LabSO

Architettura, processi e system calls

Architettura I

- **L’elemento di base di un sistema Unix-like è il nucleo del sistema operativo ovvero il kernel**
- **Al kernel si demanda la gestione delle risorse essenziali: CPU, memoria, periferiche, etc…**
- **Al boot il kernel verifica lo stato delle periferiche, monta la prima partizione in read-only e lancia il primo programma (/sbin/init)**
- **Tutto il resto, come l'interazione con l’utente, viene realizzato con programmi eseguiti dal kernel tramite init**

Architettura II

- I programmi utilizzati dall'utente accedono alle periferiche chiedendo al kernel di farlo per loro.
- I kernel Unix-like sfruttano alcune caratteristiche intrinseche ai processori come la gestione hardware della memoria virtuale e la modalità protetta
- Solo il kernel è eseguito in modalità privilegiata, con il completo accesso all’hardware
- Tutti gli altri programmi sono eseguiti in modalità protetta

User e Kernel space

- **Due sono i concetti fondamentali su cui si basa l'architettura dei sistemi Unix-like:**
  - **User space**: ambiente in cui sono eseguiti i programmi
  - **Kernel space**: ambiente in cui viene eseguito il kernel
- **Ogni programma vede se stesso come unico possessore della CPU**
- **Per questo fatto, non è possibile ad un singolo programma disturbare l'azione degli altri**
- **Da qui deriva la stabilità dei sistemi Unix-like**

File system I

- **Il sistema Unix fornisce due tipologie di interfacce per la programmazione attraverso file:**
  - Stream. Fornisce strumenti come la formattazione dei dati, bufferizzazione, ecc…
    `FILE*` objects
  - File descriptors. Interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel
    `INT` objects

File system II

- **Stream**
  - un file è descritto da un puntatore a una struttura di tipo FILE (definita in `stdio.h`)
  - I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volte, ecc.) ed essere interpretati
- **File descriptors**
  - Un file è descritto da un semplice intero (file descriptor) che punta alla rispettiva entry nella file table
  - I dati possono essere letti e scritti soltanto un buffer alla volta di cui spetta al programmatore stabilire la dimensione

File system III

File descriptors: Low Level I/O

Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un’interfaccia meno amichevole.

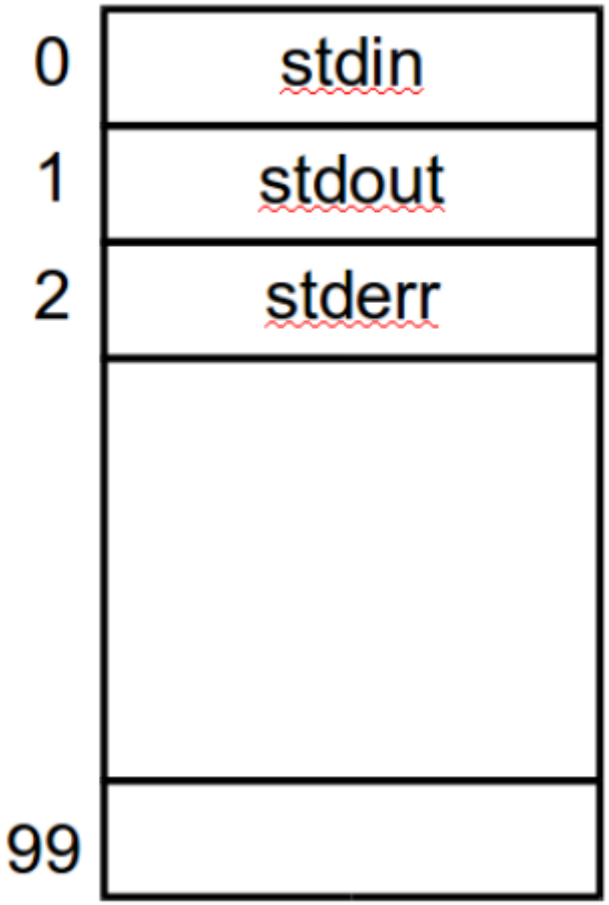
File system IV

File descriptors: Low Level I/O

- Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel
- Questo si fa aprendo il file con la system call `open` la quale localizza l'*i-node* del file e aggiorna la file table del processo
- All'interno di ogni processo i file aperti sono descritti da un intero chiamato file descriptor
- In Unix ogni processo all'avvio ha tre file aperti, standard input (valore di fd 0 "stdin"), output (1 "stdout"), error (2 "stderr")
- L'Input/Output Unix è basato essenzialmente su cinque funzioni: `open`, `read`, `write`, `lseek` e `close`
- Con la libreria `stdio.h` invece abbiamo `stdin`, `stdout` e `stderr` come file pointer predefiniti

