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

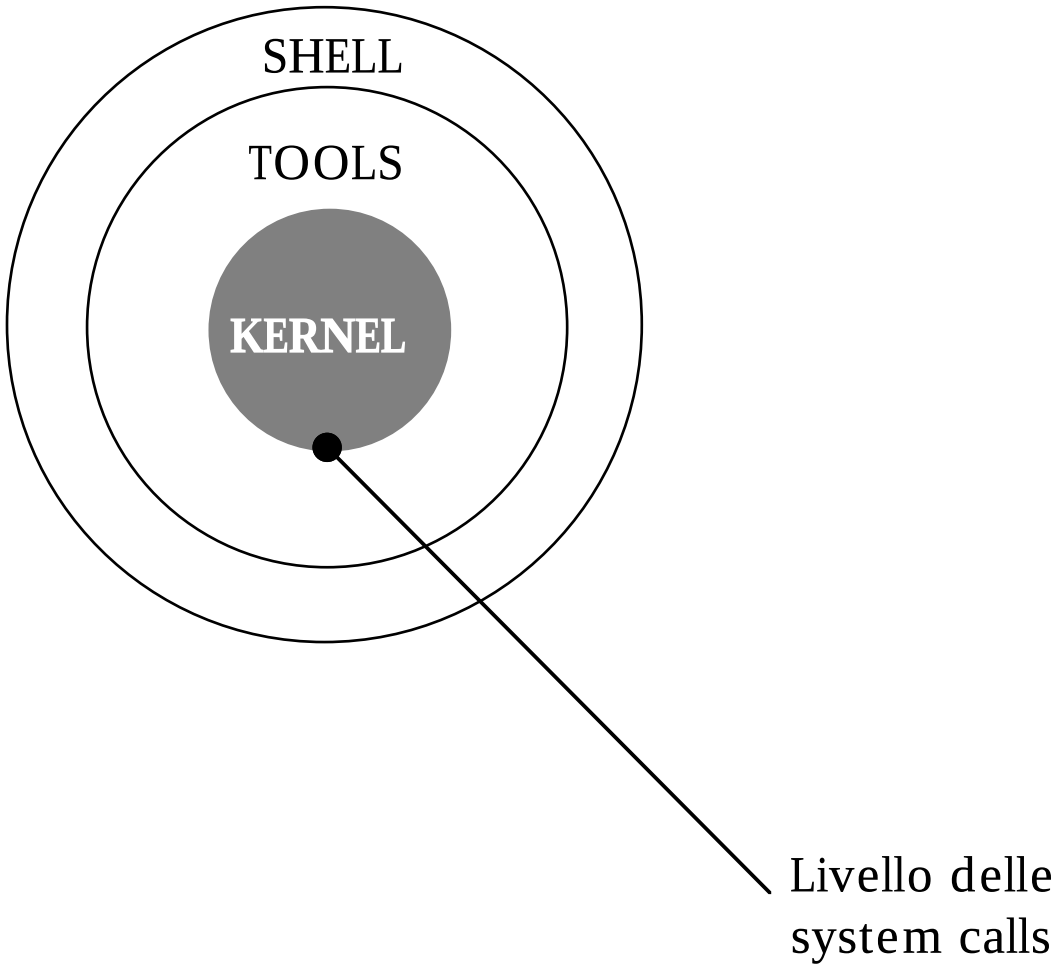
Ultimo aggiornamento: maggio 1997

TEMP. 071 - 01

INDICE

1. INTRODUZIONE

LIVELLO DI VISIBILITÀ



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").

