

IL SISTEMA UNIX

Parte Seconda

**Programmazione di sistema e
system call**

Roberto Polillo

**Corso di Sistemi Operativi
Corso di Laurea in Informatica
Università di Milano**

Ultimo aggiornamento: maggio 1997

SYSTEM CALLS

Per utilizzare i servizi del sistema operativo, il programmatore ha a disposizione una serie di funzioni di libreria, dette **system calls** (da una settantina a oltre 200, a seconda della versione di Unix)

POSIX.1 standardizza un nucleo base di tali primitive (circa 120), a livello di interfaccia sorgente C, ad es.:

```
int open(const char *path,  
        int oflag, ...);
```

(Poi anche Fortran e
Ada)

POSIX.1

- POSIX standardizza un nucleo base di funzioni usate da applicazioni di tipo "normale" (approccio "minimalista").

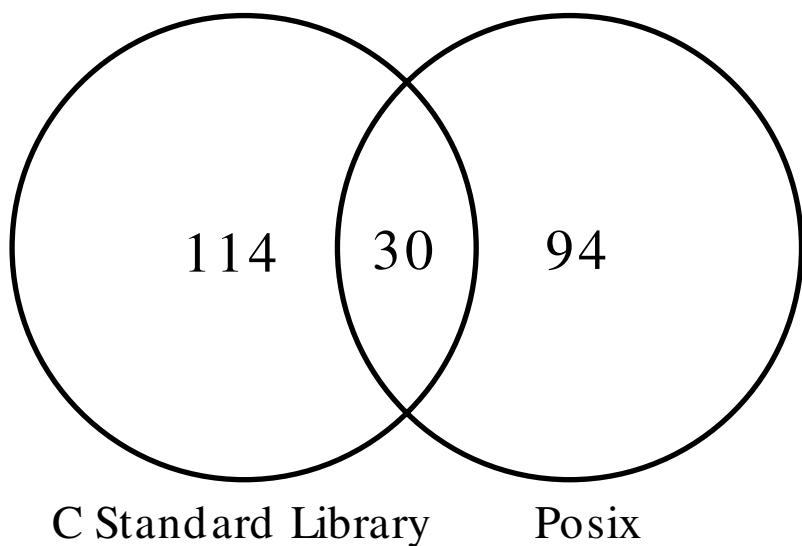
Ad es., non definisce alcuna funzione di system administration

- I sistemi POSIX forniscono normalmente delle estensioni non POSIX (primitive e/o opzioni addizionali)

POSIX E C STANDARD LIBRARY

POSIX.1 include anche alcune primitive della C Standard Library:

Numero di funzioni:

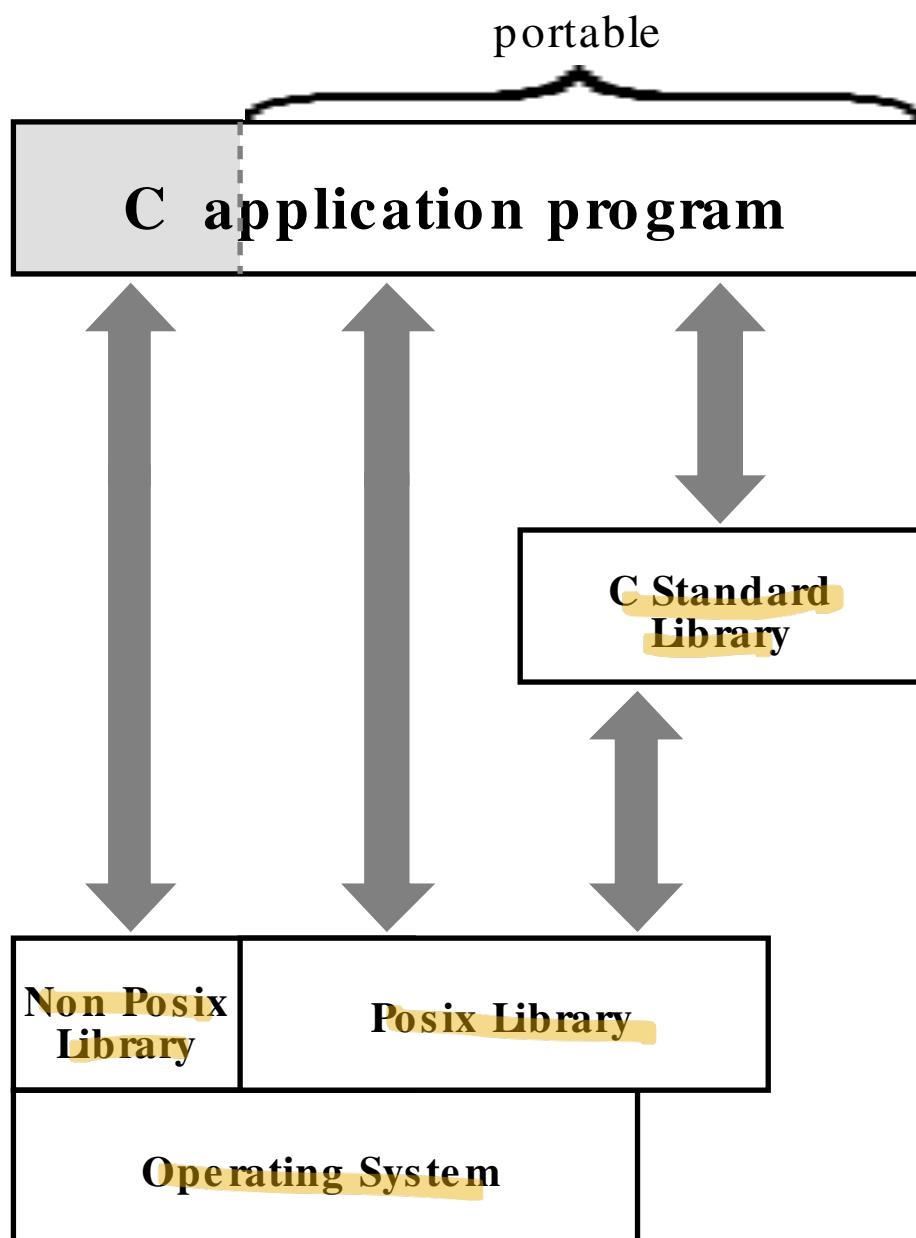


Esempio:

- | | |
|----------|-----------------------------|
| open () | è POSIX |
| fopen () | è Standard C e POSIX |
| sin () | è Standard C |

PORTABILITÀ DELLE APPLICAZIONI

Un sorgente C Standard conforme a POSIX può essere eseguito, una volta ricompilato, su qualsiasi sistema POSIX dotato di un ambiente di programmazione C Standard



SYSTEM CALL: CLASSI PRINCIPALI

- Gestione file e directory
- Gestione processi
- Comunicazione fra processi

Nel seguito, assumeremo note le primitive della C Standard Library, e presenteremo solo le principali system call (POSIX)

Manuale utente (in linea)

man (abbreviazione di "manual") e' la forma tradizionale della documentazione in linea per i sistemi operativi Unix e Linux. Le "man pages" sono file formattati in modo speciale; Digitando **man nomecomando** verra' visualizzata la pagina del manuale relativa al comando denominato **nomecomando**.

