

Summary for CPEN 212

Tom Wang

Spring 2024

1 Calling conventions

In assembly, sometimes we need to make system calls to the OS. We need to follow the calling conventions to make sure that the OS can understand our request.

1.1 Start with examples

Start with an example:

```
.text
.global _start
_start:
    mov x0, 42
    mov x8, 93
    svc 0
```

Here, we are trying to make a system call to exit the program. We need to put the system call number in x8, and the argument in x0. The system call number for exit is 93. We then use `svc 0` to make the system call.

- *.text* tells the assembler/linker that code follows.
- *.global* tells the assembler/linker that the symbol `_start` is visible to the linker. This is the entry point of the program.
- Then we put the value in x0 and x8. Here x0 is the argument for the system call, and x8 is the system call number.

To compile this and run (on Linux/arm64):

```

as -o foo.o foo.s
ld -o foo foo.o
./foo
echo \$?

```

The last line prints the exit code of the program. In this case, it should be 42.

- *as* is the assembler. It takes the assembly code and produces an object file.
- *ld* is the linker. It takes the object file and produces an executable.
- *./foo* runs the executable.
- *echo\$?* prints the exit code of the program.

Another example:

```

    .text
    .global _start
_start:
    mov x8, 64
    mov x0, 1
    adr x1, txt
    mov x2, len
    svc 0
    mov x8, 93
    mov x0, 42
    svc 0
txt: .ascii "goodbye, cruel world!\n"
len = . - txt

```

This program prints "goodbye, cruel world!" and exits with code 42. Here, we are using the write system call. The system call number for write is 64. The arguments are:

- system call number 64 in x8 means write
- file descriptor 1 in x0 means stdout, which is the standard output.
- address of the string in x1
- length of the string in x2

- system call number 93 in x8 means exit
- exit code 42 in x0
- system call 0 to make the system call
- the string to print
- The length is calculated by subtracting the address of the string from the current location. (note that in the stack, the address grows downwards)

One more example:

```
length:
    mov x2, -1
length_rec:
    add x2, x2, 1
    ldrb w1, [x0], 1
    cbnz w1, length_rec
    mov x0, x2
    ret

.global _start
_start:
    adr x0, txt
    bl length
    mov x2, x0
    mov x0, 1
    adr x1, txt
    mov x8, 64
    svc 0
    mov x8, 93
    mov x0, 0
    svc 0
txt: .asciz "goodbye, cruel world!\textbackslash n"

//Here we decompose bl and ret
str lr, [sp, -4]! //store the lr original value
```

```

mov lr, [pc, 4]
//load the address of the next instruction
mov pc, length //jump to the function
...\dots
//now returning
mov pc, lr //return to the address in lr
ldr lr, [sp] //restore the lr value
add sp, sp, 4 //restore the stack pointer
//that is it

```

Here instead of using the length variable, we are using a function to calculate the length of the string. The function is called `length`. It takes the address of the string as the argument and returns the length of the string in `x0`. The function is called using `bl`, which is a branch and link. It branches to the function and saves the address of the next instruction in the link register. The function returns using `ret`, which branches to the address in the link register.

Explain code line by line:

- `movx2, -1` sets `x2` to -1. This is the initial value of the length.
- `addx2, x2, 1` increments `x2` by 1. Increase the length by 1 for each loop.
- `ldrbw1, [x0], 1` loads a byte from the address in `x0` into `w1`. The address in `x0` is then incremented by 1. This loads the next character in the string. If we want to increment by 1 then load, we can use `ldrbw1, [x0, 1]!`. The `!` means increment after.
- `cbnz w1, length_rec` checks if `w1` is not zero. If it is not zero, then it branches to `length_rec`. This is the loop condition. If the character is not zero, then we continue the loop.
- `movx0, x2` moves the length in `x2` to `x0`. This is the return value.
- `ret` returns to the address in the link register.
- The rest is the same as before.

1.2 calling convention

Calling convention is not enforced by hardware, but it is a convention to make sure that the caller and callee agree on how to pass arguments and return values.

The caller is the function that calls another function. The callee is the function that is called. The calling convention is a contract between the caller and the callee. The caller must follow the calling convention to call the callee. The callee must follow the calling convention to be called by the caller. The calling convention includes:

Here are some rules:

- bl is for address relative to the current instruction. b is for absolute address. blr is for register relative to the current instruction. br is for absolute register.
- function arguments are passed in registers x0-x7. If there are more than 8 arguments, then the rest are passed on the stack.
- function results are in x0. If there are more, we use x1 to x7 then the stack. x8 is used for system call number.
- return address is in lr (x30). lr is the link register. It is used to store the return address when we call a function. It is also used to store the address of the next instruction when we use bl.
- Caller is responsible for saving registers x0-x18. The callee can use them without saving them. The callee is responsible for saving registers x19-x30. The caller can use them without saving them.
- The stack grows downwards. The stack pointer is in x31. The stack pointer points to the last item in the stack. The stack pointer is decremented when we push something onto the stack. The stack pointer is incremented when we pop something from the stack.

1.3 call frame

The call frame is the part of the stack that is used to store information about the function call. It includes the return address, the saved registers, and the local variables.

The call frame is created when we call a function. It is destroyed when the function returns. The call frame is created by the caller. The caller is responsible for saving the registers and creating the local variables. The callee is responsible for saving the return address.

Usually we have stack pointer which points to the last item in the stack.

We also have frame pointer which points to the start of the call frame. The frame pointer is used to access the local variables.

The frame pointer is not necessary, but it makes it easier to access the local variables.

But if we have dynamically sized stack allocation (memory allocation functions), then fp is obligatory.

1.4 Jumping to a function

To jump to a function, we use bl. bl is a branch and link.

It branches to the function and saves the address of the next instruction in the link register. The function returns using ret, which branches to the address in the link register.

Back in the time when there is no bl, we have to do this manually. We have to save the return address in the stack, then jump to the function. The function then returns by popping the return address from the stack and jumping to it.

Another way is to specify the line of return when creating the callee function. And then when we call a function, the caller will write the return address to the specified line. And then the callee will jump to that line when it returns. This is called a trampoline. Not too nice to use, but it is a way to do it back in the day.

1.5 ABI

ABI is the application binary interface. It is a standard for binary interfaces. It specifies the calling convention, the system call numbers, and the system call arguments. It is used to make sure that the caller and callee agree on how to pass arguments and return values.

1.5.1 stack-based ABI

- stacks are sequential. If we spawn two functions, the stack must grow into two different directions. how to achieve this? We can use two different stacks.
- stack spaces are deallocated on return, so we can reuse the stack space.

2 Haxx

Start with a C example:

```
int main() {
    volatile long ans = 42;
    char buf[16];
    gets(buf);
    //latex does not support printf properly
    printf(ans);
    return 0;
}
```

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

So this program is vulnerable to buffer overflow. We can overflow the buffer and overwrite the value of ans.

2.1 Analysis of the victim

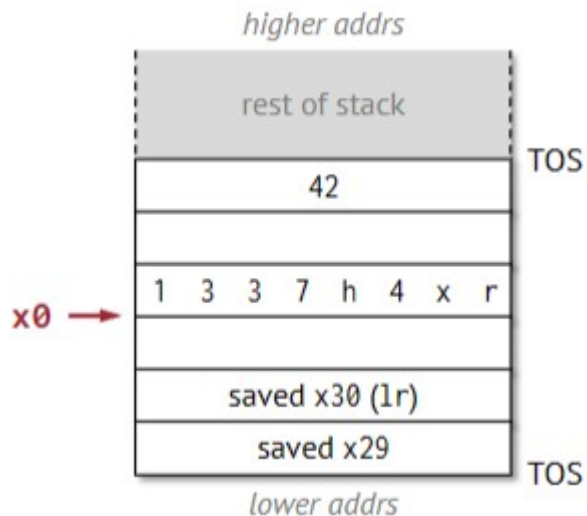
By observing, volatile makes sure that ans lies on the stack. Then we allocated 16 bytes for buf (char is 1 byte each). Now the pointer of buf is 16 bytes away from ans. With that, have a look at the compiled assembly code:

```
main:
    stp x29, x30, [sp, -48]!
    mov x29, sp
    mov x0, 42
    str x0, [x29, 40]
    add x0, x29, 24
    bl gets
    ... rest of the code
```

stp is store pair. It stores two registers in the stack. The first register x29 (Stack pointer, pointing to the top of the stack of the parent function) is stored at the address in sp. The second register x30 (Link register, parent function address) is stored at the address in sp minus 8. The stack pointer is then decremented by 48. So 48 lines of memory is allocated for the call frame.

Then x29 keeps the current program sp. x0 is loaded with 42. Then x0 is stored in the address in x29 plus 40. So the address of ans is x29 plus 40 which

is the bottom of the frame (Stack grows downwards, so right next to the parent function's top of stack). Then x0 is loaded with x29 plus 24. So x0 is loaded with the address of buf. Then we call gets.



Now if we input more than 16 bytes, we can overflow the buffer and overwrite the value of ans.

For example, we can input “ffffff fffffff 3905”. This will overwrite the value of ans with 3905.

2.2 Typical attack pattern

- Find a program running on target (maybe as root)
- scan binary for potential buffer overflows. E.g, look for gets, strcpy, strcat, sprintf, etc.
- craft input that will overflow the buffer and overwrite something interesting
- objective: execute arbitrary code

2.2.1 Jump to arbitrary code

In ARMv8, typically, we save lr to the top of the stack before bl to another function.

Since overwrite in this case writes to the higher address, but lr for the current call frame is at the lower address, we can't overwrite the current function's lr.

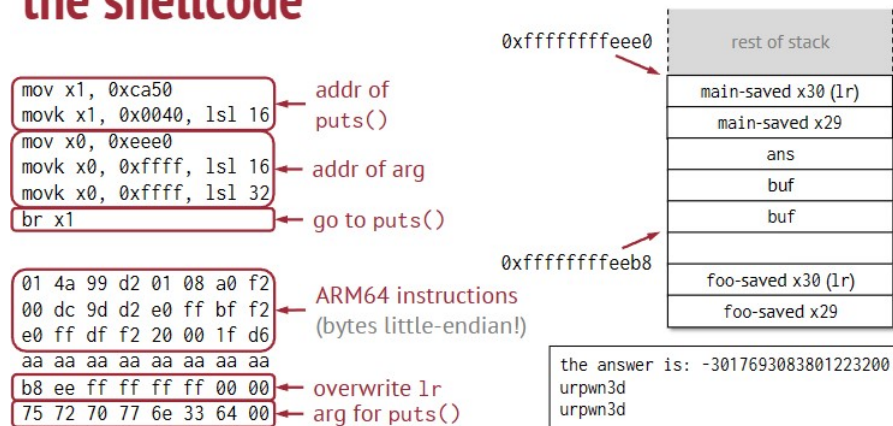
But we can overwrite the return address of the parent function which will lie just above the bottom of the call frame. So we can overwrite the return address of the parent function with the address of the code we want to execute. Then when the parent function returns, it will jump to the code we want to execute.

2.3 Code injection

Now we know how to jump to other addresses, but we still need to inject our code into the program. How do we pass arguments to the code you want to execute?

One way to do it is to overwrite the stack with the machine code that corresponds to the assembly code you want to execute. Then you can pass arguments to the code by putting them in the stack.

the shellcode



Note that the ARM64 code is little-endian. So the first byte is the least significant byte. We put the code to overwrite the space for `buf` and `ans`. The value printed is the third line of ARM machine code in large-endian.

we overwrite the `lr` with the start of the ARM code address "00 00 ff ff ff ff ee b8". So the `pc` will jump to your written code.

Also, since ARM only allows 16-bit constant input, we can load 16 bits of a register at a time. That is why we have to load the value of `ans` into `x0` and `x1` separately by shifting the constant input.

Now last line "`br x1`" would allow us to jump to the address in `x1`. The argument we want to pass has its address in `x0`, which is the last line we overwritten.

2.3.1 Code injection limitations

- stack/heap often non-executable.

- enforced by virtual memory system (Segmentation fault)
- It will crash the program if we try to execute code in the stack or heap.
- stack/heap locations often randomized (ASLR)
 - enforced by the virtual memory system
 - It will randomize the location of the stack and heap. So we can't predict the location of the stack or heap.

2.4 Gadgets

Gadgets are small pieces of code that end with a return instruction.

Since code injection is not executable, we can find some existing code (gadgets) and chain them together to do what we want. It could be either in the program itself or in a library.

2.4.1 return-oriented programming (ROP)

1. find gadgets in the program or libraries
2. put the address of the gadget in the stack by overflowing the buffer
3. prepare the arguments for the gadget by overflowing the buffer
4. construct a chain of gadgets that does what you want

Depending on the found gadgets, we inject the useful value into the stack by overflowing the buffer. The ultimate goal is to change the value of the link register and pass the desired arguments to the gadget.

