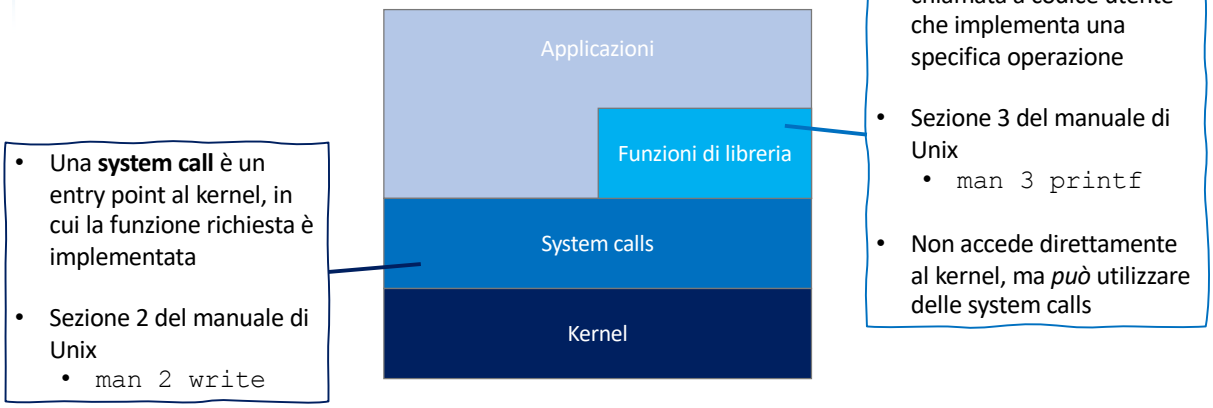
Introduzione alla programmazione di sistema

Architettura di Unix



- Una **system call** è un entry point al kernel, in cui la funzione richiesta è implementata
- Sezione 2 del manuale di Unix
  - `man 2 write`

- Una **library call** è una chiamata a codice utente che implementa una specifica operazione
- Sezione 3 del manuale di Unix
  - `man 3 printf`
- Non accede direttamente al kernel, ma *può* utilizzare delle system calls

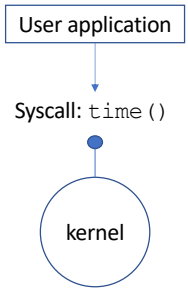
Il kernel Linux conta poco meno di 400 system calls, che sono descritte nella sezione 2 del manuale di Unix. Le library calls sono chiamate a funzioni di libreria, sviluppate e precompilate da qualcun altro, che implementano una specifica operazione. Per farlo, possono (non necessariamente) aver bisogno di accedere al kernel, e lo fanno tramite le system calls. Molte di esse sono descritte nella sezione 3 del manuale di Unix.

Nota: entrambe appaiono come funzioni C, invocabili dall'applicazione

System vs library calls

- Le system calls generalmente restituiscono informazioni «grezze»
- Eventuali operazioni più complesse sono lasciate all'implementazione del processo chiamante

- Esempio: qual è la data di oggi?



Deve occuparsi di manipolare il valore restituito per ottenere la data in formato human-readable

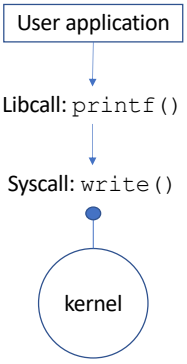
Restituisce il numero di secondi passati da 1/1/1970

Esempio: per sapere qual è la data di oggi, possiamo utilizzare la system call time(), la quale restituisce un numero intero che rappresenta il contatore dei secondi trascorsi dal 1/1/1970. Questa data è chiamata **Epoch** ed è stata scelta in modo arbitrario. Permette di rappresentare l'ora e data in formato molto compatto.
Per avere una data leggibile e usabile per l'utente) dovremo scrivere del codice per convertire il numero dei secondi in una data.

Problema dell'anno 2038: l'informazione restituita da time() è immagazzinata in un intero con segno a 32 bit --> non potrà contenere numeri più grandi di 2^(31), e quindi non potrà rappresentare date più lontane del 19 gennaio 2038, Quando si raggiunge il numero massimo di secondi si torna indietro al 13 dicembre 1901 (!), causando possibili malfunzionamenti nei software che utilizzano tale data.
Su wikipedia trovate una animazione che mostra il bug.

System vs library calls

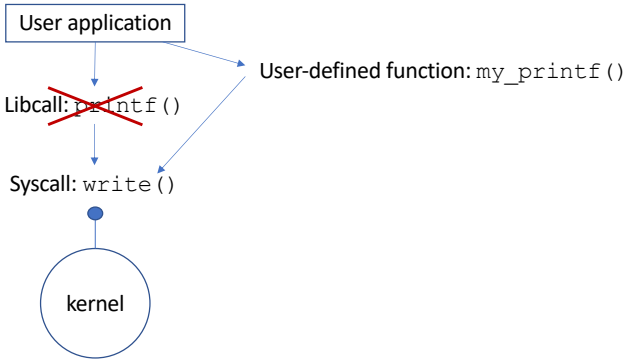
- Le librerie forniscono routine/funzioni che possono essere riutilizzate dai programmi dell'utente
  - La libreria `string.h` fornisce funzioni per operare sulle stringhe (`char*`)
- Le funzioni di libreria possono sfruttare una o più system calls per svolgere il proprio compito
- Esempio: scrittura su standard output (heap)



