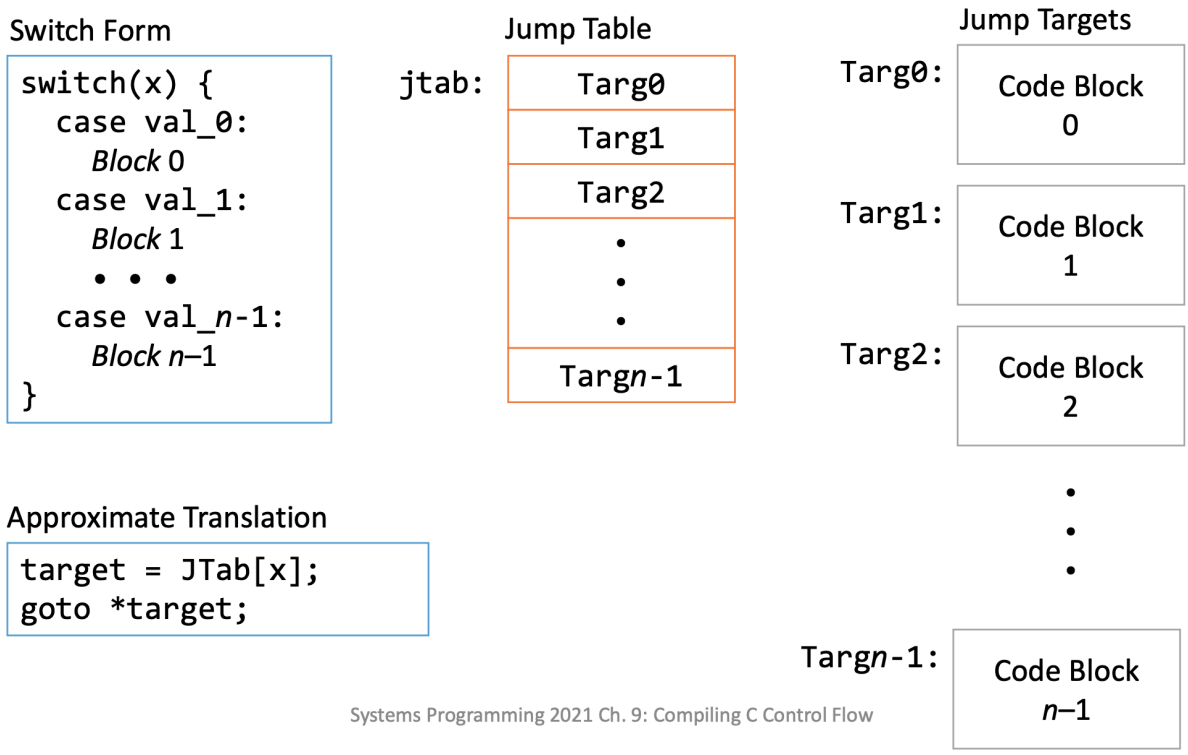
```
  goto middle;
Loop:
  ...
middle:
  if (Test) goto Loop;
```

9.4 `for` loops

```
Init:
  if(!Test) goto Done;

Loop:
  ...
  Update;
  if(Test) goto Loop;

Done:
```

9.5 Compact `switch` statements

Jump table structure



This transfers to the following assembly setup:

```
switch_eg:
  movq   %rdx, %rcx
  cmpq   $6, %rdi        # x : 6 ?
  ja     .L8             # if > goto default
  jmp    *.L4(, %rdi, 8) # goto Jtab[x]
```

We differ between *direct* and *indirect jumping:*

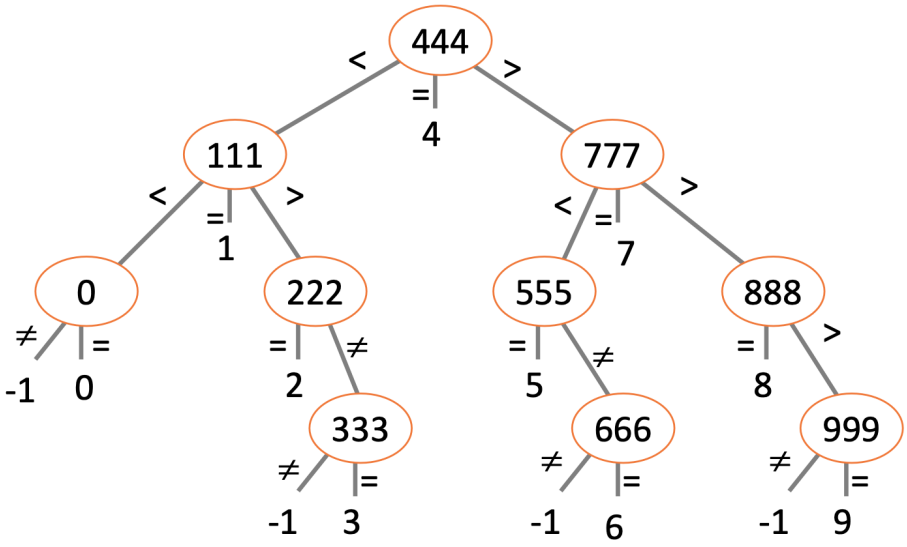
- Direct: `jmp .L8` → Jump target is denoted by label `.L8`
- Indirect: `jmp .L4(, %rdi, 8)` → Fetch target from effective address `.L61 + rdi*8`, must scale by factor 8 (since labels are 64-bit = 8 Bytes on x86_64 machines)

9.6 Sparse `switch` statements

We look at the following piece of code:

```
/* Return x/111 if x is multiple && <= 999. -1 otherwise */
int div111(int x) {
  switch(x) {
    case   0: return 0;
    case 111: return 1;
    case 222: return 2;
    case 333: return 3;
    case 444: return 4;
    case 555: return 5;
    case 666: return 6;
    case 777: return 7;
    case 888: return 8;
    case 999: return 9;
    default:  return -1;
  }
}
```

Here it wouldn't be practical to use a jump table, since this would require 1000 entries, of which 990 entries would not be meaningful at all (i.e. would be default cases). The compiler proposes the following ordering and execution of the above code with `if-`statements:



- Organizes cases as binary tree
- Logarithmic performance

9.7 Procedure call and return

x86_64 Stack

A ***stack*** is a region of memory managed with stack discipline. Register `%rsp` (register stack pointer) contains the lowest stack address, i.e. the address of the "**top**" element.

Push

In x86_64 we have a `pushl` *`Src`* function:

- Fetches operand at *Src*
- Decrements `%rsp` by 4
- Writes operand at address given by `%rsp`

Pop

The `popl` *`Dest`* does the following:

- Reads operand at address `%rsp`
- Increments `%rsp` by 4
- Writes operand to *Dest*

Procedure control flow

We use a stack to support procedure call and return.

- *Procedure call:* `call` *`label`*
  - Push return address on stack
  - Jump to *label*

- *Procedure return:* `ret`
  - Pop address from stack
  - Jump to address

9.8: x86_64 calling conventions

When a procedure `yoo` calls a function `who`, we say that yoo is the ***caller*** and who is the ***callee***. If we use registers for temporary storage, we differ between two conventions:

- *Caller Save:* Caller saves the temporary in its frame before calling
- *Callee Save:* Callee saves the temporary in its frame before using

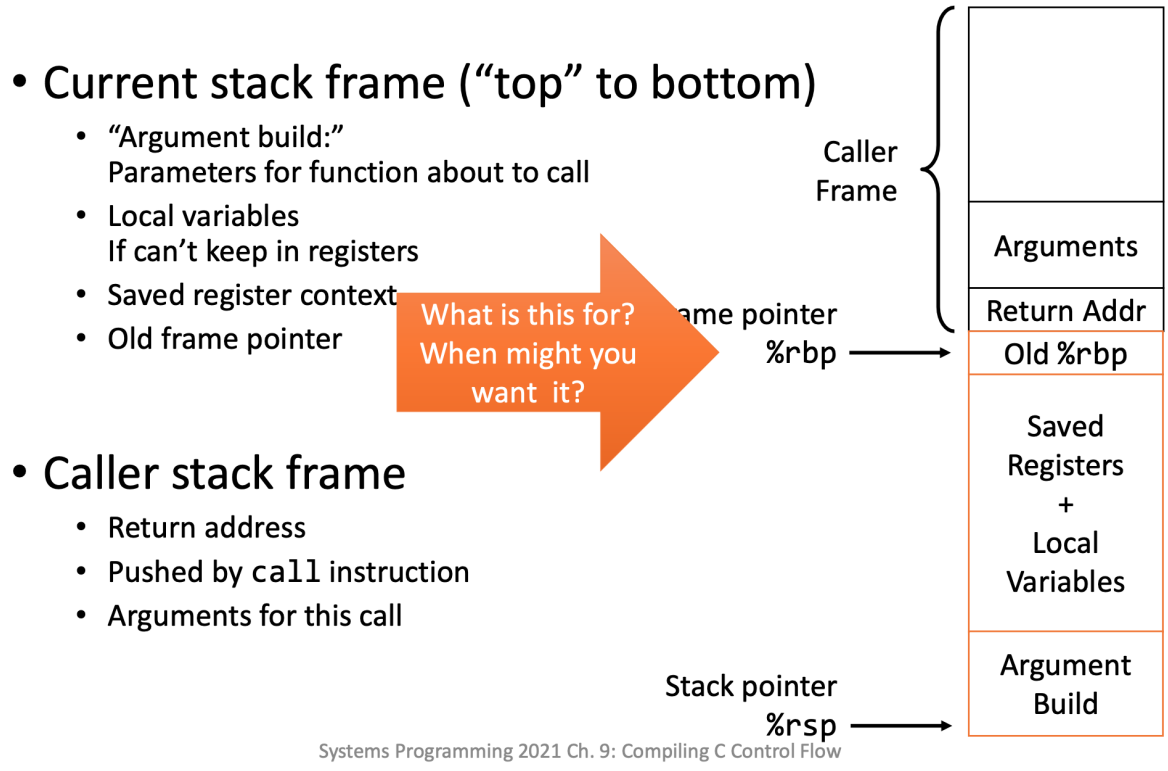
---

Registers:

- `%rax & %eax` used without first saving
- `%rbx & %ebx` used, but saved at beginning and restored at end
- Arguments passed to functions via registers:
  - If more than 6 integral parameters are needed, then pass the rest onto the stack
  - These registers can be used as caller-saved as well
- All references to stack frame via stack pointer

%rax	Return value, # varargs	%r8	Argument #5
%rbx	Callee saved; base ptr	%r9	Argument #6
%rcx	Argument #4	%r10	Static chain ptr
%rdx	Argument #3 (& 2nd return)	%r11	Used for linking
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved; frame ptr	%r15	Callee saved

Full x86_64 / Linux stack frame



We use the `%rbp` for cases where we have an unknown amount of arguments.

10. Compiling C Data Structures

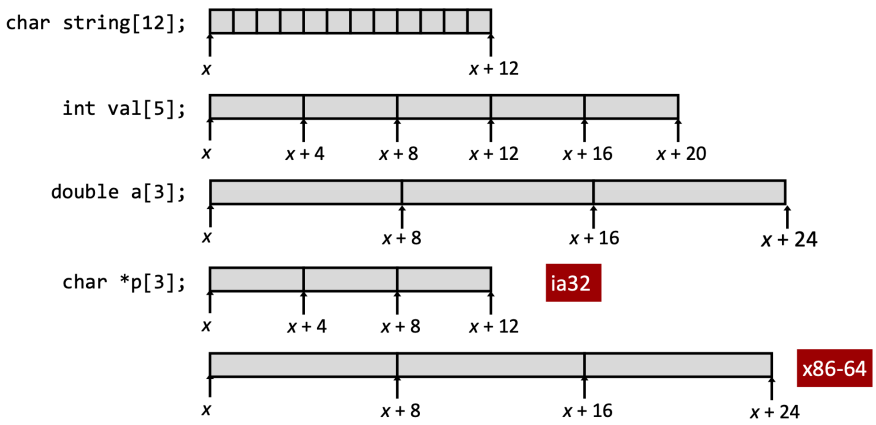
10.1 One-dimensional arrays

Following a quick recap of ***basic data types*** we have already seen:

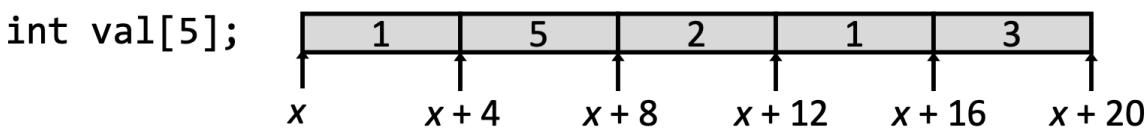
- Stored and operated on in general integer registers
- Signed vs. unsigned depends on instructions used
- Stored and operated on in floating point registers

Array allocation

- T A[L] : Array of
  - data type T
  - length L
- Contiguously allocated region of L * sizeof(T) bytes



Array access



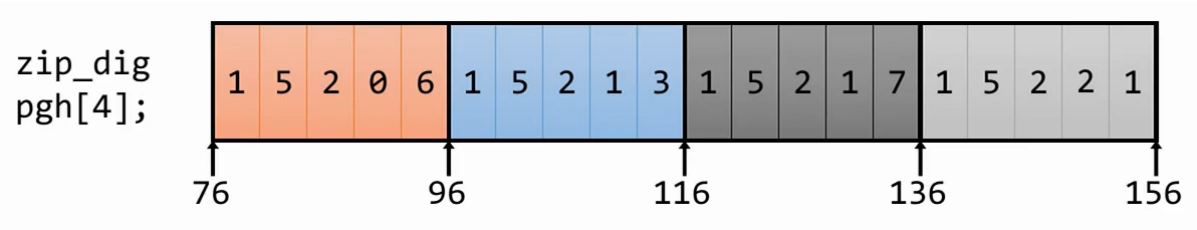
Reference	Type	Value
val[4]	int	3
val	int *	x
val + 1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 *I*

10.2 Nested arrays

We take a look at the following example:

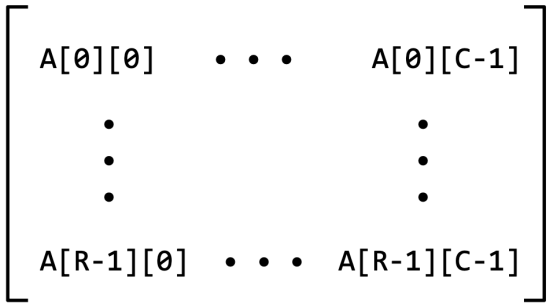
```
typedef int zip_dig[5];
#define PCOUNT 4

zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3},
   {1, 5, 2, 1, 7},
   {1, 5, 2, 2, 1}};
```

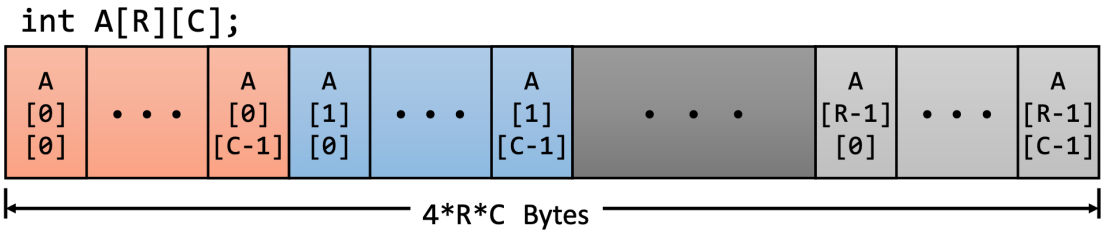
Allocated in memory the above defined array looks like this:



We can generalize the idea of ***multidimensional nested arrays*** as follows:

- Declaration
  T A[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- Array Size
  - *R* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering



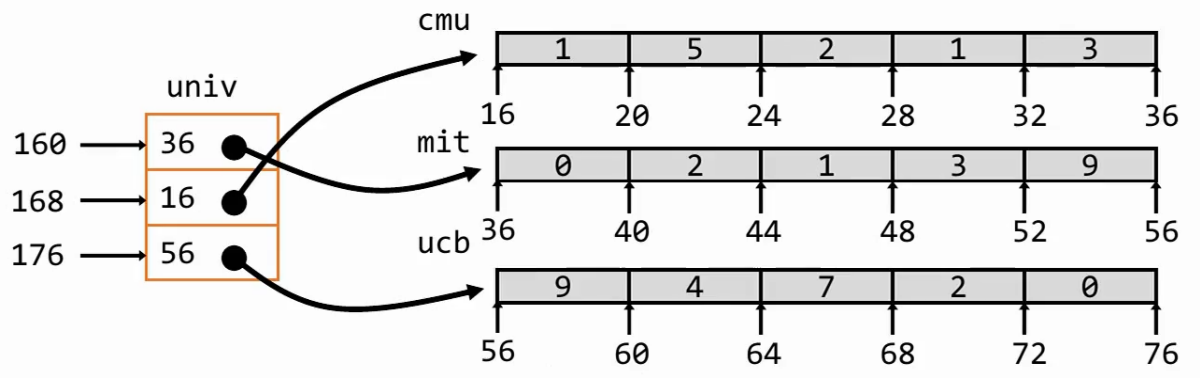


10.3 Multi-level arrays

Example:

```
zip_dig cmu = {1, 5, 2, 1, 3};
zip_dig mit = {0, 2, 1, 3, 9};
zip_dig ucb = {9, 4, 7, 2, 0};

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer (8 bytes)
- Each pointer points to an array of int's



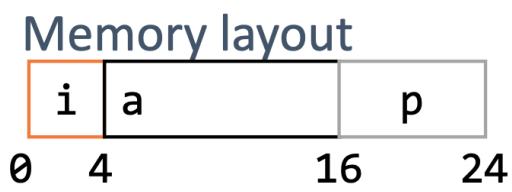
Element access on multi-level arrays is done with `Mem[Mem[univ + 8*index] + 4*dig]`. This is different to the nested array, as the arrays here are not in consequtive memory locations.

10.4 Structures

Concept

A `struct` is a contiguously-allocated region of memory. One refers to members within structures by names and members may be of different types.

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



Accessing strcture member

```
void set_i(struct rec *r, int val) {
  r -> i = val;
}
```

```
set_i:
  movl %esi, (%rdi)
  ret
```

Generating pointer to structure member

```
int *find_a(struct rec *r, ind idx) {
  return &r -> a[idx]
}
```

```
find_a:
  movslq %esi, %rsi
  leaq   4(%rdi, %rsi, 4), %rax
  ret
```

Structure referencing

```
void set_p(struct rec *r) {
  r -> p = &r -> a[r -> i];
}
```

```
set_p:
  movslq (%rdi), %rax
  leaq   4(%rdi, %rax, 4), %rax
  movq   %rax, 16(%rdi)
  ret
```

10.5 Alignment

Primitive data types require *K* bytes → Address must be a ***multiple*** of *K*.

1 byte: `char`, …
- no restrictions on address

2 bytes: `short`, …
- lowest 1 bit of address must be 0_2

4 bytes: `int`, `float`, …
- lowest 2 bits of address must be 00_2

8 bytes: `double`, `char *`, …
- Windows & Linux:
  - lowest 3 bits of address must be 000_2

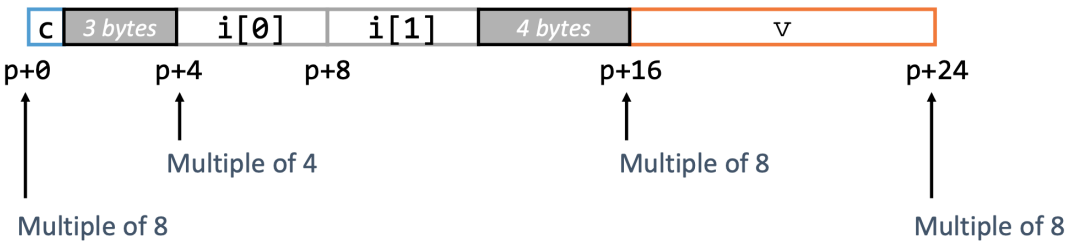
16 bytes: `long double`
- Linux:
  - lowest 3 bits of address must be 000_2
  - i.e., treated the same as a 8-byte primitive data type

Satisfying alignment with structures

Each structure has an alignment requirement *K*, where *K* is the largest alignment of any element. The initial address and structure length must be multiples of *K*. Inside the structure, every element has to be aligned according to its own rules.

Example
(under Windows or x86-64):
- K = 8, due to double element

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



10.7 Unions

Unions are allocated according to the largest element.

11. Linking

We consider the following example C program:

```
int buf[2] = {1, 2};

int main() {
  swap();
  return 0;
}
```

```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap() {
  int temp;

  bufp1  = &buf[1];
  temp   = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Programs are translated an linked using a compiler driver `gcc -O2 -g -o p main.c swap.c`. Linking enables us to write a program as a collection of smaller source files, rather than one monolithic mass.

What do linkers do?