Le pagine del manuale vengono raggruppate in sezioni numerate. Ad esempio e' possibile incontrare l'indicazione **man(1)**. Questa indicazione significa che **man** (il comando) e' documentato nella sezione 1 (comandi utente); Specificare la sezione nella quale man ricerca la pagina da visualizzare e' utile nel caso di argomenti multipli con lo stesso nome.

Sezione 1 comandi utente (soltanto l'introduzione)

Sezione 2 chiamate di sistema

Sezione 3 chiamate delle librerie C

Sezione 4 dispositivi (es.: hd, sd,)

Sezione 5 formati dei file e protocolli (es.: /etc/passwd, nfs)

Sezione 6 giochi (introduzione)

Sezione 7 convenzioni, pacchetti macro, etc. (es.: nroff, ascii)

Sezione 8 manutenzione di sistema

In aggiunta a **man(1)**, ci sono i comandi **whatis(1)** e **apropos(1)**, il cui scopo comune e' quello di agevolare la ricerca di informazioni all'interno del sistema delle pagine del manuale.

whatis produce una descrizione molto breve dei comandi di sistema, una sorta di guida di riferimento tascabile per i comandi.

apropos viene utilizzato per la ricerca delle pagine di manuale contenenti un dato argomento.

GESTIONE ERROTI

La maggior parte delle system call restituisce il valore -1 in caso di errore

... ed assegna lo specifico codice di errore alla variabile globale

```
extern int errno;
```

Nota:

Se la system call ha successo, errno non viene resettato

LO HEADER FILE `errno.h`

Lo header file `errno.h` contiene la definizione dei nomi simbolici dei codici di errore

Esempio:

```
# define EPERM      1     /* Not owner */  
  
# define ENOENT     2     /* No such file or directory */  
  
# define ESRCH      3     /* No such process */  
  
# define EINTR      4     /* Interrupted system call */  
  
# define EIO        5     /* I/O error */  
  
...
```

LA PRIMITIVA perror

```
void perror (const char *str )
```

"print error" (C&P)

- converte il codice in errno in un messaggio in inglese, e lo stampa anteponendogli str:

str : messaggio di errore

Esempio:

```
...
fd=open ("nonexist.txt", O_RDONLY);
if (fd== -1) perror ("main");
...
--> main: No such file or directory
```

2. PRIMITIVE DI GESTIONE FILES E DIRECTORIES

Cosa fa SO quando gli chiediamo di aprire / leggere un file? DOMANDA ORALE

FUNZIONI PRINCIPALI

- Creazione di files, directory, file speciali
- Apertura e chiusura di files
- Accesso a files
- File e record locking
- Creazione e distruzione di link
- Lettura degli attributi di un file
- Cambiamento degli attributi di un file
- Cambiamento della directory corrente
- Redirezione e pipeline
- Montaggio e smontaggio di un file system
(non POSIX)

GESTIONE FILES: FILOSOFIA

Un file per essere usato deve essere aperto (open)

La open: Non modifica il file, ma mette al SO un'indicazione di accesso al file

- localizza il file nel file system attraverso il suo pathname
- copia in memoria il descrittore del file (i-node)
- associa al file un intero non negativo (file descriptor), che verrà usato nelle operazioni di accesso al file, invece del pathname

Un processo usa anche file standard. Non sono vani propri file ma sono usati per lettura

I file standard non devono essere aperti, perché sono aperti dalla shell. Sono associati ai file descriptor 0 (input), 1 (output) e 2 (error). Questo permette di realizzare la redirezione

Un processo che punta ha già dei file aperti disponibili: stdin, stdout, stderr

La close rende disponibile il file descriptor per ulteriori usi

↓
Buona norma chiudere file che non più serve

ESEMPIO

```
int fd;  
...  
fd=open(pathname, ...);  
if (fd== -1) { /*gestione errore*/ }  
...  
read(fd, ...);  
...  
write(fd, ...);  
...  
close(fd);
```

Nota:

Un file può essere aperto più volte, e quindi avere più file descriptor associati contemporaneamente

APERTURA DI UN FILE: `open`

→ Può avere 0, 2 o 3 parametri

```
int open(const char *path,  
        int oflag, ...);
```

↳ Significa che ci sono 3 parametri

- apre (o crea) il file specificato `pathname`, secondo la modalità specificata in `oflag`
- restituisce il file descriptor con il quale ci si riferirà al file successivamente (o -1 se errore)

path `pathname` assoluto o relativo

oflag	<code>O_RDONLY</code>	read-only
	<code>O_WRONLY</code>	write-only
	<code>O_RDWR</code>	read and write
	<code>O_APPEND</code>	append
	<code>O_CREAT</code>	creazione del file

... ↳ Non è una vera e propria macro, da cui l'origine

... permessi iniziali (se `O_CREAT`)

↳ 3° parametro con 8 permessi:

Rappresentati con 3 bit:

`rwx` `rwx` `rwx`
read write execution respectively

1 consentito
0 non consentito

2° Terna: utente che hanno stesso group-id dal proprietario. Group-id rappresenta un insieme di utenti vicini al proprietario (che in genere hanno permessi simili).

Ultimi 3 servono per tutti gli altri.

Esempio: un utente che è proprietario prova ad aprire file per leggerlo:

Si confronta user-id con quello che vuole provare ad aprire. Se sono uguali, vede la prima terna per vedere se qualcosa si può fare.

Se no, si confronta il group-id. Altrimenti gli ultimi 3.

Oltre questi 3 ce n'è un altro chiamato **Setuid**, associato a dei file eseguibili.

Tutti che serve a consentire a certi utenti che non hanno il permesso di fare certe op.

Esempio: file passwd che porta una memoria criptografata la pw. Problema: non scava.