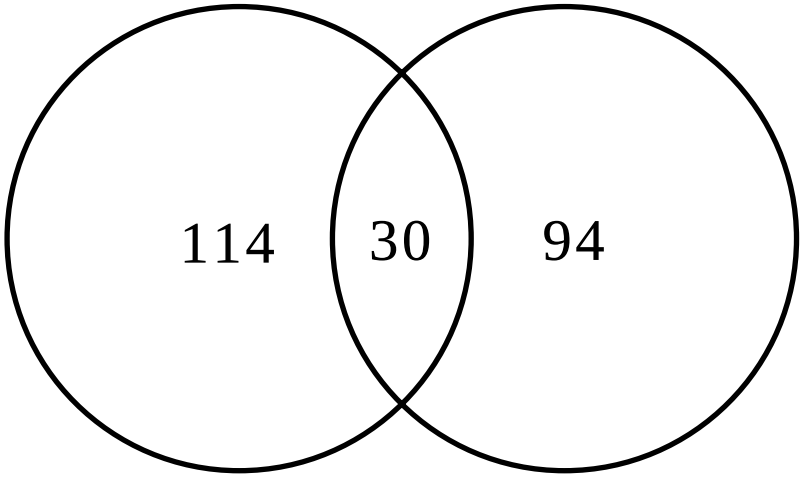
  Ad es., non definisce alcuna funzione di system administration

- 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:

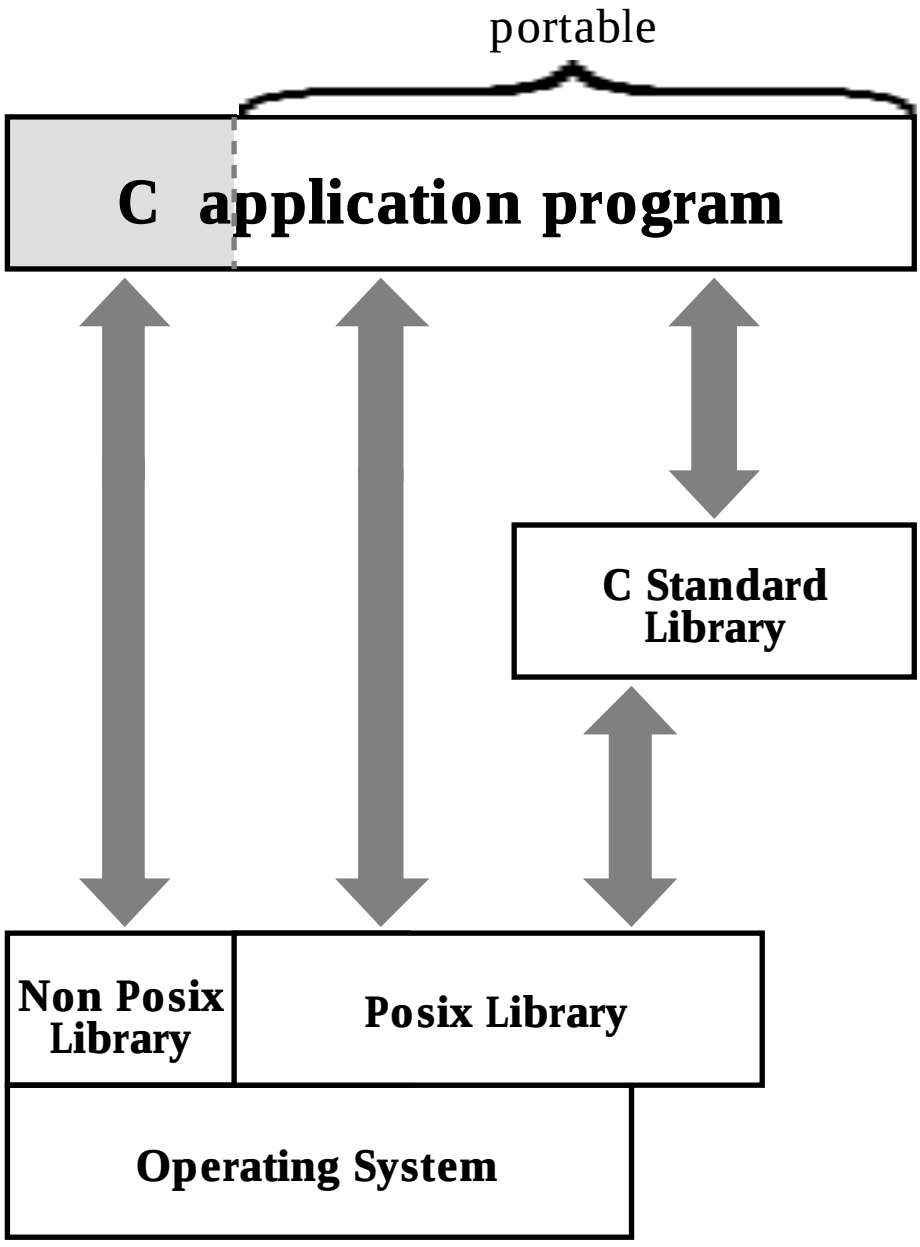
`open()` è POSIX

`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 qualunque 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)

HEADER FILES POSIX

Per scrivere un programma conforme a POSIX, bisogna includere gli header files richiesti dalle varie primitive usate, scelti fra:

- header files della C Standard Library (contenenti opportune modifiche POSIX)

- header files specifici POSIX (una diecina)

Inoltre bisogna inserire, prima di ogni `#include`:

```
#define _POSIX_SOURCE 1
```

GESTIONE ERRORI

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 )
```

"**p**rint **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
```

RIFERIMENTI

I seguenti libri trattano in dettaglio sia le system call che le primitive della C Standard Library:

W.R.Stevens
Advanced Programming in the Unix Environment
Addison-Wesley, 1992

W.R.Stevens
Unix Network Programming
Prentice Hall, 1990

D.Lewine
POSIX Programmer's Guide
O'Reilly & Associates, 1991

2. PRIMITIVE DI GESTIONE FILES E DIRECTORIES

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`:

- 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

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

La `close` rende disponibile il file descriptor per ulteriori usi

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

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

- 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`	`O_RDONLY`	read-only
	`O_WRONLY`	write-only
	`O_RDWR`	read and write
	`O_APPEND`	append
	`O_CREAT`	creazione del file
	...	
`...`	permessi iniziali (se `O_CREAT`)	

CREAZIONE DI UN FILE NORMALE: `creat`

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

- crea un nuovo file normale di specificato pathname, e lo apre in scrittura
- `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`

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

- 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);
```

- 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 nè chiusi

ESEMPIO: COME SCEGLIERE BUFFSIZE

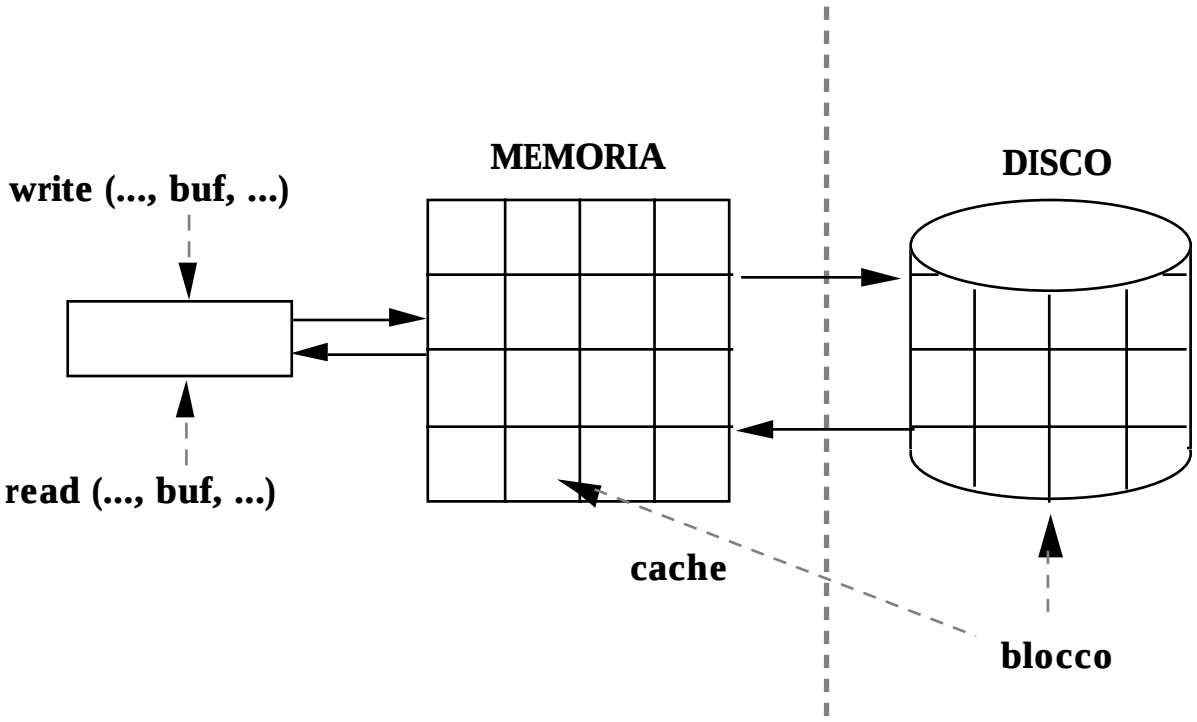
Prestazioni dell'esempio precedente, per lettura di 1.468.802 bytes, al variare di BUFFSIZE, su un file system con blocchi di 8192 bytes (Berkeley fast file system)

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	# loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	**0.0**	**0.3**	**0.3**	**180**
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

Da: W.R.Stevens, Advanced Programming in the Unix Environment, pag 57

CACHING

Il kernel utilizza un meccanismo di caching per ridurre il numero di operazioni di I/O:



POSIZIONAMENTO SU UN FILE: `lseek`

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

- 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`, o `-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);
```