Step 1: Symbol resolution

- Programs define and reference ***symbols*** (variables and functions)
- Symbols definitions are stored (by the compiler) in a ***symbol table***
- Linker associates each symbol ***reference*** with exactly one symbol ***definition***

Step 2: Relocation

- ***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
- Updates all references to these symbols to reflect their new positions

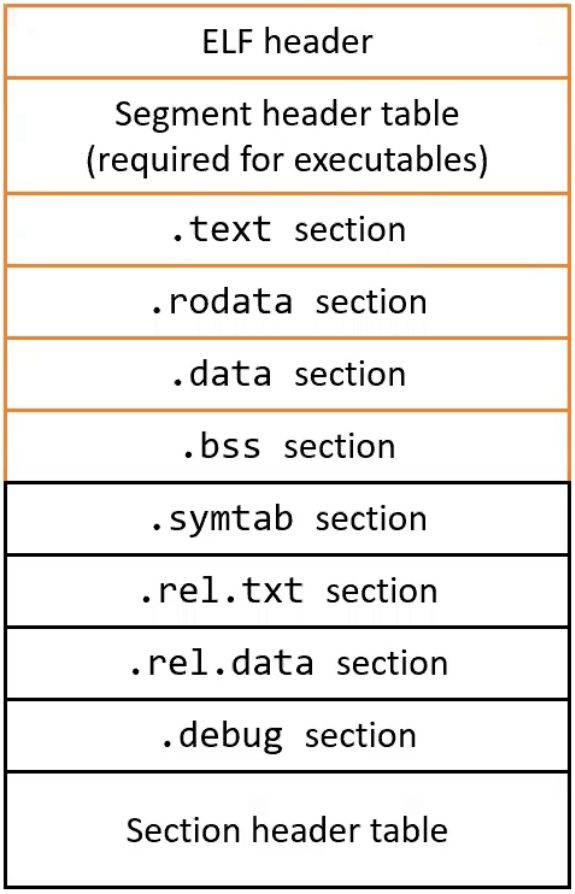
11.1 Object files

We distinguish between three types of object files (modules):

- Relocatable object file (`.o` files)
  - Contains code and data in form that can be combined with other relocatable object files to form an executable object file
  - Each `.o` file is produced from exactly ***one source file*** (`.c` file)
- Executable object file
  - Contains code and data in a form that can be copied directly into memory and then be executed
- Shared object file (`.so` file)
  - Special type of relocatable object file that can be loaded into memory and linked *dynamically,* at either load time or run-time
  - Called *Dynamic Link Libraries* (DLLs) by Windows

The ***ELF format*** (executable and linkable format) is a standard binary format for object files. It looks like follows:

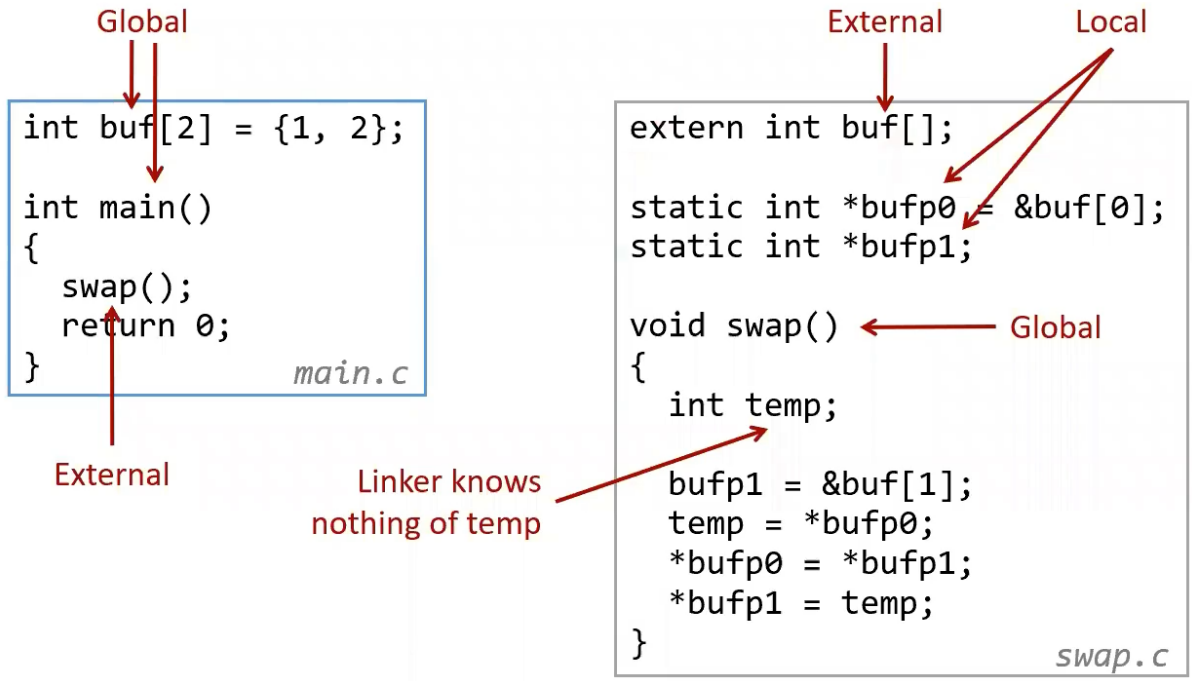
- Elf header: Word size, byte ordering, file type, machine type, etc.
- Segment header table: Page size, virtual addresses memory segments, segment sizes
- `.text` section: Code
- `.rodata` section: Read only data like jump tables etc.
- `.data` section: Initialized global variables
- `.bss` section: Uninitialized global variables etc.
- `.symtab` section: Symbol table, procedure and static variable names, section names and locations
- `.rel.text` section: Relocation info for `.text` section, instructions for modifying
- `.rel.data` section: Relocation info for `.data` section
- `.debug` section: Info for symbolic debugging (`gcc -g`)
- Section header table: Offsets and sizes of each section



11.2 Linker symbols

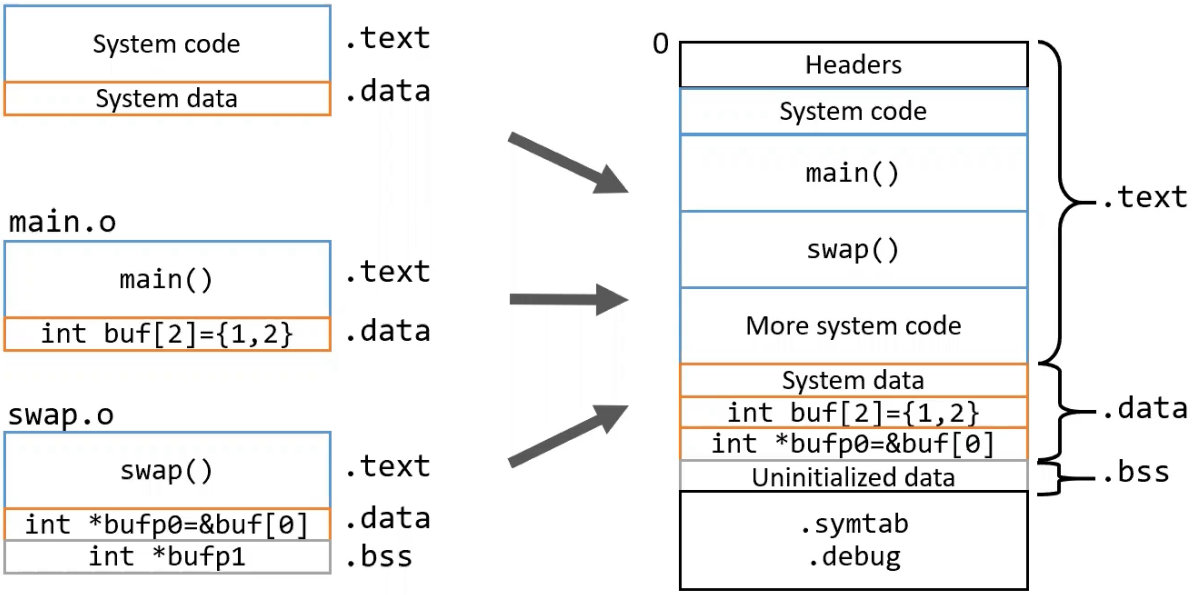
We distinguish between three types of linker symbols:

- 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
- External symbols
  - Global symbols that are referenced by module *m* but defined by some other module
- Local symbols
  - Symbols that are defined and referenced exclusively by module *m*
  - e.g. C functions and variables defined with the static attribute
  - *Local linker symbols are not local program variables! (e.g. temp in the example below)*



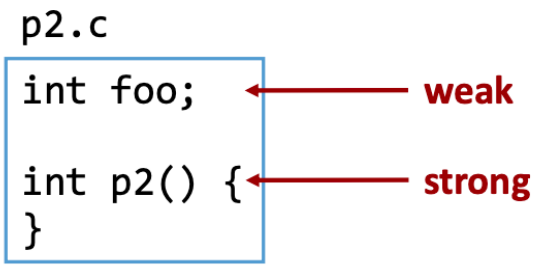
Relocating code and data

We can see how the relocation of code and data works with our *swap*-example:



For program symbols we furthermore distinguish between **strong and weak symbols**:

- Strong: procedures and initialized globals
- Weak: uninitialized globals

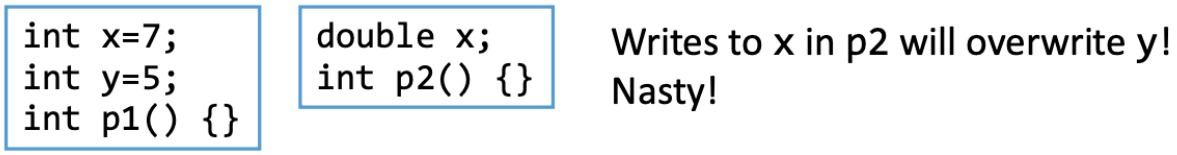


The linker's symbol rules

The linker has to check if there are multiple symbols with the same name.

1. Multiple strong symbols are not allowed
   - Each item can be defined only once
   - Otherwise we get a linker error
2. Given a strong symbol and multiple weak symbols, the linker chooses the strong symbol
   - References to weak symbols resolve to the strong symbol
3. If there are multiple weak symbols, the linkers picks an arbitrary one
   - Can override this with `gcc -fno -common`

This example shows some interesting behaviour. Since `int x = 7` is a strong symbol, we will allocate 4-byte of memory. Now if we would try to write to `x` from `p2`, we would try to write a `double`! Since we only allocated 4-bytes and a `double` takes 8-bytes, we would overwrite the following symbol, in this case `y`.



11.3 Static libraries

We can describe ***static libraries*** (`.a` archive files) the following way:

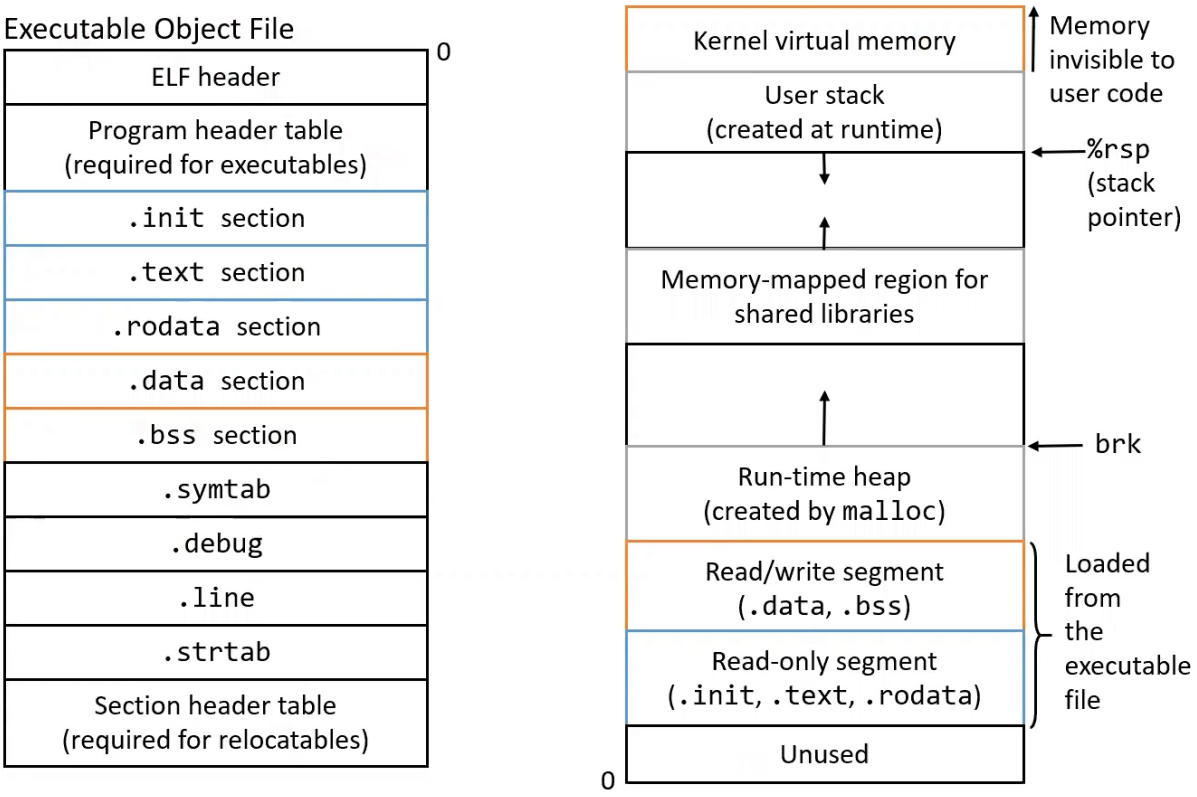
- Concatenate related relocatable object files into a single file with an index (called an archive)
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives
- If an archive member file resolves a reference, link it into the executable

It is important that command line order matters when compiling a file. Always put the static libaries at the end of the command!

Commonly-used libraries

- `libc.a` : the C standard library
- `libm.a` : the C math library

Loading executable object files



11.4 Shared libraries

Static libraries have the following disadvantages:

- Duplication in the stored executable
- Duplication in the running executables
- Minor bug fixes of system libraries require each application to explicitly relink

The solution to the above mentioned problems are ***shared libraries:***

- Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
- Also called: dynamic link libraries (DLLs) or `.so` files

We can either have dynamic linking when the executable gets loaded (**load-time linking**) or the linking can also occure after the program has begun (**run-time linking**).

Such shared libaries, routines can be shared by multiple processes! E.g. `printf()` needs to be loaded once and not by every program running. This is the reasons that mostly the standard C library (`libc.so`) is dynamically linked.

> ❗ By default `gcc` / `clang` includes some standard C libaries, therefore we do not have to include it in the command line.

12. Code Vulnerabilities

12.1 Worms and Viruses

A ***Worm*** is a program that:

- Can run by itself
- Can propagate a fully working version of itself to other computers

A ***Virus*** is a code that:

- Adds itself to other programs
- Cannot run independently

→ Both are usually designed to spread among computers and to wreak havoc.

12.2 Stack overflow bugs

Consider the following code of `gets()` :

The problem here is that there is no way to specify a limit on number of characters to read (buffer overflow).

> ❗ Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines. To achive this we load such much data into the buffer until we overwrite the return address, allowing us to decide where to jump.

```
/* Get string from stdin */
char *gets(char *dest) {
  int c = getchar();
  char *p = dest;
  while(c != EOF && c != '\\n') {
    *p++ = c;
    c = getchar();
  }
  *p = '\\0';
  return dest;
}
```

12.3 Stopping overrun bugs

We can avoid overflow vulnerabilities by using functions that limit string lengths.

- `fgets()` instead of `gets()`
- `strncpy()` instead of `strcpy()`

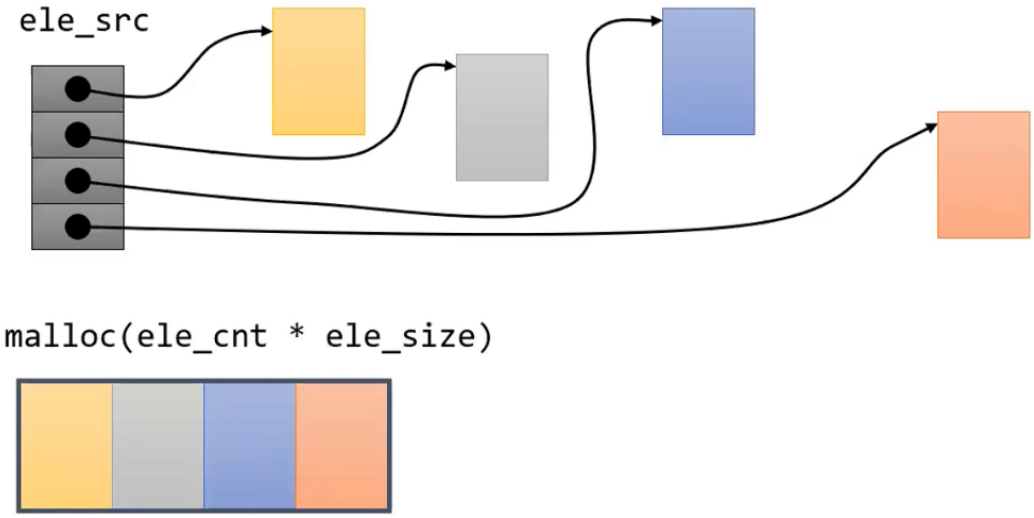
System-level protections

- Compiler-inserted checks on functions
- Randomized stack offsets: At the start of a program, allocate a random amount of space on stack, this makes it difficult to predict the beginning of inserted code.
- Nonexecutable code segments, marking regions of memory as *read-only* or *writeable*.

12.4 Another example: XDR

The SUN XDR library is a widely used library for transferring data between machines (e.g. for **N**etwork **F**ile **S**ystems). A common use is to send an array of blocks to different machines:

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



Notice that `int ele_cnt` is signed, but `size_t ele_size` is unsigned. What if, on a 32-bit machine, we have:

- `ele_cnt` = $2^{20} + 1$
- `ele_size` = $4096 = 2^{12}$

We will overflow the 32-bit limit and only allocate 1 byte, which in the end results in us overwriting *a lot* of data!

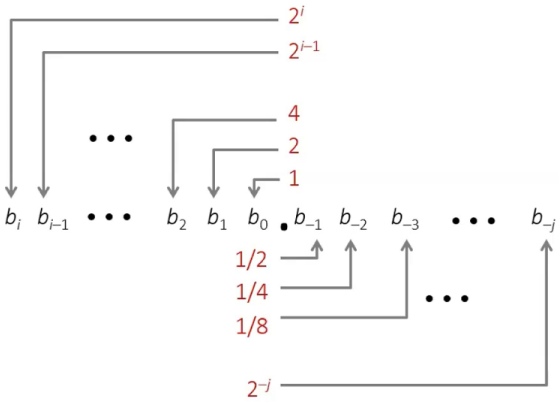
13. Floating Point

13.1 Representing floating-point numbers

Fractional binary numbers

We represent rational numbers as:

$$\sum_{k=-j}^{j} b_k \cdot 2^k$$



We make the following observations:

- Dividing by 2 can be done by shifting to the right
- Multiplying by 2 can be done by shifting to the left

We can easily represent numbers of the form $x/2^k$, but other rational numbers have repeating bit representations and can't be represented accuratly.

IEEE Floating Point

The ***IEEE Standard 754*** was established in 1985 as an uniform standard for floating point arithmetic.

Floating point representation

Numerical Form:

$$(-1)^S * M * 2^E$$

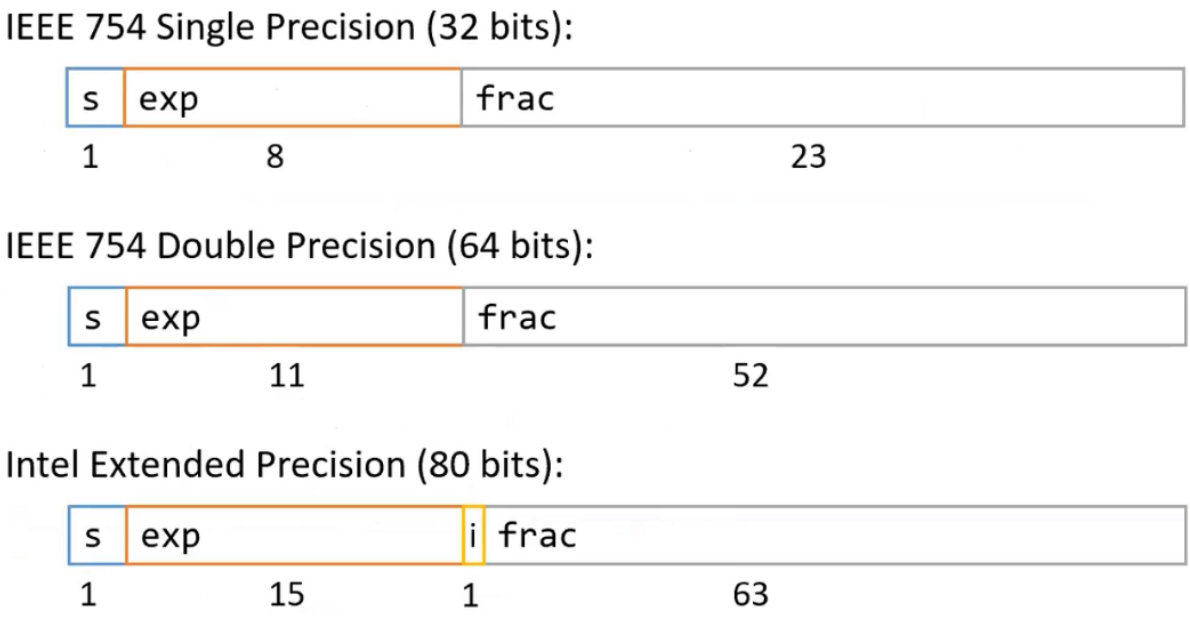
- Sign bit S determines whether the number is negative or positive
- Significand M is normally a fractional value in the range $[1.0, 2.0)$
- Exponent E weights value by power of two

Encoding

- MSB s is the sign bit S
- `exp` field encodes E (but is not actually equal to E)
- `frac` field encodes M (but is not actually equal to M)

13.2 Types of IEEE floating-point numbers

Precisions



There are many more different types of representations, used for different applications.

Floating point in C

C99 gruarantees two levels:

- `float` → single precisions
- `double` → double precisions
- `long double` → can be double, extended, or quadruple precision

The exponent is coded as ***biased*** values: $E = Exp - Bias$

- *Exp*: unsigned value `exp`
- *Bias* $= 2^{e-1} - 1$, where e is the number of exponent bits
  - Single precision: 127 (*Exp*: 1...254, E = -126...127)
  - Double precision: 1023 (*Exp*: 1...2046, E: -1022...1023)

The significand is coded with implied ***leading 1:*** $M = 1.xxx...x_2$

Normalized encoding example (exp $\neq$ 0)

- Value: `float F = 15213.0;`
  15213_{10} $= 11101101101101_2$
  $= 1.1101101101101_2 \times 2^{13}$
- Significand
  - *M* = $1.\underline{1101101101101}_2$
  - `frac` = $\underline{1101101101101}0000000000_2$
- Exponent
  - *E* = 13
  - *Bias* = 127
  - *Exp* = $140 = 10001100_2$
- Result:

0	10001100	11011011011010000000000
s	exp	frac

Denormalized values

Condition: `exp` $=$ 000...0

- **Exponent value is $E = -Bias + 1$**
- Significand is coded with implied ***leading 0:*** $M = 0.xxx..x_2$

Special values

Condition: `exp` $=$ 111...1

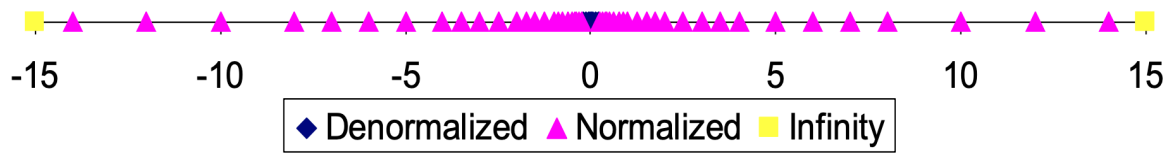
- `frac` $=$ 000...0 → ∞
- `frac` $\neq$ 000...0 → NaN

> ❗ This is the best possible representation for floating point numbers we can get, if we want more precision we end up with a lot more storage space used and way longer computation times.

