

Laboratorio di Sistemi Operativi

a.a. 200/2001

Mario Guarracino

Introduzione

Tutti i sistemi operativi forniscono **servizi**

Eseguire un nuovo programma, aprire un file, allocare la memoria sono esempi di servizi forniti da un sistema operativo

Descriveremo adesso alcuni servizi forniti da varie versioni del sistema operativo Unix

Autenticazione

Quando accediamo ad un sistema Unix forniamo la nostra **login** e la **password**

La nostra login viene cercata nel file di password del sistema (**es. /etc/passwd**)

```
$ grep mariog /etc/passwd  
mariog:x:212:50: Mario Guarracino:/home/mariog:/bin/csh  
$
```

Logging in

Una volta eseguito il login, possiamo dare **comandi** al sistema, utilizzando un interprete dei comandi (*shell*)

Una *shell* è un **interprete** che accetta l'input dell'utente ed esegue i **comandi**.

Tra le shell più utilizzate ci sono:

- La **Bourne shell**, /bin/sh
- La **C shell**, /bin/csh
- La **Korn shell**, /bin/ksh
- La **Basic shell**, /bin/bash

Ciascun utente ha una **shell** che viene attivata **di default**.

File e Directory

Il *filesystem* di Unix è un insieme di file e directory organizzato in maniera gerarchica.

La radice (*root*) del filesystem è una directory di nome /.

Una *directory* è una *tabella* che contiene un *nome* e un puntatore ad una *struttura* di informazioni, per ciascun file o directory in essa contenuto.

Gli attributi in tale struttura riguardano:

- tipo di file,
- grandezza,
- proprietario,
- permessi,
- ...

Esempio 1

```
#include <sys/types.h>
#include <dirent.h>
#include "ourhdr.h"

int main(int argc, char *argv[ ])
{
    DIR          *dp;
    struct dirent *dirp;

    if (argc != 2)
        err_quit("a single argument (the directory name) is required");

    if ( (dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

File I/O

Quando un programma apre un file o lo crea, il kernel ritorna un *descrittore* utilizzato per tutte le successive operazioni di lettura e scrittura.

I *descrittori di file* sono degli interi non negativi che identificano i file utilizzati da un particolare programma.

Quando un programma viene mandato in esecuzione, la shell apre tre descrittori:

0 std input

1 std output

2 std error

Di norma questi descrittori sono collegati con il terminale.

Esempio 2

```
#include    "ourhdr.h"

#define BUFSIZE  8192

int
main(void)
{
    int    n;
    char  buf[BUFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Esempio 3

```
#include    "ourhdr.h"

int main(void)
{
    int     c;

    while ( (c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Programmi e Processi

Un *programma* è un file *eseguibile* che risiede nel filesystem.

Esso viene *caricato in memoria* ed eseguito dal kernel quando viene eseguita una chiamata ad una delle funzioni di **exec**.

Un *programma* in *esecuzione* viene detto *processo*.

Ciascun *processo* è *identificato* univocamente da un *intero* non negativo.

Esempio 4

```
#include    "ourhdr.h"

int
main(void)
{
    printf("hello world from process ID %d\n", getpid());
    exit(0);
}
```

Esempio 5

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"
```

```
int main(void)
```

```
{
```

```
    char buf[MAXLINE];
```

```
    pid_t pid;
```

```
    int status;
```

```
    printf("%% "); /* print prompt (printf requires %% to print %) */
```

```
    while (fgets(buf, MAXLINE, stdin) != NULL) {
```

```
        buf[strlen(buf) - 1] = 0; /* replace newline with null */
```

```
        if ( (pid = fork()) < 0)
```

```
            err_sys("fork error");
```

```
        else if (pid == 0) { /* child */
```

```
            execlp(buf, buf, (char *) 0);
```

```
            err_ret("couldn't execute: %s", buf);
```

```
            exit(127);
```

```
        } /* end child */
```

```
    }
```

Esempio 5 (cont.)

```
/* parent */
    if ( (pid = waitpid(pid, &status, 0)) < 0)
        err_sys("waitpid error");
    printf("%%\n");
}
exit(0);
}
```

Caratteristiche dell'ANSI C

Nell' header `<unistd.h>` si trovano i prototipi di funzioni per molti servizi di Unix

```
ssize_t read(int, void *,size_t);  
ssize_t write(int, const void *,size_t);  
pid_t getpid(void);
```

Tali prototipi forniscono informazioni aggiuntive al compilatore e permettono di eseguire dei controlli sulla correttezza degli argomenti.