Usato da utente per cambiare la sua password. Comando che cambia password è un exe-Da proprietà di root (molti privilegi), ma ha bit setuid alto. Ml sì, quando ha questo bit alto, assegna a utente un effective-user-id (insieme al real-user-id). Efficacità può usare un altro valore. Eff. è quello che viene usato per il confronto. Utente per cambiare la sua password utente acciuffa (un modulino comune) e prelegge chi root. Con setuid

ULTERIORE MECCANISMO DI SICUREZZA: umask.

Se creo file e do' come parametro 777 di esadecimale (tutti 1) per la creazione però non ho il permesso come ho avuto salvati; perché la stringa è messa in AND col negativo di una maschera (default 022): $000010010 \rightarrow$ quando avrò 0 sul permesso di scrittura per il gruppo e per gli altri utenti esterni. Salvaguardia il file.

Se voglio avere tutti 1 devo cambiare la maschera. La system call umask(0) restituisce maschera a tutti 0.

CREAZIONE DI UN FILE NORMALE: **creat**

```
int creat(const char *path,  
          mode_t mode);
```

↳ Sono io che voglio assegnare a quel file. Creat è anche un'apertura.

- crea un nuovo file normale di specificato pathname, e lo apre in scrittura
 ↳ path sempre con estensione .m scrilla
- mode specifica i permessi iniziali; l'owner è l'effective user-id del processo
- se il file esiste già, lo svuota (owner e mode restano invariati)
- restituisce il file descriptor, o -1 se errore

CHIUSURA DI UN FILE: **close**

```
int close(int fildes);
```

- chiude il file descriptor **fildes**
- restituisce l'esito dell'operazione (0 o -1)

Nota:

Quando un processo termina, tutti i suoi files vengono comunque chiusi automaticamente.

LETTURA DI UN FILE: **read**

indirizzo di memoria in
↑ con subire wo' che legge

```
ssize_t read(int fildes, void *buf,  
           size_t nbyte);
```

↳ Quanti byte vuoi leggere

- legge in ***buf** una sequenza di **nbyte** bytes dalla posizione corrente del file **fildes**
- aggiorna la posizione corrente
- restituisce il numero di bytes effettivamente letti, o **-1** se errore

SCRITTURA DI UN FILE: **write**

```
ssize_t write(int fildes,  
const void *buf, size_t nbyte);  
↑ da dove inizia a scrivere in memoria ↓ Quanti byte
```

- scrive nel file **fildes** **nbyte** bytes da ***buf**, a partire dalla posizione corrente
- aggiorna la posizione corrente
- restituisce il numero di bytes effettivamente scritti, o -1 se errore

ESEMPIO

Copiare lo standard input sullo standard output

```
#include ...
#define BUFFSIZE 8192
int main(void)
{
    int n;
    char buf[BUFFSIZE];
    while( (n=read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            perror("main");
    if (n<0)
        perror("main");
    exit(0);
}
```

Note:

- È corretto usare, come file descriptor:
STDIN_FILENO e STDOUT_FILENO (definiti
in <unistd.h>) al posto di 0 e 1
- standard input e output non vengono aperti né
chiusi: perché sono già aperti

POSIZIONAMENTO SU UN FILE: `lseek`

```
off_t lseek(int fildes,  
off_t offset, int whence);
```

\uparrow numero byte relativo \uparrow indica da dove puntare per spostare la testina

- sposta la posizione corrente nel file `fildes` di `offset` bytes a partire dalla posizione specificata in `whence`:

`SEEK_SET` dall'inizio del file

`SEEK_CUR` dalla posizione corrente

`SEEK_END` dalla fine del file

- restituisce la posizione corrente dopo la `lseek`, -1 se errore

Nota:

La `lseek` non effettua alcuna operazione di I/O

CREAZIONE DI UN HARD LINK: **link**

```
int link(const char *existing,  
         const char *new);
```

- crea il nuovo (hard) link new al file existing
- restituisce 0 se ok, -1 se errore

Nota:

Per cambiare nome a un file:

```
int rename(const char *old,  
            const char *new);
```

RIMOZIONE DI UN HARD LINK: `unlink`

```
int unlink(const char *path);
```

- distrugge il link (hard) path e, se si tratta dell'ultimo, dealloca il file
- restituisce l'esito dell'operazione (0 o -1)

Note:

- Se qualche processo sta usando il file, viene solo deallocata la directory entry: il file verrà rimosso quando tutti i file descriptors relativi al file saranno stati chiusi
- Così, un eseguibile può fare `unlink` di se stesso, e proseguire comunque la esecuzione

GESTIONE DI DIRECTORIES

Per creare una directory (vuota, con . e ..) :

```
int mkdir(const char *path,  
mode_t mode);
```

Per rimuovere una directory:

```
int rmdir(const char *path);
```

Per cambiare la working directory:

```
int chdir(const char *path);
```

PERMESSI DI UN FILE: **chmod**

```
int chmod(const char *path,  
          mode_t mode);  
    > deve essere il proprietario per funzionare
```

"**change mode**"

- **cambia i permessi del file path, come specificato in mode (S b,r)**
- **restituisce l'esito dell'operazione (0 o -1)**

Nota:

Per cambiare owner e gruppo di un file:

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

Per cambiare i tempi di ultimo accesso e di ultima modifica:

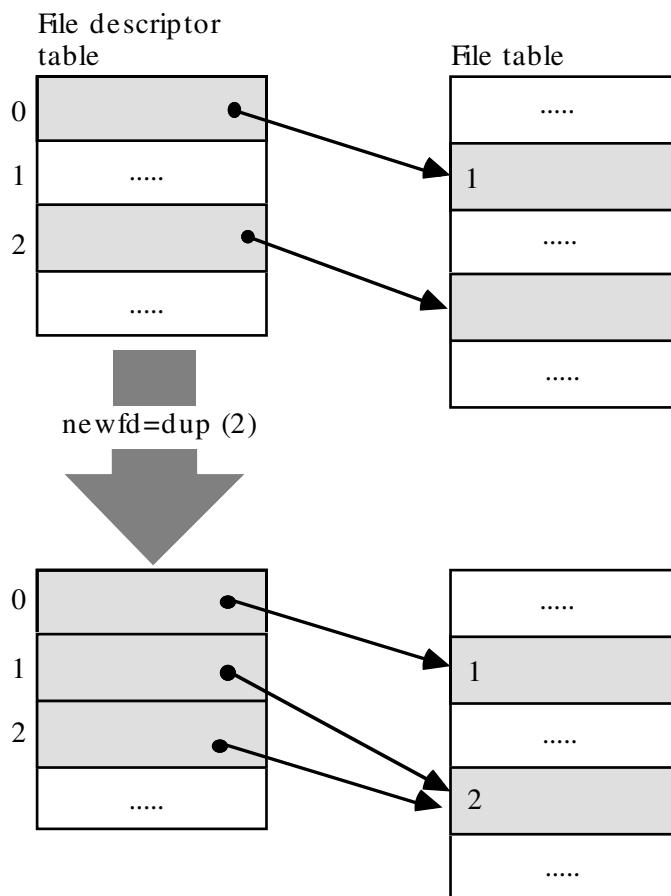
```
int utime(const char *path,  
          const struct utimbuf *times);
```