GESTIONE DI DIRECTORIES (II)

Per leggere una directory in modo portabile, la directory entry è definita in `dirent.h` come:

```
struct dirent {
 ino_t d_ino;/*i-number*/
 char d_name[NAME_MAX+1]/*filename*/
};
```

e sono disponibili le seguenti primitive:

```
DIR *opendir(const char *dirname);

struct dirent *readdir(DIR *dirp);

int closedir(DIR *dirp);

void rewinddir(DIR *dirp);
```

Nota: Solo il kernel può scrivere su una directory

STATO DI UN FILE: `stat e fstat`

```
int stat(const char *path,
    struct stat *buf);

int fstat(int fildes,
    struct stat *buf);
```

- carica in `*buf` alcune informazioni di stato relative al file `path` (o `fildes`) e restituisce l'esito (`0` o `-1`)
- i membri della `struct stat` sono definiti in `sys/stat.h`:

`st_dev`	numero del device
`st_ino`	i-number
`st_mode`	permissions
`st_nlink`	numeo di (hard) links
`st_uid`	user id
`st_gid`	group id
`st_size`	dimensione del file
`st_atime`	last access time
`st_mtime`	last modification time
`st_ctime`	last status change time

PERMESSI DI UN FILE: chmod

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

"**ch**ange **mod**e"

- cambia i permessi del file `path`, come specificato in `mode`
- 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:

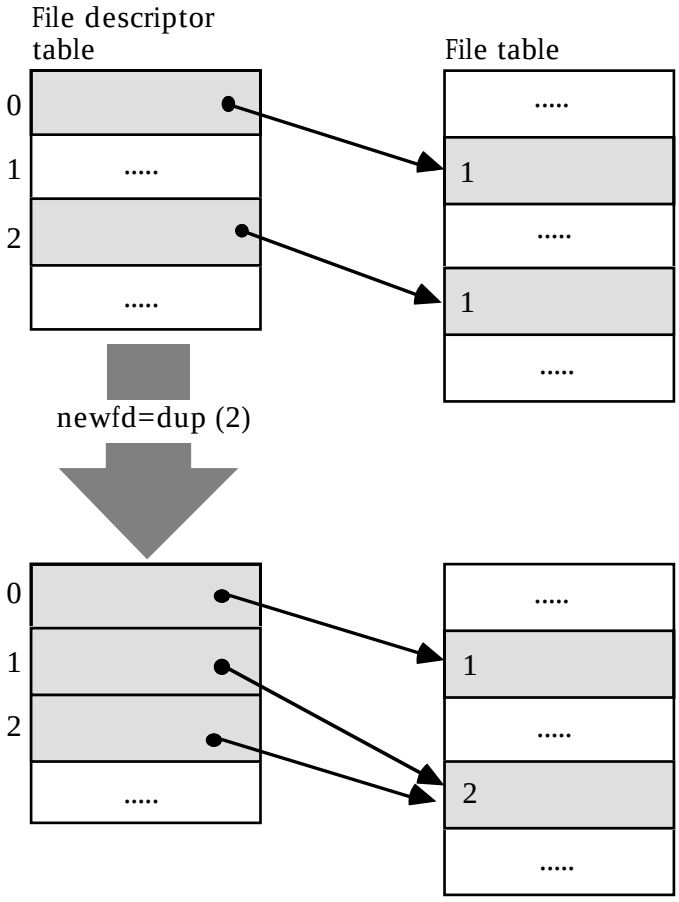
```
int utime(const char *path,
          const struct utimbuf *times);
```