13.3 Floating-point ranges

	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	largest norm
	0 1111 000	n/a	inf	

Looking at a tiny 6-bit IEEE-like format with 3 exponent bits and 2 fraction bits, we see the following distribution, where it becomes denser towards zero.



13.4 Floating-point rounding

The standard rounding mode in IEEE is ***nearest even.*** But there are also other rounding modes, including: towards zero, round down, round up.

In binary even is when the least significant bit is 0. We round up if the bits to the right of the rounding position are $100\ldots_2$.

Value	Binary	Rounded	Action	Result
$2\,3/32$	10.00011_2	10.00_2	< ½ : down	2
$2\,3/16$	10.00110_2	10.01_2	> ½ : up	$2\,1/4$
$2\,7/8$	10.11100_2	11.00_2	= ½ : up	3
$2\,5/8$	10.10100_2	10.10_2	= ½ : down	2 ½

Creating a floating point number

We have the following steps:

1. Normalize to have leading 1

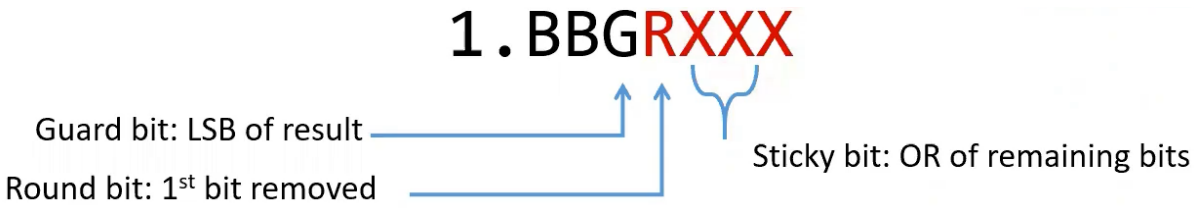
2. Round to fit within fraction
3. Postnormalize to deal with effects of rounding

We examine the following example:

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

For ***rounding*** we divide the fraction bits into four different types:

- B-bits: the surviving bits
- G-bit (*Guard*): the last surviving bit
- R-bit (*Round*): the first bit to be removed
- S-bit (*Sticky*): the OR of the remaining bits



We the follow the ***round up conditions:***

- $R = 1$ AND $S = 1 \Rightarrow > 0.5$
- $G = 1$ AND $R = 1$ AND $S = 0 \Rightarrow$ round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Rounding now might have caused overflow, if this is the case we need to **postnomalize**. This means we shift once to the right and increment the exponent by 1.

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

13.5 Floating-point addition and multiplication

Floating-point multiplication

We consider the following ***multiplication:***

$$(-1)^{S1} M_1 2^{E1} \cdot (-1)^{S2} M_2 2^{E2}$$

The exact result of this multiplication is given by $(-1)^S M 2^E$ where:

- Sign $S = S1 \hat{} S2$
- Significand $M = M_1 * M_2$
- Exponent $E = E_1 + E_2$

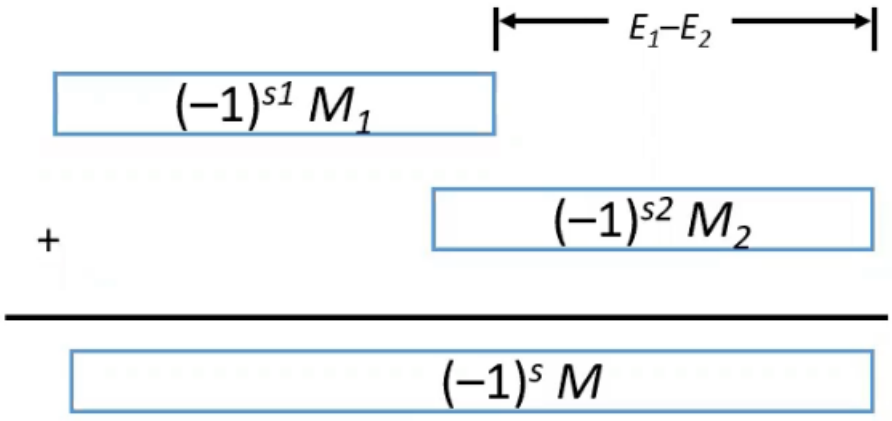
Possibly occuring problems can be fixed the following way:

- If $M \geq 2$, shift M to the right and increment E
- If E is out of range, we have an overflow
- Round M to fit `frac` precision

Floating-point addition

W.L.O.G. we assume $E_1 > E_2$.

When ***adding*** two floating-point numbers, both the sign S and the significand M are given by signed aligned addition. The exponent E is equal to E_1.



Possibly occuring problems can be fixed the following way:

- If $M \geq 2$, shift M to the right, increment E
- If $M < 1$, shift M to the left k positions and decrement E by k
- If E is out of range we have an overflow
- Round M to fit the `frac` precision

13.7 SSE floating point

SIMD (single-instructiom, multiple data) vector instructions allow for parallel operation on small (length 2-8) vectors of integers or floats. Floating point vector instructions are available with Intel's SSE (streaming SIMD extensions) family.

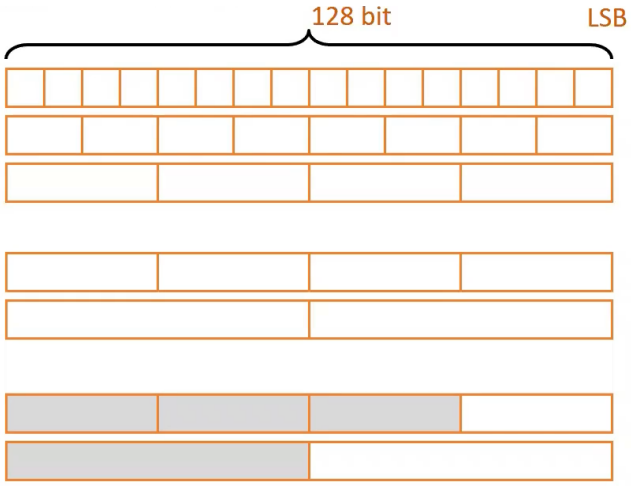
SSE3 register

All SSE3 registers are caller saved and 128 bit (= 2 doubles = 4 singles) wide. `%xmm0` is used for floating point return values.

%xmm0	Argument #1	%xmm8
%xmm1	Argument #2	%xmm9
%xmm2	Argument #3	%xmm10
%xmm3	Argument #4	%xmm11
%xmm4	Argument #5	%xmm12
%xmm5	Argument #6	%xmm13
%xmm6	Argument #7	%xmm14
%xmm7	Argument #8	%xmm15

Those registers do have different data types and associated instructions:

- Integer vectors:
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
- Floating point vectors:
  - 4-way single
  - 2-way double
- Floating point scalars:
  - single
  - double



SSE3 basic instructions

We will focus on scalar operations.

- Moves:

Single	*Double*	*Effect*
movss	movsd	D ← S

  - Usual operand form: reg → reg, reg → mem, mem → reg

- Arithmetic:

Single	*Double*	*Effect*
addss	addsd	D ← D + S
subss	subsd	D ← D – S
mulss	mulsd	D ← D x S
divss	divsd	D ← D / S
maxss	maxsd	D ← max(D,S)
minss	minsd	D ← min(D,S)
sqrtss	sqrtsd	D ← sqrt(S)

x86-64 FP code example

The task is to compute the inner product of two vectors.

```
float ipf(float x[], float y[], int n) {
  int i;
  float result = 0.0;

  for(i = 0; i < n: i++) {
    result += x[i]*y[i];
  }
  return result;
}
```

```
ipf:
  xorps   %xmm1, %xmm1             # result = 0.0
  xorl    %ecx, %ecx               # i = 0
  jmp     .L8                      # goto middle
.L10:                              # loop:
  movslq  %ecx, %rax               # icpy = i
  incl    %ecx                     # i++
  movss   (%rsi, %rax, 4), %xmm0   # t = y[icpy]
  mulss   (%rdi, %rax, 4), %xmm0   # t *= x[icpy]
  addss   %xmm0, %xmm1             # result += t
.L8:                               # middle
  cmpl    %edx, %ecx               # i:n
  jl      .L10                     # if <, goto loop
  movaps  %xmm1, %xmm0             # return result
  ret
```

Constants

```
double cel2fahr(double temp) {
  return 1.8 * temp + 32.0;
}
```

```
cel2fahr:
  mulsd .LC2(%rip), %xmm0   # Multiply by 1.8
  addsd .LC4(%rip), %xmm0   # Add 32.0
  ret
```

```
.LC2:
  .long 3435973837   # Low order four bytes of 1.8
  .long 1073532108   # High order four bytes of 1.8
.LC4:
  .long 0            # Low order four bytes of 32.0
  .long 1077936128   # High order four bytes of 32.0
```

To check that 1077936128 corresponds to 32.0 we first convert the number to hexadecimal:

$$1077936128 \Rightarrow 0x40400000$$

We have 11 exponent bits and 1 sign bit. The first 12 bits correspond to the first three digits of the hex format, and since the sign bit is 0, 404 corresponds to the exponential. The bias is given by $2^{e-1} - 1 = 1023$ and $0x404 = 1028$, we therefore have an exponential of 5. Assuming the leading 1 for normalized decimals:

$$1077936128 \Rightarrow 1 \cdot 2^5 = 32$$

14. Optimizing Compilers

The reality is that runtime performance is a lot more than asymptotic complexit! One can easily loose 100x in runtime or even more. To get the most out of our code we have to be familiar with the compiler and what it does.

`gcc` can optimize code by giving it the specific `-OX` -flag. Good choices for `gcc` are:

- O0 (no optimization!), -O2, -O3, -march=xxx (to specify the architecture), -m64 (to specify 64 bits)

One might also wants to try different compilers, `icc` is often faster than `gcc`.

14.1 Code motion and precomputation

The idea is to reduce the ***frequency*** with which a computation is performed. This is sometimes also called ***precomputation.*** This is something that the compiler does for us!

The right one is better than the left one:

```
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];
}
```

14.2 Strength reduction

Another thing that the compiler does is strength reduction. For example in the following code we can replace a multiplication by an addition:

```
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) {
    a[n*i + j] = b[j];
  }
}
```

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

The key idea is to replace ***costly operations*** with simpler ones. Another frequent example would be the replacement of divisions or mutliplications by shifts.

14.3 Common subexpressions

Following an example of sharing ***common subexpressions:***

```
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

```
int inj = i+n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

This reduces the amount of multiplications vom 3 (*Before*) to just 1 (*After*)! This is something that we can achieve with the compiler flag -O1, but keep in mind the compilar can eliminate some common subexpressions, but not all.

14.4 Optimization blocker: procedure calls

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

When we run the above code we observe that it's runtime is quadratic, i.e. in $O(n^2)$. But we only have one loop, so why is that?

The problem is that `strlen(s)` is called in every iteration and the prcoedure itself is in $O(n)$!

A better version therefore would be:

```
void lower2(char *s) {
  int i;
  int len = strlen(s);
  for(i = 0; i < len; i++) {
    if(s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

The compiler does not optimize this by himself! This is caused by the fact, that procedure calls can have side effect, that are not known to the compiler.

14.5 Optimization blocker: memory aliasing

Memory aliasing describes the situation when multiple memory references refer to the same location in memory.

We consider the following example where we sum up the rows of an $n \times n$ matrix into a vector of length n.

```
void sum_rows1(double *a. double *b, long n) {
  long i, j;
  for(i = 0; i < n; i++) {
    b[i] = 0;
    for(j = 0; j < n; j++) {
      b[i] += a[i*n + j];
    }
  }
}
```

```
# sum_rows1 inner loop
.L53:
  addsd   (%rcx), %xmm0
  addq    $8, %rcx
  decq    %rax
  movsd   %xmm0, (%rsi, %r8, 8)
  jne     .L53
```

The problem/observation here is that the code updates `b[i]` on every iteration. The compiler assumes possible side effect and therefore will not be optimizing this away.

How to remove aliasing

The key idea/solution is to use ***scalar replacement,*** i.e. copy array elements that are reused into *temporary variables.*

```
void sum_rows2(double *a, double *b, long n) {
  long i, j;
  for(i = 0; i < n; i++) {
    double val = 0;
    for(j = 0; j < n; j++) {
      val += a[i*n + j];
    }
    b[i] = val;
  }
}
```

```
.L66:
  addsd   (%rcx), %xmm0
  addq    $8, %rcx
  decq    %rax
  jne     .L66
```

14.6 Blocking and unrolling

We look at matrix multiplication $C = A * B + C$:

```
void mmm(double *a, double *b, double *c, int n} {
  int i, j, k;
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      for(int k = 0; k < n; k++)
        c[i*n+j] += a[i*n+k] * b[k*n+j];
}
```

We should be thinking about **data locality**. Data that gets reused should still be in the cache! In this example we have a lot of data that does not get reused.

Blocking (also called tiling) is partial unrolling of the loop. This assumes associativity, something the compiler will never do. Optimizing this we end up with this rather complicated code:

- Every array element a[…], b[…],c[…] used twice
- Now scalar replacement can be applied

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
         for (j = 0; j < n; j+=2)
             for (k = 0; k < n; k+=2)
                  <body>
}
```

```
<body>
c[i*n + j]         = a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j]
                     + c[i*n + j]
c[(i+1)*n + j]     = a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j]
                     + c[(i+1)*n + j]
c[i*n + (j+1)]     = a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)]
                     + c[i*n + (j+1)]
c[(i+1)*n + (j+1)] = a[(i+1)*n + k]*b[k*n + (j+1)]
                     + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)]
```

Moral: Help the compiler to help you

- Turn on optimization
- Remove obstacles to optimizer
- Do it yourself if necessary

15. Architecture and Optimization

The goal of this chapter is to understand how we can improve the performance of our code beyond the compiler optimizations from chapter 14.

During this chapter we are going to use the following structure and function for benchmarks:

```
/* data structure for vectors */
struct vect {
  size_t len;
  data_t *data;
};
```

```
/* retrieve vector element and store at val */
int get_vec_element(struct vec* v, size_t idx, dat
a_t *val) {
  if(idx >= v -> len) {
    return 0;
  }
  *val = v -> data[idx];
  return 1;
}
```

The actual benchmark we are going to run looks as follows:

```
void combine1(struct vec *v, data_t *dest) {
  long int i;
  *dest = IDENT;
  for(i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

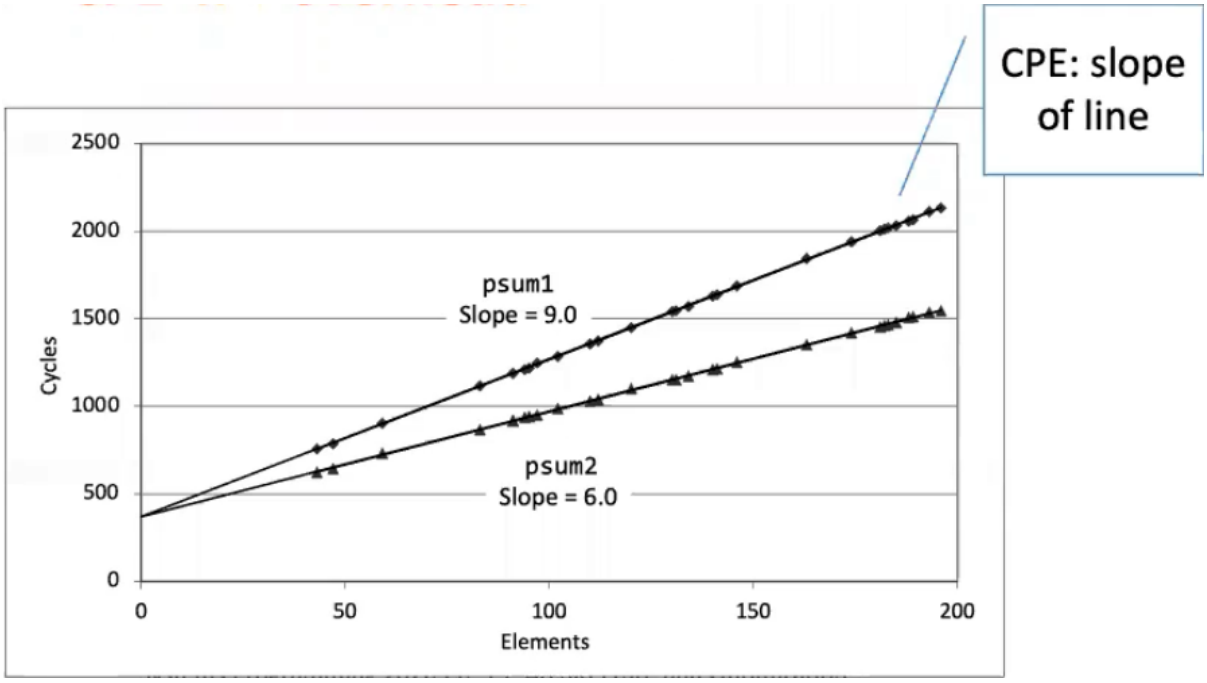
We are going to do two different benchmarks, i.e. we are going to run the benchmark with the following two pairs of `IDENT` and `OP` :

- 0 / + (*addition*)

- 1 / * (*multiplication*)

Cycle per Element (CPE)

The CPE is a way to express the performance of a program that operates on vectors or lists.



The ***execution time*** can then be given by $\mathrm{CPE} \cdot n + \mathrm{overhead}$, where CPE is equal to the slope of the graph and the $\mathrm{overhead}$ is equal to the y-intercept. The first benchmark yields the following CPE:

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Basic Optimizations

1. Move `vec_length` out of the loop
2. Avoid bounds check on each cycle
3. Accumulate in temporary

