LabSO 2021

Laboratorio Sistemi Operativi - A.A. 2020-2021

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Architettura

Kernel Unix

L’elemento di base di un sistema Unix-like è il nucleo del sistema operativo, ovvero il kernel. Il kernel è incaricato della gestione delle risorse essenziali: CPU, memoria, periferiche, etc...

Ad ogni boot il kernel verifica lo stato delle periferiche, monta la prima partizione in read-only e lancia il primo programma (/sbin/init).

Il resto delle operazioni, tra cui l’interazione con l’utente, vengono gestite con i programmi eseguiti dal kernel.

Kernel e memoria virtuale

I programmi utilizzati dall'utente che vogliono accedere alle periferiche chiedono al kernel di farlo per loro.

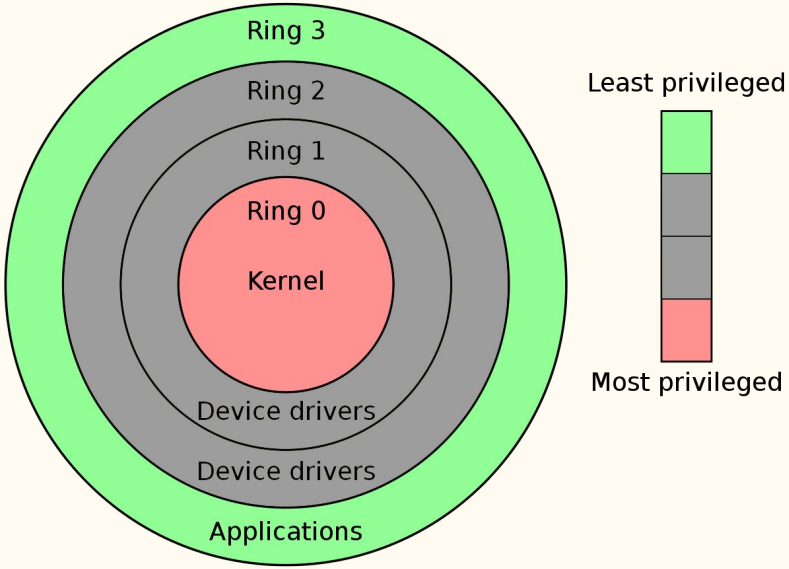
L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione hardware della memoria virtuale e la modalità protetta.

Ogni programma vede se stesso come **unico possessore della CPU** e non gli è dunque possibile disturbare l'azione degli altri programmi → stabilità dei sistemi Unix-like!

Privilegi

Nei sistemi Unix-like ci sono due livelli di privilegi:

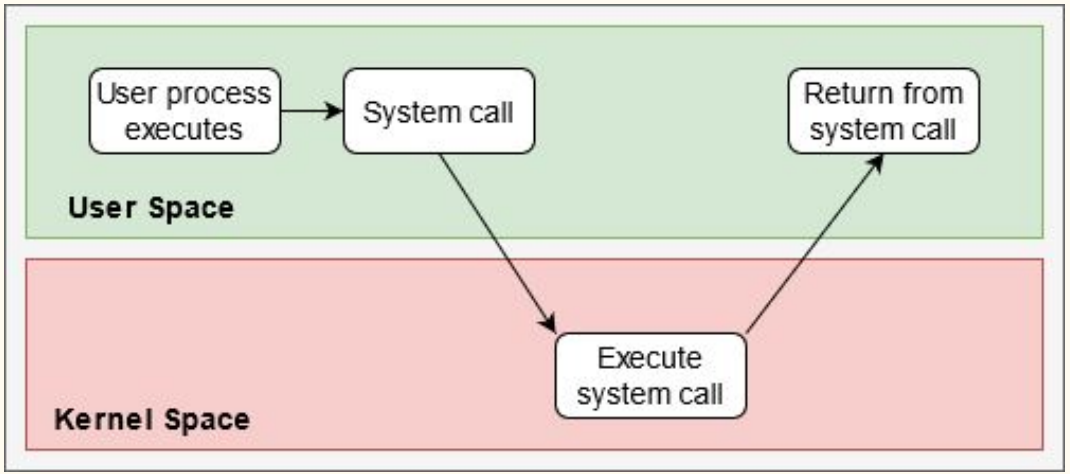
- **User space:** ambiente in cui vengono eseguiti i programmi
- **Kernel space:** ambiente in cui viene eseguito il kernel



System calls

System calls

Le interfacce con cui i programmi accedono all'hardware si chiamano **system calls.** Letteralmente "chiamate al sistema" che il kernel esegue nel kernel space, restituendo i risultati al programma chiamante nello user space.



