Systems Programming and Computer Architecture

The C Programming Language

Declaration: Says that something exists, somewhere.

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

Definition: Describes the exact implementation.

```
char *strncpy(char *dest, const char *src, size_t n) {
    // some implementation here
    return dest
};
```

Compilation Unit: A C file, plus everything it includes.

`extern`: Definition is somewhere else, either in this comilation unit or in another.

`static`: The Definition is in this compilation unit, and it can't be seen outside it. (Attention: `static` inside a block means that the value persists between calls)

Interface of a module: Everything externally visible: Functions, typedefs and macros. Interfaces are specified with header files.

GNU `gcc` Toolchain

1. Preprocessor
   - Includes the header files
   - Macro substitution
2. Compiler
   - Compiles each file into assembly
3. Assembler
   - Assembles each file into object files
4. Linker
   - Links the object files into a program binary

Integers

- Both signed and unsigned integers wrap around if an overflow/underflow occurs.
- Both signed and unsigned integers implement modular arithmetic.
- Both signed and unsigned integers form a commutative ring, they are isomorphic.
- Integer division is biased towards zero.

Pointers

- Every variable in C has
  - Name
  - Address
  - Type
  - Value
- Pointers are just variables that contain the addresses of other variables.

- For security reasons, the base of the stack is randomized on Linux.
- Addition and substration on pointers cause the address to increase/decrease by the size of the type times the added/substracted value.
- The array name is treated as a pointer to the first element of the array (`A[i]` is always rewritten to `*(A+i)`).
- Arrays degrade into a pointer when passed into a function.
- Arrays can't be renamed.

Memory Allocation

Static Memory Allocation

- Allocated with global variables.

Automatic Memory Allocation

- Allocated using variables inside blocks.

Dynamic Memory Allocation

- Allocated using `malloc`, `calloc`, `realloc` and `free`.
- `size_t` is large enough to hold the size of the largest possible array in memory, can be used to store a pointer.

Explicit Memory Allocation: Application allocates and frees space (C, C++).

Implicit Memory Allocation: Application allocates, but the freeing is automatically done with Garbage Collection or other technology (Java, Rust).

Throughput: Number of completed requests.

Peak Memory Utilization: Aggregate payload (Sum of allocated payloads) divided by heap size.

The goal of memory allocators is to maximize both throughput and peak memory utilization.

Internal Fragmentation: Occurs if a payload is smaller than a block size. Caused by overhead of maintaining heap data structures, or by padding for alignment purposes.

External Fragmentation: Occurs if there is enough aggregate heap memory, but no single block is large enough.

The key policies of any allocator are the *placement policy*, the *splitting policy*, and the *coalescing policy*.

Let n be the number of blocks, m the number of free blocks.

	Allocation Cost	Free Cost
Implicit List	$O(n)$	$O(1)$
Explicit List	$O(m)$	$O(1)$
Segregated List	$O(log(n))$ for power-of-two size classes	$O(1)$

Implicit List

- For each block, store the length and an allocated flag in one word.
- First Fit: Start at beginning, choose the first free block.
- Next Fit: Start where previous search finished, choose the next free block.
- Best Fit: Choose the block with the fewest bytes left over.
- Free a block by setting allocated flag, check if it can be coalesced.

Explicit List

- Store pointers to both next and previous free blocks

Segregated List

- Each size class has its own free list

Garbage Collection **Garbage Collection**: Automatic reclamation of heap-allocated storage, such that the application never has to free itself.

Basic x86 Architecture

Instruction Set Architecture (ISA): Parts of a processor design that needs to be understanded to write assembly code.

Microarchitecture: Implementation of the ISA.

Complex Instruction Set (CISC): Add instructions to perform typical programming tasks (Easy for compiler, smaller code size).

Reduced Instruction Set (RISC): Few simple instructions instead of many (Better for optimizing compilers, runs fast with simple chip design).

Data Types

- Integer Data of 1, 2, 4 or 8 bytes.
- Floating point data of 4, 8, or 10 bytes.

Moving Data

`movx s, d`, where `x` is either `b` ("byte", 1 byte), `w` ("word", 2 bytes), `l` ("long word", 4 bytes), `q` ("quad word" 8 bytes).

Operand Types

Immediate: Constant integer data (`$0x400`).

Register: One of the register (`%eax`).

Memory: Bytes of memory at address given by register (`D(Rb, Ri, S)`, where `D` and `S` are constants, and `Rb` and `Ri` are registers. The memory address is calculated by $Addr = Rb + S * Ri + D$).

The `lea` instruction exploits this complex memory addressing and computes an address, but instead of accessing it, it returns the resulting address.

Condition Codes

Condition Code	Description
CF	Carry out from the most significant bit (Unsigned overflow)
ZF	Set if result is zero
SF	Set if smaller than zero as a signed integer
OF	Set if a two's complement signed integer overflows

All arithmetic operations set these flags according to the result.

`cmpx a, b` computes `a - b` without setting the destination.

`testx` computes `a & b` without setting the destination.

Condition codes can be read using the `setx` instructions.

Operation	`cmpx a, b` tests `a` *op* `b`
`sete`	Equal
`setne`	Not equal
`sets`	Negative
`setns`	Nonnegative

Operation	`cmpx a, b` tests `a` *op* `b`
`setg`	Greater (Signed)
`setge`	Greater or equal (Signed)
`setl`	Less (Signed)
`setle`	Less or equal (Signed)
`seta`	Above (Unsigned)
`setb`	Below (Unsigned)