LABORATORIO SISTEMI OPERATIVI: SYSTEM CALLS

- **SISTEMA E ARCHITETTURA**:
  - L'architettura in cui andiamo ad operare in generale è costituita da un **Sistema Operativo** (strutturato a strati) dove è presente uno **strato fondamentale** chiamato **Kernel**.
  - Il **Kernel** è il **nucleo operativo** che contiene una **serie di funzionalità di base** per gestire:
    - CPU
    - Memoria
    - Dispositivi
    - Accesso alle risorse a basso livello (hardware).

  - All'avvio del sistema, il **Kernel predispone tutto l'ambiente operativo** e **dà l'avvio** al sistema stesso, **eseguendo un primo processo di base** chiamato **init()**, processo che **rimane attivo per tutto il ciclo** d'esecuzione del sistema stesso (gestendolo).
  - Fa iniziare una serie di processi che a loro volta eseguiranno altri processi (effetto a cascata).
    - Init fa da root process a tutti i processi che vengono generati (**init** ha PID=1).

- **KERNEL**:
  - Insieme di **funzioni di basso livello** che accedono all'hardware.
  - I **processi Kernel** sono **eseguiti** in **modalità privilegiata**, hanno un **accesso completo all'hardware** sottostante (tutti gli **altri processi** sono **eseguiti in modalità protetta**, una **modalità limitata**).

- **USER PROGRAMS**:
  - Per accedere ai dispositivi, i **processi utente chiedono al Kernel l'accesso** (viene usato come intermediario).
  - Il Kernel **controlla**, **accetta** ed **esegue** la richiesta.

- **SPACES ENVIROMENTS**:
  - Tutti i programmi sono eseguiti all'interno di uno spazio di lavoro:
    - **Kernel Space**:
      - **Ambiente in cui vengono eseguiti i processi del Kernel**, quando le sue funzionalità vengono richiamate.

    - **User Space**:
      - **Ambiente dove si trovano ad operare i singoli programmi** che vengono eseguiti (programmi che non fanno parte del kernel).
      - Ogni programma utente vede sé stesso come se fosse l'unico programma in esecuzione non curandosi della gestione delle risorse, della CPU, ecc. (sono gestiti da altri processi).
      - I **processi utente possono scambiarsi dati** e **interagire con altri processi** (usando delle chiamate di sistema specifiche che fanno affidamento al kernel).

- **SYSTEM CALL**:
  - Il **Kernel** è composto da un **insieme di funzionalità** per **accedere alle risorse del sistema** e **offre agli altri processi un insieme di funzioni** (definite in librerie standard) per accedere a queste funzionalità.
  - Queste **funzioni** (disponibili per tutti) sono chiamate **System Calls** e possono avere argomenti (definiti prima della chiamata).
  - Sono **eseguite nel Kernel Space** (con privilegi alti) e una volta eseguite, i **dati vengono restituiti al chiamante** (nel User Space).
  - Quando un processo utente esegue una **syscall**, **notifica al Kernel dove salvare i dati generati**, così dopo può recuperarli.

- Le System Calls hanno sempre un codice d'uscita (valore intero):
  - **0**: **successo**
  - **1-255**: **errore** (o significati particolari)
  - **-1**: **fallimento**
    - La **chiamata non è stata eseguita** (problemi con permessi, se la risorsa esiste o no, ecc) fa riferimento a singole chiamate a basso livello che non sono state eseguite.
    - Bisogna sempre verificare che il codice di ritorno non sia -1 (è comodo crearsi la propria funzione che esegue la Syscall e controlla che sia andato tutto bene).
    - Oltre a restituire il codice d'uscita -1 di fallimento, le **chiamate di sistema che falliscono settano anche la variabile errno** (variabile globale), specificando l'errore che si è verificato.

- **CATEGORIE DI SYSCALL**:
  - **Processi e controllo memoria**: richiesta di memoria, info su processi, ecc.
  - **Accesso al file system**: funzionalità che ci permettono di accedere a file o di crearli, leggerli, ecc.
  - **Gestione dispostivi**: accessi a dispositivi particolari e funzionalità relative ai driver.
  - **Informazioni** (generiche).
  - **IPC**: consentono la comunicazione tra processi, creano canali dedicati per la comunicazione.

- **SYSCALLS**:
  - **Exit**:
    - **void exit(int status):**
      - Chiamata di sistema a cui posso passare una variabile intera (status) che **viene usata per uscire da un programma** (termina il processo).
      - Tutti i **descrittori vengono chiusi automaticamente** (**rilascia** in modo automatico tutte le **risorse** e file).
      - **L'argomento d'uscita è restituito al chiamante** (0 in caso di successo o 1-255 per un errore).

```
root@LABSO:/tmp/syscall/exit# gcc -std=gnu99 exit.c -o exit
root@LABSO:/tmp/syscall/exit# cat exit.c
#include <stdlib.h>
#include <stdio.h>

int main(){
  printf("Main\n");
  exit(2); //chiamata di sistema per terminare
  printf("NON VIENE ESEGUITA!");
}
root@LABSO:/tmp/syscall/exit# ./exit
Main
root@LABSO:/tmp/syscall/exit# echo $?
2
root@LABSO:/tmp/syscall/exit#
```

  - **Desctiptors**:
    - I descrittori sono degli indici che ci permettono di accedere al file system e di leggere/scrivere su un file:
      - **int open(…)**
      - **int read(…)**
      - **int write(…)**
      - **int close(…)**
    - Ci sono diverse modalità d'accesso ai file e i permessi a loro associati.