Refer to mie's slides for examples.

2.4.2 more advanced ROP

- can be any indirect jump instruction, not just ret. For example

```
ldp x1, x2, [sp], 16
br x2
```

- can overwrite virtual function pointers in C++ objects.
- SROP: play with signals to set all registers.

2.5 Non-executable stack/data

For most systems, the stack and data are non-executable.

Virtual memory system enforces this. It will crash the program if we try to execute code in the stack or heap. Also called W^X policy (write xor execute).

However, we can use ROP to make a system call to make the stack executable. Then we can inject our code into the stack and execute it.

2.6 Address space layout randomization (ASLR)

ASLR randomizes the location of the stack and heap. So we can't predict the location of the stack or heap.

Virtual memory system enforces this. It will randomize the location of the stack and heap. So we can't predict the location of the stack or heap.

2.6.1 Limitations of ASLR

- not all bits are random, it is possible to brute-force the address space. Just need more trials.
- Library locations are not randomized since programs need to find libraries to use them once the program starts. So we can use gadgets in libraries.
- Programs on machines like servers are not restarted often. So we can use brute-force to find the address of the stack and heap.

2.7 Array bounds checking

Some programming languages like Java and Python have array bounds checking. It checks if the index is inside the bounds for each operation. So we can't overflow the buffer.

This prevents writing to memory that we don't own. However, for languages that could do pointer arithmetic, we can still do buffer overflow. C and C++ are typical examples.

Also, people sometimes disable array bounds checking for performance reasons. This kind of fast but unsafe code is hackers' best friend.

2.8 Stack canaries

- put some value on the stack below the return address (or frame pointer if it exists)
- check the value, see if it is modified before ret.
- if modified, abort the program. Might print something like “stack smashing detected” (Should work just like an exception)
- the variants can be 0, random, or random XOR with the return address.

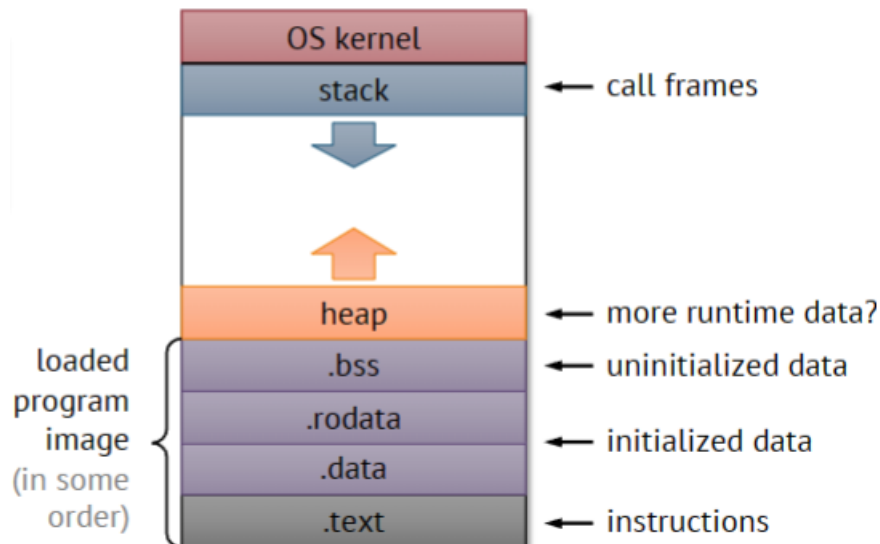
However, there are still limitations to it:

- if exception handling is hacked, we can still bypass the stack canaries.
- the pattern can only detect **some** overflows.
- fixed canary value can be easily discovered.
- canary value can still be leaked.
- canary feature is usually not enabled by default.

2.9 safe(r) programming languages

- Java, Python, etc. have array bounds checking.
- Rust, Go, etc. have memory safety.
- C/C++ are still the most popular language.
- safe languages are not always safe. For example, Java has a lot of vulnerabilities.
- Some languages guarantee safety via **type system**. For example, Rust. It is checked by the compiler at compile time. However, only guaranteed if used properly. Also, these languages are not as popular as C/C++.

3 Memory management



Each process has its own memory space. The memory space is divided into segments. The segments are:

- text segment: the code of the program
- data segment: the variables
- heap: the dynamic memory allocation
- stack: the local variables and the call frames

Heap and stack grow towards each other. The stack grows downwards. The heap grows upwards.

- Stack is easy to allocate but disappears after the function returns.
- Heap is harder to allocate but can be used after the function returns.

3.1 Allocation strategies

- automated:
 - garbage collection: the memory is automatically deallocated when it is no longer used. It is used in languages like Java and Python.

- reference counting: the memory is automatically deallocated when the reference the count is zero. It is used in languages like C++. (There are limitations to this, for example, circular references)
- Hard to implement but easy to use
- manual: allocate and deallocate pairs.
 - malloc()+free() or new()+delete() comes in pairs. (It is a contract that must be obeyed)
 - easy to implement but hard to use. (lots of bugs)

3.2 Memory allocation

- want high utilization and low fragmentation
 - physical memory is limited among all processes
 - OS only gives big chunks of memory to processes (usually 4KB pages)
 - most allocs are small, few words to a few KB
- want high allocation throughput. Minimize the management overhead.

Recall the stack allocation:

- to allocate, just move the stack pointer downwards
- to deallocate, just move the stack pointer upwards
- easy to implement and efficient
- However it does not support dynamic allocation which does not work for heap.

3.2.1 Challenges

- how to keep track of block size?
- how to keep track of free blocks?
- how to find a free block?
- how to allocate a block? which free block to use?

3.2.2 tracking allocation size

For each block, we can store the size of the block which can be used to find the start and the end of the current block. Thus we can find the neighboring blocks.

Note that the blocks can also request the size of both neighboring blocks. Just having the size at the start of the block is not enough. We need the size at both the end and the start of the block.



Note that we may or may not contain the padding there to follow the alignment rules of the system. At the same time, we can omit the lower bits of the size to store the status of the block since size should always occupy the whole line in the system no matter what the system is working with. Lastly, the *a* there can indicate the status of the block. 1 means allocated, 0 means free.

3.2.3 finding a free block

let p = heap start: for each block b at p :

- if b is free **and** b is big enough, return b
- else $p = p + \text{size of } b$
- keep going until the end of the heap
 - either we find a free block
 - or we run out of memory
 - or ask OS for more memory

This is a “first fit” strategy. It is simple and fast, but it can lead to fragmentation.

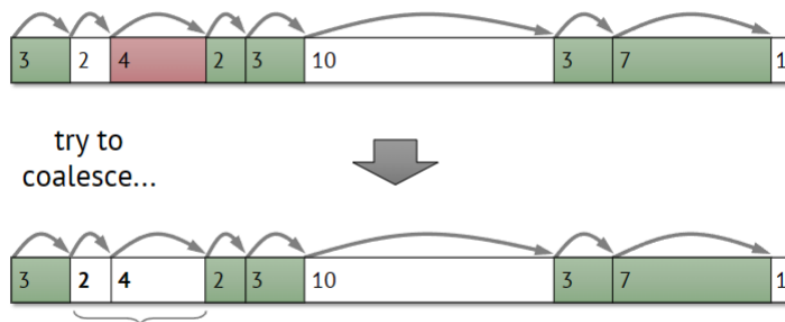
We can have “next fit” as well:

- keep track of the last block we allocated, return the next free and big enough block after that
- can be faster than first fit
- but still can lead to fragmentation (even worse than first fit)

Also we can have “best fit”:

- return the smallest free block that is big enough
- but it is slow (overhead of searching for the smallest block)

3.2.4 coalescing and boundary tags



As shown in the figure, if we just free the 4-byte block, we will have two free blocks connecting together which we can merge by coalescing. This is done by checking the status of the neighboring blocks. If they are free, we can merge them together. This is called coalescing.

Note that we can use either the header or the boundary tag to track the neighboring blocks. So it does not matter if the free neighboring block is at the higher address or the lower address. We can always find it by checking the status and the size of the neighboring blocks.

In some cases, we might need to connect both neighboring blocks together if it happens that they are both free.

3.3 explicit freelist

The above section is an example of an implicit freelist. The freelist is implicit because it is not explicitly stored. It is stored in the heap itself. We need to find them by checking the status of the neighboring blocks.

We can also have an explicit freelist. We can have a linked list of free blocks. Each free block contains a pointer to the next free block. This is called an explicit freelist because the freelist is explicitly stored in the heap.



3.3.1 data structure

For a double-linked list, we can have a pointer to the next free block and a pointer to the previous free block. This is called a doubly linked list. It is useful when we want to remove a block from the list. We can just change the pointers of the previous and the next block to remove the current block from the list.



This way, we can find the free block by just following the pointers. We can also remove the block from the list by changing the pointers. We can also add a block to the list by changing the pointers.

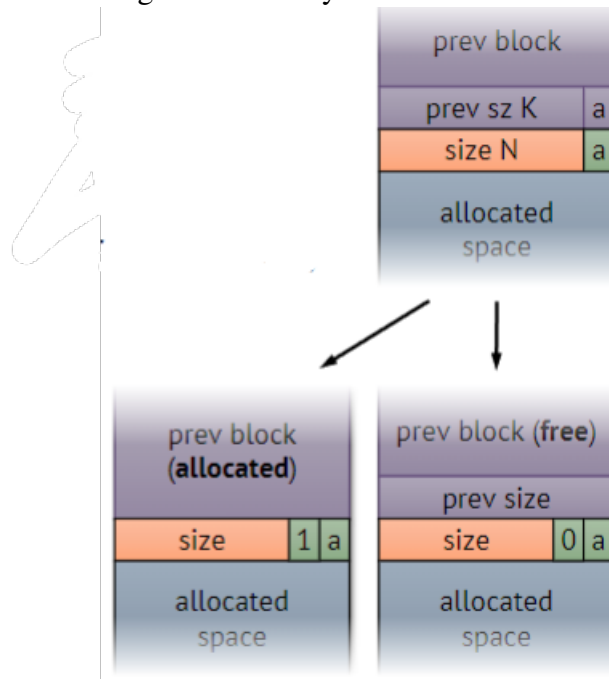
3.3.2 Coalescing

Note that coalescing is not achieved by the explicit freelist since it does not know the neighboring blocks. We need to use the implicit freelist to coalesce the blocks. So we still need to header and footer to track the neighboring blocks.

3.3.3 Optimizing boundary tags

Consider when we use the boundary tags? We use it when we want to coalesce the blocks. So we can optimize the boundary tags by only using it when we want to coalesce the blocks which would only happen for the free blocks.

To know if the previous block is free, we use another bit in the header to indicate if the previous block is free. Note that lower bits of size can be omitted because of alignment with bytes reason.



3.4 segregated free lists

If we mix all sizes of free blocks in one list, then it becomes hard to track the sizes and find the best one to use and do coalescing.

It is better to keep a list for some special size of blocks. Usually, 2^k bytes or Fibonacci bytes. The reason for choosing these two is that they add up to each other, making them easy to do coalesce.

3.4.1 buddy allocator

- what if one FL is empty? Just steal another block from the FL with larger sizes and break it into two.
- what if we coalesce? Just put the coalesced block into the FL with larger sizes.

3.4.2 slab allocator

- what if common sizes are not powers of 2? For example, we use size of 3 all the time. Then we waste internal fragmentation overhead of 25%.
- We can fix it by using slab allocator. We can allocate a slab of memory and divide it into blocks of the same size. Then we can use the blocks to allocate memory. For example, we have so many size 3 right? Just pick a 256 block and break them to allocate size 3. Overhead is only $\frac{255}{256} = 0.4\%$.

4 Garbage collection

To automatically free the memory that is no longer used. It is used in languages like Java and Python.

Explore a few ways to achieve it:

4.1 Delete upon destruction

When variables go out of scope, the destructor is called. Why don't we just free the memory there?

```
template <typename T> class ptr{
    T* p;
    public:
        //some other code:
        virtual ~ptr() {delete p;}
        //upon destruction, delete the memory
    }
    void foo() {
        auto c = ptr<complex>(new complex(1,2));
        //do something with c
        //c is destroyed upon exiting foo()
    }
    void foo1() {
        auto c = ptr<complex>(new complex(1,2));
        bar(c);
        //some other use of pointers
    }
}
```

```

void bar(ptr<complex> c) {
    //do something with c
    //c is destroyed upon exiting bar()
}

```

Note that it works well with `foo()`, however, it can cause trouble with `foo1()`. The reason is that we can't free the memory until we exit `foo1()`. But we have already deleted it upon exiting `bar()`. So we can't use it anymore.