DUPLICAZIONE DI FILE DESCRIPTORS: dup

```
int dup(int fildes);
```

- associa al file descriptor `fildes` un file descriptor aggiuntivo (il primo libero a partire da 0)
- restituisce il nuovo file descriptor, o -1 se errore

Esempio:



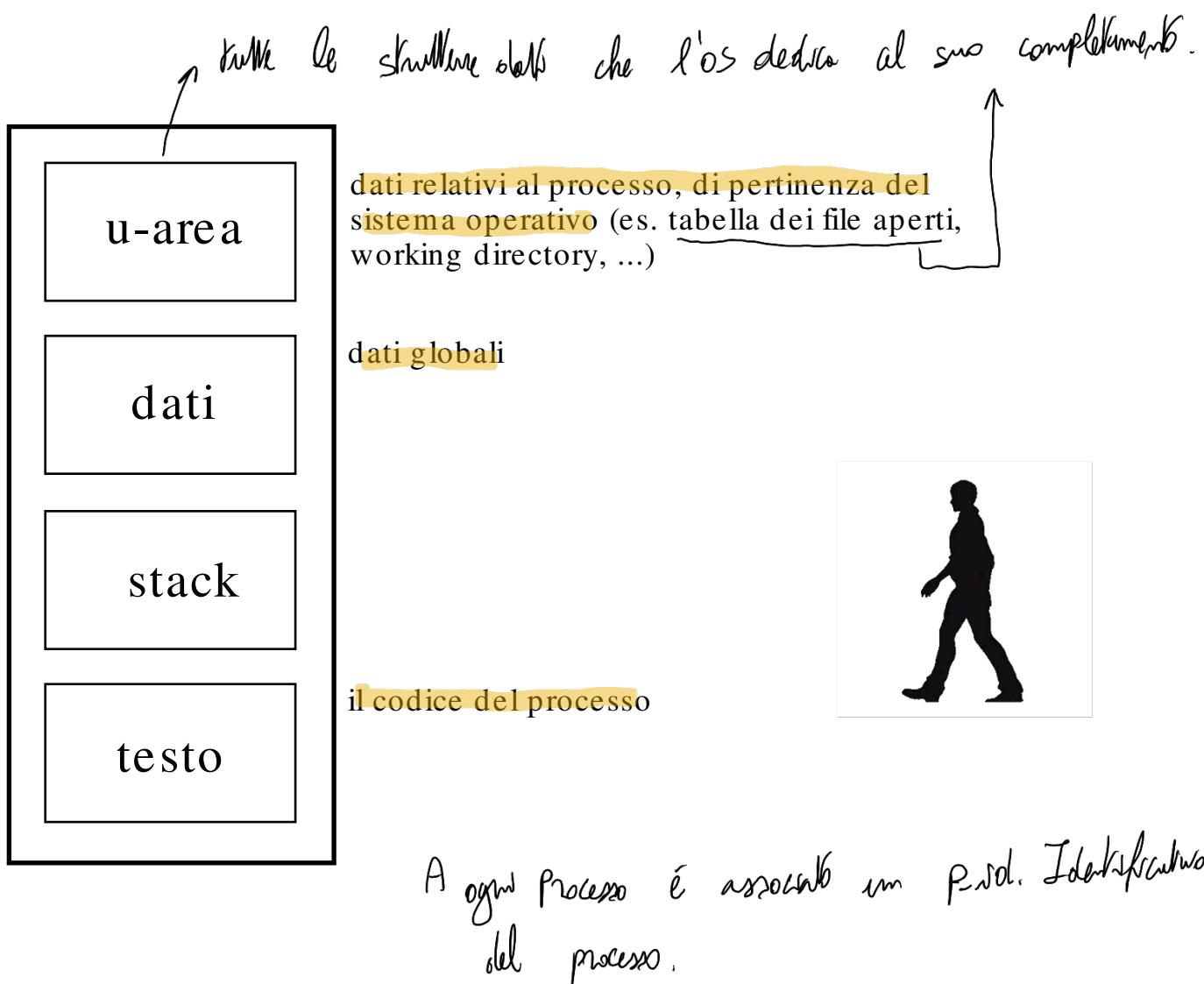
SINTESI PRINCIPALI PRIMITIVE DI GESTIONE FILE

Primitive POSIX	Descrizione
fildes=creat(path, mode)	crea un nuovo file
fildes=open(path, oflag, ...)	apre un file
r=close(fildes)	chiude un file aperto
n=read(fildes, &buf, nbyte)	legge i dati da un file nel buffer
n=write(fildes, &buf, nbyte)	scrive i dati da un buffer nel file
newoffset=lseek(fildes, offset, whence)	muove il puntatore di file in qualche punto del file
r=link(existing, new)	crea un nuovo link a un file
r=unlink(path)	rimuove un link
r=rename(oldpath, newpath)	cambia nome a un file
r=stat(path, &buf)	fornisce informazioni sullo stato del file
r=fstat(fildes, &buf)	
r=mkdir(path, mode)	crea una nuova directory
r=rmdir(path)	rimuove una directory vuota
r=chdir(path)	cambia la working directory
newfildes=dup(fildes)	duplica il file descriptor fildes
s=pipe(fd_pipe)	crea una pipe senza nome
s=chdir(dirname)	cambia la directory di lavoro
r=chmod(path, mode)	cambia le protezioni di un file
r=chown(path, owner, group)	cambia l'owner e/o il gruppo di un file
r=utime(path, &time)	cambia il tempo di accesso e modifica di un file

3. PRIMITIVE DI GESTIONE PROCESSI

IMMAGINE DI MEMORIA DI UN PROCESSO

Un processo Unix è costituito da quattro parti principali, che costituiscono la sua immagine:



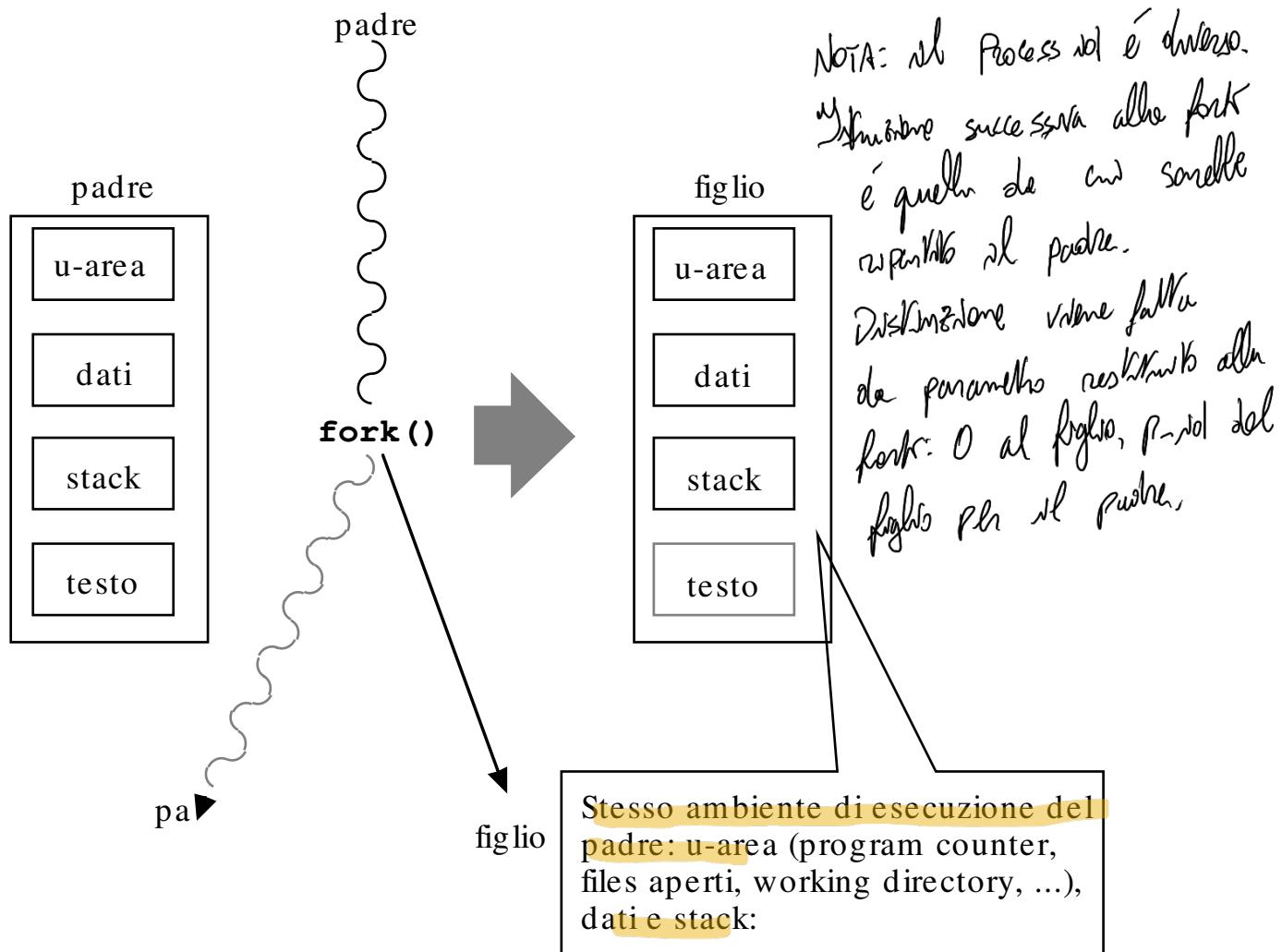
Nota:

Questi aspetti verranno approfonditi nella Parte Terza

CREAZIONE DI UN PROCESSO

In Unix, ogni processo (tranne il primo) è creato da un processo "padre" mediante la chiamata di sistema **fork** ...

... che duplica l'immagine del padre, creando un processo figlio identico:



Mecanismo più semplice per creare un processo.

NOTA: Area codice non viene copiata dal figlio, perché è da sola lettura. Può essere comune fra i 2 sistemi.

LA PRIMITIVA **fork**

```
pid_t fork(void);
```

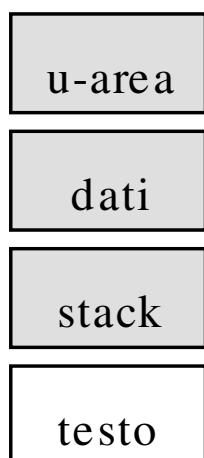
- crea un processo figlio "identico" al padre, e restituisce:

al figlio:	0
al padre:	PID del figlio
in caso di errore:	-1

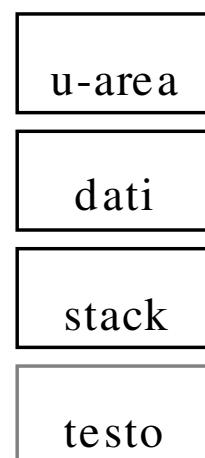
ESEMPIO

```
pid = fork();  
if (pid<0)          /* la fork è fallita */  
else if(pid>0)      /* operazioni del padre */  
else                /* operazioni del figlio */
```

padre



figlio



LE PRIMITIVE **wait** E **waitpid**

↳ Primitiva che fa un processo padre. Si sblocca quando finisce uno dei figli.

```
pid_t wait(int *statloc);
```

- sospende il processo chiamante, fino a che uno dei suoi figli termina
- restituisce il PID del figlio terminato, o -1 se non ci sono figli
- assegna a statloc l'exit status del figlio

```
pid_t waitpid(pid_t pid,  
int *statloc, int options);
```

- permette di specificare di quale figlio si attende la terminazione

LA PRIMITIVA `_exit`

```
void _exit(int status);
```

- termina il processo chiamante
- rende disponibile il valore di `status` al processo padre (che lo otterrà tramite `wait`)