  - **Permissions**:
    - **int chmod(…)**: modifica i permessi del file (lettura/scrittura/esecuzione)
    - **int chwn(…)**: modifica l'autore del file
    - Se abbiamo i permessi di farlo, possiamo modificare i permessi associati ad un file.

- Esempio:

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

#define bufsize 255

//mia funzione open() dove eseguo la exit con controllo
//mia funzione per la read() dove controllo e in caso libero le risorse

int main(){
  int fd; //variabile che contiene identificativo del file aperto (descrittore)
  char buf[bufsize]; //variabile dove salvo i dati del file
  int size; //contiene il numero di byte effettivamente letti
  int loop = 0;

  //chiamata di sistema che apre un canale con un file e resituisce l'identificativo
  fd = open("fd.c", O_RDONLY);
  //Controllo se la e' andata a buon fine
  if(fd==-1){
    printf("?Error: open");
    exit(2); //restituisco un mio codice d'errore
  };
  printf("FD=%d\n", fd);

  size=1;
  while(size>0){
    loop++;
    printf("\nLoop %d\n", loop);

    //Chiamata di sistema che legge i dati contenuti nel file rappresentato da fd
    //di salvare i dati letti all'interno dell'area di memoria puntata da buf
    //e di leggere fino a bufsize - 1 byte (minore della dimensione della variabile buf)
    size=read(fd, buf, bufsize-1); //permette di leggere il file
    if(size==-1){
      printf("?Error: read");
      close(fd); //libero le risorse
      exit(3);
    };

    if(size > 0){
      //se ho letto qualcosa, imposto come carattere terminatore lo 0 e stampo
      buf[size]='\0';
      printf("%s", buf);
    }
  };

  printf("size=%d\n", size);
  close(fd); //chiusura del canale e liberazione delle risorse
  return 0;
}
```

- **Processes**:
  - Forniscono indicazioni ed informazioni riguardo ai processi.
  - Funzioni:
    - **pit_t getpid()**: identificativo del processo.
    - **pit_t getppid()**: identificativo del padre del processo.
    - **int getuid()**: identificativo dell'utente.

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(){
  //Posso usare int o pid_t (sono compatibili)
  int pid; //processo
  int ppid; //padre
  int uid; //utente

  pid=getpid(); //recupero pid per processo
  ppid=getppid(); //recupero pid del padre
  uid=getuid(); //recupero id dell'utente

  printf("PID=%d\n", pid);
  printf("PPID=%d\n", ppid);
  printf("UID=%d\n", uid);
  return 0;
}
```

- **External call**:
  - **int system(char *cmd)**:
    - Il processo che la esegue **crea un nuovo processo figlio**.
    - Questo **processo figlio esegue**, **con la shell di sistema**, **il comando che viene passato** come parametro alla funzione.
    - Una volta **eseguito il comando**, il **processo figlio termina** e il **processo padre continua la sua esecuzione**.

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

int main(){
  printf("Going to run commands!\n\n");
  if(system("ls -l ; sleep 3s")==-1){
    printf("?Error: %d\n", errno);
  };
  printf("\n\nEnd (%d)\n", errno);
  return 0;
}
```

ESERCIZIO:

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

//al posto di char * ora posso usare string (e' un alias)
typedef char* string;

//Errori vengono stampati sul canale stderr=2 non su stdout=1
int warn(string msg, int code){
    //printf("%s\n", msg);
    //uso la funzione fprintf() che permette di specificare il canale d'uscita (stderr)
    fprintf(stderr, "%s\n", msg);
    return code;
};

void nl(){
   printf("\n");
};

void feedback(string filename, int size){
    printf("%s : %d bytes\n", filename, size);
}

int computeSize(string filename){
    int code=2;
    int fd;
    int size;

    fd=open(filename, O_RDONLY);
    if(fd>-1){
      //int dim = lseek(file, offset, option); //restituisce la posizione corrente calcolata dall'inizio
      //lseek(fd, 5, SEEK_SET); //SEEK_SET impongo che il corsore parta con l'offset scelto
      //lseek(fd, 10, SEEK_CUR); //SEEK_CUR prende il cursore corrente e ci aggiunge 10 (spostamento)
      //lseek(fd, -1, SEEK_END); //SEED_END posiziona il cursore alla fine del file e torna indietro
      size=lseek(fd, 0, SEEK_END); //cursore viene posizionato all'ultima posizione possibile (dimFile-0)
      feedback(filename, size);
      code=0; //andato tutot bene
      close(fd); //rilascio le risorse
    }else{
      code = warn("?Error opening file", 3);
    };
    return code;
};
```

```c
int main(int argc, char *argv[]){
  //char *argv[] e' un puntatore di stringhe (quindi puntatore di puntatori)
  //argv[] viene valorizzato dal nome dell'eseguibile e poi da eventuali parametri
  //printf("argc=%d, argv=%s,%s,%s\n", argc, argv[0], argv[1], argv[2]);

  nl();
  int ret=0;
  string filename;

  //Se l'eseguibile non viene invocato con "nomeExe" e con un solo parametro genero un errore
  if(argc!=2){
    //printf("?Errore\n");
    //ret=1;
    ret = warn("?Errore", 1);
  }else{
    filename=argv[1]; //prendo il nome del file
    ret = computeSize(filename);
  };
  nl();
  return ret;
}
```