Grazie alle funzioni di libreria non abbiamo bisogno di riscrivere una funzione che fa la copia di una stringa perché qualcuno lo ha già fatto per noi - funzione strcpy()

System vs library calls

- Le librerie forniscono routine/funzioni che possono essere riutilizzate dai programmi dell'utente
  - La libreria `string.h` fornisce funzioni per operare sulle stringhe (`char*`)
- Le funzioni di libreria possono sfruttare una o più system calls per svolgere il proprio compito
- Esempio: scrittura su standard output (heap)



Non siamo obbligati a utilizzare le funzioni di libreria, quindi se non vi piace come è implementata una certa funzione di libreria siete liberi di definire una vostra funzione che utilizza in maniera alternativa le system call (a vostro rischio e pericolo!).

Tipi di dato di sistema primitivi

```
...
int n = sizeof(object);
...
```

`int` occupa generalmente 32 bit
Cosa succede se il programma viene eseguito su un sistema in cui `sizeof()` restituisce un valore su 64 bit?

- Quando ci si riferisce a oggetti del sistema (file, dispositivi, processi,...) è consigliato usare tipi la cui dimensione dipende dall'implementazione del sistema sottostante
- Tipi definiti dalla libreria POSIX `<sys/types.h>`
  - Suffisso `_t`

```
...
size_t n = sizeof(object);
...
```

`size_t`	Dimensione di oggetti (senza segno)
`ssize_t`	Dimensione di oggetti (con segno)
`time_t`	Secondi da 1/1/1970
`mode_t`	Tipo di file e modalità di accesso
`pid_t`	Identificatore di processo
`ino_t`	Numero di i-node
`...`	...

Gli standard (ISO e/o POSIX) definiscono **l'interfaccia** delle funzioni, non la loro implementazione. È possibile che due sistemi Unix diversi implementino la stessa funzione in modo diverso (basta che abbiano una implementazione di tale funzione per rispettare lo standard)

Esempio: la funzione sizeof() potrebbe restituire un numero su 32 o 64 bit su sistemi diversi → problema se assegniamo il risultato a una variabile di tipo int.

Soluzione: usare i **tipi di dato di sistema primitivi**. Sono rappresentati su un numero di bit che dipende dal sistema, ma se li uso nel mio programma posso stare tranquillo che non avrò problemi di conversione.

Per esempio, la funzione sizeof restituisce un tipo size_t - che può essere 32 o 64 bit. Indipendemente dall'implementazione del tipo size_t, possiamo assengare il risultato di sizeof a una variabile di tipi size_t, evitando problemi di conversione. Questa versione del programma funziona sia su sistemi a 32 che a 64 bit.

Gestione degli errori

- System e library calls possono terminare con un errore
  - e.g. tentare di leggere un file che non esiste
- Alcune di esse, in caso di errore:
  - restituiscono un intero negativo o `NULL`
  - settano il valore di una variabile `errno`

Variabile intera definita in `<errno.h>`
Contiene un codice che "spiega" cosa è andato storto

- La libreria `<string.h>` offre una funzione che "converte" `errno` in un messaggio
  - `char* strerror(int num);`

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
...
int result = <invocazione di una system/library call>;
if (result < 0) {
    printf("Error! errno=%d: %s \n", errno, strerror(errno));
    /* gestione dell'errore */
}
...
```


errno è una variabile definita dalla libreria <errno.h>, il valore che ci viene scritto dentro dalla funzione specifica quale è stato l'errore. Per cui, ogni volta che invochiamo una funzione di sistema, possiamo poi testare il valore di ritorno per capire se c'è stato un errore, e in caso affermativo, possiamo vedere di che tipo di errore si tratta, ispezionando la variabile errno.

man errno → elenca tutti i possibili codici di errore

Possiamo dare informazioni più esplicative all'utente del programma usando la funzione strerror(), che converte il valore di errno in una stringa. Tale stringa può essere mostrata a video.

Gestione degli errori

- Cosa fare in caso di errore?
- Politica comune → terminare l'esecuzione del programma
- Funzione della libreria `<stdlib.h>`
  - `void exit(int status);`

`<stdlib.h>` definisce delle costanti da usare per specificare il motivo della terminazione
- `EXIT_SUCCESS = 0`
- `EXIT_FAILURE = 1`

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
...
int result = <invocazione di una system/library call>;
if (result < 0) {
    printf("Error! errno=%d: %s \n", errno, strerror(errno));
    exit(EXIT_FAILURE);
}
...
```


Una volta che ho scovato l'errore, devo mettere in atto delle politiche di gestione. La politica più brutale e semplice, ma anche la più comune, è quella di interrompere l'esecuzione del programma, invocando la funzione di libreria exit().

Si raccomanda di **non** mettere "numeri magici" nel codice, ma di usare delle costanti definite dalle librerie, in questo caso EXIT_SUCCESS o EXIT_FAILURE, per lo stesso motivo di maggiore portabilità del codice su sistemi diversi visto prima con i tipi di dato primitivi. Infatti, non c'è garanzia che EXIT_FAILURE = 1 su tutti i sistemi.

- exit(1) → NO
- exit(EXIT_FAILURE) → SI