So we need a better way of doing it.

4.2 Reference counting

Why don't we just count the number of references to the memory? When the number of references is zero, we can free the memory.

- upon initialization, set the reference count to 1
- upon assignment, increment the reference count of the new pointer and decrement the reference count of the old pointer
- upon destruction, decrement the reference count and free the memory if the reference count is zero

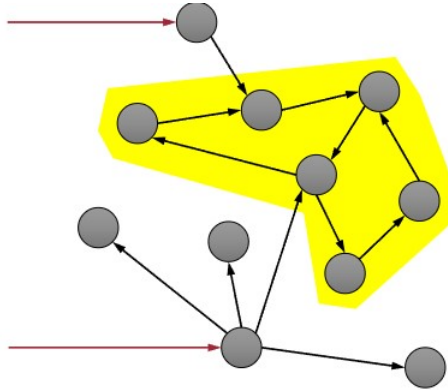
Note that we store the number inside the memory itself instead of the pointers. This is because if we store it in the pointers, for each update, we need to update all of the pointers to keep them present. This is not efficient.

4.2.1 Overhead

- Space overhead: each smart memory has a reference count. Then smart pointers have two pointers, one to the memory and one to the reference count.
- Performance overhead: each access is just 1 pointer dereference. However, each copy/destroy is 1 pointer dereference and 1 arithmetic operation.

4.2.2 Limitations

circular references: if two objects refer to each other, then the reference count will never reach zero.



In the picture there, it is possible that we lost the reference to the memory while the reference count is not zero. This is a memory leak.

4.3 reachability

So the real problem is reachability. If the memory is not reachable, then we can free it. So we need to find a way to check if the memory is reachable.

Observation: once a memory is not reachable, it will never be reachable again. So we can free it.

Now we need to know how to check if the memory is reachable. We can use a graph to represent the memory. Each node is a memory block. Each edge is a reference from one block to another. Then we can use graph traversal to check if the memory is reachable.

4.3.1 A tracing garbage collector

- start from the roots (global variables, local variables, etc.)
- follow the references to other memory blocks
- mark the memory blocks as reachable
- free the memory blocks that are not marked

4.3.2 The tri-colour abstraction

- white: not visited
- grey: visited but not finished (A data that contains multiple references to other data, and we did not visit all of them)

- black: visited and finished

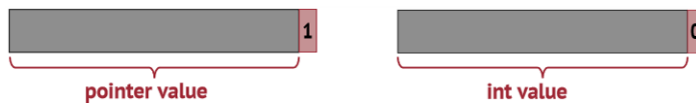
Using this abstraction, we can use some graph algorithms to do the garbage collection. In the end, all the block that we mark as white can be freed.

4.3.3 mark and sweep

- pause the program
- traverse the graph from the roots
- mark the reachable memory blocks
- sweep the memory blocks that are not marked
- resume the program

4.3.4 mark and sweep implementation

- Note that we need a bit in the memory to indicate if the memory is marked. We can just steal another bit from the size. Since we already have one for allocation status and one for the previous block, it requires it to be 8-byte aligned. So we can just steal the lower bit of the size to indicate if the memory is marked. A negligible overhead.
- We also need to distinguish between pointers and integers. We can use the lower bit of the pointer to indicate if it is a pointer or an integer.



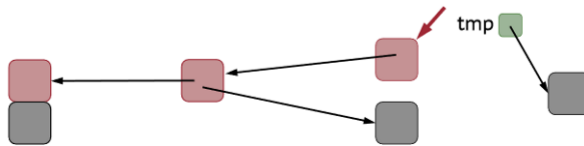
Note that we chose integers to be even numbers. This is to make sure that pointer arithmetics with integers keep the lower bit to be 1 which keeps consistency with the pointers.

drawback of this approach is the smaller range of integers.

Note that this might not be a problem in some programming languages where the data structure makes clear of where the pointers are. Java and Python are examples of this.

- Backtracking during traversal: the current way of doing it would store all the pointers as we mark, which will take up a lot of memory space. However, by the time we are doing the garbage collection, we are close to running out of memory, which could be a problem.

One of the way to solve it is by reversing the pointers. We use a temporary pointer to store the current branch and reverse the visiting node to point to the parent node. This way we can always trace back to the parent node without storing all the pointers.



Refer to Miezko's slides for an animation of this process.

This approach only takes $O(1)$ space. However, it is slower than the previous approach and it stops the program for a longer time since pointers are moved, the program cannot visit all the nodes in progress. No **concurrent** garbage collection.

4.4 Stop and Copy

Mark and sweep is nice, but it creates fragmentation. Free space is **not contiguous**.

Stop and copy is a way to solve this problem. It is a two-space collector. It has two spaces: from-space and to-space. It copies the reachable memory blocks from from-space to to-space. Then it swaps the two spaces.

Refer to Miezko's slides for an animation of this process.

- Stop and Copy would require us to only use half of the memory space. This is a big overhead. But it is not that bad. Most of the time, the objects we left in the heap does not live for too long. So a much more frequent garbage collection is not that bad given that we get huge chunk of free space instead of fragmented space.
- It is **Not Concurrent**. It stops the program since it moves the pointers.
- Note the use of two pointers **scan** and **free**.

- scan: if all pointers inside the node are visited, and the pointed blocks are moved to the to-space, then we can move the pointer to the next node. We paint the node black following the tri-colour abstraction.
- free: if the node is reachable. Paint the node grey if so and move on.

4.5 refcount vs mark and sweep s stop and copy

	refcounts	mark & sweep	stop & copy
collects cycles?	no	yes	yes
compacts?	no	no	yes
space overhead?	rc (even on stack!)	none	semispaces
alloc time?	2× malloc	same as malloc	free++
copy time?	wrapper copy, rc++	pointer copy	pointer copy
end-of-scope time?	rc--, maybe free	nothing	nothing
out-of-mem time?	nothing	trace + free	trace + copy

4.6 mark and compact

Wonder if there is a way to use the entire heap nad still avoid fragmentation (compact)?

Idea: save the working pointers somewhere else so that we can use the whole sapce.

- keep an extra space to store them.
- keep the pointers in the heap itself as a break table and move it accordingly if needed.

Refer to miezko's slides for an animation of this process.

4.7 ephemeral garbage collection

Observation: most allocs have short lifetimes. Why not just garbage collect the young objects?

Split heap into generations:

- young generation (nursery): for new objects

- when the young generation is full, garbage collects it
- if an object survives a few garbage collections, move it to the old generation
- we can have multiple levels of generations, apply gc as well if they are full

Note that gc is fast if we keep the young generation (nursery) small.

4.8 Weak references

- a weak reference is a reference that does not prevent the object from being garbage collected
- it is useful for caches, observers, etc.
- The object is reachable, but the event is not happening anymore. So we can free the memory.
- it is used in languages like Java and Python

4.9 non-deterministic finalization

- finalization is the process of freeing the memory
- For example, in C++, the destructor is called when the object goes out of scope, and sockets and locks in Java and Python are closed when the object is garbage-collected
- it is non-deterministic because we don't know when the memory will be freed

4.10 Conservative garbage collection

Some languages do not distinguish between pointers and integers. They can also do pointer arithmetics with pointers.

We can use some logic to find the difference, for example, small numbers are generally not pointers. However, this is not always true. If not pointing inside blocks, then it is not a heap pointer.

These tricks are not always reliable. So we can use conservative garbage collection. It is a way to do garbage collection without knowing the exact location of the pointers. We can just assume that the integers are pointers and do garbage collection.

5 Virtual memory

Virtual memory is needed since we want to run multiple instances simultaneously and use the same memory addresses.

Each program would think that they have the entire memory to itself.

5.1 Virtual memory abstraction

- each program has its own virtual memory space
- the virtual memory space is divided into pages
- the pages are mapped to physical memory
- the mapping is done by the operating system

In a 64-bit architecture, we usually divide into 4KB pages. Thus the last 12 bits of a Virtual Address is the offset of the page. The rest of the bits are the page numbers.

5.1.1 levels of page tables

Page tables can be divided into levels.

- PT0: 47-39
- PT1: 38-30
- PT2: 29-21
- PT3: 20-12
- offset: 11-0

Each cell in PT_i is a pointer to the next level PT_{i+1} , which is a new page. The last level is the physical memory.

Each entry of the page table contains the physical address of the next level page table. The actual data page is stored in the last level, accessed by the offset which contains bytes.

5.2 PTE (Page Table Entry)

There are lots of other bits in the entry of the virtual page table to indicate something else:

- valid bit: if the page is valid (if it is not valid, then it is not in the physical memory)
- dirty bit: if the page is modified (if it is modified, then we need to write it back to the disk when we swap it out)
- read-only bit: if the page is read-only (if it is read-only, then we can't write to it)
- user-accessible bit: if the page is accessible by the user (if it is not accessible, then we can't access it)
- non-executable bit: if the page is non-executable (if it is non-executable, then we can't execute code in it)
- physical page number: the address of the page in the physical memory
- recently used bit: if the page is used recently (if it is used recently, then we can keep it in the physical memory, otherwise, it becomes a candidate for swapping out)

If there is insufficient permission, then we will get a segmentation fault.

5.3 VM uses

5.3.1 more memory

We have limited physical memory. We can use virtual memory to use the disk as memory. We can swap the pages in and out of the physical memory. This is called paging.

Upon more memory requests, we either terminate some programs to free up the RAM or move some pages to the disk to free up the RAM.

Use the idea of **locality** to make use of the physical memory. We can use the pages that are used recently and the pages that are used together.

5.3.2 Page fault

When we access a page that is not in the physical memory, we get a page fault. The operating system will then load the page from the disk to the physical memory and then continue the program.

It will evict some pages from the physical memory to make space for the new page. It will use the idea of locality to decide which pages to evict.

5.3.3 shared memory

We can use virtual memory to share memory between processes. We can map the same page to different processes. This is called shared memory.

So same physical memory can be mapped to different processes with different VM. This is useful for inter-process communication.

This saves the memory space and the time to copy the data. For example, multiple processes can share the same library. So we only need to load the library once.

Note that they should be read-only, otherwise, we will have a problem with the synchronization or haxx attack.

At the same time, it can share the same file as IO. For example, we can map the file to the memory and then read and write to the memory. This is called memory-mapped IO.

5.4 Speed up memory access

VA to PA translation is needed for each memory access. It is slow. We can use a TLB (Translation Lookaside Buffer) to speed it up.

5.4.1 TLB(Translation Lookaside Buffer)

It is a separate cache for the page table. It caches the recently used page table entries. It is much faster than the page table.

- if the VA is in the TLB, then we can use the PA directly
- if the VA is not in the TLB, then we need to use the page table to find the PA

TLB requires **coherence** mechanism. If the page table is updated, then the TLB should be updated as well. Otherwise, we will have a problem with the inconsistency. It is called a **TLB shutdown**.

- upon a TLB miss, we evict a TLB entry and load the new entry
- upon a context switch, we flush the TLB (different processes have different page tables)

6 Processes

If we run `top` in a Linux terminal, we can see a list of processes. Each process has its own memory space. The memory space is divided into segments. The segments are:

- **PID**: process ID
- **USER**: the user who runs the process
- **PR**: priority of the process (the higher the number, the lower the priority)
- **NI**: the nice value of the process (if it is negative, then it has a higher priority)
- **VIRT**: virtual memory used by the process
- **RES**: resident memory used by the process (the physical memory used by the process)
- **SHR**: shared memory used by the process (the memory shared with other processes)
- **S**: status of the process (R: running, S: sleeping, D: waiting, Z: zombie, etc.)
- **%CPU**: percentage of CPU used by the process
- **%MEM**: percentage of memory used by the process
- **TIME+**: time used by the process
- **COMMAND**: the command that runs the process

6.1 Process abstraction

OS kernel handles the process abstraction. It is the kernel that creates, schedules, and terminates the processes.

6.1.1 OS kernel

Heart of a running system:

- **interface** between the hardware and the software
- provides **abstraction** of the hardware (e.g., file system, network, memory, **processes, threads, VM** etc.)
- resident in memory in each program's address space (kernel space)
- runs at an elevated privilege level (ring 0)
- interacts with programs via **system calls** and **interrupts/signals**

6.1.2 Process

Processes run under an OS kernel. Each has its own **virtual addresses space** and **logical control flow**. They are isolated from each other. They can communicate with each other via **inter-process communication** (IPC).

- single **instance** of a running program has its own process ID (PID)
- each process thinks it has the entire memory to itself
- kernel **schedules** the processes on CPU. It can **interrupt** the processes. Switching between processes is called **context switch**

6.1.3 Threads

Threads are processes that share the same virtual memory space. They are isolated from each other. They can communicate with each other via **thread communication** (TC). However, they have their own **logical control flow**.

