Elementi di programmazione nei SO

I SO sono essenzialmente programmi giganteschi scritti in C.

> ⓘ **C**
>
> Creato da Dennis Rirhice nel 1972 per sviluppare programmi UNIX, l'idea principale dietro al C è che tutto è un file (Socket, Dischi, Modem ecc...).

Come ogni linguaggio di programmazione abbiamo uno standard input e due standard output:

Nome	Short name	File number	Descrizione
Standard In	stdin	0	Input dalla tastiera
Standard Out	stdout	1	Output alla console
Standard Error	stderr	2	Output di errore alla console

Hello world

Esistono essenzialmente tre modi per scrivere questo programma basilare il cui compito è stampare la strainga "Hello World" sulla console (Output Standard):

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

int main()
{
        printf("Hello World");
        return 0;
}
```

Semplicemente chiamiamo la funzione printf è portiamo nello standard output la stringa. printf non è una funzione di sistema, ma una funzione in C che chiama una funzione di sistema.

```c
#include<unistd.h>
#define STDOUT 1

int main(int argc, char **argv)
{
```

```
        char msg[] = "Hello World!\n";
        write (STDOUT, msg, sizeof(msg));
        return 0;
}
```

Definiamo l'output della nostra funzione su 1, cioè lo standard output.
`write` è una funzione più basilare di printf che permette di scirvere byte su file o dipsositivi tramite il suo **file descriptor**.

> **File descriptor**
>
> E' un **intero** che rappresenta un **canale** attraverso il quale un programma può accedere a un file o a un dispositivo in un sistema operativo.
> Ogni file o dispositivo aperto è associato a un file descriptor che il programma utilizza per leggere e scrivere dati.

```
#include<unistd.h>
#include<stdio.h>
#include<sys/syscall.h>
#define STDOUT 1

int main(int argc, char **argv)
{
        char msg[] = "Hello World\n";
        int nr = SYS_write;
        syscall(nr, STDOUT, msg, sizeof(msg));
        return 0;
}```
In quest'ultimo codice invece stiamo invocando la **chiamata di sistema**
write tramite l'interfaccia **syscall**.
Fattostà che stiamo assegnando il numero per la chiamata di sistema di write
alla variabile `nr`, infatti `SYS_write` è una costante definita in
`<sys/syscall.h>` che ==rappresenta il numero di chiamata di sistema per la
scrittura.==

**`syscall(nr, STDOUT, msg, sizeof(msg));`**: Chiama la funzione di sistema
`syscall`, passando:

- Il ==numero della chiamata di sistema== (`nr`), che è **`SYS_write`**.
- Il **file descriptor** di destinazione (in questo caso `STDOUT`, che è 1).
- Il **puntatore al buffer** (`msg`), che contiene la stringa che vuoi
scrivere.
- La **dimensione** dei dati da scrivere, che è **`sizeof(msg)`**, che
```

include anche il carattere di terminazione null `\0`.
Processo di costruzione di un file C

Il processi di costruzione di un file C è diviso in diversi passaggi:

0) Scrittura del codice C.

1) Analisi del codice da parte del **Preprocessore** e ==creazione del file chè verrà analizzato dal compilatore.==

2) Il compilatore trasforma le istruzioni scritte in C in **istruzioni assembly** che vengono tradotte il **linguaggio macchina.** In questo modo il codice è nella forma tale da poter ==essere eseguito dal calcolatore== ed esso viene indicato con il termine **codice oggetto** che ==non rappresenta ancora il codice eseguibile==.

3) Per passare dal codice oggetto al programma eseguibile devono essere ==risolti eventuali riferimenti a funzioni esterne al codice==, compito affidato al **linker** che unsce al codice oggetto porzioni di codice oggetto presenti nelle librerie, completanto questo passaggio otteniamo il **file eseguibile**.

>[!Info] Standard Library
>Una standard library di un linguaggio di programmazione è un ==insieme di funzioni, classi, moduli e strumenti predefiniti che sono inclusi nel linguaggio stesso== e che forniscono funzionalità comuni e di base.
>Un esempio di standard library in C è **libc** che ci permette di implementare istruzioni come: **read, write, exit**.
>Per chiamare queste istruzioni è necessario chiamare la **syscall**, ci sono davvero tante syscall in C (Quasi 450).
>le standard library sono poi quelle librerie che ci permettono di implementare tutte quelle funzioni a "livello più alto" come printf.
Passare da read() a sys_read()

Cosa succede quando invoco una funzione della standard library, per esempio read:

1) La funzione read() è implementata in gblic e funge da ==wrapper per la chiamata di sistema== read.

2) La macro **SYSCALL_CANCEL** viene utilizzata in read.c per effettuare la chiamate in modo ==sicuro==.

3) **SYSCALL_CANCEL** si basa su **INTERNAL_SYSCALL** per ==configurare i registri e invocare l'istruzione syscall==, che fa passare il controllo dal contesto utente a quello kernel.

4) L’istruzione syscall provoca il passaggio alla modalità kernel e ==indirizza l’esecuzione== a **entry_SYSCALL_64**, definito in assembly.

5) **entry_SYSCALL_64** chiama **do_syscall_64**, una funzione C che ==determina quale funzione kernel chiamare== basandosi sul numero della chiamata di sistema.

6) **do_syscall_64** consulta **sys_call_table** per ==mappare il numero di system call alla funzione kernel appropriata.== Per read, il numero è 0 e punta a __x64_sys_read.

7) La funzione **__x64_sys_read** è il ==wrapper per read nel kernel,== definita con **SYSCALL_DEFINE3**, che richiama ksys_read per la logica di lettura.