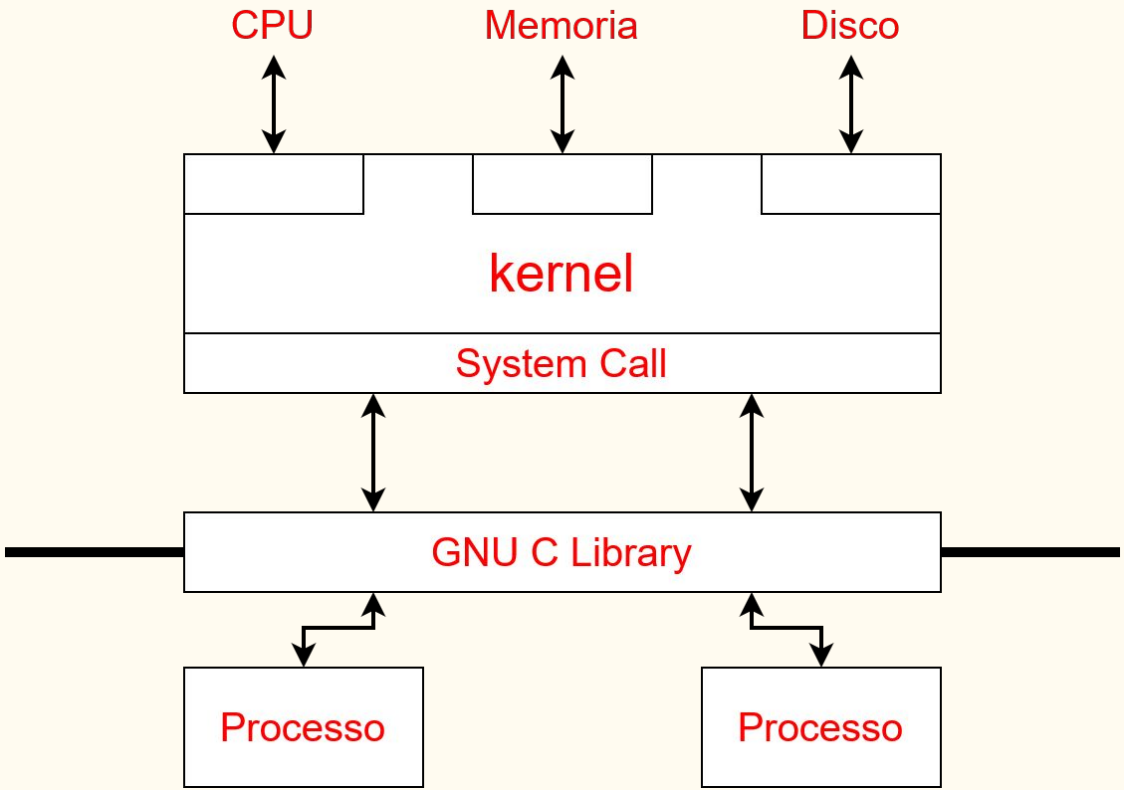
Le chiamate restituiscono "-1" in caso di errore e settano la variabile globale `errno`. Errori validi sono numeri positivi e seguono lo standard POSIX, il quale definisce degli alias.

System calls: librerie di sistema

Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono tipicamente anche *ld-linux.so*, e *libc.so*.

- **ld-linux.so**: quando un programma è caricato il sistema operativo passa il controllo a ld-linux.so anzichè al normale punto di ingresso dell'applicazione, così da caricare le librerie non ancora "risolte" e poi passarle il controllo.
- **libc.so**: la libreria GNU C solitamente nota come glibc che contiene le funzioni basilari più comuni

System Calls: librerie di sistema



Get time: time() e ctime()

```
time_t          time(          time_t          *second          )
char * ctime( const time_t *timeSeconds )
```

```
#include <time.h>                                          //time.c
#include <stdio.h>

void main(){
    time_t whatTime = time(NULL); //seconds since 1/1/1970
    //Print date in Www Mmm dd hh:mm:ss yyyy
    printf("Current time = %s", ctime(&whatTime));
}
```

Working directory: chdir(), getcwd()

```
int      chdir(      const      char      *path      )
char * getcwd( char *buf, size_t sizeBuf );
```

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

void main(){
    char s[100];
    chdir(".."); //Change working dir
    printf("%s\n", getcwd(s,100)); //Print current working dir
}
```

Operazioni con i file

```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);


FILE *fopen(const char *filename, const char *mode)
int fclose(FILE *stream)

....
```

Duplicazione file descriptors: dup(), dup2()

```
int dup(int oldfd)
int dup2(int oldfd, int newfd)
```

```
#include <unistd.h>                                                    //dup.c
#include <stdio.h>
#include <fcntl.h>
void main(){
    char buf[50];
    int fd = open("file.txt",O_RDWR); //file exists
    read(fd,buf,50); printf("Content: %s\n",buf);
    int cpy = dup(fd); // Create copy of file descriptor
    dup2(cpy,22); // Copy cpy to descriptor 22 (close 22 if opened)
    lseek(cpy,0,SEEK_SET); //move I/O on all 3 file descriptors!
    write(22,"This is a fine ",15); // Write starting from 0-pos
}
```