DUPLICAZIONE DI FILE DESCRIPTORS: dup

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

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

Esempio:



FUNZIONI DI AMMINISTRAZIONE DEL FILE SYSTEM

POSIX non standardizza le funzioni di amministrazione del sistema; ogni sistema tuttavia mette a disposizione opportune system call

Esempi:

mknod(pathname, type, device);

- crea un nuovo file di tipo specificato (regolare, directory, device o pipe con nome)

mount(device, directory, mode)

- monta il file system contenuto nel device sulla directory del root file system, con i permessi specificati

umount(device)

- smonta il file system contenuto nel device (precedentemente montato)

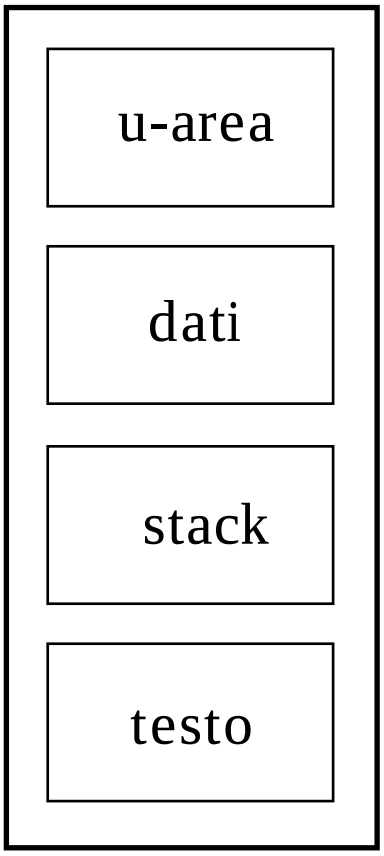
SINTESI PRINCIPALI PRIMITIVE DI GESTIONE FILE

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)` `r=fstat(fildes, &buf)`	fornisce informazioni sullo stato del file
`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**:



dati relativi al processo, di pertinenza del sistema operativo (es. tabella dei file aperti, working directory, ...)