Threads have its own **stack** and **registers**. They share the same **heap** and **code**.

6.1.4 logical control flow

logical control flow is the sequence of instructions that the process executes. It is the **program counter** (PC). It is the address of the next instruction to be executed.

This is why threads are isolated from each other. They have their own logical control flow. They can execute different instructions at the same time. Thus different stacks and registers are needed.

6.1.5 Context switch

upon a context switch:

- Interrupt the CPU
- Save the registers and the PC of the current process (architectural state). Usually saved in the kernel stack.
- Load the registers and the PC of the next process
- Resume the CPU

6.2 Process tree

Processes can create other processes. The created processes are called **child processes**. The process that creates the child process is called the **parent process**.

Processes can **wait** for children to finish, **kill** children, and **communicate** with children. And it will be notified when the child process dies.

interface provided by OS to applications:

- `fork()`: create a child process by duplicating the current process
- `exec()`: replace the current process with a new program

to spawn a new process, we can use `fork()` to create a child process and then use `exec()` to replace the child process with a new program.

6.2.1 `fork()`

upon a `fork()` syscall:

- declaration:

```
pid_t fork(void);
```

- success: returns the PID of the child process to the parent process and 0 to the child process (the child process is a copy of the parent process, so they will see the exact same instruction, but different return value)
- failure: returns -1 to the parent process (no child process is created)

child is an exact duplicate of the parent process except that:

- the child has a different PID
- the child and the parent have a separate memory space
- the child's parent PID (PPID) is the parent's PID

```
int main() {  
    printf("Child does not print this, PC has passed\n");  
    pid_t pid = fork();  
    printf("PID is %d\n", pid);  
    printf("PPID is %d\n", getppid());  
}
```

Note that the above code would produce the following output:

```
Child does not print this, PC has passed  
PID is 1234  
PPID is 1233  
PID is 0  
PPID is 1234
```

So there is only one line of "Child does not print this, PC has passed" in the output. This is because the child process is a copy of the parent process. So it will not execute the line that is after the fork() syscall.

The child process will have a different PID and PPID. The child process will have a PID of 0. This is how we can distinguish the child process from the parent process.

Note that the order of the output is not guaranteed. The parent process and the child process can run in any order. So the output can be different each time we run the program. This is due to the scheduling of the OS and speculatively execution of the CPU.

6.2.2 zombie process

When a child process dies and the parent is still running, the child process becomes a zombie process. It is still in the process table, but it is not running. It is waiting for the parent to collect its exit status.

The parent can use the `wait()` syscall to collect the exit status of the child process. The child process will be removed from the process table after the parent collects the exit status.

If the parent dies before the child, the child becomes an orphan process. It is then adopted by the `init` process. The `init` process will collect the exit status of the child process and remove it from the process table.

6.3 Signals

Signals are a way to notify a process that an event has occurred from the kernel or another process. It is a way to do IPC (inter-process communication).

It is mediated by the OS kernel. Can be caused by

- exceptions (SEGV, FPE, etc.)
- user actions (KILL)

The kernel or process takes action depending on signals.

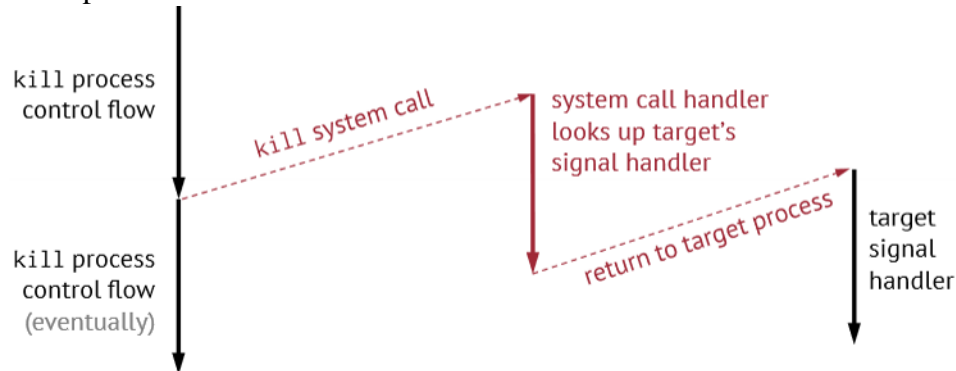
- KILL and STOP must be handled by the kernel
- others can be handled by **signal handlers**

6.3.1 signal handlers

Programs can register signal handlers with the OS.

- We register via the **signal()** syscall
- We can register a function to handle the signal, so it is a user-defined code.
- The function is called when the signal is received
- It is kind of like an interrupt handler in microcontrollers

For example:



Note that the signal handler is a separate function that runs in parallel. Before executing, it will check with the OS to see if the caller has permission to do certain tasks. If not, it will not execute the signal handler.

6.3.2 Signal handling

Issues with signal handling:

- Signals are **not queued**.
 - If a signal is received during *I/O*, the *I/O* will be interrupted. The signal handler will be executed.
 - To resume, we need to set the “SA_RESTART” flag in the “sigaction” struct. This is to make sure that the *I/O* will be resumed after the signal handler is executed.
- Signals are **not synchronized**.
 - Imagine updating the list of children while receiving a signal of another child dying. If not synchronized, we will have **data race**
 - Need to block signals via “sigprocmask” syscall while accessing shared data
 - new signals may occur while running the signal handler. We need to **block** the signals while running the signal handler.
- Some functions are signal-unsafe.

- For example, “printf” is not signal-safe. It uses a buffer to store the output. If the signal handler is executed while the buffer is being updated, the buffer will be corrupted.
- We can use “write” instead of “printf” to avoid this problem.

Blocking and unblocking signals:

```
sigset_t mask;
sigemptyset(&mask);
// create an empty set to store the signals
sigaddset(&mask, SIGCHLD);
// block signals before forking
sigprocmask(SIG_BLOCK, &mask, NULL);
pid_t pid = fork();
if (pid == 0) {
    // child process
    // unblock signals in the child process
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    // do something
    exit(0);
}
else {
    // parent process
    // add the child to the list, then unblock signals
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
}
```

Note that we need to add the child to the list before unblocking the signals.

Question: can we delay blocking until after forking? No, we can't. The child process will inherit the signal mask from the parent process. So we need to block the signals before forking.

6.3.3 Signal Summary

- Allow processes to be **notified** of events
- Mediated through **kernel exceptions mechanism**
- **KILL and STOP** are handled by the kernel

- **Others** can be handled by **signal handlers**
- Signals are **not queued**, cannot be used to **count events**
- Most libc functions are **not signal-safe**
- Need to **block signals** while accessing shared data to avoid **data race**

6.4 Exceptions

6.4.1 Control flow

The life of CPU is controlled by the program counter (PC). It is the address of the next instruction to be executed.

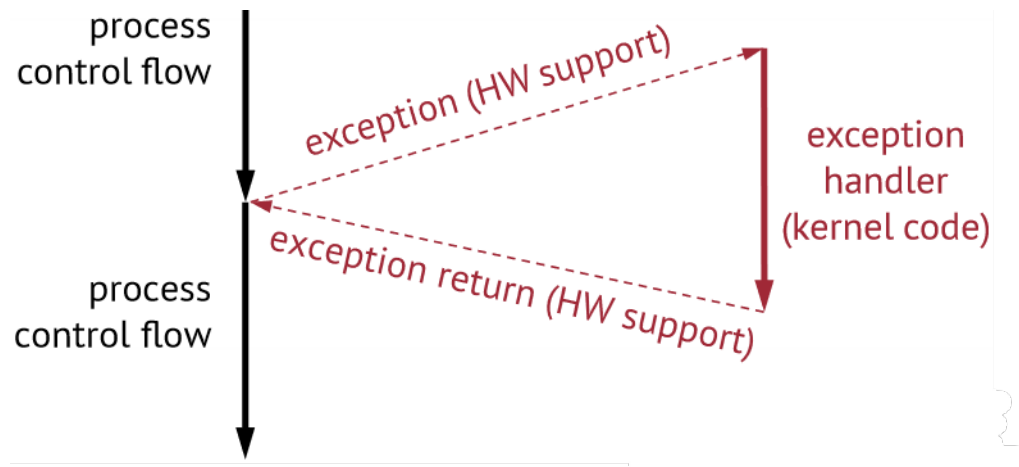
Besides the usual increment of PC we can have

- Conditional/Unconditional jump/branches (if/else, loops, etc.)
- Call/Return (function calls)

It is possible that there are unexpected control flow changes:

- key press (interrupt)
- incoming network packet (interrupt)
- illegal instruction fetched (exception)
- illegal memory access (exception)
- page fault (exception)
- divide by zero (exception)
- etc.

6.4.2 Exception handling



Note that this exception is hardware-based, not software-based exceptions like in Java.

There are two types of exceptions:

- synchronous: caused by the program itself (e.g., divide by zero, illegal instruction, etc.)
- asynchronous: caused by external events (e.g., key press, incoming network packet, hardware timer, etc.)

There are inconsistent terminology use in the industry:

- exception: synchronous or just everything
- interrupt: asynchronous or just everything
- trap: synchronous or just specific debug support

To communicate effectively, better to specify whether it is synchronous or asynchronous.

6.4.3 Privilege level

Exception handlers need more permissions than the program itself. They need to

- access hardware devices directly

- switch/update the page table
- etc.

Normally, the exception handler runs at OS kernel level (ring 0).
CPU has multiple privilege levels.

- For example in ARM64: EL0 (user), EL1 (kernel), EL2 (hypervisor), EL3 (secure monitor)
- Some **instructions** can only be executed at certain privilege levels
- Some **memory** can only be accessed at certain privilege levels
- CPU will fault if the privilege level is not high enough (e.g., trying to access kernel memory from user space, segmentation fault)

6.4.4 Abstraction

- Processor **interrupts** the program
- **Saves** the program counter and the registers
- Control is **transferred** to the exception handler in OS (Interrupt Service Routine, ISR)
- what happens next is **OS-specific** (e.g., kill the process, handle the network packet, write to the disk, context switch, page fault, etc.)

6.4.5 Exception during exception

What if an exception occurs while handling an exception?

- For example, page fault while handling a page fault
- some architecture need handler for this case
- interrupt hardware ensures exceptions are not lost (e.g., nested page fault)
- once device quiesced, it can be re-enabled
- sometimes parts of handlers are re-entrant
- some exceptions cannot be masked (e.g., NMI)

6.5 Debugger breakpoint

Debugger can set breakpoints on

- hardware: CPU will generate an exception when the program counter reaches the breakpoint
 - set CPU control register to breakpoint address
 - when the program counter reaches the breakpoint, the CPU will generate an exception
 - the exception handler will be called
 - hardware watchpoints are similar
 - the exception runs at **escalated privilege level** usually OS level
- software: the program will generate an exception when the program counter reaches the breakpoint
 - replace the instruction at the breakpoint with a special instruction (trap, int 3, etc.)
 - control transfer is non-local, but still within the program
 - no hardware support needed

7 Caches

7.1 The need for Cache

- Temporal locality: if a memory location is accessed, it is likely to be accessed again soon
- Spatial locality: if a memory location is accessed, nearby memory locations are likely to be accessed soon

So we keep cache as a **small, fast memory** to be used repeatedly.

Have **cache line** to store a range of data brought from the memory at once to exploit spatial locality.

7.2 Cache hierarchy

Memory latency $\approx \sqrt{\text{cache size}}$. So we have a hierarchy of caches to reduce the latency.

- L1 cache: small, fast, close to the CPU
- L2 cache: larger, slower, further from the CPU
- L3 cache: even larger, even slower, even further from the CPU
- main memory: large, slow, far from the CPU

7.3 Cache organization

- **Direct-mapped cache:** each memory block can only be stored in one cache line
- **Fully associative cache:** each memory block can be stored in any cache line
- **Set-associative cache:** each memory block can be stored in a set of cache lines

7.4 Replacement policy

Ideal policy: Bélády's optimal algorithm. Replace the cache line that will not be used for the longest time.

However, we can't predict the future. So we use approximation algorithms.

- **Random:** randomly choose a cache line to replace
- **FIFO:** replace the cache line that has been in the cache the longest
- **LRU (Least Recently Used):** replace the cache line that has not been used for the longest time
- **LFU (Least Frequently Used):** replace the cache line that has been used the least

7.5 Using cache effectively

focus on the reuse of L1 cache, then L2 cache, then L3 cache, then main memory.

Take an example: find reuse in matrix multiplication. We can use the idea of blocking to make use of the cache.

Tiling: divide the matrix into small tiles that fit in the cache. Then we can reuse the cache effectively.