File system V

- A ogni processo è associata una tabella dei file aperti di dimensione limitata (circa 100 elementi)
- Ogni elementod della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero (il “file descriptor”)
- I file descriptor 0, 1 e 2 individuano normalmente standard input, output ed error (aperti automaticamente)

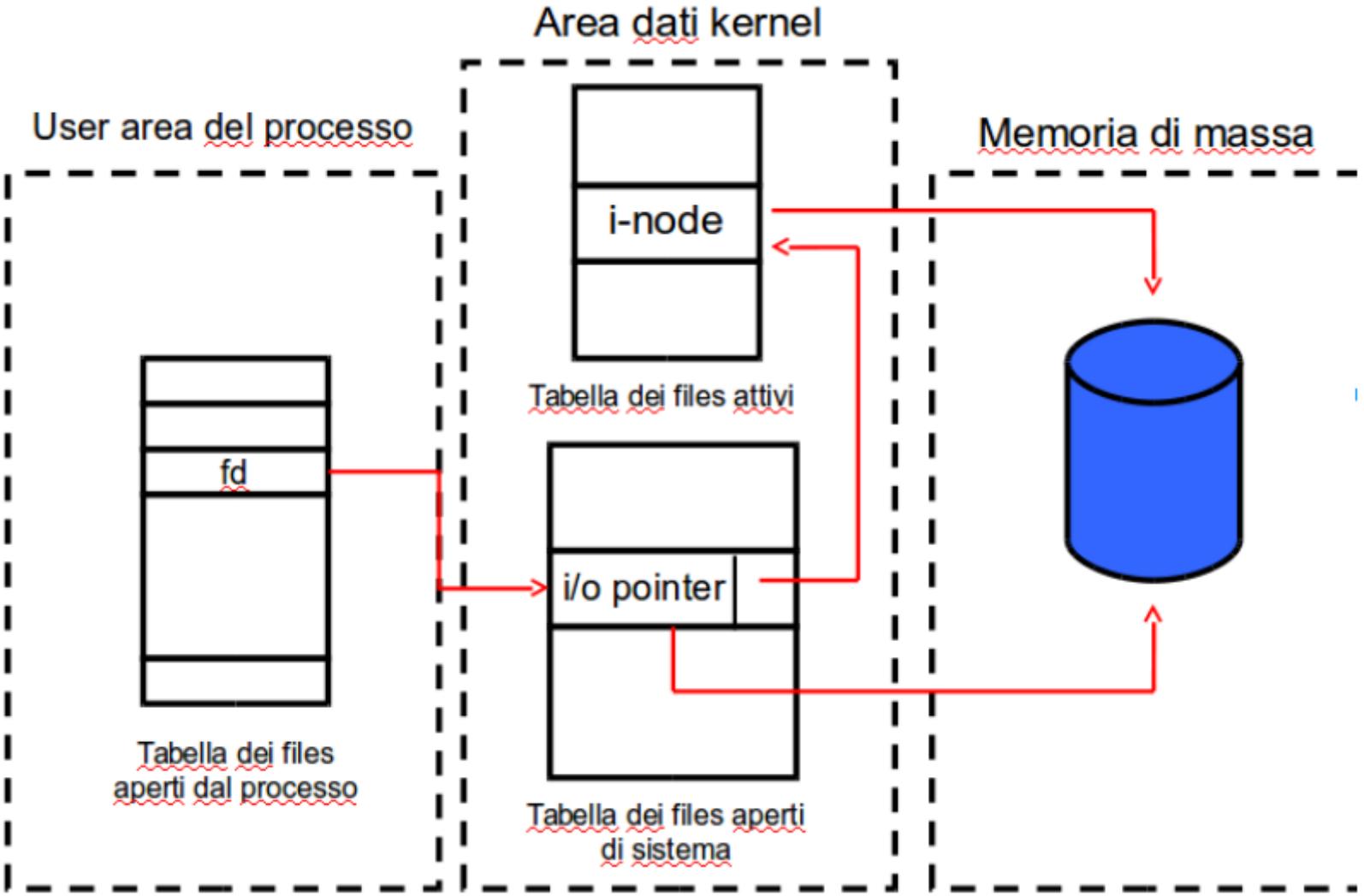


File system VI

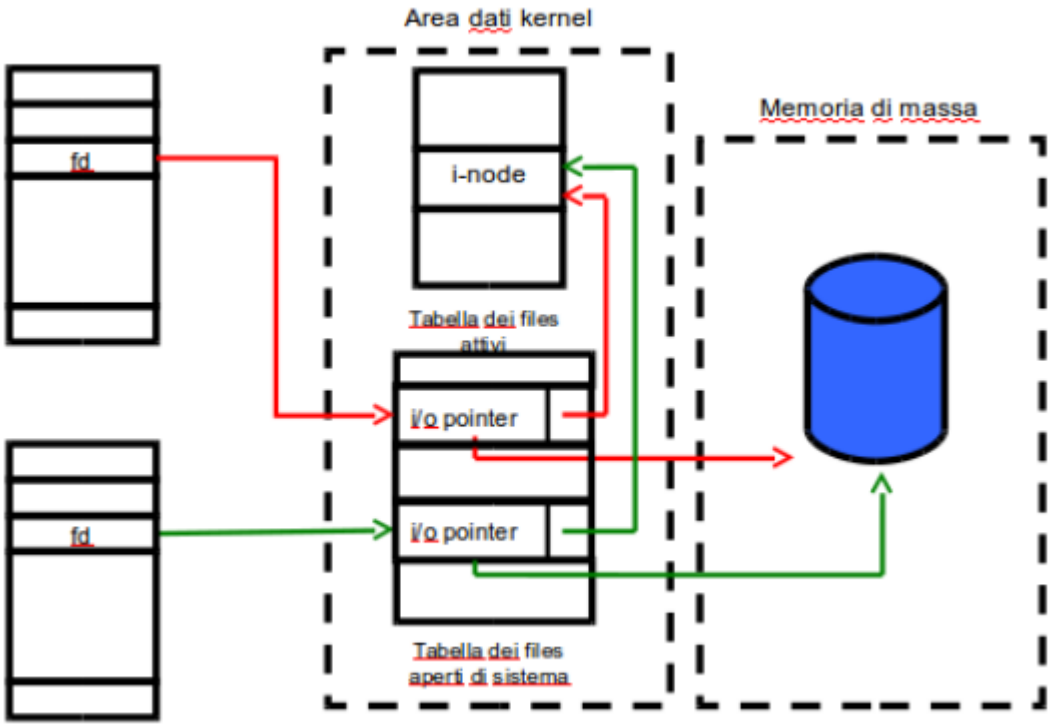
- **Per l'accesso ai file il S.O. utilizza due strutture globali (allocate nell'area del kernel):**
- **Tabella dei file attivi. Per ogni file aperto contiene una copia del suo i-node (efficienza d'accesso per ottenere attributi - come permessi e proprietario – informazioni temporali, etc.)**
- **Tabella dei file aperti. C'è un elemento per ogni operazione di apertura relativa a file aperti e non ancora chiusi, che contiene:**
  - I/O pointer (posizione corrente nel file)
  - puntatore indiretto a i-node attraverso la tabella dei file attivi

  nota: se più processi aprono separatamente lo stesso file, ci sono altrettanti elementi distinti (che però puntano allo stesso elemento dell'altra tabella, ma con I/O pointer differente)