```
void combine4(struct vec *v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start();
  data_t t = IDENT;
  for(i = 0; i < length; i++) {
    t = t OP d[i];
  }
  *dest = t;
}
```

From ths optimization we get the following CPE:

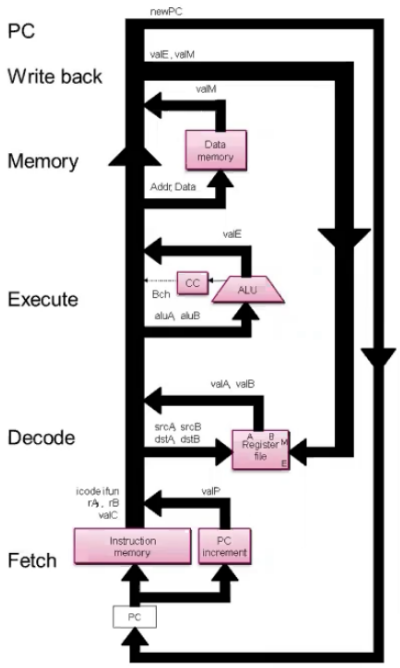
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

Already a massive improvement! But can we do better?

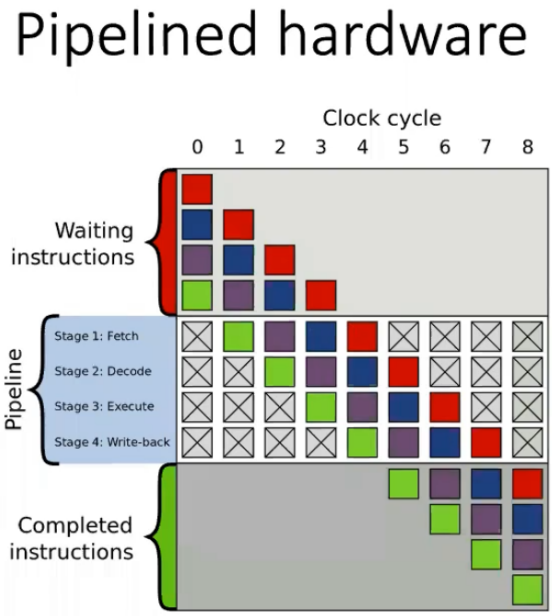
15.1 A bit about modern processor design

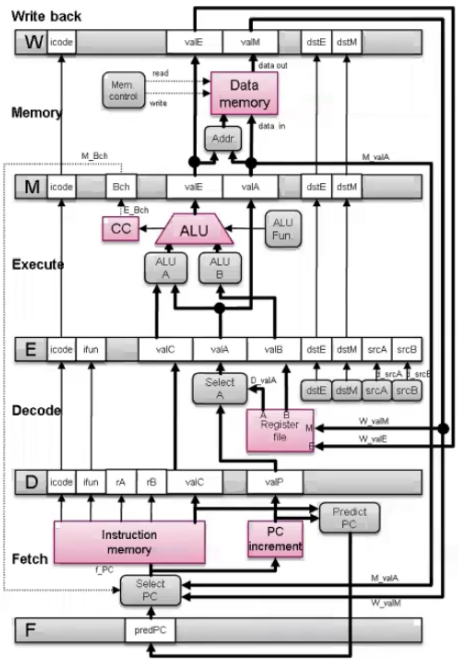
Sequential processor stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



Such a sequential processor is slow, this is due to the fact that a signal has to propagate through every stage in one cycle. The clock therefore can't go very fast and single hardware units are only active for a fraction of the cycle.



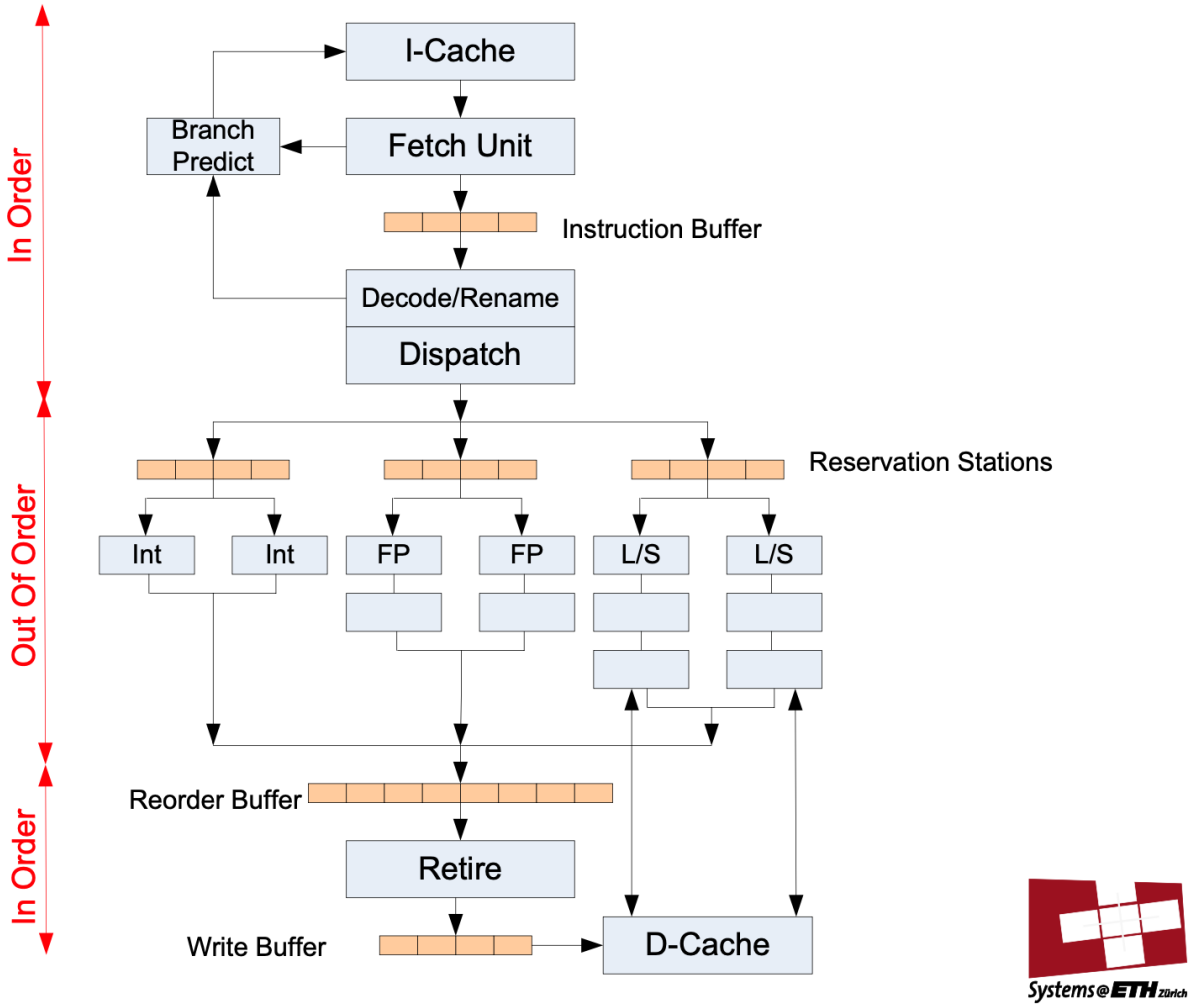


Pipelined processors are already a lot fast, but they introduce data / control hazards that need to be dealt with.

For processor we can estimate the performance with the ***program execution time,*** which is euqal to $IC \cdot CPI \cdot CCT$, where:

- IC is the instruction count
- CPI are the cycles per instruciton ($1/IPC$)
- CCT is the clock cylce time ($1/\text{Frequency}$)

Superscalar processor

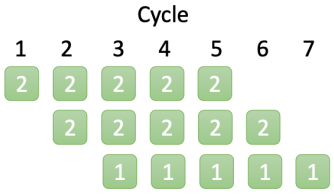


A ***superscalar processor*** can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically. This has the benefit, that without any programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have.

15.2 Superscalar processor performance

Latency versus Throughput

- Last slide: *latency* *cycles/issue*
  FP Multiply: 5 1



- How fast can 5 independent FP multiplies be executed?
  - `t1 = t2*t3; t4 = t5*t6; …`
- And 5 sequentially dependent FP multiplies?
  - `t1 = t2*t3; t4 = t5*t1; t6 = t7*t4; …`
- Major problem for fast execution: **Keep pipelines filled**

Recall: Data hazards

We distinguish the following types of data hazard:

- Read after Write (RAW)
- Write after Write (WAW)
- Write after Read (WAR)

The two last types of data hazards can be avoided with ***register renaming.***

What does this mean for our previous example?

We can state the following about ***performance bound:***

- Performance is *latency bound* when operations must execute sequentially.
- Performance is *throughput bound* when operations can execute in parallel. (typically faster)

Comparing our benchmarked CPE with an Intel Haswell as an example, we can see that we, in most cases, reached our latency bound.

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine4	1.27	3.01	3.01	5.01
Latency bound	**1.00**	**3.00**	**3.00**	**5.00**

This is the case because we execute all our computations in the for-loop sequentially. What happens if we do a 2x1 ***loop unrolling?***

```
void unroll2a_combine(struct vec *v, data_t *dest) {
  long length = vec_length(v);
  long limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x = IDENT;
```

```
  long i;
  /* combine 2 elements at a time */
  for(i = 0; i < limit; i+=2) {
    x = (x OP d[i]) OP d[i+1];
  }
  /* finish any remaining elements */
  for(; i < length; i++) {
    x = x OP d[i];
  }
  *dest = x;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	**1.00**	**3.00**	**3.00**	**5.00**

This only improves integer addition, since the other operations are still sequentialy dependent.

15.3 Reassociation

Lastly we take a lookt a 2x2 ***loop unrolling:***

```
void unroll22_combine(struct vec *v, data_t *dest) {
  long length = vec_length(v);
  long limit = length-1;
  data_t *d = get_vec_start(v);
  data_t x0 = IDENT;
  data_t x1 = IDENT;
  long i;
  /* combine 2 elements at a time */
  for(i = 0; i < limit; i+=2) {
    x0 = xo OP d[i];
    x1 = x1 OP d[i+1];
  }
  /* finish any remaining elements */
  for(; i < length; i++) {
    x0 = x0 OP d[i];
  }
  *dest = x0 OP x1;
}
```

This can change the result for floating point numbers. This helps to break up the sequential dependencies that we encountered bevor. These changes yields the following performance:

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	**0.81**	1.51	1.51	2.51
Latency Bound	**1.00**	**3.00**	**3.00**	**5.00**
Throughput Bound	**0.50**	**1.00**	**1.00**	**0.50**

15.4 Combining multiple accumulators and unrolling

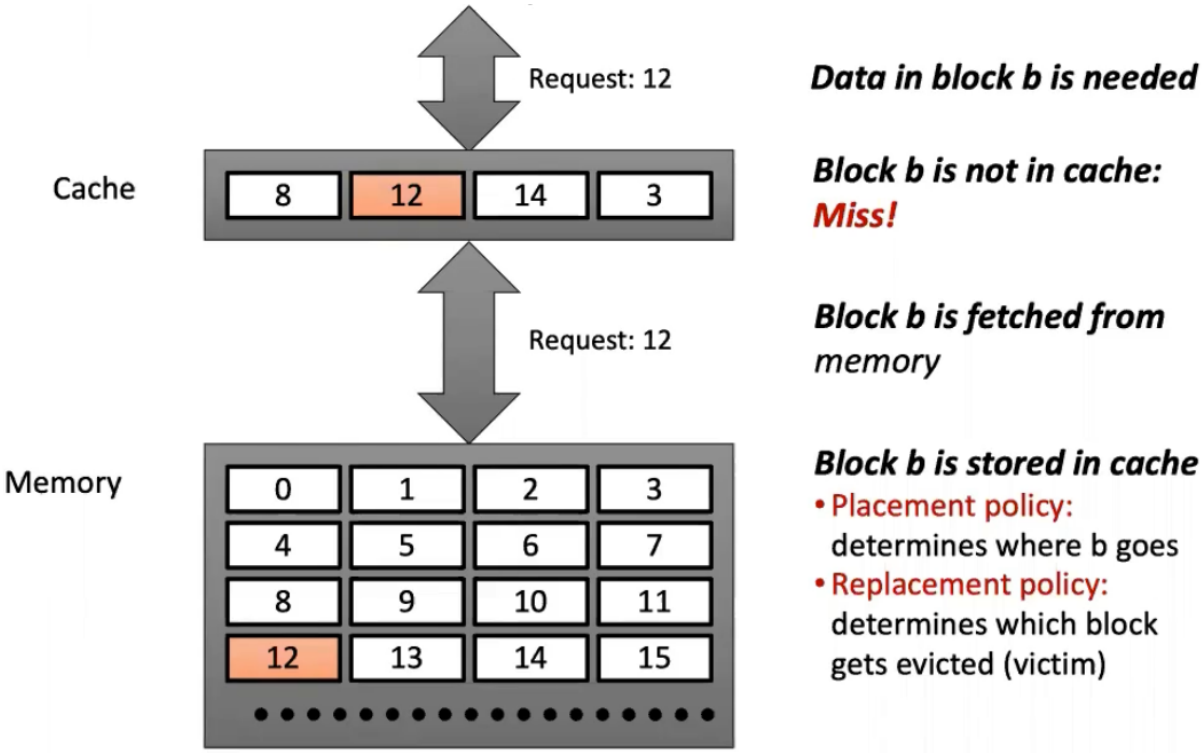
The idea is that we can unroll to any degree L and accumulate K results in parallel, where L has to be a multiple of K. When doing this we will encounter the limitations of diminishing return with growing overhead. With some trial and error we can find the sweet spot and find up to $42\times$ more performance compared to the original, unoptimized code. Using vector instructions we can imrove this number again by a drastic factor.

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	**0.50**	**3.00**	**3.00**	**5.00**
Throughput Bound	**0.50**	**1.00**	**1.00**	**0.50**
Vector Throughput Bound	**0.06**	**0.12**	**0.25**	**0.12**

16. Caches

Historically the time it takes to load data from memory to the processor has always been a bottleneck. The solution for this problem is caches.

General cache concept



Cache performance metrics

We have three different metrics for cache performance:

- Miss rate: Fraction of memory references not found in cache $1 - \text{hit rate}$
- Hit time: Time to deliver a line in the cache to the processor. Typical numbers are:
  - 1-2 cycles for L1
  - 5-20 cycles for L2
- Miss penalty: Additional time required because of a miss, typically 50-200 cycles for main memory.

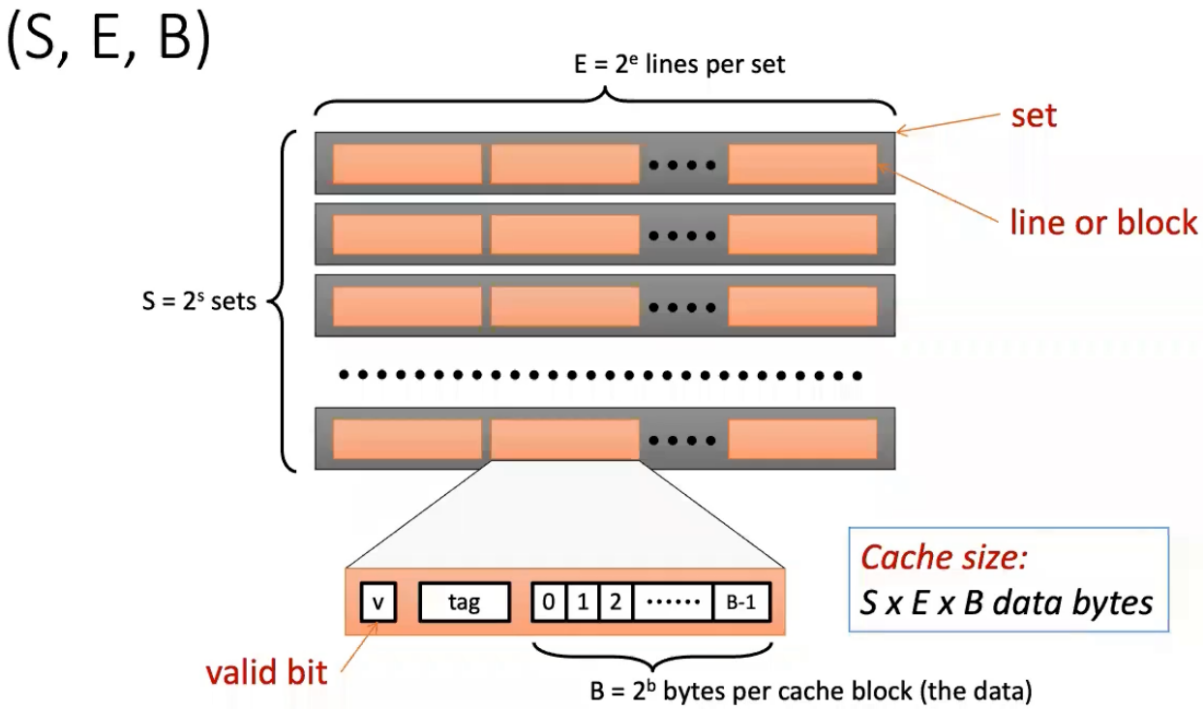
When looking at these numbers we see that there is a huge difference between hit and miss rate, 99% hit rate is twice as good as 97%!

Types of cache misses

We differ between different types of cache misses:

- Cold (compulsory) miss: occurs on first access to a block
- Conflict miss: caches limit the placement to a small subset of available slots
- Capacity miss: occurs if the set of active cache blocks (i.e. the working set) is larger than the cache
- Coherency miss: occure in multiprocessor systems - *see later in the course*

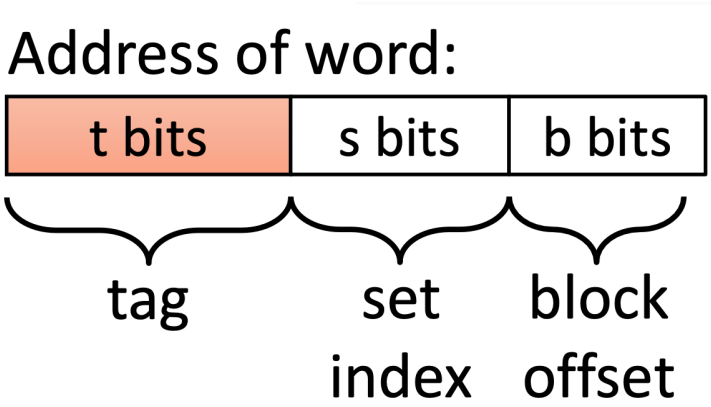
16.1 Cache organization



16.2 Cache reads

For a cache read we performe the following steps:

1. Locate the set
2. Check if any line in the set has a matching tag
3. If yes and the line is valid we have a cache hit
4. Locate the datat starting at the offset



There are different types of caches:

- Direct mapped cache (E = 1): only one line per set
- k-way set-associavtike cache (E = k): k lines per set

16.3 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	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

Cache writes

What should we do on a write-hit? We have two solutions:

- Write-through: write immediately to memory
- Write-back: introduce a dirty bit, write to memory when the line is evicted

Similar on a write-miss, we can either directly write to memory (no-write-allocate) or load the block into the cache (write-allocate).

16.4 Cache optimizations

Programs tend to use data and instructions with addresses near or equal to those they have used recently. We can therefore make use of the following two things:

- ***Temporal locality:*** Recently referenced items are likely to be referenced again in the near future.
- ***Spatial locality:*** Items with nearby addresses tend to be referenced close together in time.

16.5 Blocking

Blocking and loop unrolling, as seen in previous chapters, can help to reduce chase misse, by exploiting temporal and spatial locality.

17. Exceptions

In general, processors only do one thing:

- From startup to shutdown, a CPU simply reads and executes a sequence of instructions, one at a time.
- This sequence of instructions is the CPU's **control flow**.

Up to now we have only seen two mechanisms for changing the control flow:

- Jumps and branches
- Call and return

We now introduce exceptional control flow. An ***exception*** is a transfer of control to the OS in response to some event (i.e. change in processor state).

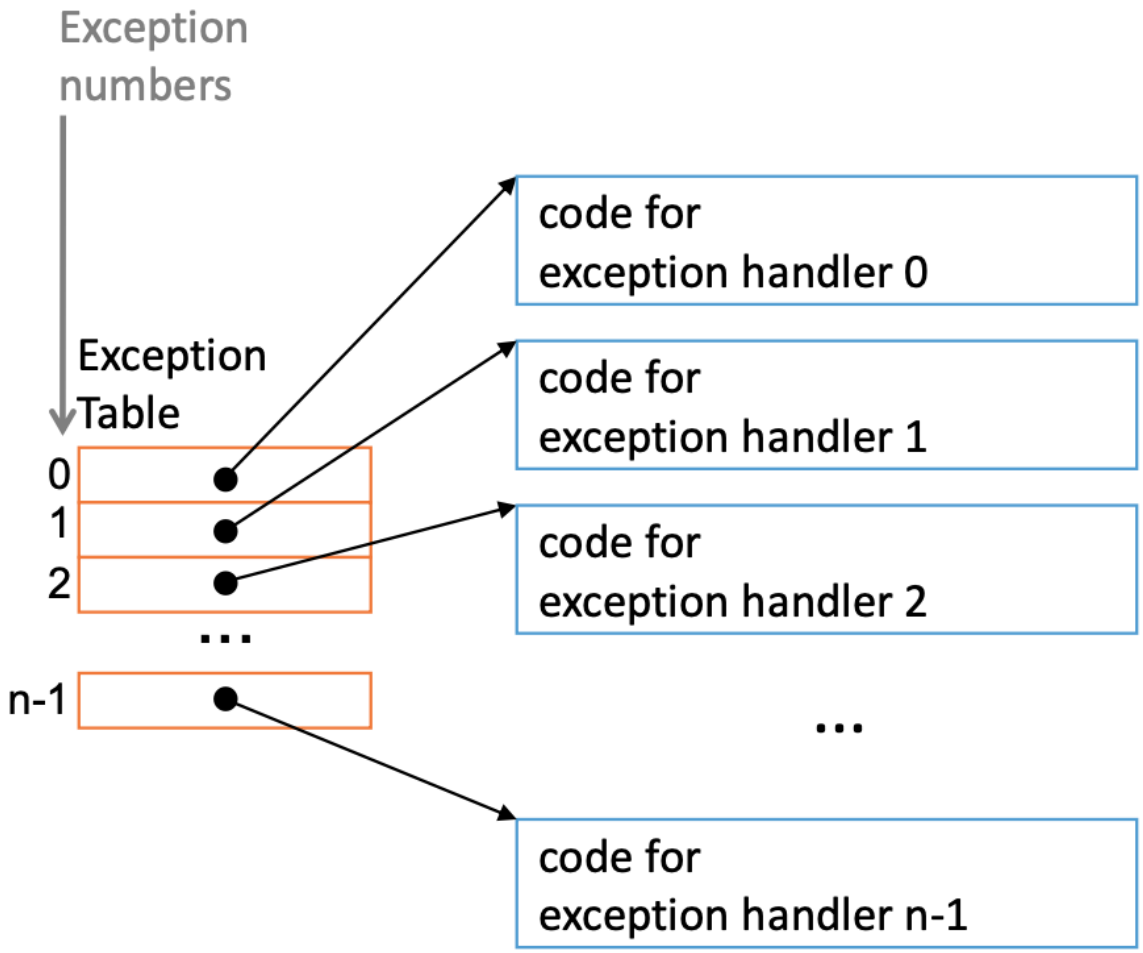
We can classify exceptions into a few different categories:

- A ***synchronous*** exception occurs as a result of executing an instruction.
- An ***asynchronous*** exception occurs as a result of events that are 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

17.1 Exception vectors and kernel mode

At boot time, the OS allocates and initializes the ***exception table.*** Each type of event has a unique exception number k with which we can index into the exception table (aka. interrupt vector).



Exceptions cause a switch to the kernel mode (supervisor mode). The kernel mode has some special things, it can access system state and has many more privileges.

17.2 Synchronous exceptions

We categorise synchronous exceptions into the following categories:

- ***Traps:*** Intentional; Example: system calls, breakpoint traps; Returns control to "next" instruction
- ***Faults:*** Unintentional but possibly recoverable; Example: page faults, protection faults, floating point exceptions; Either re-executes faulting instruction or aborts
- ***Aborts:*** Unintentional and unrecoverable; Examples: parity error, machine check; Aborts the current program

17.3 Asynchronous exceptions

Asynchronous exceptions (*interrupts*) are caused by events ***external*** to the processor. Common examples are:

- I/O interrupts
- Hard reset interrupt
- Soft reset interrupt

Basic x86 interrupts

In x86 we have two interrupt pins:

- INTR: interrupt request
- NMI: non-maskable interrupt

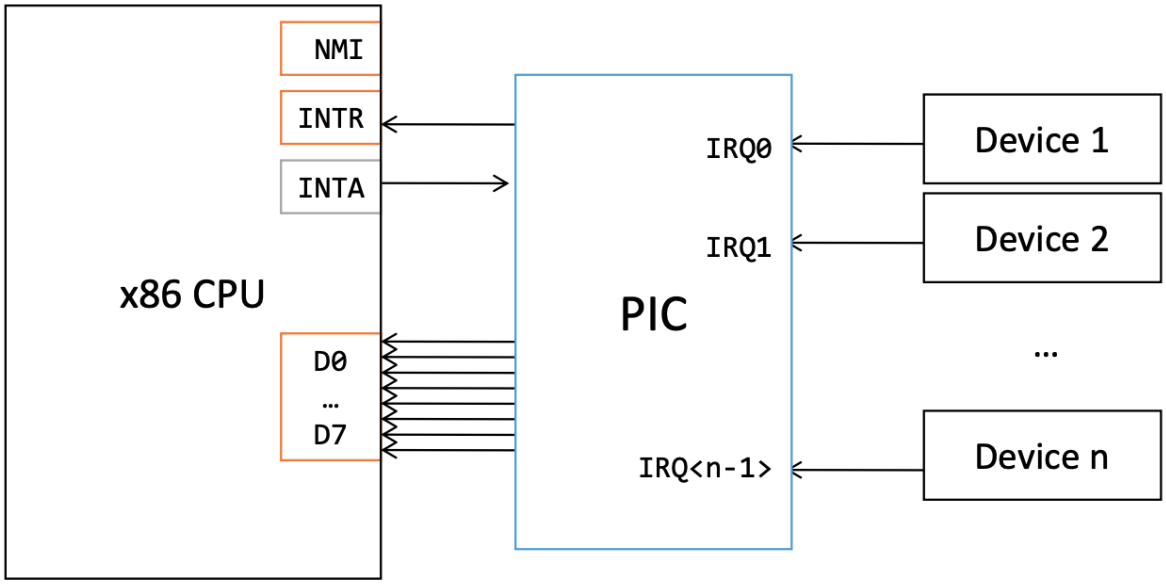
If ***NMI*** is asserted, the CPU completes the current instruction and then issues the exception. This cannot be disabled by the processor.