- Small tile that fits in L1
- Middle tile that fits in L2
- Large tile that fits in L3

7.5.1 Multicore reuse

Note that in multicore systems, L3 is shared while L1 and L2 are private. So we can use L3 to communicate between cores.

Make sure that each core is visiting the same L3 tile. This way no conflicting cache lines in L3.

7.6 Cache coherence

Invariant: single writer, multiple readers. If a writer writes to a cache line, all the readers should see the update.

Considerations:

- If two cores modify the same cache line, they will ping-pong the cache line between the cores. This is called **cache line bouncing**.
- **True sharing**: two cores modify the same cache line and the same byte.
- **False sharing**: two cores modify the same cache line but different bytes.

7.7 Cache prefetching

Problem: cache misses create long delays. Solution: prefetch the data before it is needed.

- Add **prefetch instructions** to the program, so the CPU will fetch the data before it is needed. But we need to know the access pattern before compile time. This is not always possible.

- Use **hardware prefetchers** to predict the access pattern. The CPU will fetch the data before it is needed. This is more flexible.
 - The prefetching might not have a high hit rate. So we need to be careful with the prefetching.
 - It might evict useful data from the cache. So we turn off the prefetching if it has a low hit rate/confidence.
 - Prefetching needs **off-chip memory bandwidth**. So if not high outcome, would rather use the bandwidth for other purposes.

8 Linking

8.1 The need for linking

When compiling a program, we need to link the object files together to create an executable file.

1. compiles the source code into object files e.g. `gcc -c file.c`
2. link the object files into an executable file e.g. `gcc file1.o file2.o -o file`
3. load the executable file into memory and run it

8.1.1 the hidden pieces

- **crt0**: the C runtime startup code. It initializes the program and calls the main function.
 - defines `_start`
 - sets up the stack
 - initializes the C runtime. e.g. `malloc`, `free`, `printf`, **stack canaries**, **stdio**, **errno**, etc.
 - runs C++ constructors for global objects
 - calls `main`
- **libc**: the C standard library. It provides the standard C functions. e.g. `printf`, `malloc`, `free`, etc.

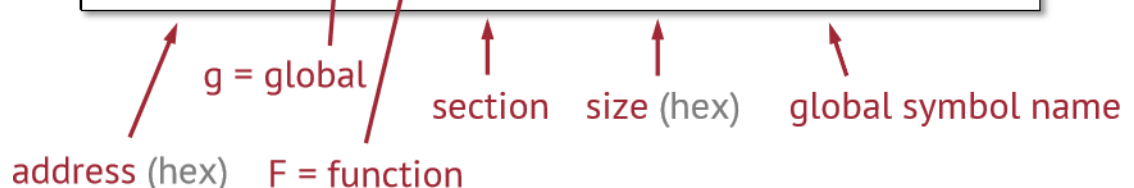
8.1.2 Advantages of linking

- **reusability**: we can reuse the object files in different programs
- **modularity**: we can split the program into different object files
- **efficiency**: we can compile the object files separately and only recompile the changed object files
- **separation of concerns**: we can separate the implementation from the interface
- **dependency management**: we can manage the dependencies between the object files

8.2 Object files

By running `objdump -t xxx.o`, we can see the symbol table of the object file. It contains the symbols and their addresses.

crazylist.o: file format elf64-littlearch64				
SYMBOL TABLE:				
0000000000000000	g	F .text	0000000000000008	enclosing_struct
0000000000000010	g	F .text	0000000000000030	cons
0000000000000000		*UND*	0000000000000000	malloc
0000000000000040	g	F .text	0000000000000008	first
0000000000000050	g	F .text	0000000000000008	rest
0000000000000060	g	F .text	0000000000000028	find
0000000000000090	g	F .text	00000000000000c0	insert_sorted
0000000000000150	g	F .text	0000000000000050	print_list
0000000000000000		*UND*	0000000000000000	__printf_chk
0000000000000000		*UND*	0000000000000000	putchar



crazylist.o: file format elf64-littleaarch64

SYMBOL TABLE:

0000000000000000	g	F .text	0000000000000008	enclosing_struct
0000000000000010	g	F .text	0000000000000030	cons
0000000000000000		*UND*	0000000000000000	malloc
0000000000000040	g	F .text	0000000000000008	first
0000000000000050	g	F .text	0000000000000008	rest
0000000000000060	g	F .text	0000000000000028	find
0000000000000090	g	F .text	00000000000000c0	insert_sorted
0000000000000150	g	F .text	0000000000000050	print_list
0000000000000000		*UND*	0000000000000000	__printf_chk
0000000000000000		*UND*	0000000000000000	putchar

starts at address 0

not defined in crazylist.o

main.o: file format elf64-littleaarch64

SYMBOL TABLE:

0000000000000000	g	F .text	0000000000000088	main
0000000000000000		*UND*	0000000000000000	cons
0000000000000000		*UND*	0000000000000000	print_list
0000000000000000		*UND*	0000000000000000	insert_sorted
0000000000000000		*UND*	0000000000000000	find

also starts at address 0

Note that addresses start at zero. The addresses are relative to the start of the object file. The addresses are not the actual addresses in the memory. Otherwise, we get conflicts and segmentation faults.

To save the Issues, we urge for linkers:

- **resolve the addresses:** replace the relative addresses with the actual addresses
- **resolve the symbols:** replace the symbols with the actual addresses
- **combine the object files:** combine the object files into an executable file
- **resolve the relocations:** replace the relative addresses with the actual addresses

8.3 Static linking

- resolution and relocation are done at **link time**
- the linker combines the object files into an executable file
 - relocatable object files: `.o`

- * code/data/debug info → combined with other `.o` files → executable
- * static libraries: `.a` (archive)
 - collection of `.o` files
 - `ar` to create and manipulate
 - `ranlib` to create an index
- shared object files: `.so` (shared object)
 - * linked `.o` files → dynamic libraries before or during runtime

8.3.1 Meta Data

ELF header
program header
.text section
.rodata section
.data section
.bss section
.symtab section
various rel, plt, got sections
.debug section
section header

- **ELF header:** the header of the executable file. It contains the meta data of the executable file. e.g. the type of the file, the architecture of the file, the entry point of the file, word size, endianness etc.
- **program header:** the header of the program. It contains the meta data of the program. e.g. the type of the program, the offset of the program, the virtual address of the program, the physical address of the program, the size of the program, the alignment of the program, etc.
- **.text section:** the code of the program. It contains the instructions of the program. It is read-only.
- **.rodata section:** the read-only data of the program. It contains the constants of the program. It is read-only.
- **.data section:** the data of the program. It contains the global variables of the program. It is read-write.
- **.bss section:** the uninitialized data of the program. It contains the uninitialized global variables of the program. It is read-write.
- **.symtab section:** the symbol table of the program. It contains the symbols of the program. e.g. the functions of the program, the variables of the program, etc.
- **various rel, plt, got sections:** the relocation, procedure linkage table, global offset table sections of the program. They contain the relocation, procedure linkage table, global offset table of the program. They are used for dynamic linking.
- **.debug section:** the debug information of the program. It contains the debug information of the program. e.g. the source code of the program, the line numbers of the program, the types of the program, etc. Only present in the debug version of the program, e.g. the program compiled with the -g flag.
- **section header:** section types/locations → linker → program

8.3.2 Symbol resolution

- find references to **unresolved symbols** in the object files, e.g. `printf` (functions from libraries/libc)

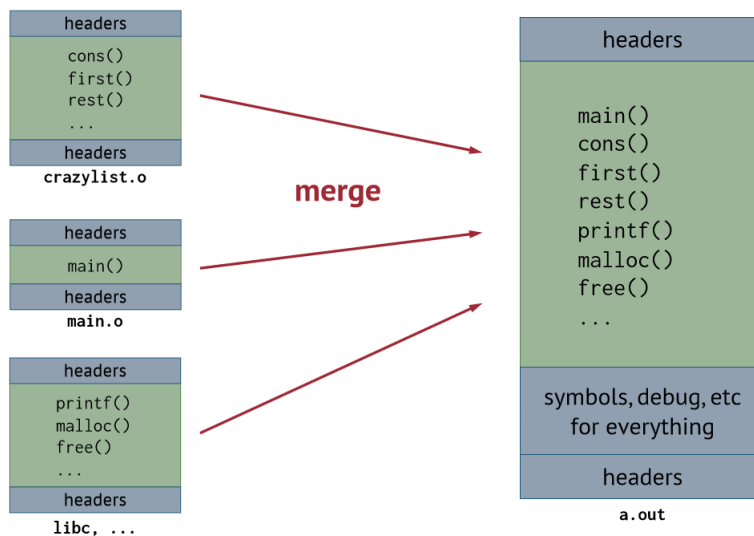
- find matching entries in the **symbol table** of the object files, e.g. `printf`
→ `printf@GLIBC_2.2.5`
- replace each reference with a **unique definition** (or fail)
 - if the symbol is not found, the linker will fail
 - if the symbol is found, the linker will replace the reference with the definition
 - if the symbol is found in multiple object files, the linker will fail

8.3.3 symbol resolution procedure

1. load **symbol table** from each object file
2. for each **referenced** but **unresolved symbol** in the object files
 - find a **matching definition** in the symbol tables
 - replace the reference with the definition
 - if the symbol is not found, the linker will fail
 - if the symbol is found in multiple object files, the linker will fail

8.3.4 Relocation

- **merge** code and data from multiple object files into one executable



- **move** each symbol to a unique address in the executable (non-overlapping)

```
$ objdump -r crazylist.o

crazylist.o:      file format elf64-littleaarch64

RELOCATION RECORDS FOR [.text]:
OFFSET            TYPE            VALUE
0000000000000028  R_AARCH64_CALL26  malloc
00000000000000b8  R_AARCH64_CALL26  malloc
00000000000000ec  R_AARCH64_CALL26  malloc
0000000000000128  R_AARCH64_CALL26  insert_sorted
000000000000013c  R_AARCH64_CALL26  malloc
0000000000000160  R_AARCH64_ADR_PREL_PG_HI21  .rodata.str1.8
0000000000000168  R_AARCH64_ADD_ABS_LO12_NC  .rodata.str1.8
000000000000017c  R_AARCH64_CALL26  __printf_chk
0000000000000194  R_AARCH64_JUMP26  putchar
000000000000019c  R_AARCH64_JUMP26  putchar
```

relocation info for
our malloc call

relocation info for
computing data address
(adrp + add pair)

ld uses this table to
replace addresses
in the object

- **replace** each reference to a symbol with the address of the symbol

**actual
addrs**

```
$ objdump -dcr crazylist_test
...
0000000000400770 <cons>:
...
400784:      d2800200      mov     x0, #0x10
400788:      94004cde      bl      413b00 <__libc_malloc>
40078c:      f9000414      str     x20, [x0, #8]
...
00000000004008b0 <print_list>:
...
4008c0:      f00002b4      adrp    x20, 457000 <_nl_unload_domain+0xa0>
4008c4:      aa0003f3      mov     x19, x0
4008c8:      9125e294      add     x20, x20, #0x978
...

Contents of section .rodata:
457970 01000200 00000000 256c7520 00000000 ..... %lu ...
...
data at
0x457978
```

real function address to call

objdump tryin' to be too clever

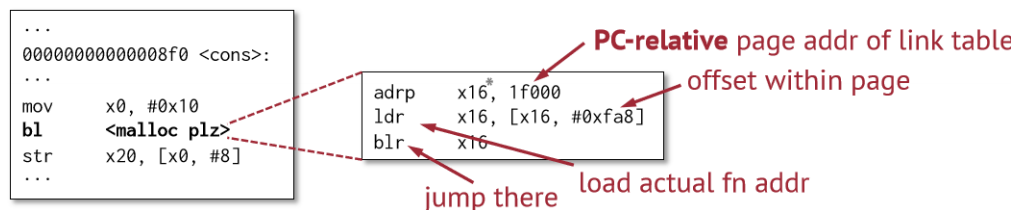
8.4 Dynamic linking

- resolution and relocation are done at **runtime**
- the linker creates a **shared object file** (e.g. `.so`)
 - the shared object file contains the object files of the program
 - the shared object file contains the object files of the libraries

- the loader loads the shared object file into memory and resolves the symbols at runtime
- the loader relocates the symbols at runtime
- the loader runs the program

8.4.1 Challenges for dynamic linking

- do not **copy** the library code.
 - To solve the problem, map existing library **somewhere** in our virtual address space
 - Put actual addresses in library's symtab
- do not **edit** the code memory.
 - startup would take too long
 - only worse because we would keep missing in the caches
 - want to make code non-writable to avoid attacks
 - To solve the problem, pick **unique offset** for each symbol.
 - Create an **indirection table** to map the symbol to the actual address
 - Make all function calls **indirect** through the table via **PLT** e.g. (blr)
 - dynamic linking: fill the table with the actual addresses
 - Illustration of indirection via global offset table (GOT):



8.4.2 Lazy binding

GOT still has to fill the offset table, which takes a long load time. So we use lazy binding to fill the GOT only when the function is called.