Caratteristiche dell'ANSI C

Il prototipo della funzione **getpid** definisce il suo valore di **ritorno** di tipo **pid_t**, così come **read** e **write** ritornano valori di tipo **ssize_t** e argomenti **size_t**.

I tipi di dati che finiscono in **_t** sono detti *tipi primitivi* di dati e sono definiti in **<sys/types.h>**.

Il loro scopo è che i programmi non utilizzino specifici tipi di dati che possono differire nelle implementazioni di Unix.

Gestione degli Errori

Per segnalare una situazione di **errore**, le funzioni di Unix spesso ritornano un valore negativo e l'intero **errno** è inizializzato ad un valore fornisce informazioni ulteriori.

```
extern int errno;
```

Nell'header **<errno.h>** si trova la corrispondenza tra i valori che **errno** può assumere e le costanti numeriche ad essi associate.



Gestione degli Errori

Le funzioni del C per gestire gli errori sono:

```
#include <string.h>
char *strerror(int errnum);
```

che associa ad **errnum** ad un messaggio di errore, e

```
#include <stdio.h>
void perror(const char *msg);
```

che stampa la stringa puntata da **msg**, seguita da due punti, uno spazio e il messaggio di errore, seguiti da un carattere di nuova linea

Esempio 6

```
#include    <errno.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

Identificazione degli Utenti

Lo *user ID* è il numero che identifica univocamente ciascun utente nel sistema.

Il superutente ha user ID uguale a 0

Esempio 7

```
#include    "ourhdr.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Identificazione degli Utenti

Ciascun **utente** appartiene ad un **gruppo** il cui identificativo viene detto **group ID**.

Più **utenti** possono appartenere allo **stesso gruppo** (es. studenti, staff, ospiti,...).

Lo scopo dei **gruppi** è di raggruppare utenti per far **condividere le risorse**.

La **corrispondenza** tra **identificativi** numerici e **gruppi** si trova di solito nel file **/etc/group**.

Segnali

Una tecnica per **notificare** ad un processo l'**occorrenza** di una certa **situazione** è quella di utilizzare i **segnali**.

Se, ad esempio un processo effettua una divisione per zero, esso riceve il segnale **SIGFPE**.

Un processo che riceve un segnale può:

1. **ignorare** il segnale,
2. lasciare che venga **eseguita** l'azione di **default**,
3. fornire una **funzione** da **eseguire** all'atto della **ricezione** del **segnale**.

Esempio 8

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include "ourhdr.h"

static void sig_int(int); /* our signal-catching function */

int main(void)
{
    char buf[MAXLINE];
    pid_t pid;
    int status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal error");

    printf("%% "); /* print prompt */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */
    }
}
```