If ***INTR*** is asserted, an interrupt request is issued. It can be disabled by the CPU. If it is enabled, we complete the current instruction, and then:

1. CPU acknowledges via the ITNA pin.
2. The interrupt vector is then supplied to the CPU via the data bus.
3. The CPU issues the exception from the vector.

17.4 Interrupt controllers

To handle multiple errors at the same time from different devices, we use so called ***programmable interrupt controllers (PIC):***



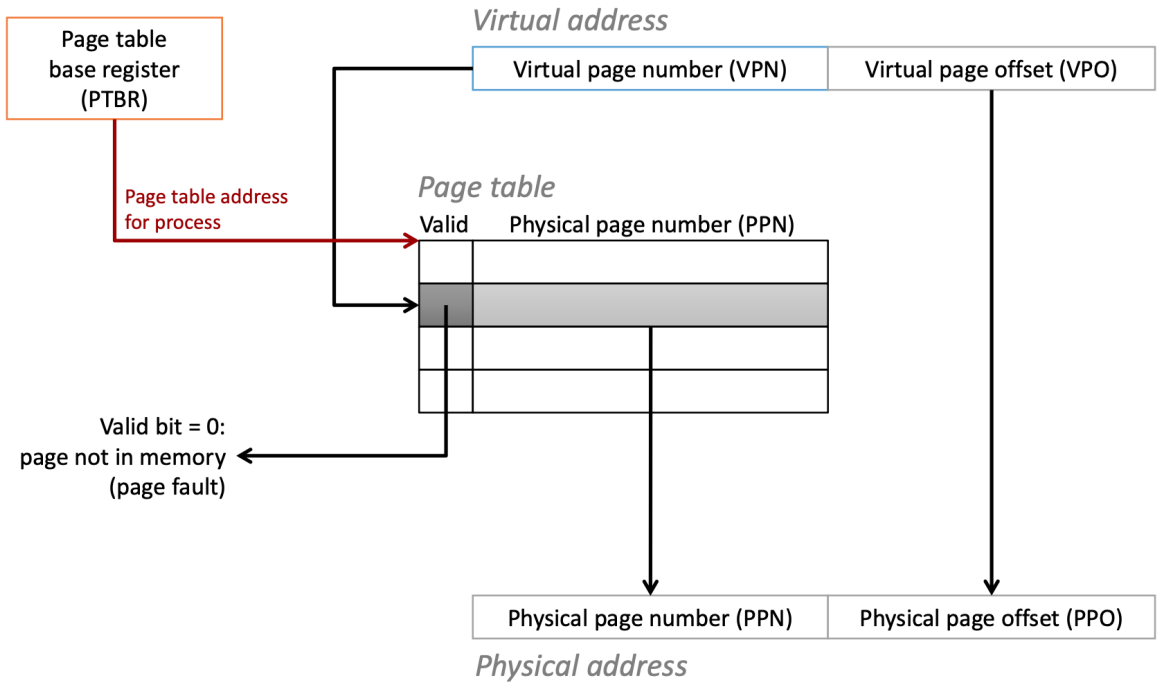
The PIC does:

- Map physical interrupt pins to interrupt vectos
- Buffer simultaneous interrupts
- Prioritize interrupts
- Selectively masks any individual device's interrupts

18. Virtual Memory

18.1 Recap: Address Translation

Address translation with a page table



18.2 Uses of virtual memory

The main reasons for using virtual memory are:

- ***Efficient use*** of limited main memory, only active part of virtual address space is kept in memory
- ***Simplifies*** memory management for programmers (Each process gets the same full, private linear address space)
- ***Isolates*** address spaces (One process can't interfere with another's memory)

Problems of virtual memory

Problem 1: Limited physical memory capacity

- 64-bit addresses allow for *16 Exabyte* of storage, but physical memory is usually only a few Gigabytes big.

Why does this work? Because of locality. At any point in time, programs tend to acess a set of active virtual pages called the working set.

Problem 2: Memory management

- Each process has a `stack` , `heap` , `.text` , etc. and the computer needs to somehow manage what data goes where in the physical main memory.

Each process has its own virtual address space which is viewed as a simple linear array. A mapping function then scatters those addresses through physical memory.

One can **share code** and data among different processes by simply mapping virtual pages to the same physical page.

Problem 3: Protection

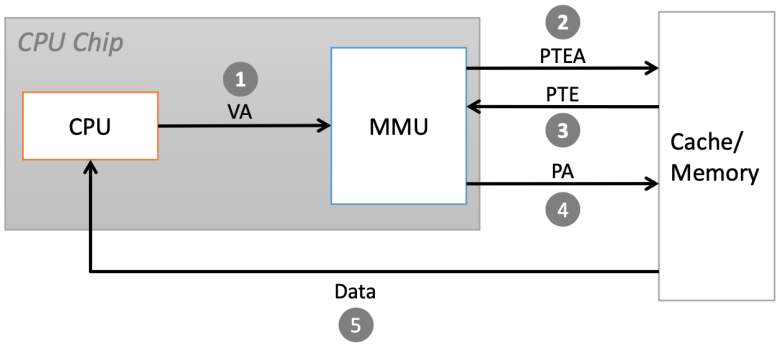
- Different processes might access the same physical page.

We extend the page table entries with permission bits. The page fault handler checks these before remapping and if it's violated, sends a segmentation fault.

Process j:	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

18.3 The address translation process

Page hit



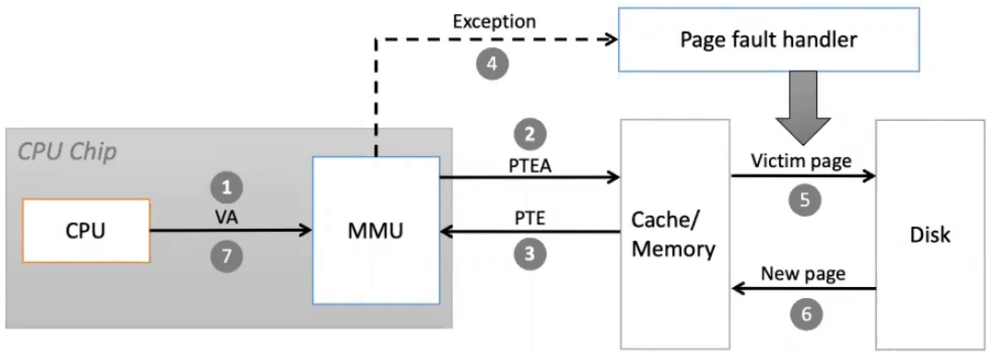
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Page fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim page to evict (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

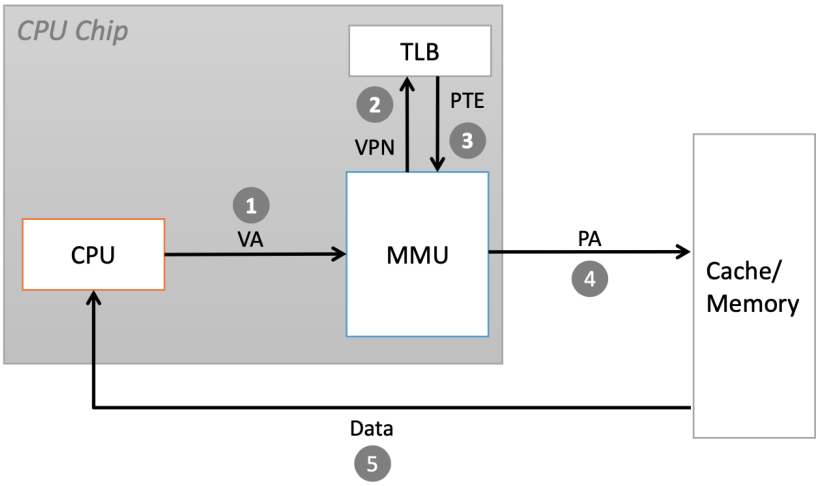
7) Handler returns to original process, restarting faulting instruction

18.4 Translation lookaside buffers

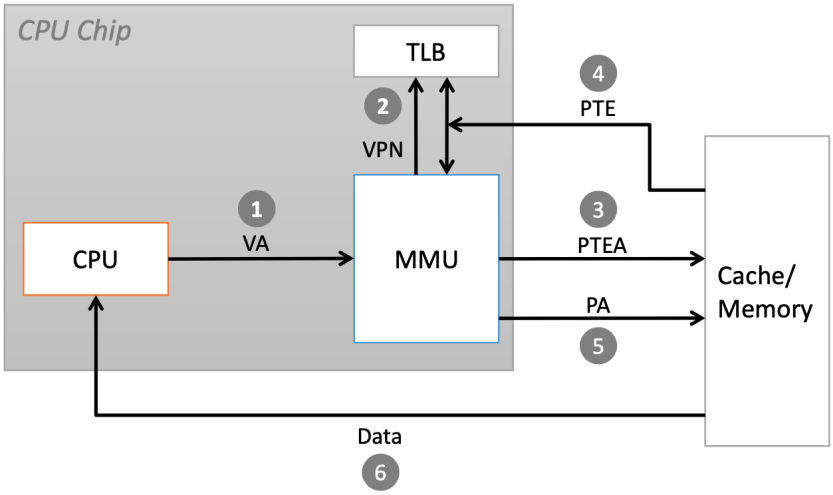
Page table entries (PTEs) are cached in L1 like any other memory word. This means that PTEs might be evicted by other data references. The solution to this problem is the ***translation lookaside buffer (TLB):***

- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

TLB hit



TLB miss



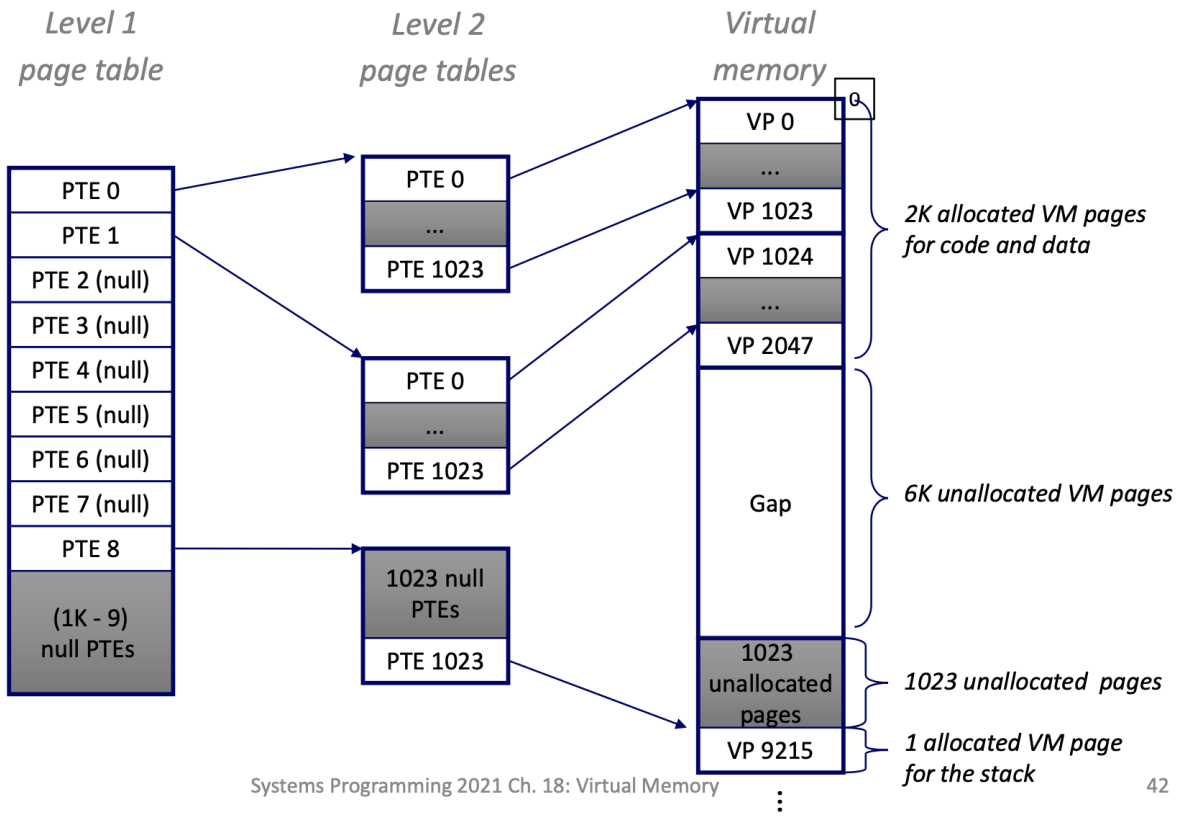
18.6 Multi-level page tables

Given 4KB page size, 48-bit address space and 8-byte page table entry (PTE). How big of a page table do we need?

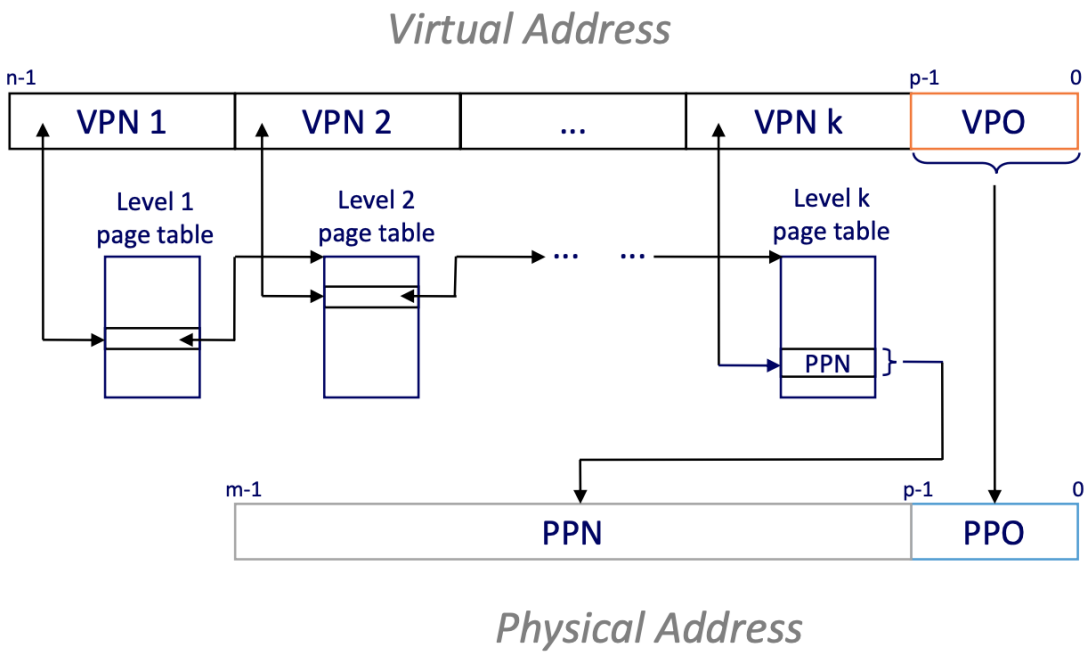
Linear page table size

We have a 48-bit (2^{48}) address space and each page has a size of 4KB (2^{12}), we therefore need $2^{48}/2^{12}$ page table entries. Since each entry is 8 bytes (2^3) bytes big, we need $2^{48}/2^{12} \cdot 2^3 = 2^{39}$ bytes of space which equals to $512\ \text{GB}$. This is a problem! To circumvent this, we introduce multi-level page tables.

2-level page table hierarchy



k-lebel page table hierarchy



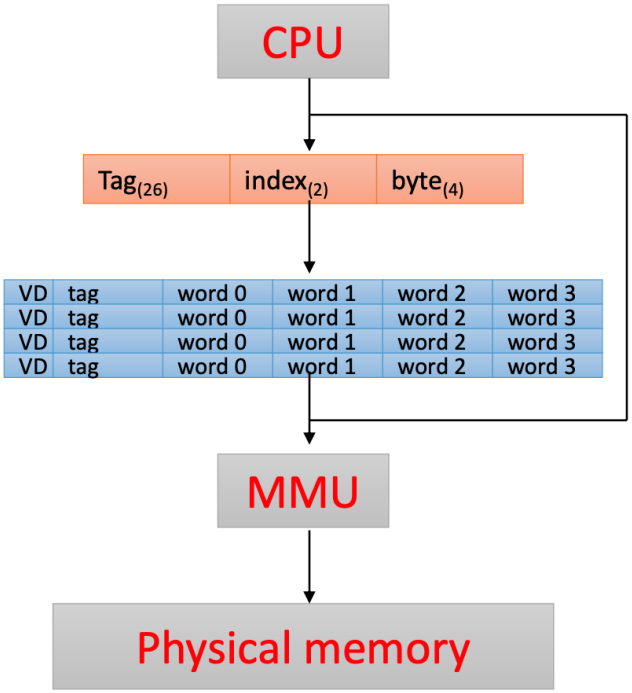
We notice that the VPO is the same as the PPO, if we now use this address to access the cache and the CI is part of the PPO, we can already start the process before the address translation is finished. But this depends on the addressing scheme of the cache.

18.9 Caches revisited

We differ between four ***cache addressing schemes:***

- Virtually indexed, virtually tagged (*VV*)
- Virtually indexed, physically tagged (*VP*)
- Physically indexed, virtually tagged (*PV*) (not covered)
- Physically indexed, physically tagged (*PP*)

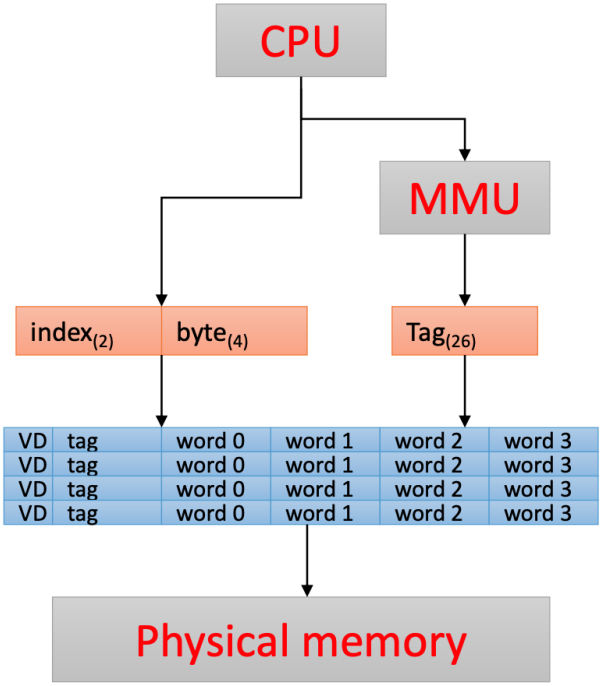
Virtually indexed, virtually tagged



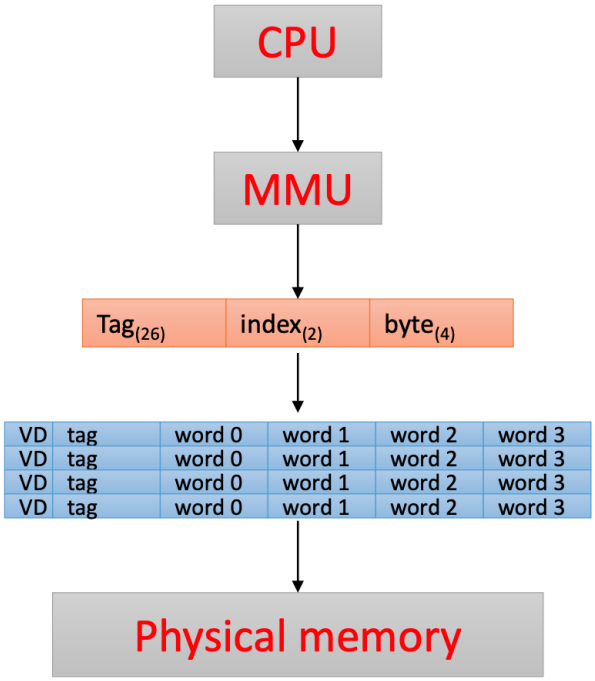
We search the cache and do the address translation parallel. Virtually indexed cache has the following issues:

- ***Homonyms:*** same names for different data
- VA used for indexing is context dependent, the same VA refers to different PAs
- Homonyms can be prevented by flushing the cache on a context switch, forcing non-overlapping address-space layouts or tagging the VA with the address-space ID

Virtually indexed, physically tagged



Physically indexed, physically tagged



- Only uses physical addresses
- Translation must complete before cache access can start
- Typically used off-core
- Slowest access time

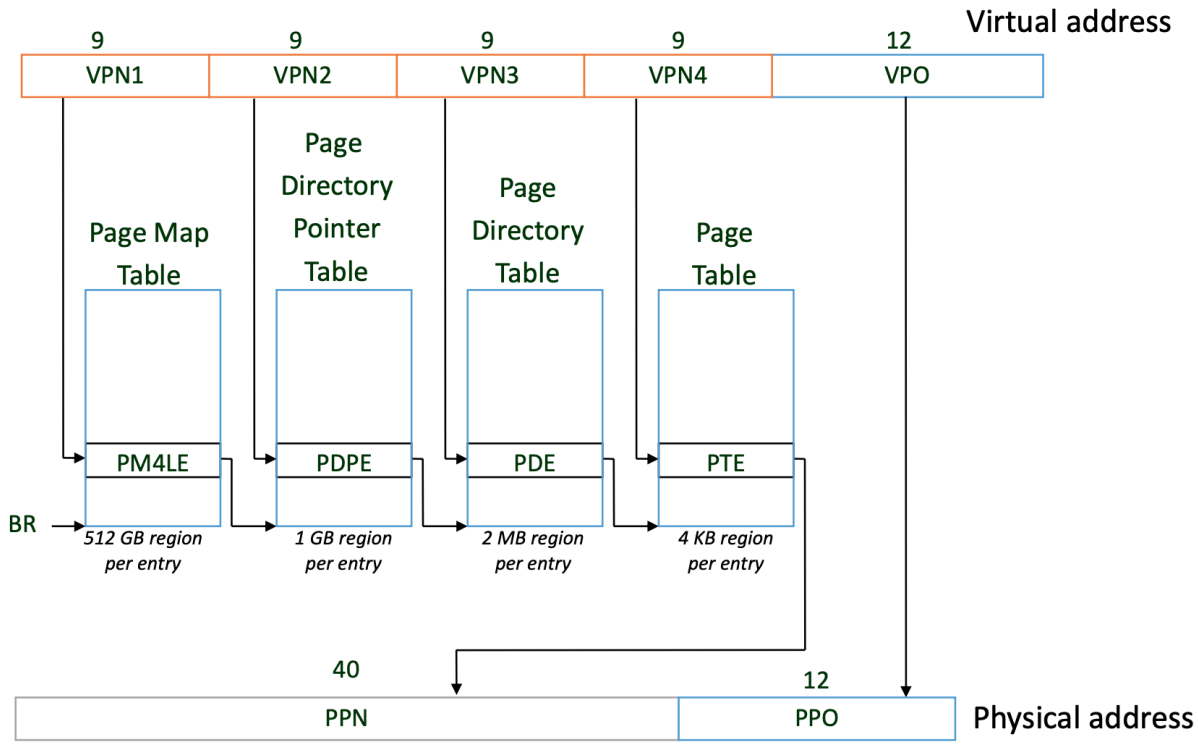
- Easy to manage (no homonyms or synonyms)