Lazy binding:

- initially, each offset table entry calls the **resolver** function
- the resolver function finds the actual address of the symbol in ELF file
- the loader puts the resolver address in the GOT
- the first time the function is called, the resolver is called
- resolver finds address in the lib's symtab, fills the GOT with the actual address

However, large overhead for the first call. Solve it with another indirection table, **PLT** (Procedure Linkage Table).

8.4.3 Procedure Linkage Table (PLT)

- linker **generates** a PLT entry for each function call
- updates PC-relative bl offsets to call the PLT entry
- also generates the resolver code (PLT resolver)

1. our code jumps to PLT entry for the function
2. PLT entry loads address from GOT, jumps to it
3. initially, GOT points to PLT resolver
4. PLT resolver finds the actual address of the function, fills the GOT with the actual address
5. next time, GOT points to the actual address

8.4.4 plugin

Idea: want to load and link additional libraries at runtime.

```

#include <dlfcn.h>
...
int main(int, char *) {
    ...
    void *lib = dlopen("./libmymath.so", RTLD_LAZY); // + err checking
    int (*f)(int, int) = dlsym(lib, "myadd"); // + err checking
    int ans = f(1, 1);
    dlclose(lib);
    ...
}

```

open library file (e.g., plugin)
 find addr of myadd()
 call myadd() by addr
 unload libmymath.so

Interposition: replace a function in a library with a function in another library.
 e.g. replace `malloc` with `my_malloc`.

Runtime Interposition:

- can tell dynamic linker to preload a library `LD_PRELOAD=/some/lib.so ./a.out`
- first checks `lib.so` then other libraries *to* can intercept calls to any function.

9 Dynamic dispatch

9.1 The need for dynamic dispatch

- **polymorphism:** the ability to treat objects of different types in a uniform way
- **dynamic binding:** the ability to call the right function at runtime
- **inheritance:** the ability to inherit the properties of a base class

9.2 Closure

So C has function pointers, but to use them in certain environments, we need to store some variables along with the function pointer. This is called a closure.

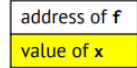
- **function pointer:** a pointer to a function
- **function object:** an object that behaves like a function

- **closure**: a function object that captures the environment

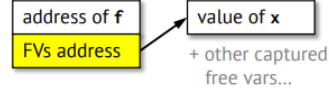
fn pointer:



simple closure:



more usual w/ first-class functions:



```
typedef struct cl_s {
    long (*fn) (struct cl_s *f, long x);
    long y;
} cl_t;
```

```
long _f(cl_t *cl, long x) {
    return x + cl->y;
}
```

```
cl_t add (long y) {
    cl_t cl = {_f, y};
    return cl;
}
```

```
int main() {
    cl_t add10 = add(10);
    add10.fn(&add10, 5); // 15

    cl_t add2 = add(2);
    add2.fn(&add2, 5); // 7
}
```

Have a look at assembly level:

```
_f:
    // load cl->y into x0
    ldr x0, [x0, #8]
    // add cl->y, x
    add x0, x0, x1
    ret
\dots
// calling the closure in x20 with argument 5
// cl address in x0
```

```

mov x0, x20
// argument 5 in x1
mov x1, 5
// call the closure
ldr x23, [x20]
// jump to the function pointer
blr x23

```

9.2.1 C++ objects and methods

Each object has **data** and **methods**. The methods are the functions that operate on the data. The methods are stored in the **virtual function table** (vtable).

Looks very like closure

9.3 Name mangling

Same field/method name can show up many times in different classes. So we need to differentiate them. This is called name mangling.

For example:

- `int foo(int x) → _Z3fooi`
- `int add(int x, int y) → _Z3addii`
- `double add(double x, double y) → _Z3adddd`
- `char* cat(char *x, const char *y) → _Z3catPcPKc`
- `void noarg() → _Z5noargv`
- `int Cls::meth(int x, int y) → _ZN3Cls4methEii`

The rule is as follows:

- start with `_Z`
- number of characters in the name and the name
 - If it is C++ and the method is in a class, the number of characters includes the class name
 - Start with N and end with E

- Between N and E is the number of characters in the class name, followed by the class name, then the number of characters in the method name, followed by the method name. e.g. `_ZN3Cls4methEii`
- the types of the arguments
- the return type is not needed here since in C/C++ we can have the same name and arguments with different return types

9.4 Objects with only non-virtual methods

Start with an example:

```
struct animal{
    int age;
    const char *name;
    animal(const char *, int);
    void say_name();
}

animal::animal(const char *name, int age){
    this->name = name;
    this->age = age;
}

void animal::say_name(){
    puts(name);
}

int main(){
    animal a("cat", 5);
    a.say_name();
}
```

9.4.1 Members of this object

- **Data fields:** age (4B, 32bit int), name(8B, 64bit pointer). Since 8B alignment, the total size is 16B, there is a 4B padding between age and name.

- **Methods:** `animal(const char *, int), say_name()`.
 - In a real implementation, there are actually two constructors. This is because it might become a virtual base class in the future. Inheritance!
- **Argument counts:** Besides the obvious arguments, there is a hidden argument, the `this` pointer. It is a pointer to the object itself. It is passed as the first argument to the method. So the actual arguments are `this`, `"cat"`, `5` for the constructor and `this` for the method.
- Since it is not virtual, it is resolved at compile time through name mangling. So the method is called directly.

9.4.2 Assembly

Let's say we call the constructor and the method in the main function.

```
int main(){
    animal a("cat", 5);
    a.say_name();
}
```

The assembly code is as follows:

```
_ZN6animalC2EPKci:
    // three arguments: this, age, name
    // Calling convention: x0, x1, x2
    str w2, [x0]
    str x1, [x0, #8]
    ret
_ZN6animal8say_nameEv:
    // one argument: this
    // Calling convention: x0
    ldr x0, [x0, #8]
    bl puts
    ret
main:
    // allocate space for the object
    sub sp, sp, #16
    // call the constructor
    mov x0, sp
```

```

mov x1, "cat"
mov x2, 5
bl _ZN6animalC2EPKci
// move the object to x0
mov x0, sp
bl _ZN6animal8say_nameEv
mov x0, 0
mov x8, 93

```

Note how the `this` pointer is passed as the first argument to the method.

9.4.3 Inheritance with non-virtual methods

Make a dog class that inherits from the animal class.

```

struct dog: animal{
    dog(const char *, int);
    void say_name();
}
dog::dog(const char *name, int age): animal(name, age){}

void dog::say_name(){
    animal::say_name();
    puts("woof");
}

int main(){
    dog d("dog", 3);
    d.say_name();
    animal *a = &d;
    a->say_name();
}

```

Expected output:

```
dogwoofdog
```

- No additional data fields for the dog class. It inherits the data fields from the animal class.
- The dog class has two methods: the constructor and the `say_name` method.

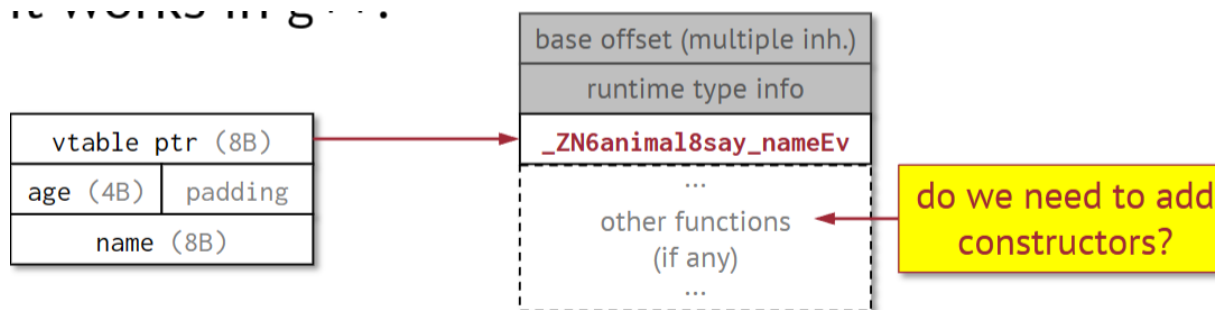
- Now we have two versions of the `say_name` method: the animal version and the dog version. The dog version calls the animal version and then prints “woof”
- The `say_name` method is resolved at compile time through name mangling. So the method is called directly. The first one thinks `d.say_name()` is a dog, but `a->say_name()` is an animal. So compiler would call different methods.

9.5 Virtual methods

What if we do not know the function addresses at compile time? We need to use virtual methods.

We might need to store the function addresses in the object itself just like closure. This is called a **vtable**.

9.5.1 Vtable



Note that constructors and destructors are not virtual. They are resolved at compile time.

- **vtable**: a table of function pointers
- **vp**: a pointer to the vtable
- **vtable entry**: a function pointer in the vtable
- **vtable offset**: the offset of the vtable entry in the vtable (the compiler will calculate this)

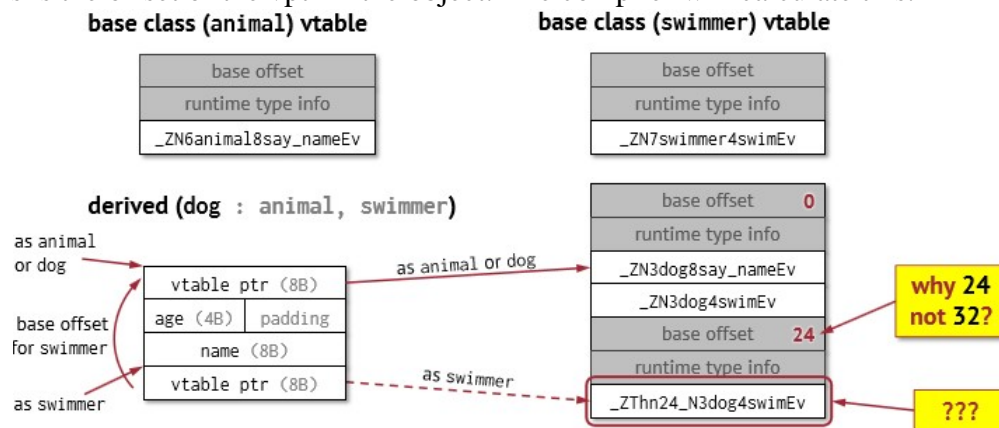
9.6 Inheritance

We can treat subclasses as their base class. This is called **upcasting**. This is because the structure of the subclass is the same as the structure of the base class with additional fields and methods that is below the base class in memory space. By upcasting, we can call the base class methods on the subclass object ignoring the additional fields and methods.

9.6.1 Multiple inheritance

What if a subclass inherits from multiple base classes? We need to store multiple vptrs in the object. This is called **vptra chaining**.

To distinguish between the vptrs, we need to store the **vptra offset** in the object. This is the offset of the vptr in the object. The compiler will calculate this.



The vtable offset is 24 here since it is the vtable pointer from the object header instead of the offset in vtable.

- When the object is casted as **dog** or **animal**, the vptr is set to the vptr of the actual top of the object.
- When the object is casted as **swimmer**, the pointer would be moved to the actual swimmer part of the object, in this case, -24 bytes by the offset.
- Note that you cannot cast a parent to its child.

9.7 Traits

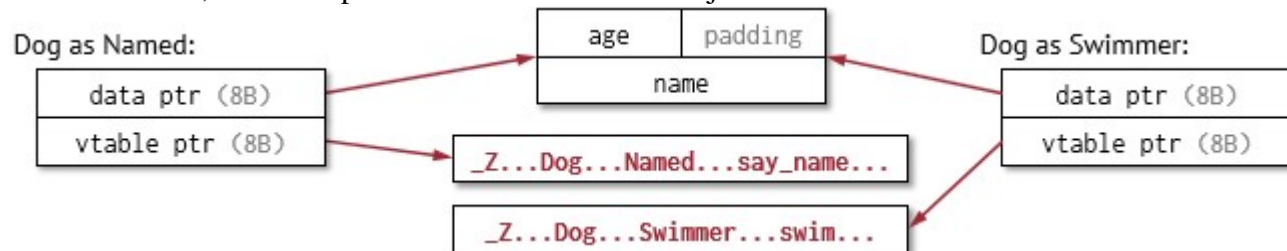
Let's say we want an object to add some other functionality after compiling, then we cannot modify the vtable to do it since memory space is set. We can use traits

to add functionality.