```

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {          /* child */
    execlp(buf, buf, (char *) 0);
    err_ret("couldn't execute: %s", buf);
    exit(127);
}                             /* parent */
if ( (pid = waitpid(pid, &status, 0)) < 0)
    err_sys("waitpid error");
printf("%d\n", pid);
}
exit(0);
}
void sig_int(int signo)
{
    printf("interrupt\n");
}

```

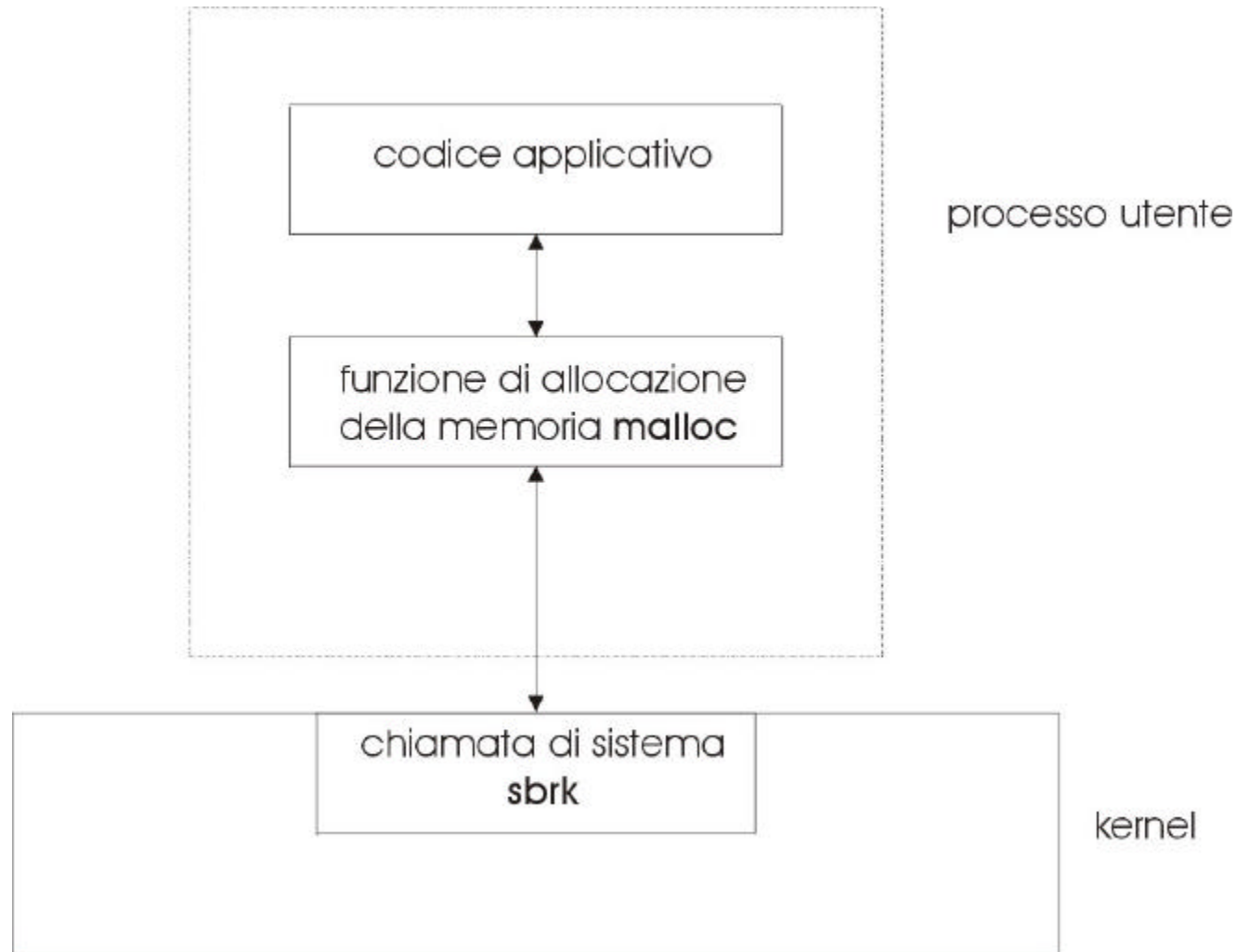
Chiamate di Sistema e Funzioni di Libreria

Tutti i **sistemi operativi** forniscono **interfacce** attraverso cui i programmi richiedono **servizi**.

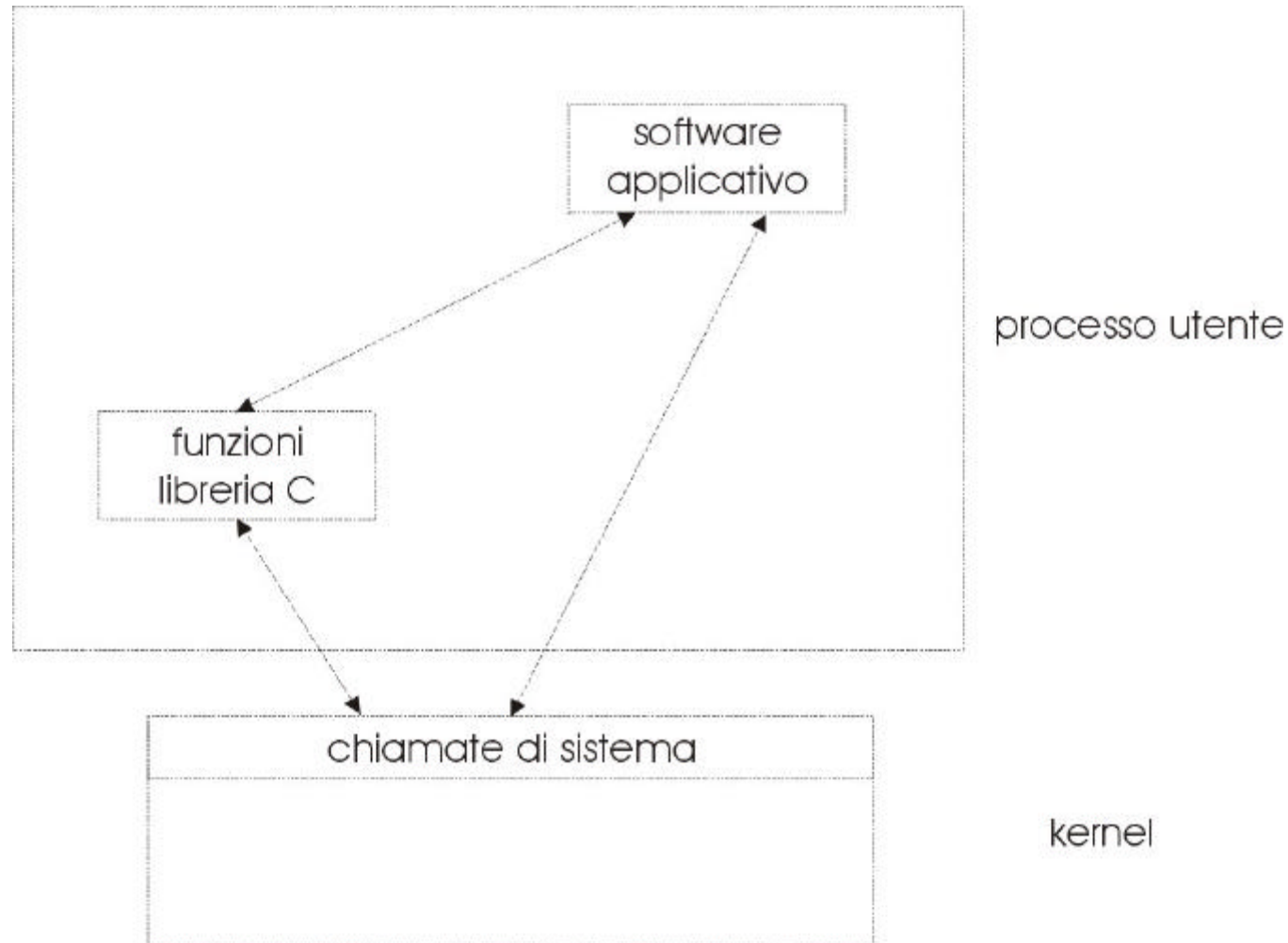
Tutte le implementazioni di Unix forniscono un insieme definito e limitato di **punti di accesso** al kernel, detti **chiamate di sistema**.

Dal punto di vista del programmatore, sia le **chiamate di sistema**, sia le funzioni della **libreria standard** del C appaiono come **funzioni del C**.

Chiamate di Sistema e Funzioni di Libreria



Chiamate di Sistema e Funzioni di Libreria



Sommario

Abbiamo visto come il sistema operativo Unix

- autentica ed identifica i suoi utenti
- permette di leggere file e directory
- controlla i processi

Vedremo in dettaglio le funzioni che ci permettono di eseguire operazioni di input output su file

Cose da fare per la prossima volta

- Scaricare **apue.tar** da www.kohala.com
- Guardare come sono fatti **ourhdr.h**, **err_sys**, **err_quit**
- Far girare gli esempi senza **ourhdr.h**

Lecture per la prossima volta

Leggere il capitolo 7 del Kernighan & Ritchie