Write buffers

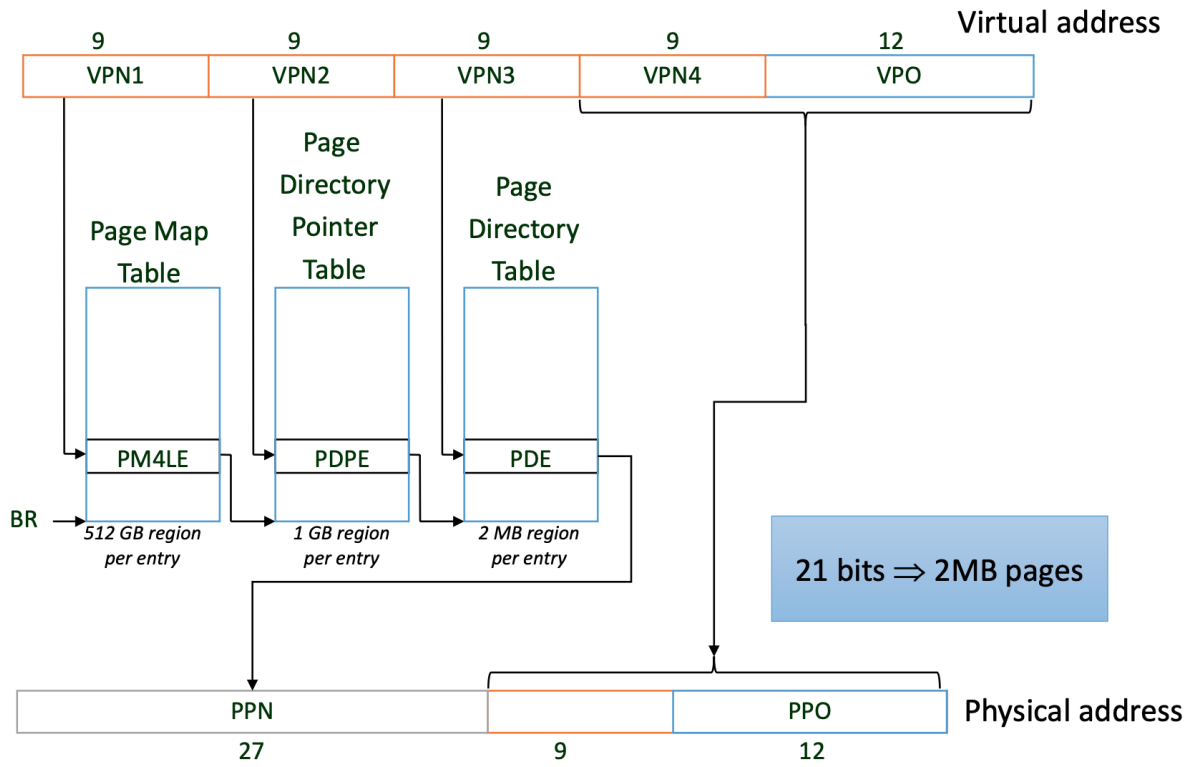
We want to avoid stalling the CPU for memory writes. We introduce a FIFO queue to buffer writes. When we want to read a address that has a yet completed write, we can directly read out of the write buffer.

18.10 Large pages

For 4KB pages we have seen the following x86-64 setting:



But what if we need larger pages? (We can save some memory if we have larger data chunks and it's easier on the TLB). We define a ***large page*** as a page of size 2MB. We can reuse the "old" hardware to support those pages in the following way:



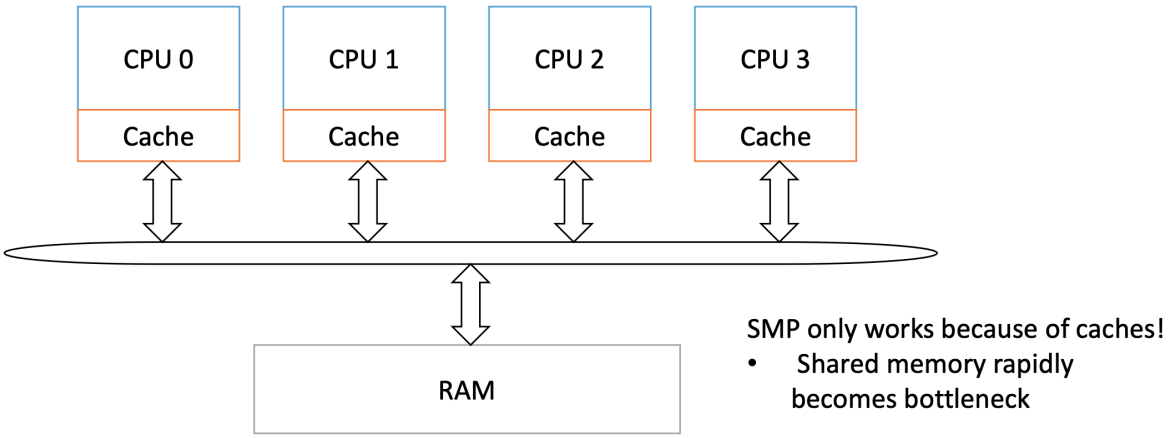
The same idea applies to ***huge pages*** of size 1GB.

19. Multiprocessing and Multicore

Symmetric multiprocessing (SMP)

Due to the power wall + ILP wall + memory wall, we are at the end of the road for serial hardware. A solution for this are multicore processors.

The first idea of multiprocessing is to attach multiple processors to the same bus (but each processor still has its own cache), so calles shared memory multiprocessors:



19.1 Consistency and Coherence

With several processors, memory can change under a cache. This fact leads to two important concepts:

- ***Coherency:*** Values in caches all match each other, processors all see a coherent view of memory.
- ***Consistency:*** The order in which changes to memory are seen by different processors.

Cache coherency

Most CPU cores on a modern machine are ***cache coherent.*** They behave as if all cores are accessing a single memory array.

The big advantage of this is the ease of programming, i.e. shared-memory programming models work. But they are complex to implement at memory is slower as a result.

Memory consistency

Terminology

- *Program order:* order in which a program on a processor apperas to issue reads and writes
  - Refers only to local reads and writes
- *Visibility order:* oder in which all reads and writes are seen by one or more processors
  - Refers to all operations in the machine

19.2 Sequential consistency

1. Operations from a single processor appear to all other processors in *program order*
2. Every processor's *visibility order* is the same interleaving of all the program orders

This has the following requirements:

- Each processor issues memory operations in program order
- RAM totally orders all operations
- Memory operations are globally atomic

19.3 Cache coherence with snooping

Cache snoops/listens into on reads and writes from other processors. If a line is valid in local cache, we initiate a remote write to line, i.e. invalidate local line.

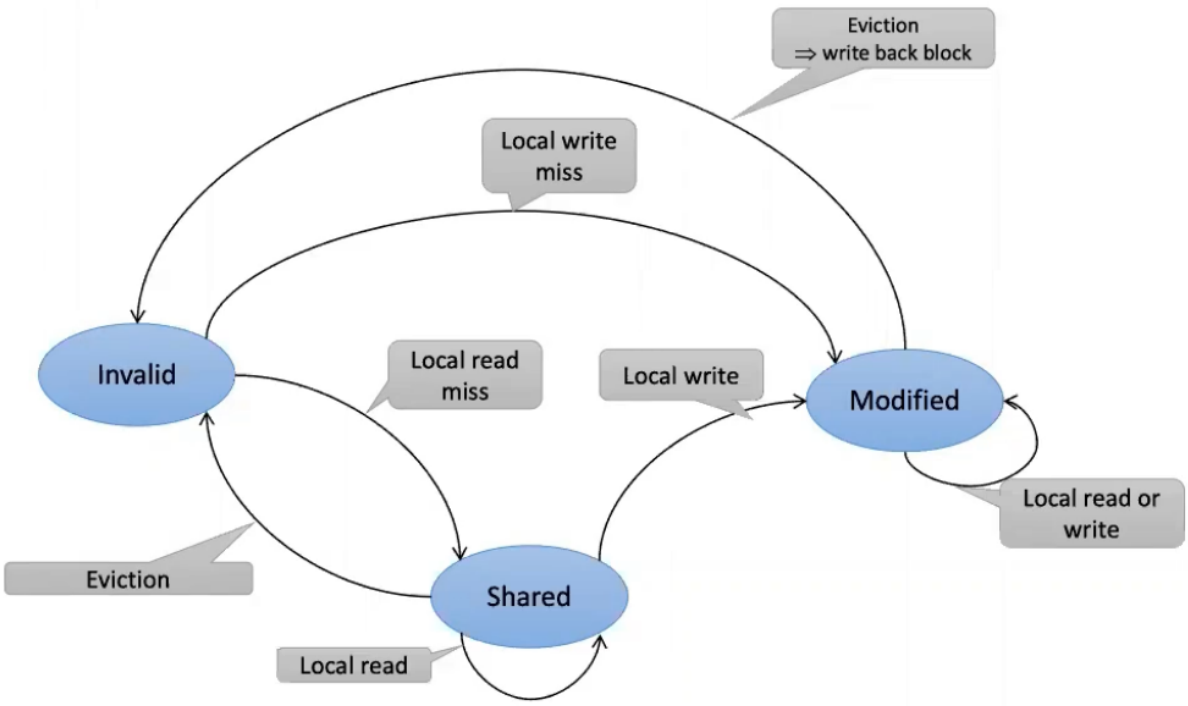
Cache lines can now be dirty/modified. This requires a ***cache coherency protocol.***

The simplest form is ***MSI:***

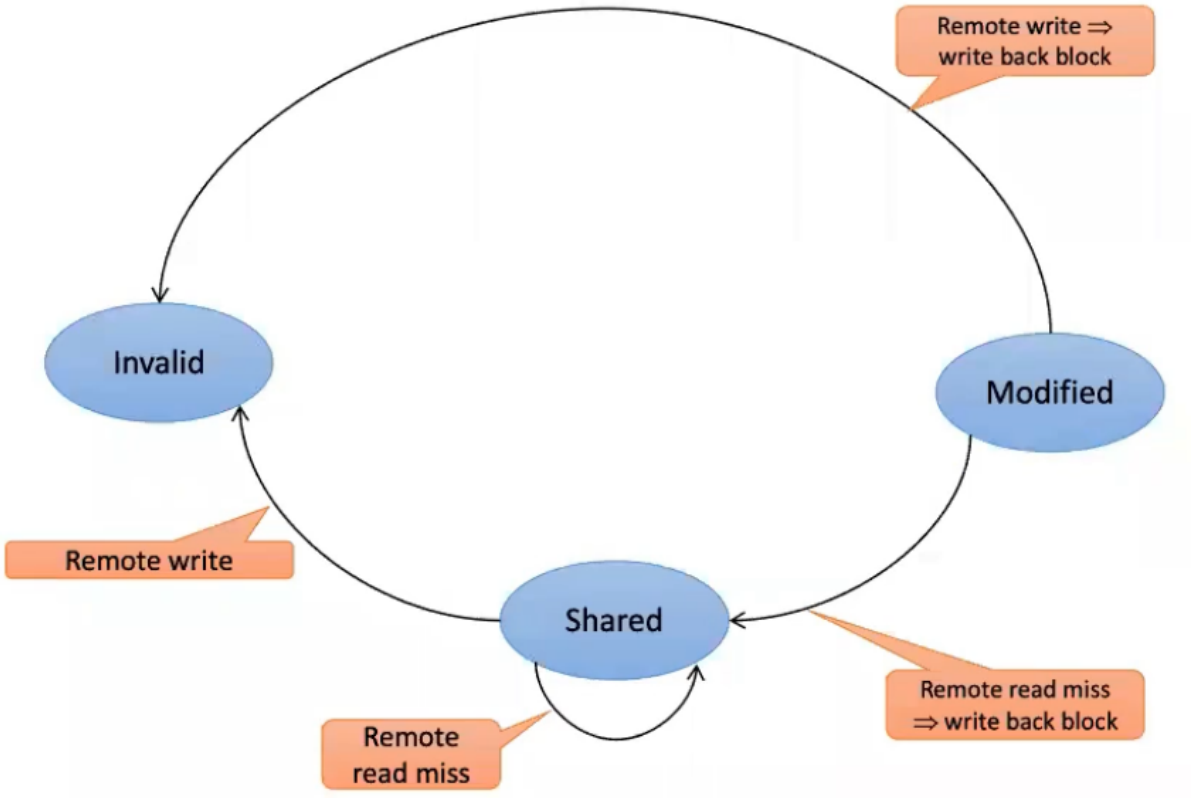
- Each line has 3 states: Modified, Shared, Invalid
- A line can only be dirty in one cache

The cache logic must respond to processor reads and writes aswell as remote bus reads and writes.

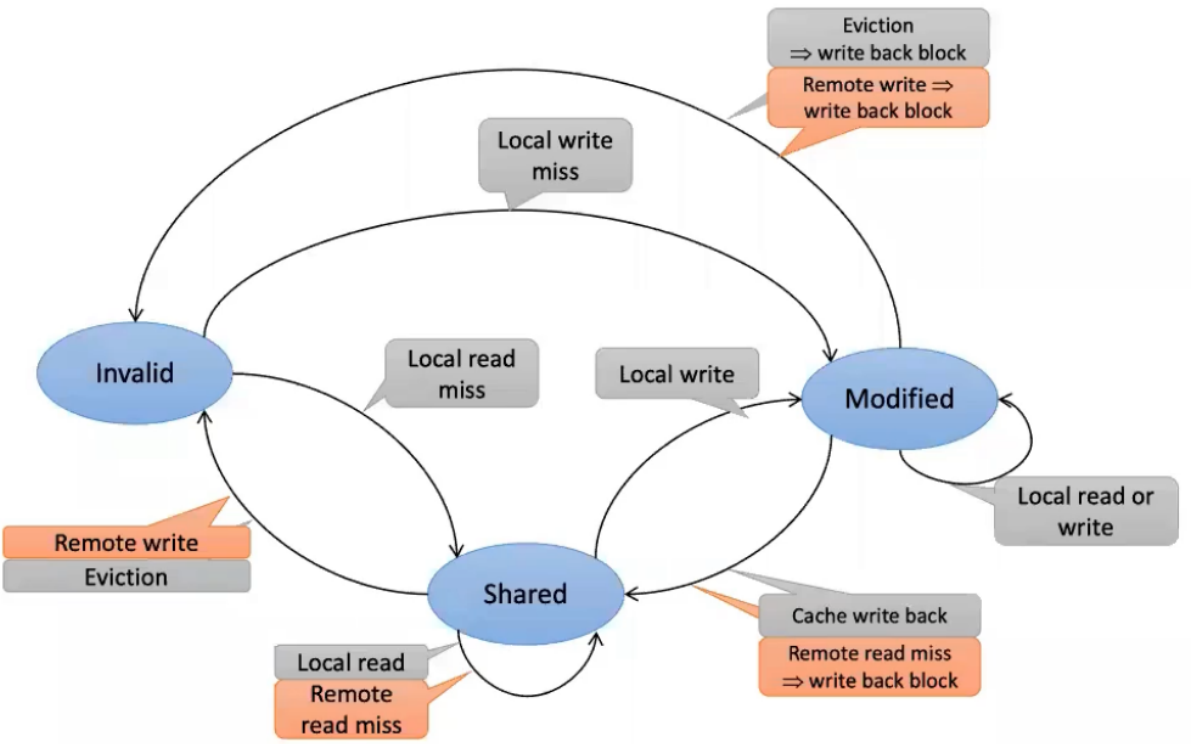
MSI state machine: local processor transitions



MSI state machine: remote snooped transitions



MSI state machine: all transitions



MSI assumes that we can distinguish remote processor read and writes misses, this is not allways the case! Another problem is that we have unnecessary bus messages.

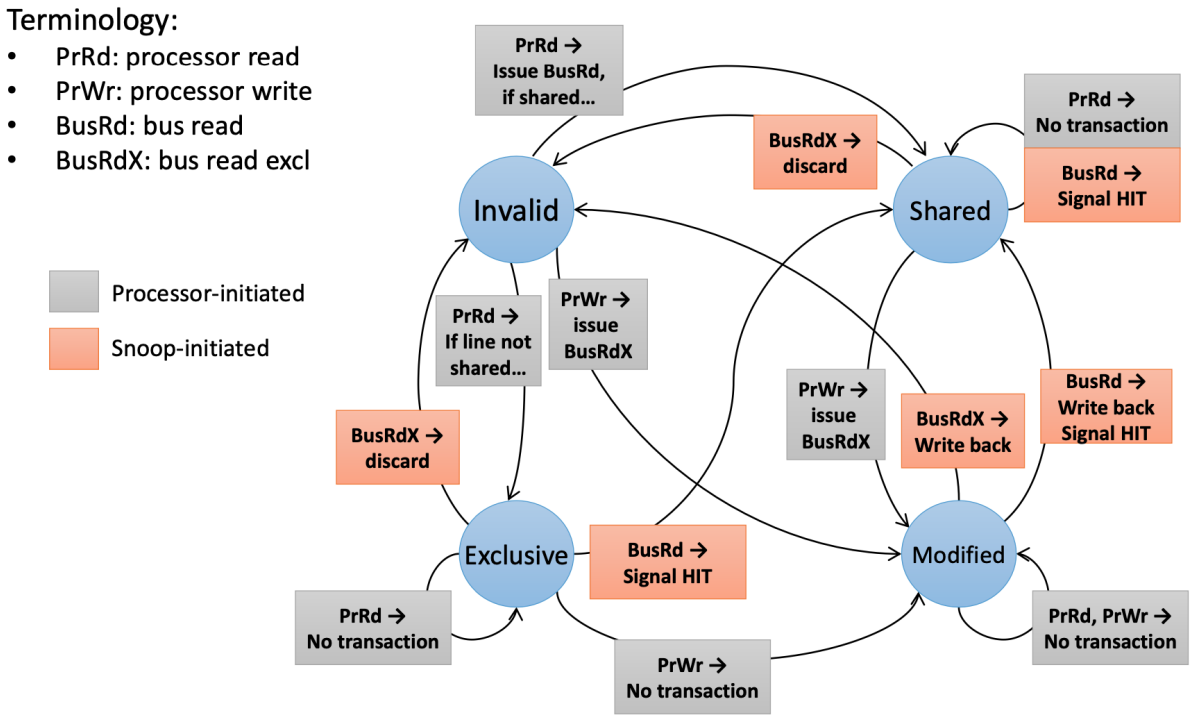
19.4 The MESI cache coherence protocol

Compared to the MSI protocol, the MESI protocol adds a new line state:

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

It basically allows us to be sure that one processor is the only owner of a cache block. We introduced the HIT signal, informing another processor that its cache block is present in local cache. The protocol furthermore adds a new bus signal ***RdX,*** which corresponds to a *read exclusive.*

MESI state machine



Intel and AMD have their own similar protocols MESIF / MOESI that have another state, but these were mentioned only shortly.

19.5 Relaxing sequential consistency

There are many different ways of relaxing sequential consistency. Some of them are:

- Write-to-read: later reads can bypass earlier writes
- Write-to-write: later writes can bypass earlier writes
- Break write atomicity (no single visibility order)
- Weak ordering: no implicit order guaranteed at all

Following that we need some kind of synchronization at chosen points in our program. Therefore x86 provides ***explicit synchronization instructions:***

- `lfence` (load fence)
- `sfence` (store fence)
- `mfence` (memory fence)

Processor Consistency

One of the most common way of relaxation is the processor consistency. It is standard for 64-bit x86 processors, sometimes also called *Total Store Ordering (TSO).* It implements the *write-to-read relaxation:*

- All processors see writes from one processor in the order they were issued
- Processors can se different interleavings of writes from different processors

(u,v) = (0,0) is possible in PC
- a2 read bypasses a1 write
- b2 read bypasses b1 write

CPU A		CPU B	
a_1:	*p = 1;	b_1:	*q = 1;
a_2:	u = *q	b_2:	v = *p;

There are many more consistency models for different processors. Even portable languages like Java have to define their own memory model.

19.6 Barriers and fences

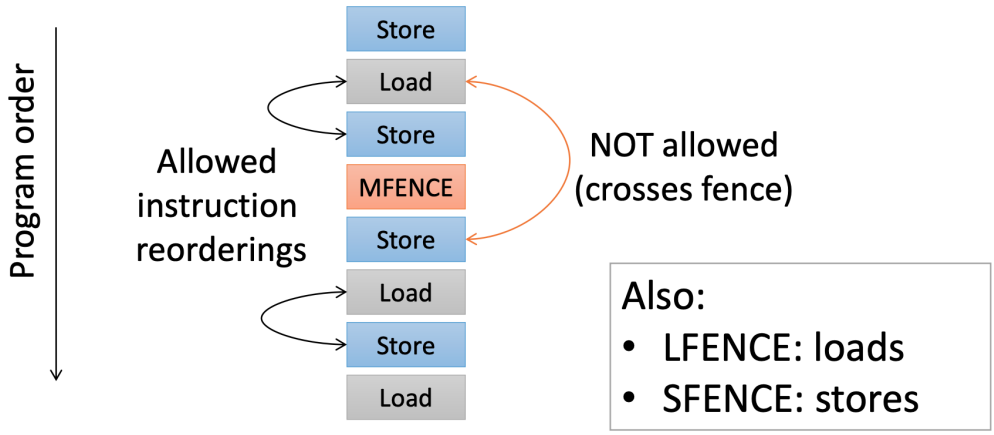
It generally holds that the weaker the consistency model is, the faster & cheaper it goes into hardware. We have seen that the visibility order is essential for correct functioning of some algorithms, but is really difficult to guarantee with many compilers and memory models.