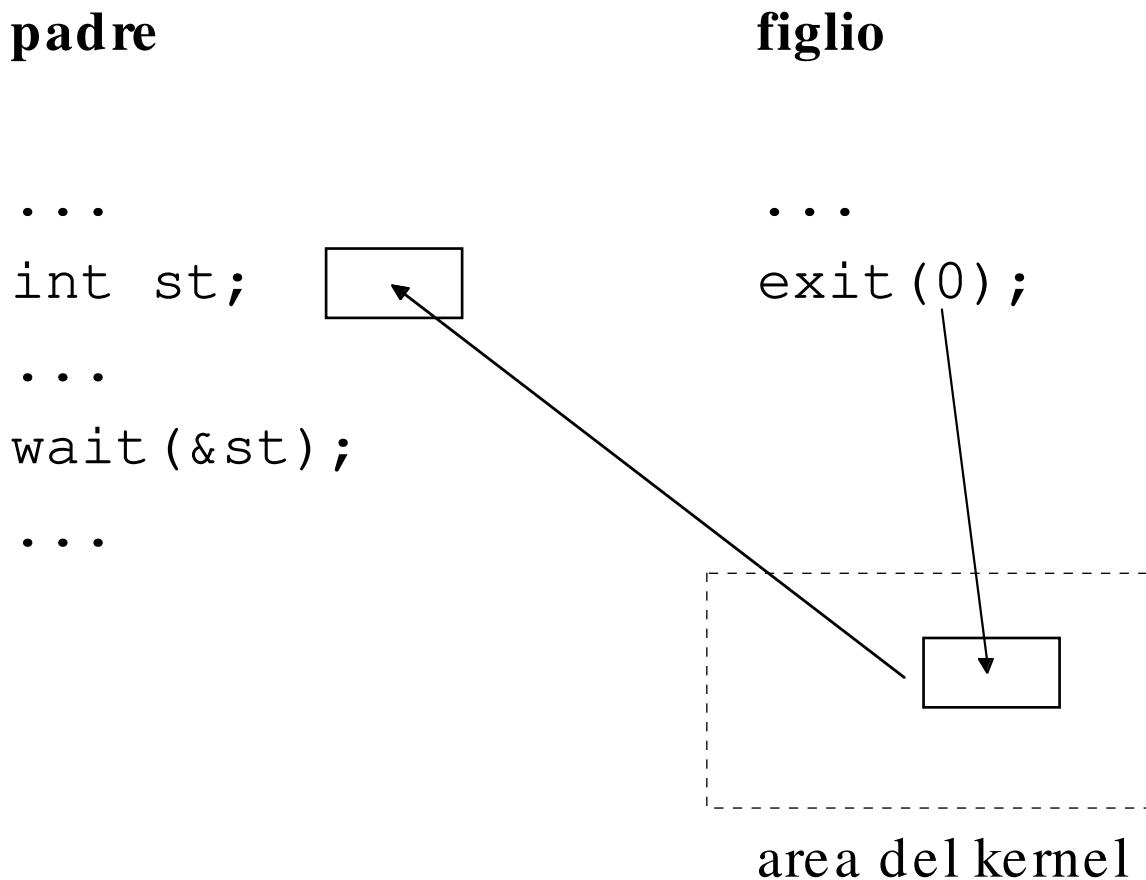
Nota:

La primitiva della C Standard Library:

```
void exit(int status);
```

chiama al suo interno la primitiva POSIX `_exit`

wait E exit



Nota:

- Un processo terminato passa nello stato di **zombie**, e viene definitivamente rimosso dopo che il padre ha ricevuto il suo stato di terminazione con una `wait`
- Un processo zombie occupa un insieme minimale di risorse

Process orfano: processo figlio con padre Kernel

PROCESSI ORFANI

Un processo "orfano" (cioé il cui padre è terminato) viene "adottato" dal processo `init`, quindi un processo ha sempre un padre

Processo che attiva due figli e ne attende la terminazione

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

main () {
    pid_t pid;
    int status;

    /* genera il 1o figlio */
    pid=fork();      Ci sono 2 problemi
    if(! pid) { /* codice del 1o figlio */
        printf("Sono il primo figlio: PID=%d PPID=%d\n", getpid(), getppid());
        exit(0);
    }

    else { /* codice del padre */

        /*genera il 2o figlio*/
        pid=fork();
        if(!pid) { /* codice del 2o figlio */
            printf("Sono il 2° figlio: PID=%d PPID=%d\n",getpid(), getppid());
            exit(0);
        }
        else {/* padre */
            pid=wait(&status); /*attende il 1° figlio che termina*/
            printf("1o figlio terminato: PID=%d STATO=%d\n",pid, status);