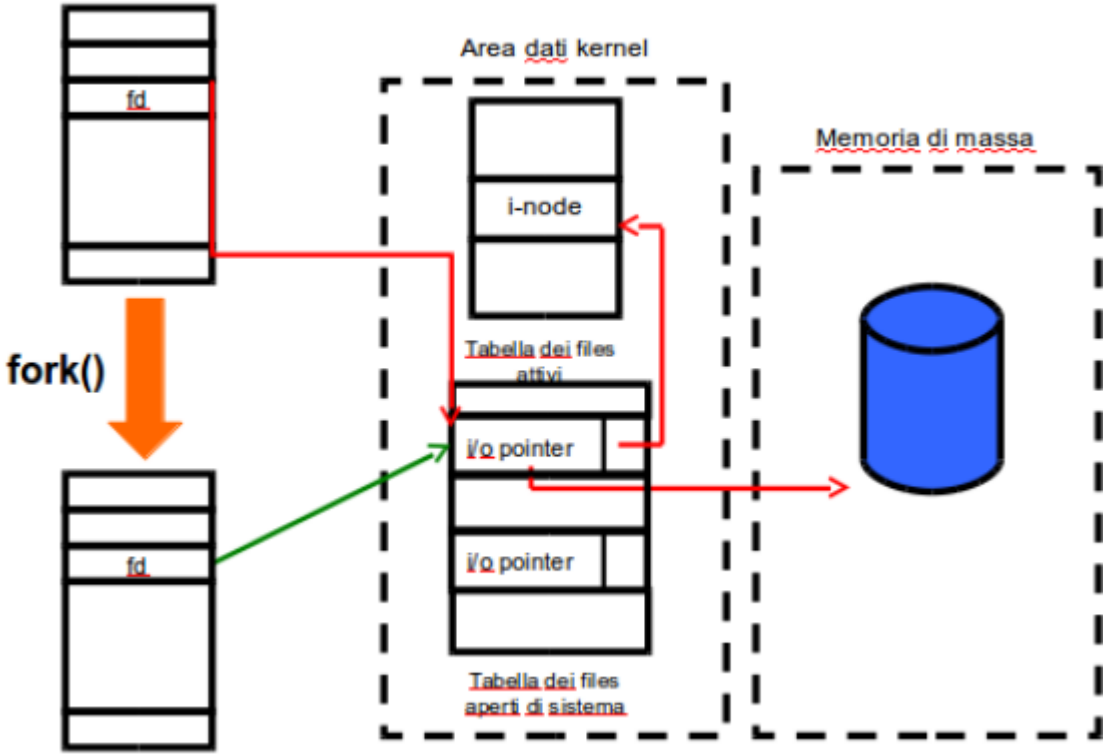
File system VII



File system VIII



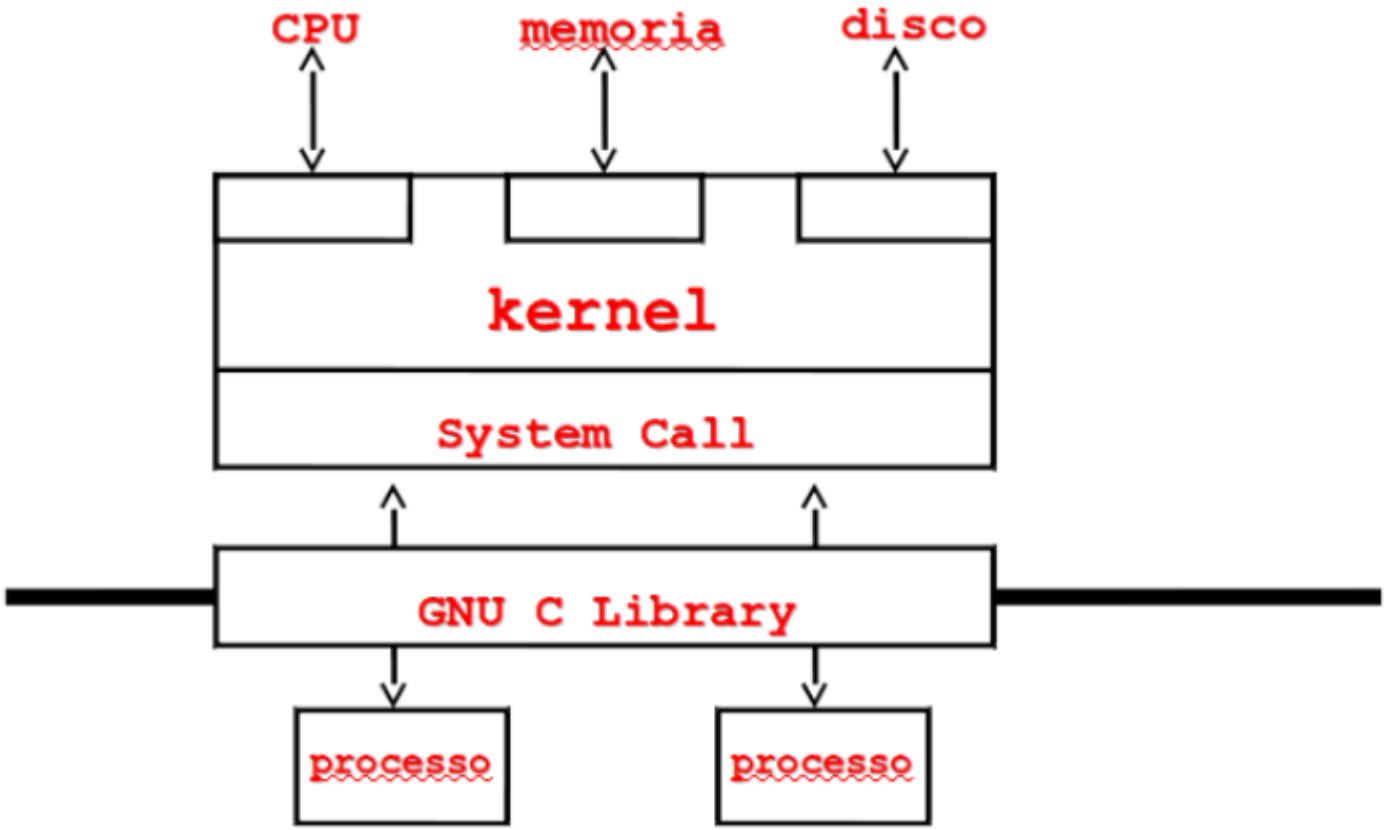
File system IX



System calls I

- Le interfacce con cui i programmi accedono all'hardware si chiamano system call
- Il kernel le esegue nel kernel space e restituisce i risultati al programma chiamante che invece opera in user space
- 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` amzichè 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
- le chiamate restituiscono "-1" in caso di errore e settano la variabile globale `errno`

System calls II



System calls III

- **Varie (tempo, uscita, ...) : `time()` e `ctime()`, `exit()`...**
- **File System: `chdir()`, `getcwd()`, `open()`, `close()`, `read()`, `write()`, `fopen()`, `fclose()`, `dup()`...**
- **Permessi : `chmod()`, `chown()`...**
- **Processi : `getpid()`, `wait()`, `fork()`, `execve()` (e simili)...**
- **Segnali : `kill()`, `pause()`, `alarm()`...**

System calls IVa – varie: time, ctime

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    time_t current_time;
    char* c_time_string;
    current_time = time(NULL); // current time
    if (current_time == ((time_t)-1)) {
        (void) fprintf(stderr, "Failure to obtain the current time.\n");
        exit(EXIT_FAILURE);
    };
    c_time_string = ctime(&current_time); // convert to readable format
    if (c_time_string == NULL) {
        (void) fprintf(stderr, "Failure to convert the current time.\n");
        exit(EXIT_FAILURE);
    };
    (void) printf("Current time is %s", c_time_string); // newline already set
    exit(EXIT_SUCCESS);
}
```

System calls IVb – varie: exit

(v. esempio precedente)