Give an example in Rust:

```
struct Dog {age: i32, name: String}
trait Named {fn say_name(&self);}
impl Named for Dog {
    fn say_name(&self){
        println!("{}", self.name);
    }
}
// a bunch of other codes
trait Swimmer {fn swim(&self);}
impl Swimmer for Dog {
    fn swim(&self){
        println!("{}", self.name);
    }
}
```

To achieve this, we can separate the vtable from the object.



Keep data in a separate place, then for each trait of the object, we keep a data pointer and a vtable pointer. If we want to add a trait, we can add a new vtable pointer and data pointer.

When the corresponding trait is called, we can call the vtable pointer to the function.

10 Concurrency

10.1 The need for concurrency

- **parallelism:** the ability to run multiple tasks at the same time
- **concurrency:** the ability to run multiple tasks in an interleaved manner

- **asynchronous programming**: the ability to run tasks in the background
- **event-driven programming**: the ability to run tasks in response to events
- **multithreading**: the ability to run multiple threads in the same process
- **multiprocessing**: the ability to run multiple processes in the same system
- **distributed computing**: the ability to run multiple processes in different systems
- **parallel computing**: the ability to run multiple tasks in parallel on multiple cores
- **concurrent computing**: the ability to run multiple tasks concurrently on a single core

10.2 Threads

Threads have their own logical control flow, registers, stack, and program counter. They share the same memory space, file descriptors, and other resources.

10.2.1 Thread creation

Process → main thread → create new threads. Upon context switch, the CPU will switch to the new thread by changing the program counter and the stack pointer, then save the old thread's registers and stack.

Here are some example functions of creating threads in C:

```
// the thread function we want it to run
void *function(void *arg){
    // do something
    return NULL;
}

// create a thread
int pthread_create(pthread_t *thread, // thread id
const pthread_attr_t *attr, // scheduling info, NULL for default
void *(*start_routine) (void *), // the function to run
void *arg); // the argument to the function
```

```

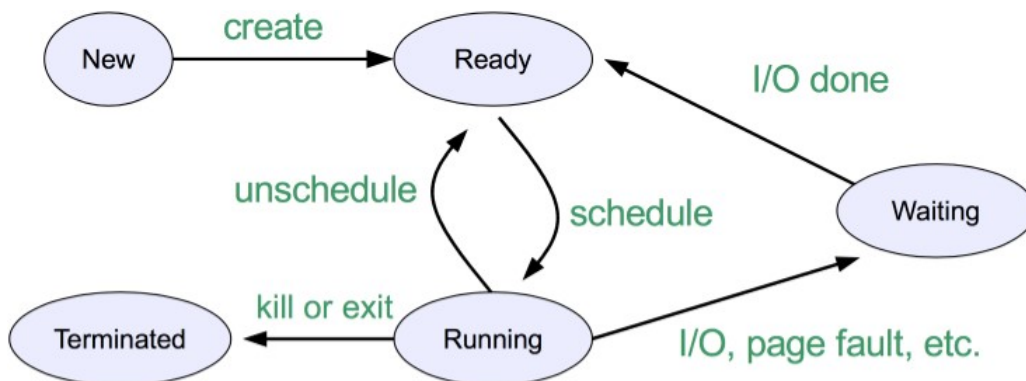
// wait for a thread to finish
int pthread_join(pthread_t thread, // the thread id
void **retval); // the return value of the thread can be found here

// exit the current thread
void pthread_exit(void *retval); // the return value of the thread

// detach a thread, OS will clean up the thread when it finishes
int pthread_detach(pthread_t thread); // the thread id
// So we don't need to call pthread_join,
// do not know the return value

```

10.3 State transitions



Processes and threads have similar transition states. The difference is that threads share the same memory space while processes do not.

10.4 Thread safety

- **reentrant**: a function that can be called by multiple threads at the same time
- **thread-safe**: a function that can be called by multiple threads at the same time without causing
- **Data race**: two threads access the same memory location at the same time, at least one of them is writing

- **Deadlock:** two threads are waiting for each other to release a lock
- **Livelock:** two threads are waiting for each other to release a lock, but they keep trying to acquire the lock
- **Starvation:** a thread is waiting for a lock, but other threads keep acquiring the lock
- **Priority inversion:** a high-priority thread is waiting for a low-priority thread to release a lock

10.4.1 Atomic operations

Atomic read-modify-write operations are operations that are guaranteed to be executed as a single operation. They are not interrupted by other threads.

For example:

- `ldadd x0, x1, [x2] → x1 = *x2; x1 += x0; *x2 = x1` which will be executed as a single operation
- `cas x0, x1, [x2] → if (*x2 == x0) *x2 = x1; x0 = *x2`
Compare and swap. x0 will get the value nevertheless.

10.4.2 Pure functions

- **pure function:** a function that does not have side effects
 - the function always returns the same value for the same arguments
 - any mathematical function is a pure function
 - `rand()` is not a pure function since it modifies the state of the random number generator and returns a different value each time
 - Any pure function is thread-safe
- **side effect:** a change in the state of the program

How to make impure functions thread safe? Have a look at the following example:

```
long int random (void){
    long int retval;
    __libc_lock_lock (lock);
```

```

        retval = rand();
        __libc_lock_unlock (lock);
        return retval;
    }

```

Although `rand()` is not thread-safe, we can make it thread-safe by locking the function.

10.4.3 Locks

One thread **may access** shared vars at a time

Libc provides some lock functions:

```

// create a lock
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);
// attr gives options like recursive, error checking, etc.

// destroy a lock
int pthread_mutex_destroy(pthread_mutex_t *mutex);

// lock a lock
int pthread_mutex_lock(pthread_mutex_t *mutex);

// unlock a lock
int pthread_mutex_unlock(pthread_mutex_t *mutex);

// try to lock a lock
int pthread_mutex_trylock(pthread_mutex_t *mutex);

```

Although it is safe, it limits the concurrency to sequentially execute. So we need to use locks wisely.

10.5 implementation of locks

Problems when implementing locks:

- Only one thread can acquire the lock at a time
 - Use **atomic** operations to acquire the lock e.g. `cas`

- Some architectures do OoO (Out of Order) execution
 - Use **memory barriers** to prevent OoO execution

10.5.1 Naive spin-lock

```
lock:
    adr x0, mutex
    mov x1, 1
spin:
    mov x2, 0
    // try to acquire the lock
    cas x2, x1, [x0]
    // if failed, try again
    cmp x2, 0
    bne spin
    // mem barrier
    dmb sy
unlock:
    adr x0, mutex
    // finish all the operations before releasing the lock
    dmb sy
    str xzr, [x0]
```

This is a correct implementation but not efficient. Here is why:

- Different cores have their own L1 cache.
- Architecture enforces cache coherence. If one core writes to a cache line, the other cores will invalidate the cache line.
- `cas` tends to write to the cache line. For each `cas`, the cache line will be invalidated and reloaded.
- This is called **cache line bouncing**.
- Although no progress, it is still consuming CPU cycles. OS would think it is busy not noticing the cache line bouncing.

10.5.2 A slightly better spin-lock

```
lock:
    adr x0, mutex
    mov x1, 1
spin:
    // read the lock instead of writing
    ldr x2, [x0]
    cmp x2, 0
    bne spin
    // looks like the lock is free, try to acquire it
    // try to acquire the lock
    cas x2, x1, [x0]
    // if failed, another thread acquired the lock
    cmp x2, 0
    // note how it go back to orgin
    bne spin
    dmb sy
```

- Reading would not invalidate the cache line, it make sure that others do not write as well.
- So no cache line bouncing.

10.6 Deadlocks

- **deadlock**: two threads are waiting for each other to release a lock
- **livelock**: two threads are waiting for each other to release a lock, but they keep trying to acquire the lock
- **starvation**: a thread is waiting for a lock, but other threads keep acquiring the lock
- **priority inversion**: a high-priority thread is waiting for a low-priority thread to release a lock

10.6.1 Deadlock prevention

- **lock ordering**: always acquire locks in the same order (topological sort)
- **lock hierarchy**: always acquire locks in the same hierarchy
- **lock timeout**: acquire locks with a timeout note that this needs to wait for a **random and increasing** time. Otherwise, it would be a livelock.
- **lock try**: try to acquire locks without blocking
- **lock yield**: yield the CPU if the lock is not available
- **lock sleep**: sleep if the lock is not available

10.6.2 Waiting on locks

It is bad if we deschedule a thread while holding a lock. This way other threads cannot do much if their critical section is locked. So we need to release the lock before scheduling.

11 File systems

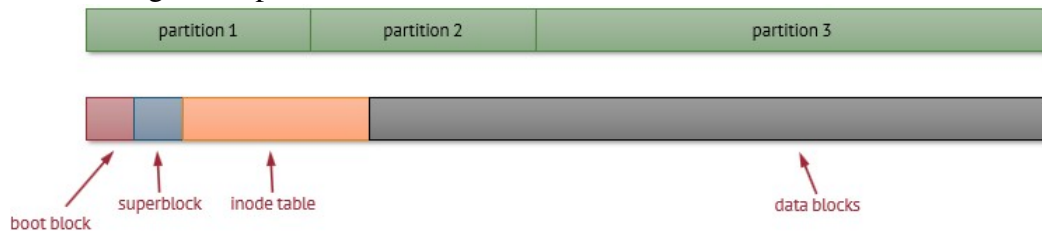
Under the **unix** abstraction:

- **everything is a file**: files, directories, devices, etc.
- a file is a sequence of bytes
 - a file *might* be stored on a disk, or just in memory, or just nowhere
 - a file *might* have a name/path, or just nothing
 - a file *might* support random access (by offset), or just sequential access

Files are just binary blobs. The OS does not know what is in the file. It is the application's responsibility to interpret the file.

11.1 Disk file systems

Disks are divided into **blocks** of fixed size. The OS reads and writes blocks. The file system manages the blocks. **Partitions** are contiguous blocks on the disk. **File systems** are organized partitions.



Each partition are made of

- **boot block:** the first block of the partition. It contains the boot loader which is the info for loading the OS when the machine starts.
- **superblock:** the second block of the partition. It contains the meta data of the file system. e.g. the size of the partition, the size of the blocks, the number of blocks, the number of inodes, etc.
- **inode table:** the third block of the partition. 1 entry per disk file, info + which blocks are used (but not file names).
- **data blocks:** the rest of the blocks of the partition. They contain the data of the files.

11.1.1 Inodes

- **inode:** a data structure that contains the meta data of a file. e.g. the size of the file, the owner of the file, the permissions of the file, the timestamps of the file, the pointers to the blocks of the file, etc.
- **inode number:** a unique number that identifies the inode
- **inode table:** a table that contains the inodes of the files
- **inode bitmap:** a bitmap that contains the free inodes

Inode example:

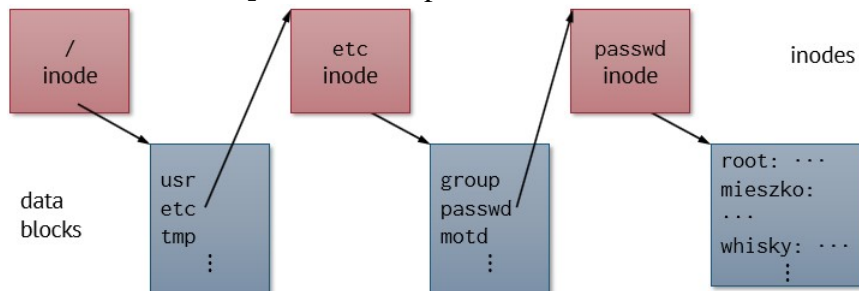
```
$ ls -li 1337.txt
35139129 1337.txt
```

```
$ sudo debugfs -R "stat <35139129>" /dev/sda1
Inode: 35139129  Type: regular  Mode: 0600  Flags: 0x80000
Generation: 2077674002  Version: 0x00000000:00000001
User: 1001  Group: 1001  Project: 0  Size: 15
File ACL: 0
Links: 1  Blockcount: 8
Fragment:  Address: 0  Number: 0  Size: 0
ctime: 0x62303552:87411474 -- Mon Mar 14 23:42:26 2022
atime: 0x62303552:87411474 -- Mon Mar 14 23:42:26 2022
mtime: 0x62303552:87411474 -- Mon Mar 14 23:42:26 2022
crtime: 0x62303552:87411474 -- Mon Mar 14 23:42:26 2022
Size of extra inode fields: 32
EXTENTS:
(0):140544014
```

Remark: You can run out of either inodes or data blocks. Check by running: `df -i` and `df -h`.

