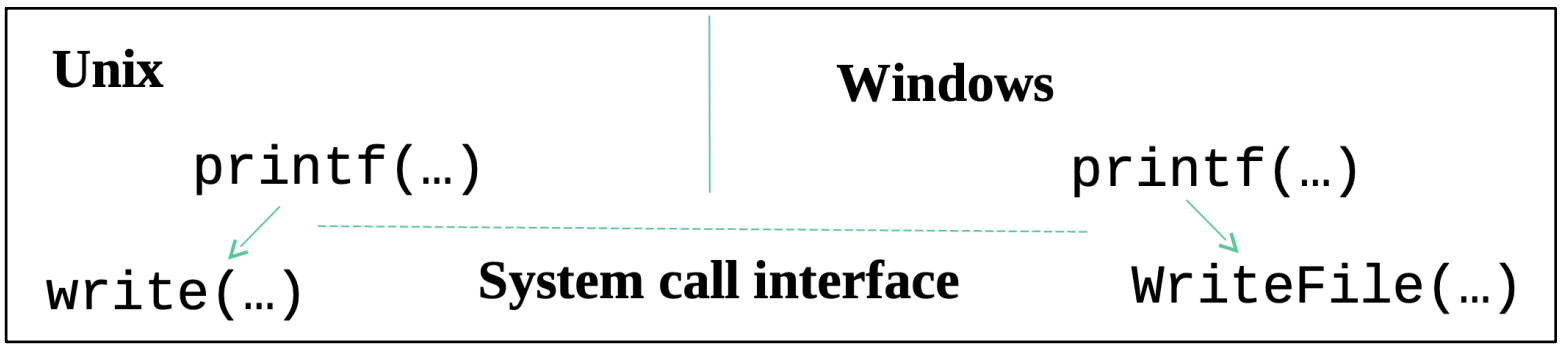
Librerie standard: gli standard di linguaggio

- gli standard di linguaggio definiscono servizi standard per la programmazione in una data tecnologia (e.g. ANSI-C)
- questi sono offerti dalle librerie standard, basate su specifica di interfaccia e semantica di esecuzione
- ogni funzione di libreria invocherà a sua volta le system call (necessarie per la sua mansione) proprie del sistema operativo su cui si opera

Una funzione di libreria per svolgere il lavoro per cui è progettata magari può a sua volta utilizzare anche un'altra porzione di software presente all'interno dell'architettura di sistema che può essere stata sviluppata da qualche altro programmatore.

Quando chiamiamo la printf in UNIX, in realtà la printf per consegnare i dati al S.O in modo tale che quest'ultimo lo consegni al preciso Device, chiama una System call che si chiama WRITE.



ESEMPIO SU LINUX UNIX

```c
#include <stdio.h>

int main(int x, char** y){
    char c;

    while(1){
        scanf("%c",&c);
        printf("%c",c);

    }
}
```

Molto semplice. Ciò che scrivo, printo sulla shell.

```
OFTWARE-EXAMPLES/C-BASICS> ./a.out
wpgw;gnmeh
wpgw;gnmeh
ehmekhnlteknh
ehmekhnlteknh
emh;tenhtelnhtelnh
emh;tenhtelnhtelnh
ehnelhntelh
ehnelhntelh
^C
```

Dobbiamo comunque parlare con il software del S.O e chiedere l'attivazione di queste funzioni, che ora mettiamo direttamente, **all'interno del software** del sistema operativo stesso.

```c
#include <unistd.h>

int main(int x, char** y){
    char c;

    while(1){
        read(0,&c,1);
        write(1,&c,1);

    }
}
```

Nella Read preleviamo 1 Byte e di scaricarlo all'interno della variabile C, e nella Write stessa cosa, ma comunichiamo l'indirizzo di memoria che poi va al software del S.O che va dentro l'address space a leggere il dato.

La programmazione non è basata sulla libreria standard, ma su un altro tipo di standard che è lo **standard di sistema** ossia un insieme di moduli software che noi possiamo utilizzare in uno specifico sistema, in questo caso su sistemi UNIX.