```
#include <stdlib.h>
void exit(int status)
```

- Funzione usata per effettuare un'uscita "normale" da un programma (v. man 3 exit)
- Tutti i file descriptor associati al processo vengono chiusi
- Il valore di uscita di main (i.e., "exit status") viene passato al processo che aveva lanciato il programma (in genere la shell)
- Questo valore fornisce indicazioni sulla riuscita o fallimento del programma. Tipicamente il valore di ritorno è compreso fra 0 e 255: 0 in caso di successo, >0 in caso di fallimento

Esempio, con la shell:
`ls ./ &`
`ls ./something &` (usare un nome inesistente + utilizzo di `echo $?`)

System calls Va – file system I (fd)

- **Access modes:**
  - `O_RDONLY  O_WRONLY  O_RDWR`
  - `O_APPEND  O_BINARY  O_TEXT`
- **Permissions:**
  - `S_IWRITE  S_IREADS_IWRITE | S_IREAD`
- **Include:**
  - `#include <fcntl.h>`
-
- **Funzioni:**
  - `int open (char *filename, int access, int permission);` (returns fd "*handle*")
  - `int read (int handle, void *buffer, int nbyte);`
  - `int write(int handle, void *buffer, int nbyte);`
  - `int close(int handle);`

System calls Vb – file system II (stream)

- **Include:**
  - `#include <stdio.h>`

- **Funzioni:**
  - `FILE *fopen(const char *filename, const char *mode)` open file
  - `int fclose(FILE *stream)` close file
  - `int fgetc(FILE *stream)` get next char in file
  - `int feof(FILE *stream)` test end of file

System calls Vc – file system (fd example)

```
#include <stdio.h>
int main () {
   FILE *fp; int c; fp = fopen("file.txt","r");
   while (1) {
      c = fgetc(fp); if (feof(fp)) { break ; };
      printf("%c", c);
   };
   fclose(fp);
   return(0);
}
```

System calls VI – permessi

```
#include <sys/stat.h>


int chmod(const char *pathname,
mode_t mode);

int chown(const char *pathname,
uid_t owner, gid_t group);
```

System calls VIIa – processi

- **Ogni processo ha un unico process ID; è un tipo di dato standard, il `pid_t`, che in genere è un intero**

  `int getpid();` restituisce il PID

- **Tutti i processi memorizzano anche il pid del genitore da cui sono stati creati, il parent process ID (PPID)**

  `int getppid();` restituisce il PPID

- **L'utente che esegue un processo è identificato dallo user ID**

  `int getuid();` restituisce lo UID

System calls VIIb – processi

- **Ogni processo ha un unico process ID; è un tipo di dato standard, il `pid_t`, che in genere è un intero**

  `int getpid();` restituisce il PID

- **Tutti i processi memorizzano anche il pid del genitore da cui sono stati creati, il parent process ID (PPID)**

  `int getppid();` restituisce il PPID

- **L'utente che esegue un processo è identificato dallo user ID**

  `int getuid();` restituisce lo UID

System calls VIIc1 – processi

```
#include <stdlib.h>
int system (char *cmd);
```

- il processo corrente crea un processo figlio
- system esegue la shell di sistema `/bin/sh`
- la shell esegue `cmd`
- la shell termina
- il processo padre continua da dove aveva interrotto

System calls VIIc2a – processi

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  printf("Run an 'ls -l' command\n\n");
  system("ls -l");
  printf("\nDone!\n\n");
}
```

System calls VIIc2b – processi

```
// sys.c | Example of usage: ./sys "ls -l"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

int main(int argc, char **argv) {
        int r;
        char cmd[100];
        printf("argc=%d\n", argc);
        if (argc>1) {
                strcpy(cmd, argv[1]);
        } else {
                strcpy(cmd, "");
        };
        printf("cmd=%s\n", cmd);
        r=system(cmd);
        printf("R,errno=%d,%d\n",r,errno);
        // in case of "logical" error running cmd
        // R is the error code, while errno should be "0"
        // as the syscall has been run
        return r;
}
```

System calls VIII – processi / esercizio

realizzare una prima semplicissima *shell* utilizzando la system che:

- **mostri un prompt all'utente**
- **esegua i comandi che l'utente inserisce**
- **ripete i passi 1 e 2 fintantoché il comando inserito non è `quit`**

Per l'input e il confronto si possono usare:

```
#include <stdio.h>

getline(&line, (size_t*)&len, stdin)
```

(*size_t = unsigned integer type*)

```
#include <string.h>

strcmp(const char *s1, const char *s2)
```

(vale 0 se il confronto riporta stringhe uguali)