            pid=wait(&status); /*attende il 2° figlio che termina*/
            printf("2o figlio terminato: PID=%d STATO=%d\n",pid, status);
            printf("Programma Terminato\n");
        }
    }
}
```

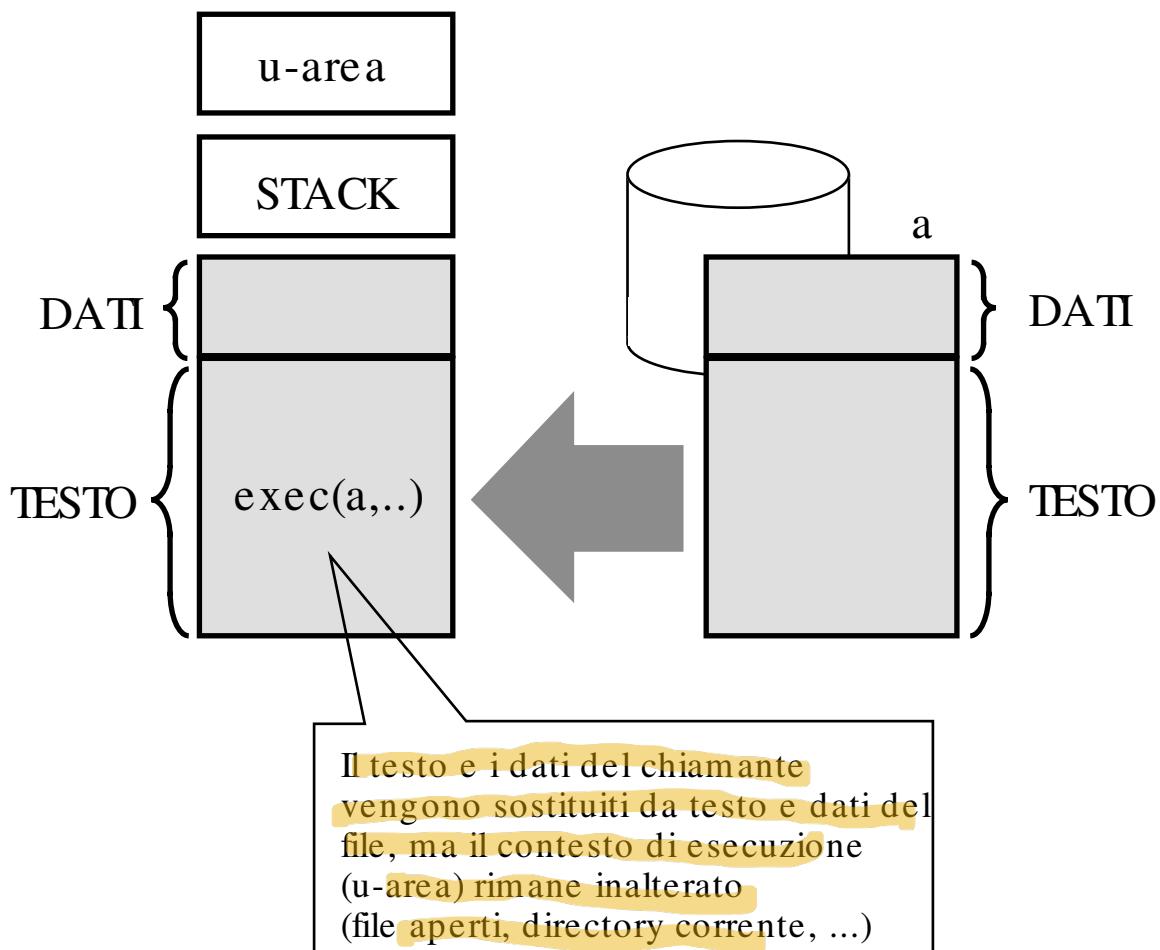
*qui c'è il
↑ Figlio del mio padre*

ESECUZIONE DI UN PROGRAMMA

Processo creato sempre così,

`exec (pathname, argomenti)`

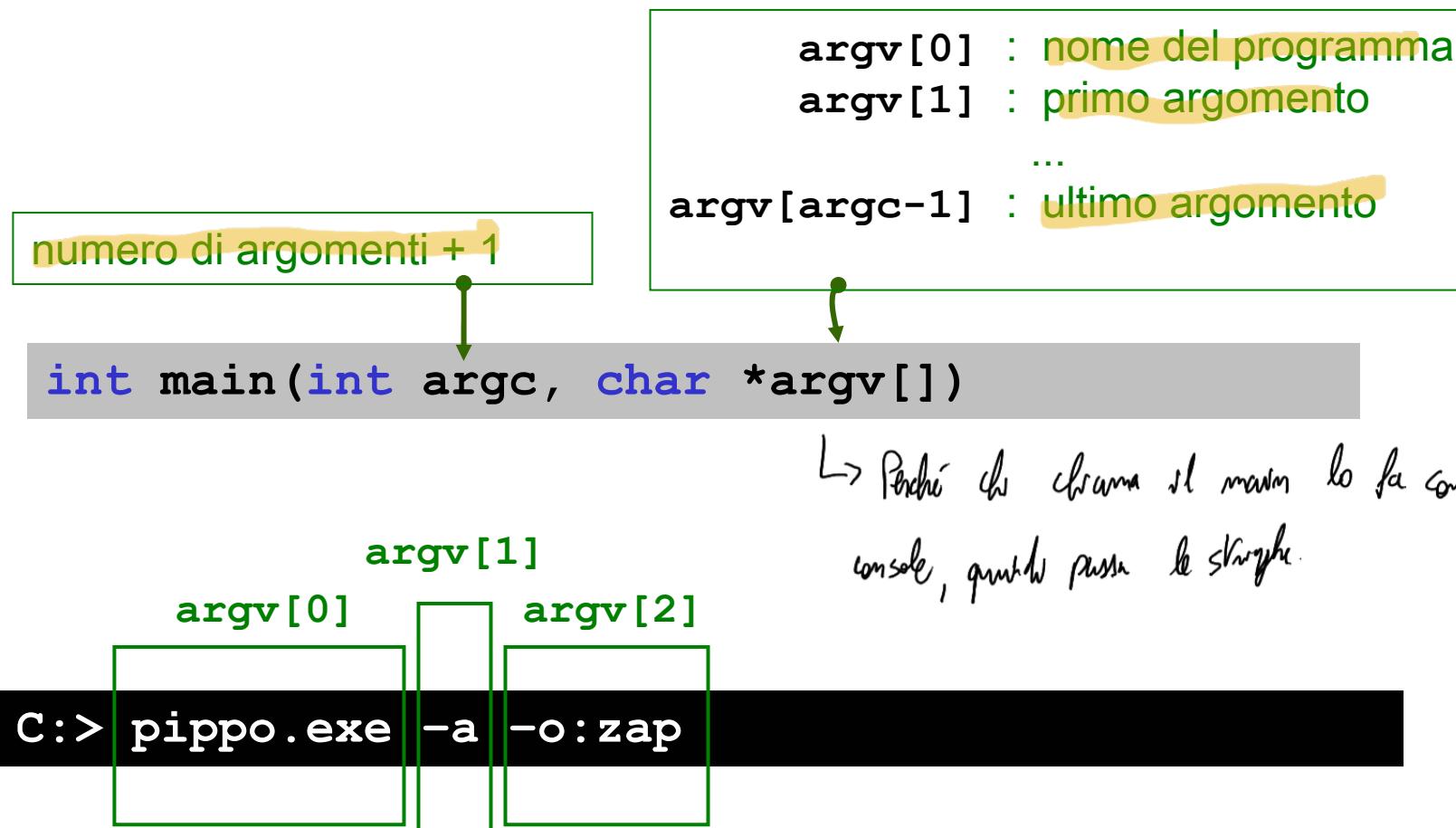
- sostituisce l'immagine del chiamante con il file eseguibile pathname, e lo manda in esecuzione passandogli gli argomenti



Qual è la prima istruzione che eseguirà? La prima del processo chiamato, a meno che non è ritornato -1.

Argomenti main

- Lettura degli argomenti da riga di comando di un programma.



Esempio

```
/* mostra le istruzioni di uso da riga di comando */
void show_usage(){
    printf("nome_programma numero");
}

int main(int argc, char *argv[]) {
    int x;
    if (argc<2) {
        /* nessun argomento: mostra l'aiuto e esci */
        show_usage(); return 0;
    }
    if (sscanf(argv[1],"%d",&x) != 1) {
        /* il primo argomento non e' un numero:
           mostra l'aiuto e esci */
        show_usage(); return 0;
    }
    printf("il numero e' %d",x);