Questo è un programma unicamente per sistemi unix.

```
OFTWARE-EXAMPLES/C-BASICS> ./a.out
wpnmgrwkgnlren
wpnmgrwkgnlren
epomgregnek
epomgregnek
```

```
epogmrekngrenren
epogmrekngrenren
```

Però possiamo scrivere un programma direttamente in Assembly.

```
1
2  //neverending echo from the standard input to the standard output
3  //please compile with "gcc -c" "and "ld" or with "-nostartfiles"
4
5  .file "asm-terminal-echo.S"
6  .text
7  .globl _start
8  .type _start, @function
9
10 _start:
11     sub $0x4, %rsp //eserve 4 bytes for hosting chars to read/write
12     mov %rsp, %rsi
13     mov $0x1, %rdx
14     .loop:
15     mov $0x0, %rax //from this line we pack registers for calling the read sycall
16     mov $0x0, %rdi
17     syscall
18     mov $0x1,%rax //from this line we update registers for calling the write syscall
19     mov $0x1, %rdi
20     syscall
21     jmp .loop
22
```

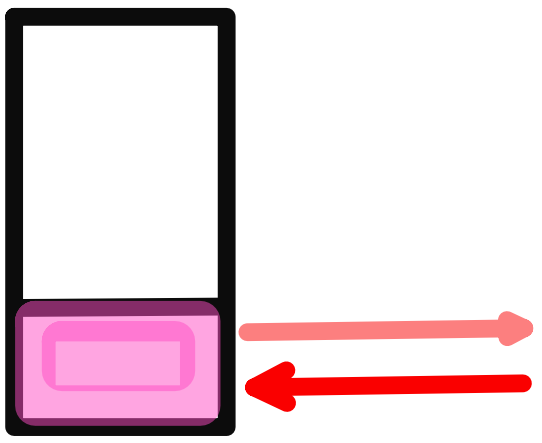
Faccio funzionalmente le stesse cose che facevo nei programmi precedenti.

Prima di passare il controllo al software del S.O con la prima chiamata della System call, con le istruzioni antecedenti stiamo mettendo apposto qualche cosa all'interno dei registri, in modo tale che quando il software del S.O parte, si troverà all'interno dei registri di processore tutta una serie di informazioni che io ho caricato.

All'interno del ciclo loop abbiamo un'istruzione macchina che si chiama Syscall, la chiamiamo due volte: esse portano il controllo al software del sistema operativo

In rax, rdi, rsi ed rdx, che sono 4 registri del processore x86, andiamo a identificare dove in memoria è la locazione dove ci deve consegnare un byte .

Riserviamo 4 byte all'interno della stack area, con la prima istruzione, all'interno dell'address space.
Così abbiamo una zona, di cui poi comunichiamo l'indirizzo quando chiamiamo il sistema operativo, dove può scaricarci almeno un byte o prelevare questo byte per immetterlo verso un altro dispositivo.



Compilazione "-nostartfiles"

```
> gcc asm-terminal-echo.S -nostartfiles
OFTWARE-EXAMPLES/C-BASICS> ./a.out
[weowngrwn
[weowngrwn
repingoengegnerkj
repingoengegnerkj
pwngwngrwngrwogrwg
pwngwngrwngrwogrwg
rengregnrekjb
rengregnrekjb
```

Non vogliamo aggiungere gli starters, cercheremo di capire che cosa sono.

Funzionalmente facciamo le stesse cose
che stavamo facendo prima.
Funzionalmente perché la struttura di questi codici è completamente differente.

All'interno di questa scanf, così come nella printf, ci sono una serie di istruzioni macchina **che servono non solo per chiamare**

semplicemente il S.O, ma anche qual'è il valore di ritorno che il sistema operativo **sta cercando di consegnare**.

Questo funziona sia per l'assembly e per il C.
Sia gli Stub read e write.
Ritornano dei valori.

Ma non stiamo controllando i valori di ritorno.

STANDARD DI SISTEMA

Esso definisce i servizi offerti su uno **specifico** sistema, per programmare applicazioni usando una specifica tecnologia.

Quando scriviamo applicazioni C **utilizziamo lo standard di sistema UNIX**.
Lo possiamo fare anche per Windows.

Gli standard di sistema classici che useremo sono:

-Sistemi Unix: Standard che si chiama **POSIX**.

-Sistemi Windows: Standard che si chiama **Win-Api**.

Lo standard di sistema definisce soltanto delle System Call, oppure abbiamo all'interno delle altre cose?

- lo standard per una famiglia di sistemi definisce tipicamente
  1) il set delle system call offerte ai programmatori (tramite libreria)
  2) il set di funzioni di libreria specifiche per quel sistema (che a loro volta possono appoggiarsi sulle system call native del sistema in oggetto)

Chi scrive le librerie da includere nei programmi classici in C, in realtà non fa altro che scrivere del codice per semplificarci la vita e non permetterci di usare le istruzioni native del sistema operativo, è un'astrazione molto elevata. Io chiamo Printf() piuttosto che chiamare Write() o peggio ancora scrivere codice Assembly .