A solution to this problem are so-called ***barriers*** (also named *fences*):

- ***Compiler barriers:*** prevents the compiler from reordering state ments
- ***Memory barriers:*** prevents the CPU from reordering instructions

Memory barriers on x86

One x86 we have the `mfence` instruction which prevents the CPU from reordering any loads or stores past it:



19.7 Multicore synchronization: Test-and-Set

There are two ways to synchronize across processors:

- ***Atomic operations*** on shared memory
  - Test-and-set, compare-and-swap
- ***Interprocessor interrupts*** (IPIs)
  - Invoke interrupt handler on remote CPU
  - Very slow (500+ cycles), often avoided except in the OS

Test-And-Set (TAS)

TAS is one of the simplest non-trivial atomic operations. We can for example use TAS to acquire a mutex:

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

This is also called a ***spinlock:*** We keep trying in a tight loop until we can acquire it. Releasing a spinlock is simple:

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

It turns out that TAS can be very expensive. The memory must be locked while a long operation occurs. Also it must do a read, followed by a write, while no one else can access the memory.

Test and Test-And-Set

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

This way, most of the spinning cycles are replaced with simple reads (and not both read and write).

19.8 Compare-and-Swap

```
CAS(location, old, new) atomically {
  1. Load location into value
  2. If value == "old" then store "new" to location
  3. Return value
}
```

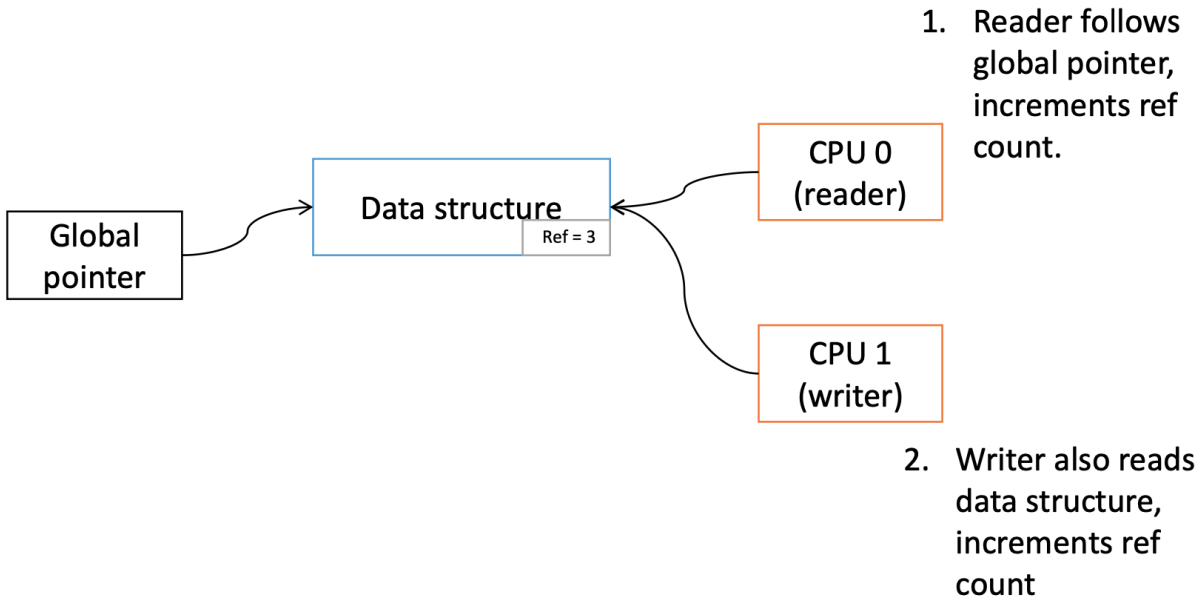
Some interesting features of CAS are:

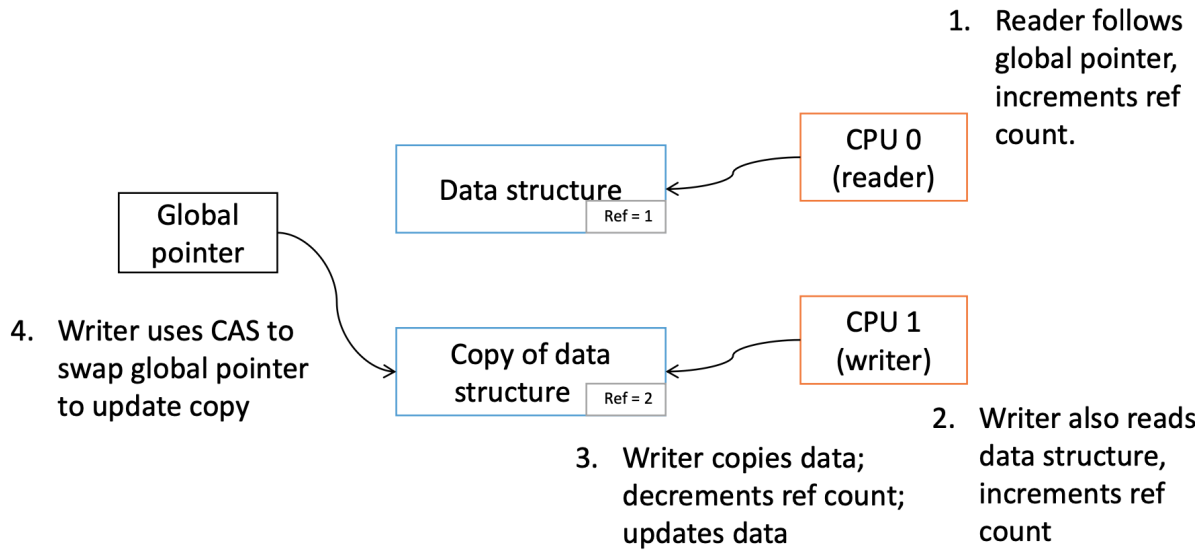
- It's theoretically more powerful than TAS, FAA, etc.
- It can implement almost all wait-free data structures

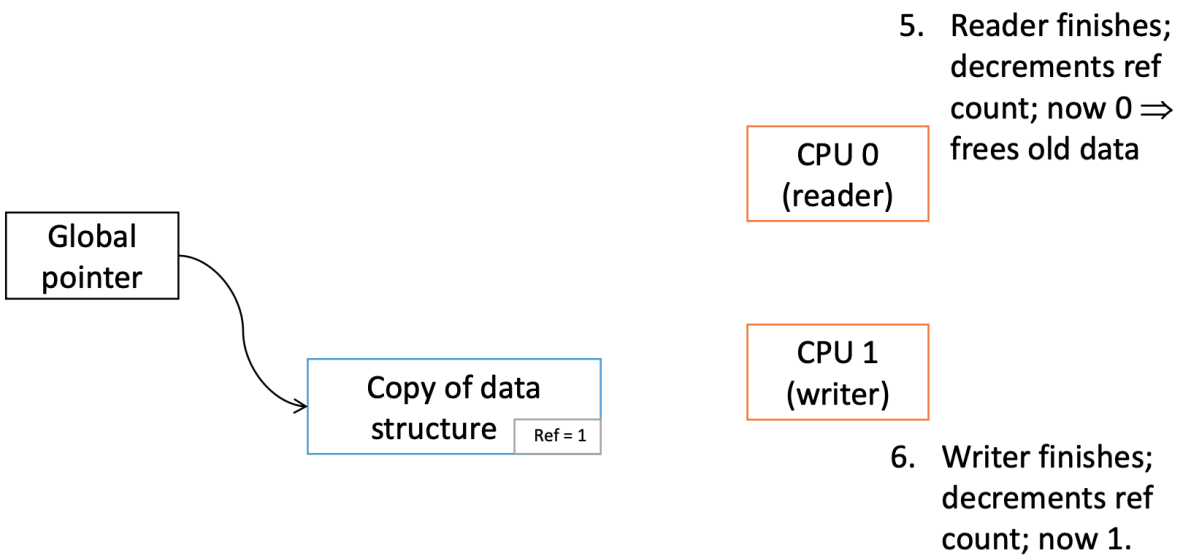
CAS follows a general structure:

- readers all read the same datastructure
- Writers take a copy, modify it, then write back the copy
- Old version is deleted when all the readers are finished

CAS for lock-free update







The ABA problem

1. CPU A reads value as x
2. CPU B writes y to value
3. CPU B writes x to value
4. CPU A reads value as x → concludes that nothing has changed

A simple solution to this problem is to make sure that the value always changes, for example by including a counter.

19.9 Simultaneous multithreading

Cache-coherent SMP still has memory as a bottleneck, all accesses to main memory stall the processor.

SMT or Hyperthreading

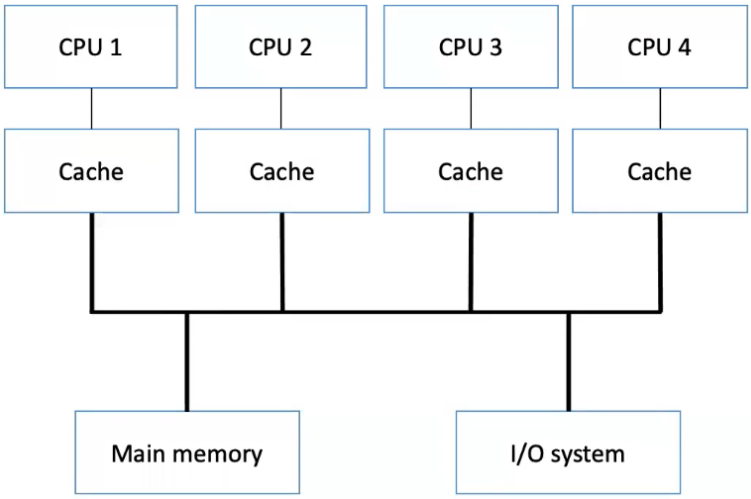
We can label instructions in hardware with their thread id. This way we have multiple independent instruction streams. We differ between two types of multi-threading:

- Fine-grained multithreading: select from threads on a per-instruction basis
- Coarse-grained multithreading: switch between threads on memory stall

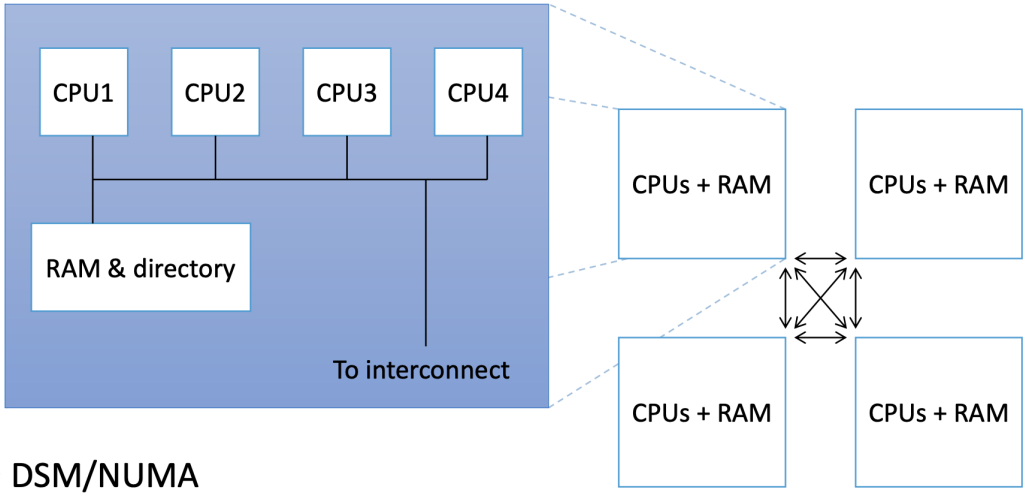
Hyperthreading can be unpredictable in term of performance improvements. We can either have performance improvements or a decrease in performance!

19.10 Non-Uniform Memory Access (NUMA)

We have seen the following ***SMP architecture:***

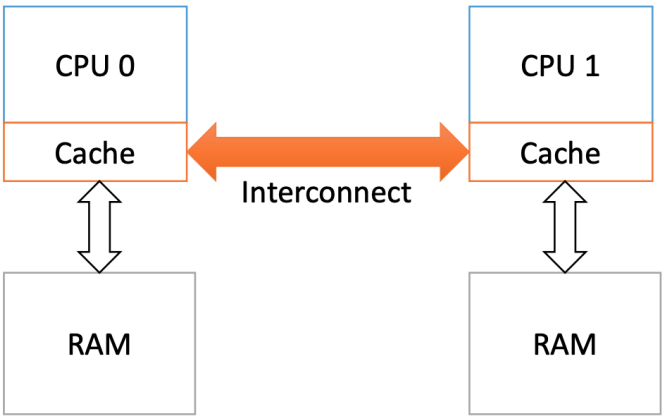


From what we have seen until now, we can end up with many coherency messages on the bus. In this part we want to look at solutions for this. We can now introduce a ***distributed memory architecture,*** where RAM is shared between smaller groups of CPUs and we can send messages in these local groups:



- DSM/NUMA
- Message-passing, eg clusters
- Could scale to 100s or 1000s of cores

Non-Uniform Memory Access (NUMA)



NUMA does:

- Remove bottleneck: Multiple, independent memory banks, processors have independent paths to memory
- Interconnect is not a bus anymore, it's a network link: carries messages between ***nodes*** (usually processor sockets)
- All memory is globally addressable

19.11 NUMA cache coherence

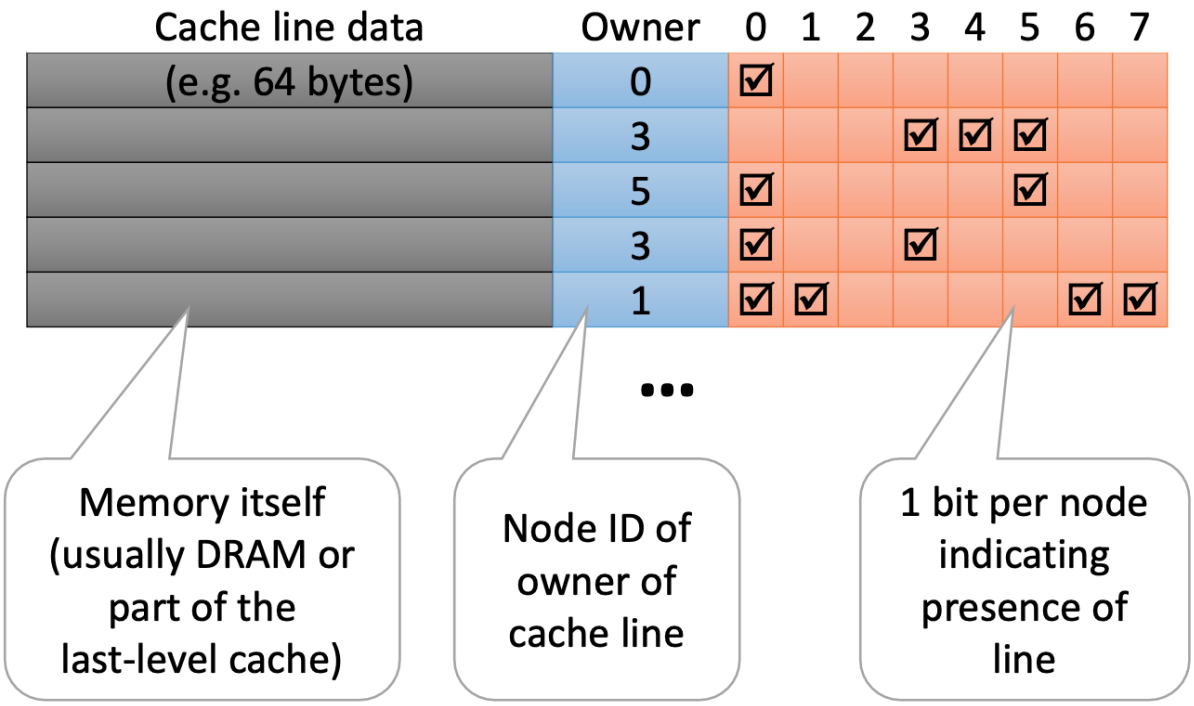
One problem of NUMA is that one cannot snoop one the bus anymore, since it isn't a bus.

Solution 1: Bus emulation

- Similar to snooping
- Each node sends a message to all other nodes ("Read exclusive")
- Waits for a reply from all other nodes

Solution 2: Cache directory

The idea is to augment each node's local memory with a cache directory. Then each "home node" maintains the set of nodes that may have a line. Now if we modify data in a cache line, we only need to notify the locations where this cache line is present.



19.13 Optimization example: MSC locks

MSC locks are possibly the best known locking system for multiprocessors. They have *excellent cache properties:*

- Only spin on local data
- Only one processor wakes up on `release()`

A general problem for locks is that a cache line containing a lock is a *hot spot.* It is continuously invalidated as every processor tries to acquire it. The solution to this problem is, that when acquiring, a processor enqueues itself on a list of waiting processors, and spins only on its own entry in the list. When releasing, only the next processor is awakened.

```
struct qnode {
   struct qnode *next;
   int locked;
typedef struct qnode *lock_t;

void acquire( lock_t *lock, struct qnode *local) {
    local->next = NULL;
    struct qnode *prev = XCHG(lock, local);
    if (prev) { // queue was non-empty
      local->locked = 1;
      prev->next = local;
      while (local->locked); // spin
  }
}

void release (lock_t *lock, struct qnode *local) {
  if (local->next == NULL) {
    if ( CAS(lock, local, NULL) ) { return; }
    while (local->next == NULL); // spin
  }
  local->next->locked = 0;
}
```

20. Devices

Specifically, to an OS programmer, a ***device*** is:

- A piece of hardware visible from software
- Occupies some location on a bus
- Has a set of registers
- Is a source of interrupts
- Something that may initiate a Direct Memory Access transfer

20.1 Device Registers

Registers

A CPU can *load* and *store* device registers. Registers are ***memory mapped***, i.e. they appear as memory locations and can therefore be accessed using loads and stores (`movb`, `movw`, `movl`, `mowq`). There are also ***I/O instructions***. Those are different 16 bit address spaces for older I/O devices.

It is important to note that register are ***not memory,*** they don't behave like RAM:

- Register contents may change without writes from CPU (status words, incoming data)
- Writes to registers are used to trigger actions (sending data, resetting state machines)

We don’t what that read / writes get “optimized” away. Often, registers are ***sets of bitfields.*** The definitions of those fields is usually given in a datasheet. When all data passes through the CPU, i.e. it explicitly reads and writes, we call it **Programmed/IO**, this is not very efficient!

20.2 Dealing with caches

Reads can't come from the cache, because if the register value changes, the cache becomes inconsistent. Write-back caches and write buffers cause problems since you don't know when the line will be written.

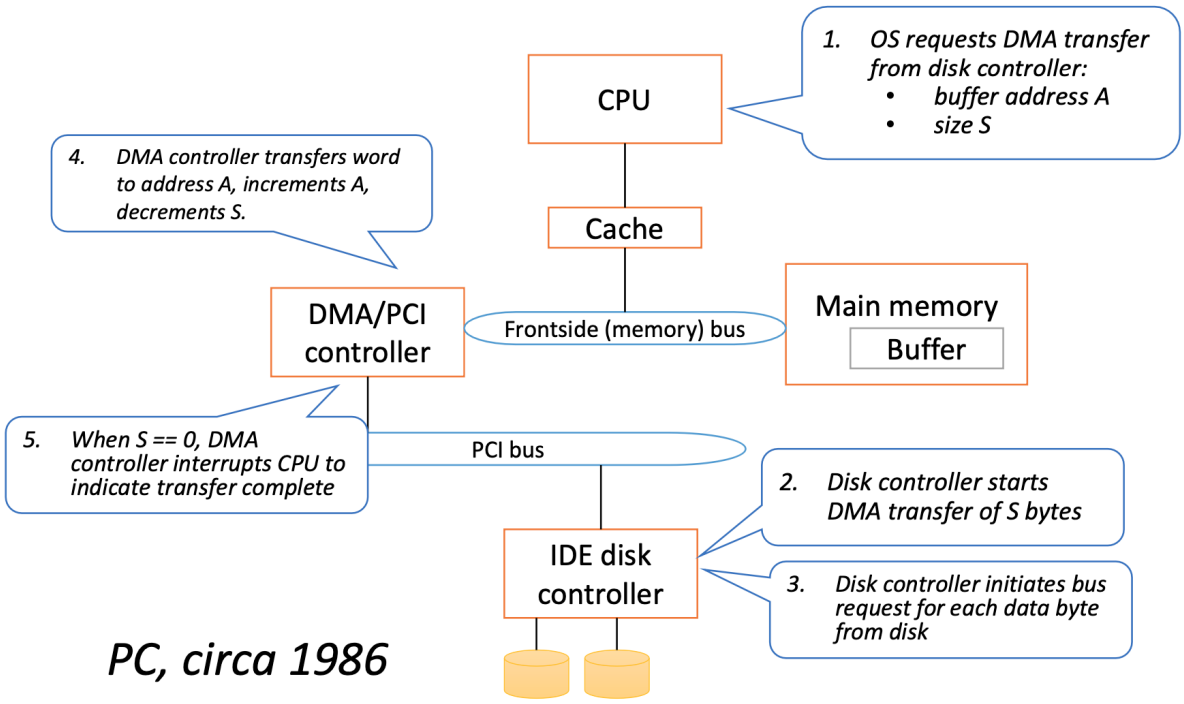
Therefore, device register access ***must bypass the cache.*** This is handled in the MMU where the corresponding PTEs have a "no cache" flag.

Other challenges are:

- How to avoid polling all the time, i.e. how does the CPU know when the device is ready? → interrupts
- How to avoid the CPU from copying all the data? → direct memory access (DMA)
- Where do these register locations come from? → dicoverable buses (PCI)

20.3 Direct Memory Access

DMA bypasses the CPU to transfer data directly between I/O device and memory. This is important, since the amount of data that is transfered can be huge and they should be transfered fast. This requires a *DMA Controller,* those are generally built-in these days.