    ...
    return 1;
}
```

LA PRIMITIVA **execve**

```
int execve(const char *pathname,  
            char *const argv[],  
            char *const envp[]);
```

- esegue il file di pathname specificato, passandogli gli argomenti argv, e l'environment envp

LA FAMIGLIA **exec**

exec è in realtà una famiglia di sei primitive, con lievi differenze nel significato dei parametri:

→ i parametri li passo come elenco
int **exec1** (const char *pathname, const char *arg0, ...);

int **execv** (const char *pathname, char *const argv[]);

int **execle** (const char *pathname, const char *arg0, ... ,
char *const envp[]);

int **execve** (const char *pathname, char *const argv[],
char *const envp[]);

int **execlp** (const char *filename, const char *arg0, ...);

int **execvp** (const char *filename, char *const argv[]);

LA FAMIGLIA `exec`: SIGNIFICATI MNEMONICI

...l (list):

```
const char *arg0, ... char *argn, (char *) 0
```

...v (vector):

```
char *const argv[]
```

...e (environment):

```
char *const envp[]
```

...p (path¹):

```
const char *filename
```

¹ Fa riferimento alla variabile di shell \$PATH

LA PRIMITIVA **times**

```
clock_t times(struct tms *buf);
```

- restituisce in buf alcuni contatori relativi al tempo di CPU usato dal processo e dai suoi figli

```
struct tms {  
    clock_t tms_utime; /*user CPU time*/  
    clock_t tms_stime; /*system CPU time*/  
    clock_t tms_cutime; /*user CPU time of terminated  
                           children*/  
    clock_t tms_cstime; /*system CPU time of terminated  
                           children*/  
};
```

- restituisce il "wall clock time" rispetto a un'origine arbitraria

Note:

Tutti i valori sono in ticks del clock (si usano in genere per differenza di due valori)

4. ESEMPIO: STRUTTURA DELLA SHELL

Shell è un programma che aspetta cosa vogliamo dal SO: posso chiedere o di eseguire un nuovo processo (e la shell fa diventare ese un processo), oppure gli chiedo dei comandi.

Shell non fa altro che fork e exec

Scheletro di un programma di shell

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

main () {
    pid_t pid, procid;
    int status, k;
    size_t n;
    char buffer[80], prompt[30];
    sprintf(prompt, "myprompt:>");
    write(1, prompt, 10);
    bzero(buffer, 80);
    while (num. caratteri letti!=0) { sempre pronto ad accettare comandi
        /* command line processing */
        k=strlen(buffer);
        buffer[k-1]=buffer[k]; /n da eliminare
        if (buffer=="esci") exit(0);
        printf("attivazione processo figlio %s \n", buffer);
        if ((pid=fork()) == 0) { attivazione processo con exec figlio. Exec passa come primo parametro la stringa
            if (attivazione processo non avvenuta con successo) exit(1);
        }
        procid=wait(&status); /n genera mai finita la vita shell
        if (status!=0) write(2, "cmd not found\n", 14);
        bzero(buffer, 80);
        write(1, prompt, 10);
    }
    exit(0);
}
```

La shell non fa fare work

NOTA: a ogni uscita sarà associata una shell, che ha user_id e group_id. In questo modo i processi avranno user_id e group_id dell'utente.

REDIREZIONE DELLO STDOUT: ESEMPIO

Nome del file eseguito > da file: tutto quello che programma eseguito scrive a video deve scrivere su un file

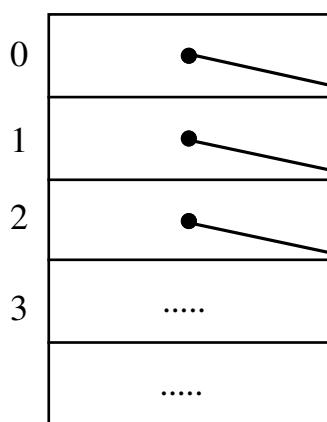
\$cmd > f1

```
if ((pid=fork()) == 0) {  
    /* processo figlio */(re-exec)  
    if (output redirection) {  
        fd=creat("f1", ...);  
        close(1);  
        dup(fd);  
        close(fd);  
    }  
    exec( .... );  
}  
...  
...
```

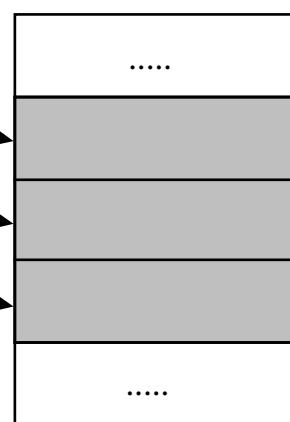
↓
Così si copia la scrittura
della dup: se voglio redirettare
creo un nuovo file usando
stessa passata. Lo ho già aperto
in modalità scrittura.

Per chiudo il fd=1, cioè lo STDOUT
Per effettua una dup di fd. Assegno
al file fd un nuovo file descriptor
prendendo primo numero libero (1). Quando
quel file sarà associato al vecchio fd
e viene chiuso il vecchio file descriptor

File descriptor
table



File table



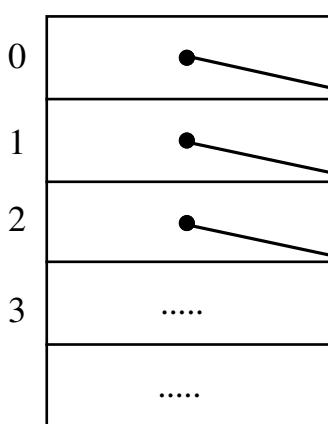
REDIREZIONE DELLO STDIN: ESEMPIO

Penso fare al contrario,
prendendo input dell'eseg. da file

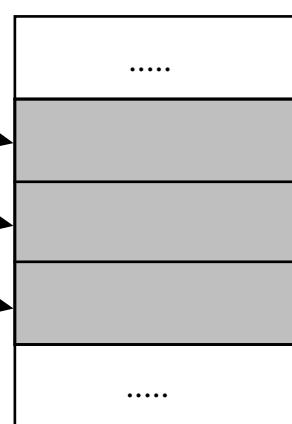
\$cmd < f1

```
if ((pid=fork()) == 0) {  
    /* processo figlio */  
    if (input redirection) {  
        fd=open("f1", ...);  
        close(0);  
        dup(fd);  
        close(fd);  
    }  
    exec( .... );  
}  
....
```

File descriptor
table



File table



POSSIAMO ANCHE CHIEDERE ALLA SHELL DI METTERE
IN BACKGROUND UN PROCESSO → IN PRACTICA LA SHELL
NON FA LA "WAIT()" ED È SUBITO DISPONIBILE NUOVAMENTE.

LA SHELL VIENE GENERATA DA UN ALTRO PROCESSO → LA FINESTRA DI
"LOGIN", CHE GENERA UNA SHELL PER QUEL' UTENTE CHE ESEGUE
L'ACCESSO.