11.2 Directories

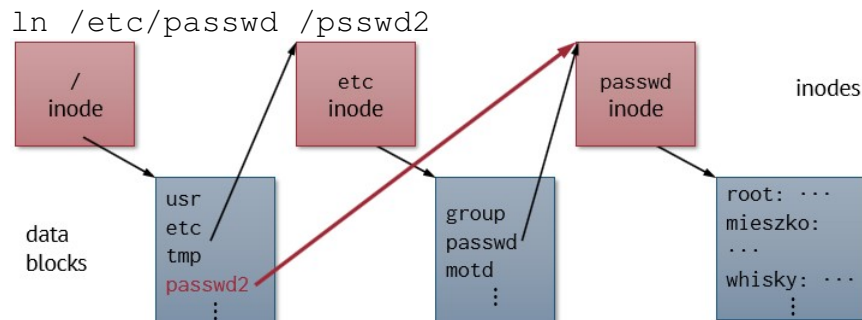
Let's find how `/etc/passwd` is represented:



- `/` is the root directory which contains all the other directories (address) as its data blocks.
- `etc` is a subdirectory that contains more files and directories.
- `passwd` is a file that contains the user information

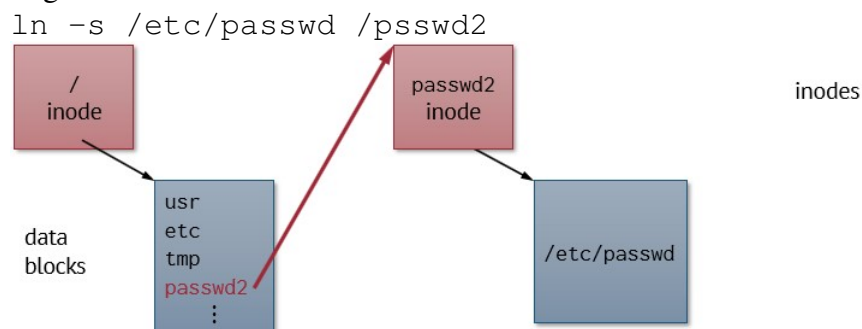
11.2.1 Hard links

A link that points to the **same** inode as another file. It is just another name for the same file.



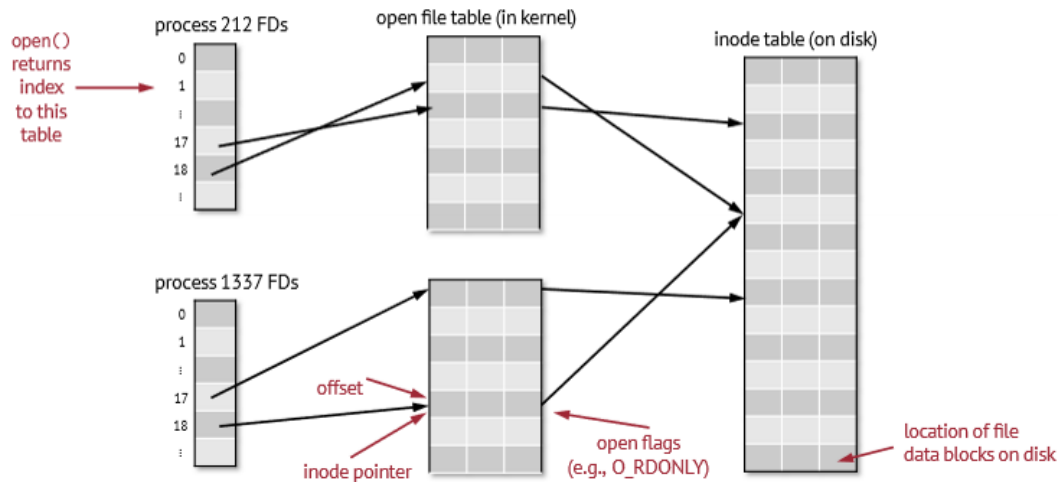
11.2.2 Soft links

A link that points to the **path** of another file. It is a file that contains the path of the target file.

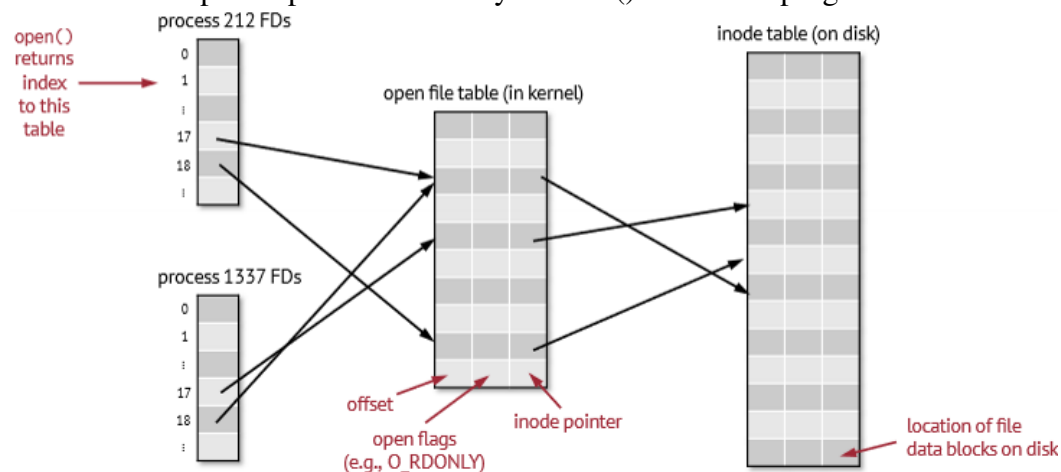


11.3 File descriptors

- **file descriptor:** a number that identifies an open file
- each process has own file descriptor indices
- these point to kernel open file table
 - multiple FDs can point to the same **file table entry**
 - child process inherits stdout by default
- each disk file may be opened multiple times



Note that parent and child processes can share a file table if the child process is forked from the parent process and not yet `execv()` to another program.



11.4 Pipes

Pipes are file descriptors that allow processes to communicate. They are unidirectional. It has no disk storage.

For example, a shell command: `ps | aux | grep root` builds a pipe chain from `ps` `aux` to `grep` `root`.

11.5 File system operations

There are two types of operations:

- syscall interface (operating system calls)
 - not portable
 - file descriptors identify open files
 - e.g. `open`, `read`, `write`, `close`
 - only cares about the binary blobs
 - **safe** for signal handlers
 - often *incorrectly* called unbuffered I/O, but it is buffered
- C library interface (`stdio.h`)
 - portable
 - `FILE` pointers identify open files
 - e.g. `fopen`, `fread`, `fwrite`, `fclose`
 - cares about the text/binary distinction
 - generally **unsafe** for signal handlers
 - usually buffered I/O

Note that file systems do not protect against synchronization. It is the application's responsibility to synchronize the access to the file system.

11.5.1 Buffereing

- **amortizes** cost of device access (bring in a block, not a byte)
- first write to buffer, then flush to disk
- caveats:
 - Programs can crash before the buffer is flushed, so that data is lost
 - makes **I/O guarantees** difficult. what if disk losses power? what if you sync a file but not the directory it is in?
 - might delay error messages. (e.g., if I/O error occurs, the error message might be delayed until the buffer is flushed)

11.6 File positions

File positions are maintained per file descriptor. They are updated after each read/write operation. They are used to determine where to read/write next.

- **file position:** the current position in the file
- **seek:** move the file position
- **read/write:** read/write from the file position
- **append:** write to the end of the file
- **truncate:** truncate the file to a certain size

Note that not all devices support seek. For example, you cannot seek on a pipe.

11.7 File redirection

11.7.1 `stdin`, `stdout`, `stderr`

Application convention:

- `stdin`: standard input (0)
- `stdout`: standard output (1)
- `stderr`: standard error (2)

By default, the shell connects these to the terminal. Note that it is not a kernel-level convention.

11.7.2 duplicate file descriptors

Redirect `stdout` to a file: `ls > file.txt`

- `dup`: duplicate a file descriptor
- `dup2`: duplicate a file descriptor to a specific file descriptor

These are used to redirect file descriptors. For example, `dup2(fd, 1)` will redirect the file descriptor `fd` to `stdout`.

11.8 Blocking vs non-blocking I/O

- **blocking I/O**: the process waits until the I/O operation is complete
- **non-blocking I/O**: the process does not wait. It continues to do other things

Open file descriptor as non-blocking: `open("file.txt", O_NONBLOCK)`

Or change it later with `fcntl(fd, F_SETFL, O_NONBLOCK | fcntl(fd, F_GETFL))`

If we do it to `read()` or `write()`, it will return `EAGAIN` if the operation is not ready.

11.9 Polling

Sometimes, we want to check **many fds** like a server with many clients. We can use `poll()` to check many fds at once.

Polling tells the OS which fds we want to check. The OS will notify when any is modified.

11.9.1 poll interface

```
struct pollfd {
    int fd; // file descriptor
    short events; // events to check
    short revents; // events that occurred
};

int poll(struct pollfd *fds, // array of fds
         nfds_t nfds, // number of fds
         int timeout); // timeout in ms
```

For example:

```
struct pollfd fds[1];
fds[0].fd = fd; // file descriptor to check
fds[0].events = POLLIN; // check for input
// wait for 10 ms
while (poll(fds, 1, 10) == 0) {
    // do something else as waiting
}
```

```
// get input from the file descriptor
```

11.10 Memory mapped file I/O

Use **virtual memory** to map a file to memory. The OS will load the file into memory when the program accesses the memory. This is called **demand paging**.

```
void *mmap(void *addr, // address to map to  
           size_t length, // length of the mapping  
           int prot, // protection (read/write/execute)  
           int flags, // flags (e.g. MAP_SHARED)  
           int fd, // file descriptor  
           off_t offset); // offset in the file
```

```
int munmap(void *addr, // address to unmap  
           size_t length); // length of the mapping
```

There are stuff that you cannot map like directories, pipes, etc. You can map a file to memory and then modify the memory to modify the file. This is called **memory-mapped file I/O**.

For example:

```
int fd = open("file.txt", ORDWR);  
uint8_t *p = mmap(NULL, // address to map to  
                  // If NULL, the OS will choose the address which is better.  
                  4096, PROT_READ | PROT_WRITE, // read/write  
                  MAP_SHARED, // shared with other processes  
                  fd, // file descriptor  
                  0); // offset  
uint8_t c = p[1]; // read the second byte  
munmap(p, 4096);  
close(fd);
```

12 Networking

12.1 local networking

- **MAC**: Media Access Control, a unique identifier for network interfaces

- Data organized in **packets** with source and destination MAC addresses
- Every host must have a unique MAC address
- Packets broadcast to all hosts on the network
- Hosts ignore packets not addressed to them and selectively read packets addressed to them

12.1.1 Bridged networking

- **bridge**: a device that connects two networks. Bridge has MAC address on each end of the network
- **bridged networking**: a network configuration where two networks are connected by a bridge
- MACs still unique across entire bridged network, packets forwarded based on MAC address
- **MAC lookup table**: a table that maps MAC addresses to ports

12.1.2 Ethernet frames

- **Ethernet frame**: a packet of data on an Ethernet network
- **preamble**: a sequence of bits that indicates the start of a frame
- **destination MAC address**: the MAC address of the destination host
- **source MAC address**: the MAC address of the source host
- **type**: the type of the frame (e.g. IP, ARP, etc.)
- **data**: the data of the frame
- **FCS**: Frame Check Sequence, a sequence of bits that checks the integrity of the frame

12.2 IP networking

Local networking is not decentralized and scalable. We need a global addressing scheme. This is where IP networking comes in.

12.2.1 An internet

- **internet:** a network of networks, Connects many networks together
- MAC addresses are not unique across the internet, only unique within a network
- **IP address:** a unique identifier for a host on the internet, IP stands for Internet Protocol
- **IP packet:** a packet of data on an IP network
- traffic routed based on IP address
- **router:** a device that connects networks and routes traffic between them

12.2.2 Routing

- **routing:** the process of forwarding packets from one network to another
- **routing table:** a table that maps IP addresses to ports
- **default route:** a route that is used when no other route matches
- **routing protocol:** a protocol that routers use to communicate with each other

Note that if the destination is a few hops away, the packet will be forwarded to the next hop and the next hop might not be a fixed route. This means that packets might take different routes to the same destination.

Now, this might bring up another problem: the network graph might contain cycles. We use **time-to-live** (TTL) to prevent packets from looping forever. The TTL is decremented at each hop and the packet is dropped when the TTL reaches 0.

If packets dropped, the sender will retransmit the packet. This is called **reliable delivery**.

12.3 Packets structure

The idea is to encapsulate the data in different layers. For example, the data is encapsulated in an Ethernet frame, then the Ethernet frame is encapsulated in an IP packet, then the IP packet is encapsulated in an Ethernet frame.