Pros:

- ***Decoupling*** of data transfer from processing
- CPU doesn't need to copy data to / from the device
- Doesn't pollute CPU cache
- Higher performance: CPU and device work in parallel

Cons:

- Higher setup overhead for very small transfers

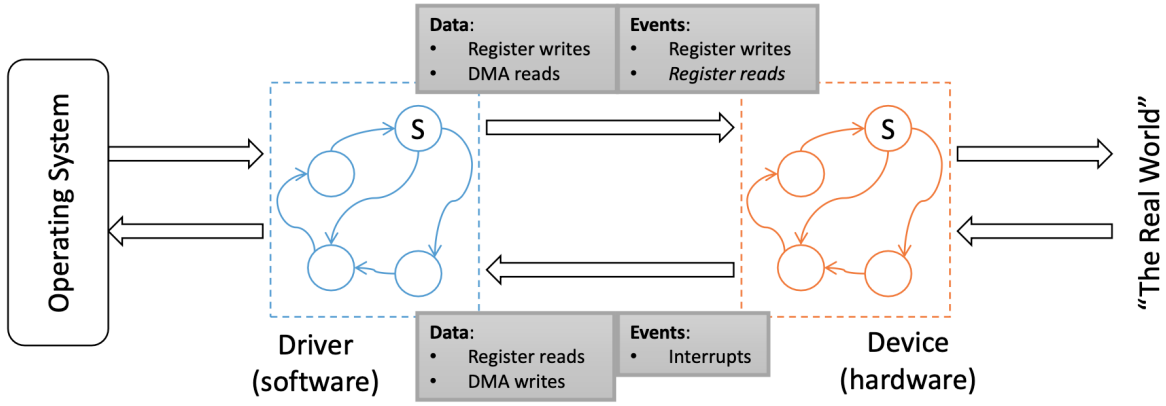
DMA and Caches

DMA means that memory becomes inconsistent with CPU caches. There are several options to fix this problem:

- CPU can map DMA buffers that are non-cacheable
- Cache can snoop DMAC bus transactions → does not scale well
- OS can explicitly flush / invalidate cache regions

20.4 Device drivers

The basic model for ***device drivers*** looks as follows:



- Diver and device are both *state machines*
- Data must be transferred between them
- Events signal a state transition

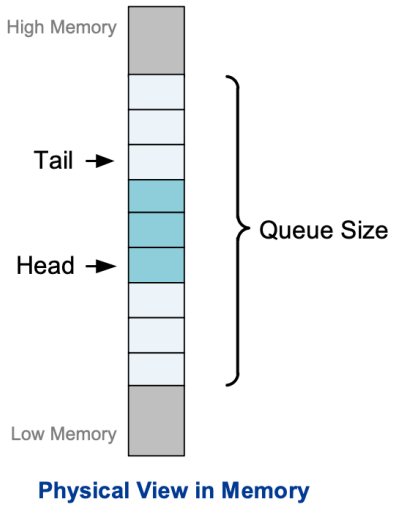
Device and CPU communication

There are four main types of device-CPU communication:

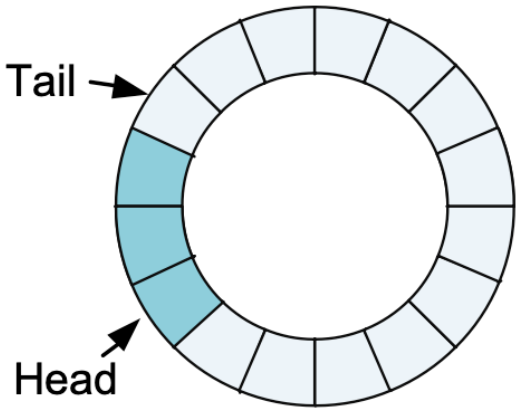
- Writing a device register: CPU → device, synchronous
- Reading a device register: CPU ↔ device, synchronous
- Device requests interrupt: Device → CPU, synchronous
- Shared memory, asynchronous
  - CPU writes to memory, DMA reads
  - DMA writes to memory, CPU reads

20.5 Buffer rings and descriptor rings

For actual data transmition, we can use descriptor rings (think of circular data structure). Normaly there is one ring for receiving data and one for sending data.

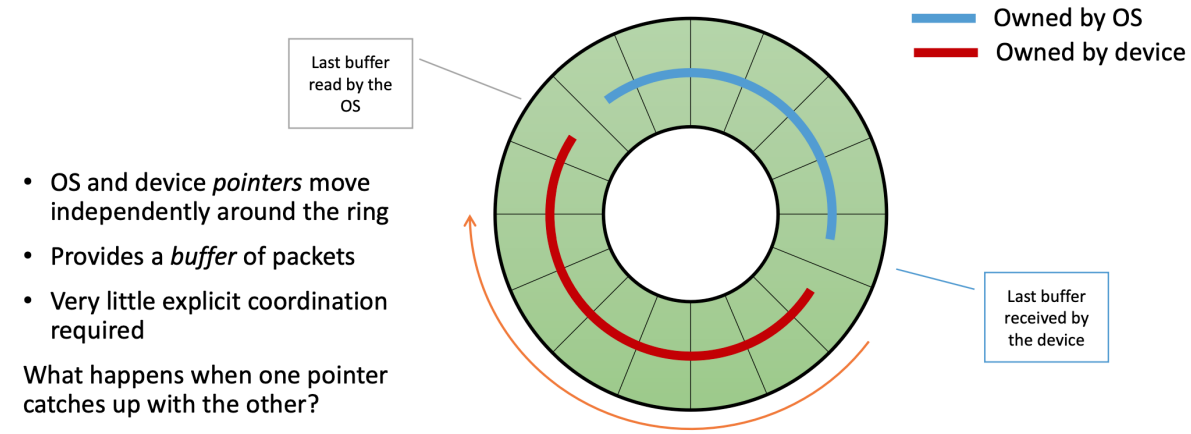


Physical View in Memory



Logical View

Here we see a receive ring:



- OS and device *pointers* move independently around the ring
- Provides a *buffer* of packets
- Very little explicit coordination required

What happens when one pointer catches up with the other?

Overruns and underruns

Transmitting

- Device has no more packets to send → it must wait
  - Could continue to poll memory until next descriptor is owned by it or could go to sleep and signal the software to wake it up
- CPU has no more slots to send packets → must wait
  - Signals the device to interrupt it when a packet has been sent

Receiving

- Device has no buffers for received packets → **starts discarding packets**

- Not as bad as it sounds, will start copying them to memory when a buffer is free

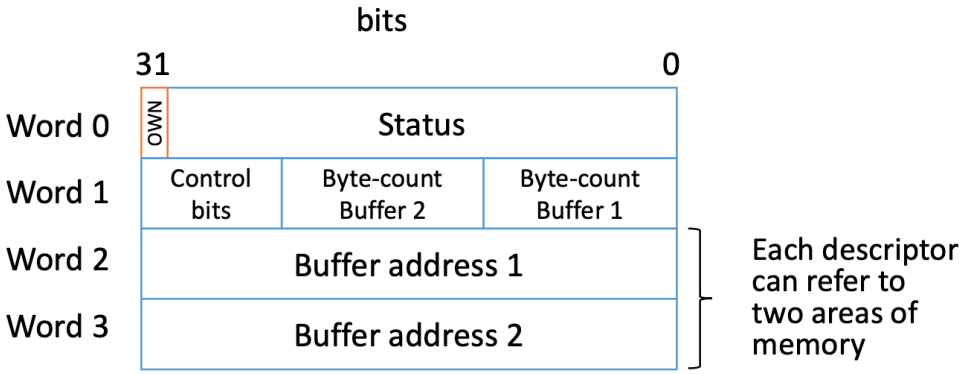
- CPU reads all received packets → it must wait
  - Signals the device to interrupt it when a new packet has been received

> Notice that these descriptor rings are producer-consumer queues!

20.6 More complex devices

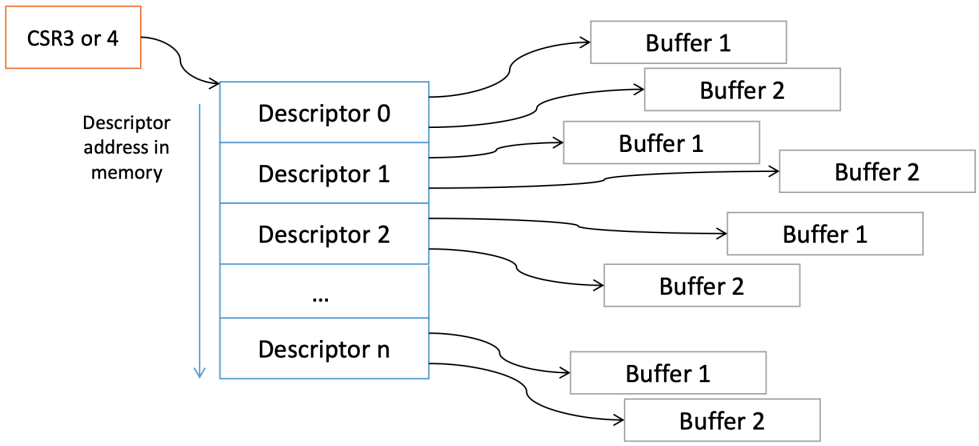
Tulip descriptors (old network card)

This is how a single descriptor ring can look like:

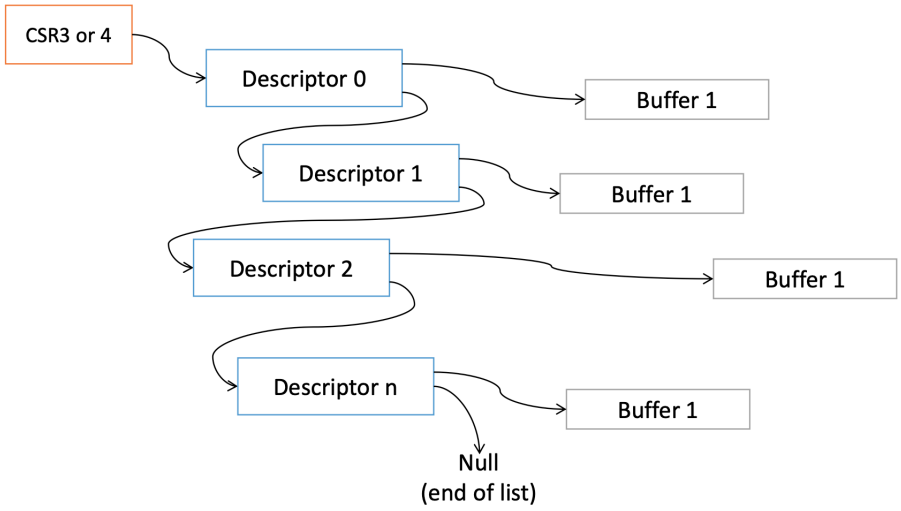


Descriptor rings

CSR3 / CSR4 are the base addresses of the descriptor rings



Descripor rings - chain mode



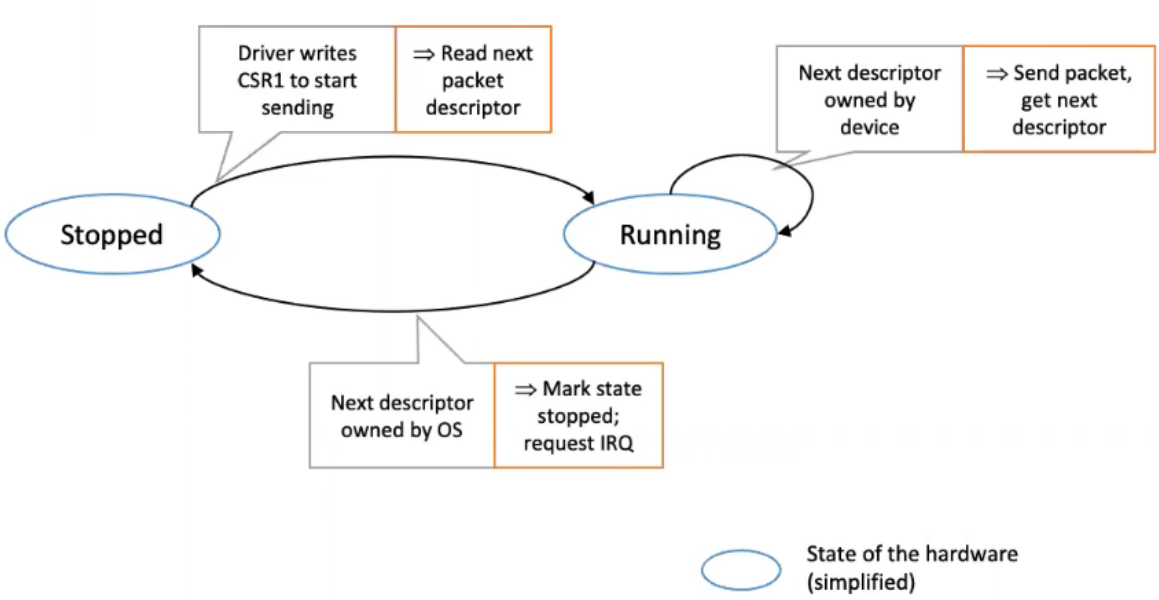
20.7 Device initialization

We do the following steps on device initialization:

1. Wait for the hardware to settle down
2. Stop the device from doing anything, just to be sure
3. Create shared data structures (i.e. descriptor rings)
4. Write registers to start the device running

20.8 I/O state machines (hardware side)

Sending packets



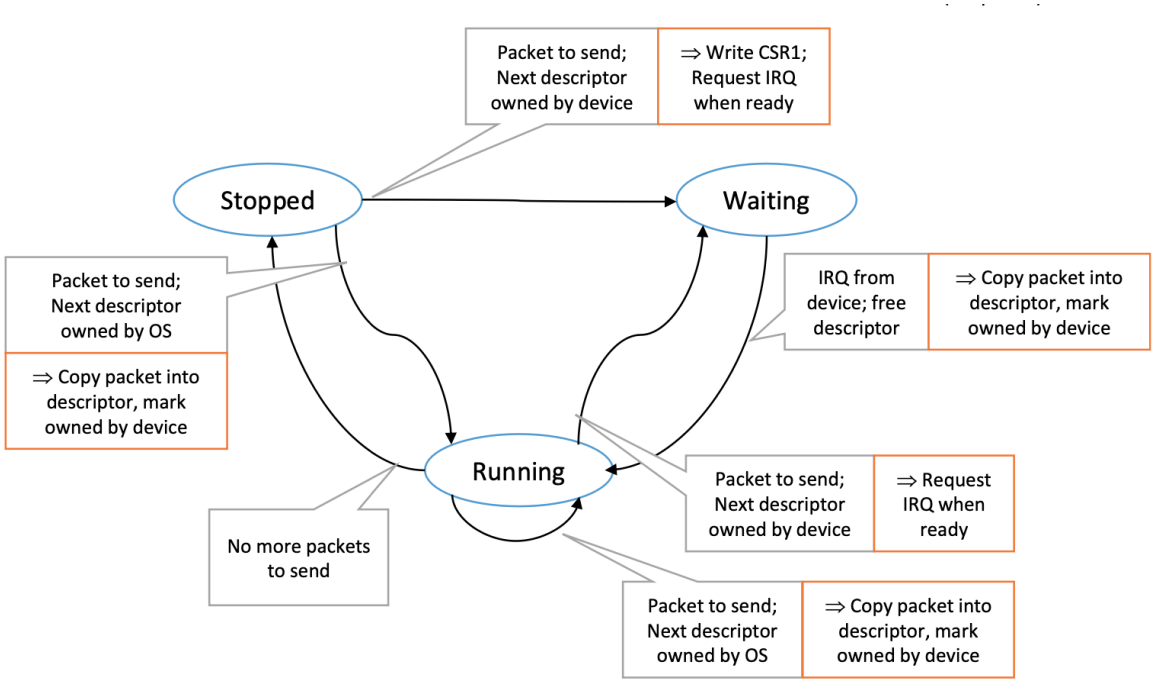
We therefore have the following DMA transactions:

1. DMA Read: descriptor
2. If *decsriptor.owner* = *OS* then enter state "stopped"

3. DMA Read: buffer
4. Send packet
5. DMA Write: *descriptor.owned* ← *"OS"*
6. Calculate next descriptor address
7. Goto 1.

20.9 I/O state machines (software side)

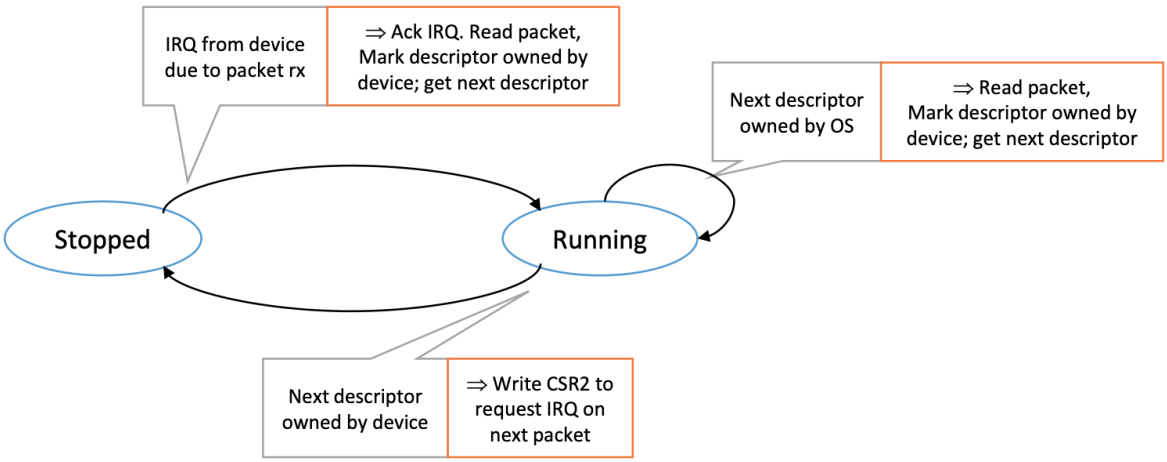
Sending packets



We have to avoid cache problems, in x86 hardware the CPU takes care of this, but no everwhere this is the case, hence:

- DMA read (device reads from memory):
  - **Before**: CPU should **flush** the cache for that address
  - **After**: CPU should **invalidate** cache for that address
- DMA writes (device writes to memory):
  - **Before**: CPU should **flush** or **invalidate** cache
  - **After**: CPU **invalidates** cache

Receiving packets



20.10 Discoverable buses: PCI

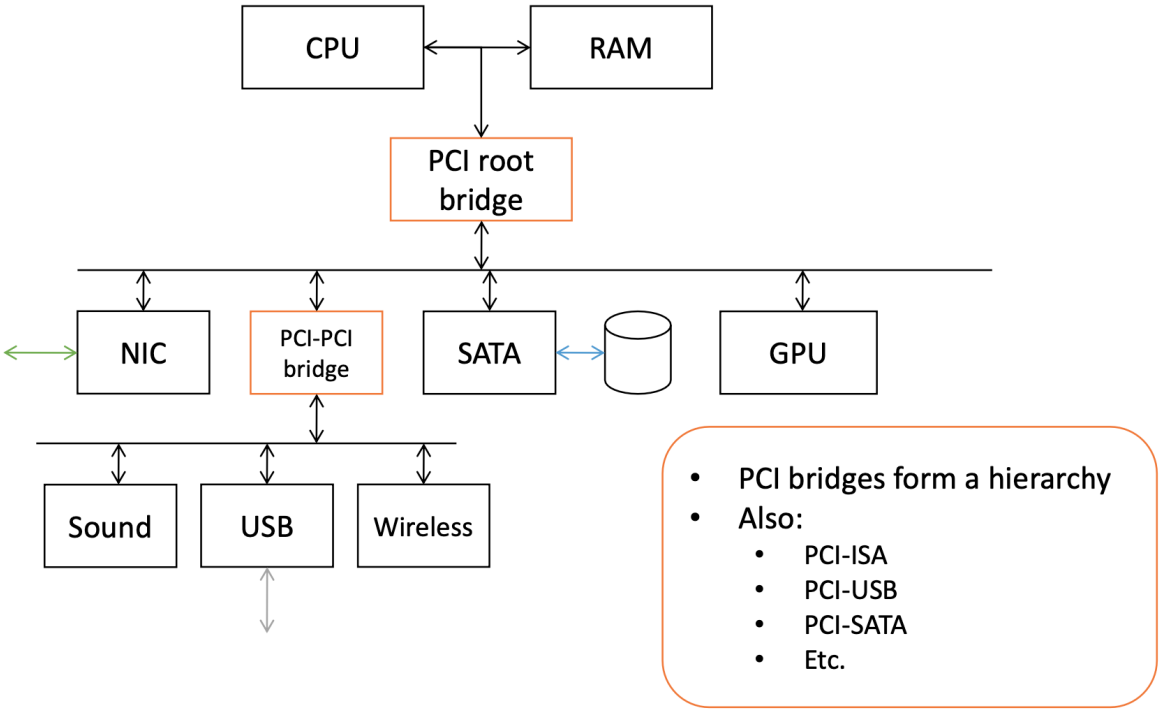
The ***PCI*** is a Peripheral Component Interconnect:

- An electrical standard for connecting devices
- A standard for physical connectors
- A set of “bus protocols” for communication between devices
- A software-visible interface to I/O hardware

The PCI tries to solve the following problems:

- Device discovery: Finding out which devices are in the system
- Address allocation: Which addresses should each device's register appear at?
- Interrupt routing: Which interrupt signals from the device should map to which exception vectors?
- Intelligent DMA: “Bus mastering” devices no longer need a DMA controller → devices can issue read / write transactions

PCI is a tree / hierarchy:



Each PCI device asks for a set of address ranges. The bridges up the tree remap addresses to the device. As a result each device appears as a set of contiguous address ranges.

Finding all the devices

PCI devices are self-describing, meaning they all come with a configuration header, that has the most important informations. To find all devices, all the configurations are read in a recursive way, starting at the PCI root bridge.

PCI Interrupts

There are four interrupt line `INTA, INTB, INTC, INTD`. The bridge allows arbitrary wiring of device lines to bridge lines. PCI Express introduced MSI (message-signalled interrupts), these are interrupt signals encoded as PCI writes to specified address range, giving us more flexibility to individually steere interrupts to particular cores.