dati globali

il codice del processo

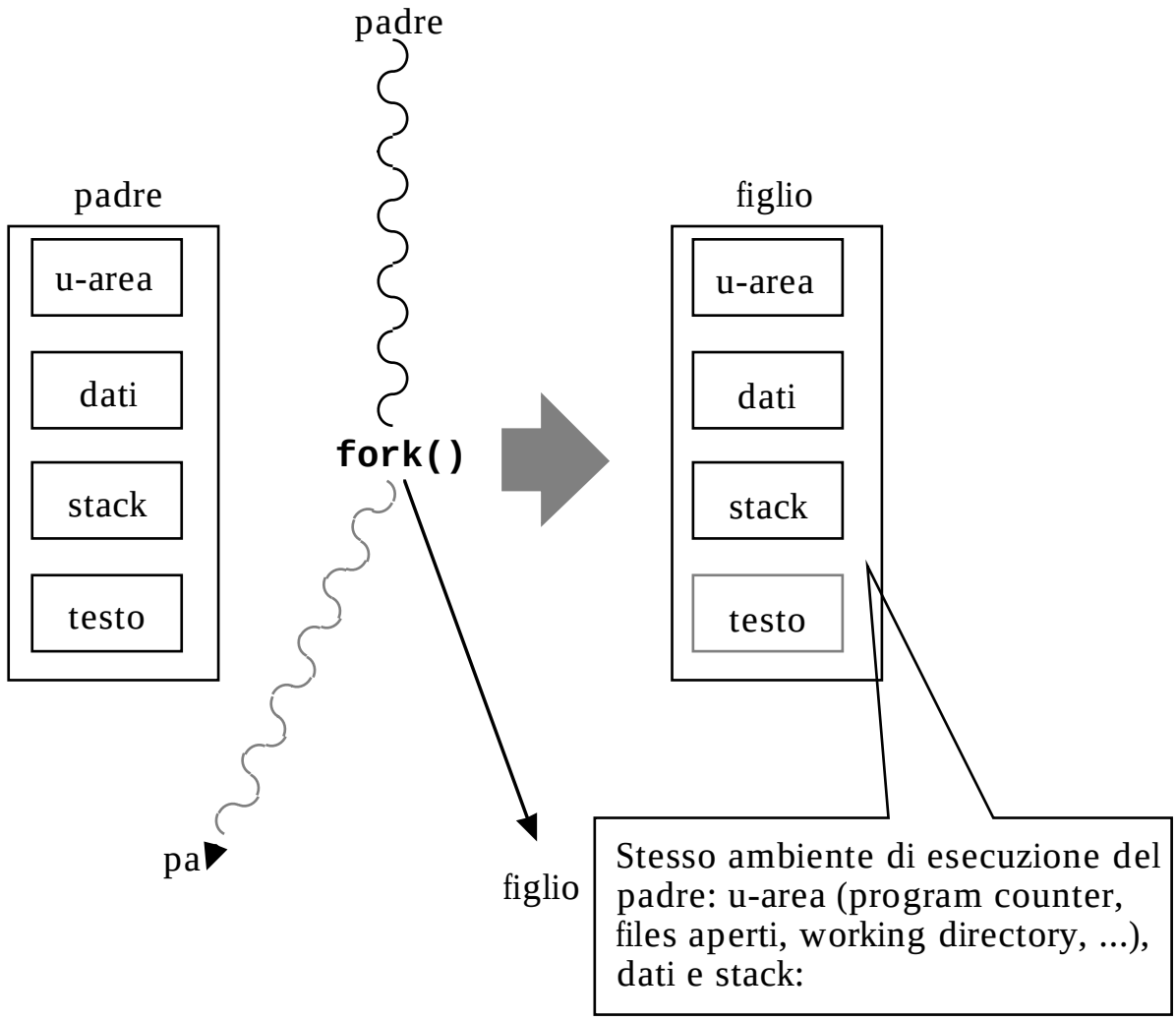
Nota:

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:**



LA PRIMITIVA `fork`

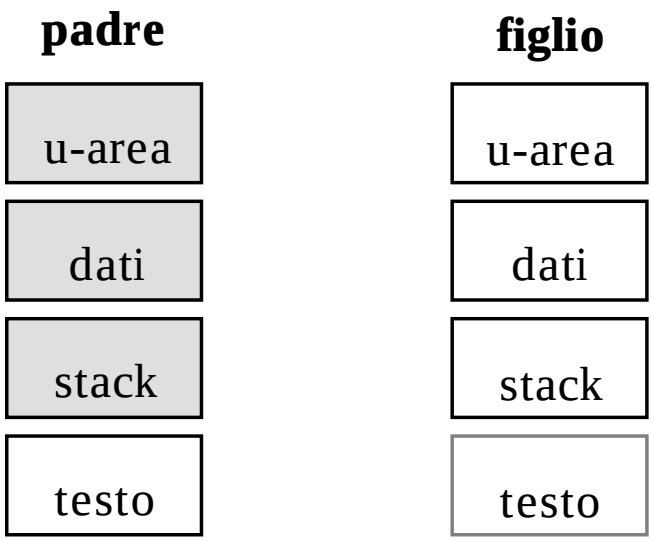
```
pid_t fork(void);
```

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

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

ESEMPIO

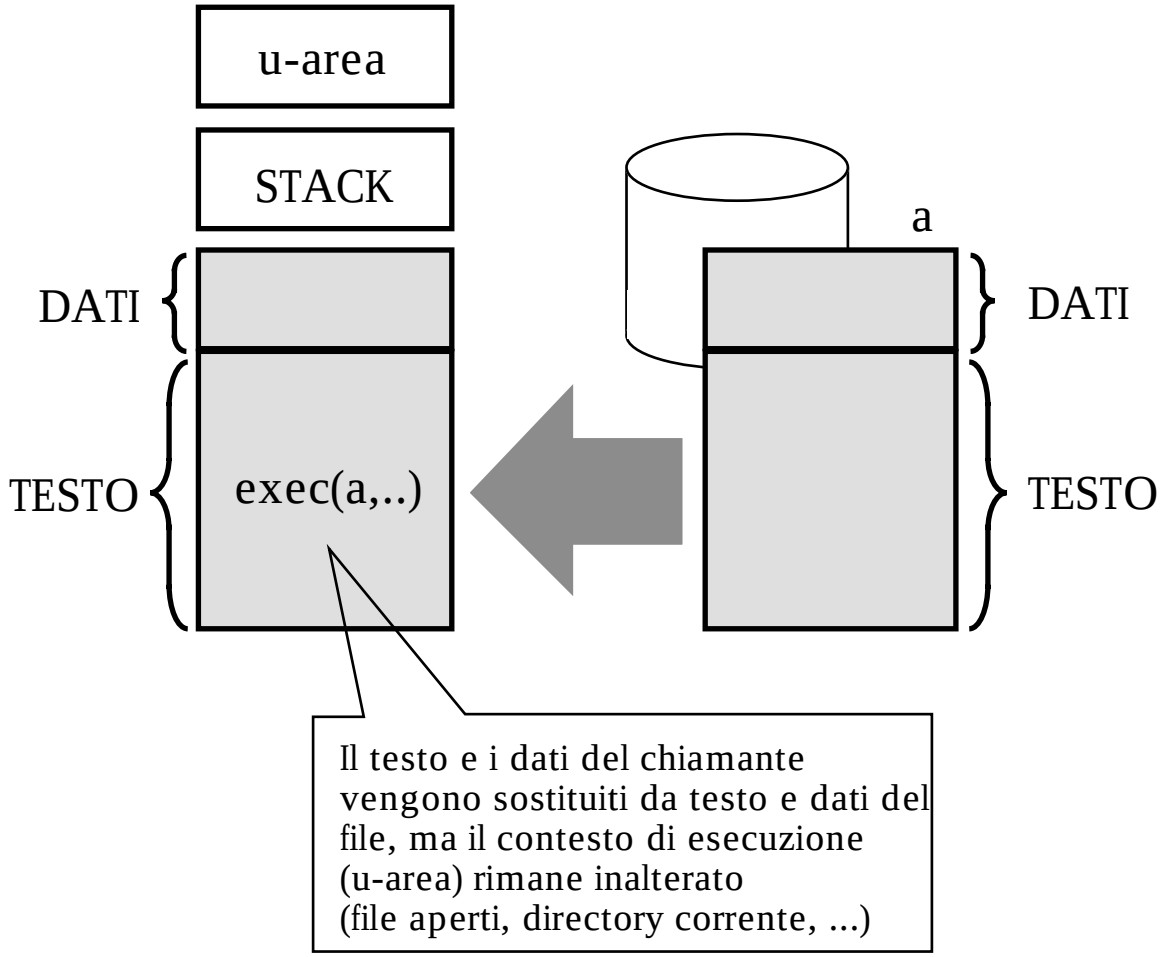
```
pid = fork();
if (pid<0)          {/*la fork è fallita */ }
else if(pid>0)      {/*operazioni del padre */}
else                {/*operazioni del figlio*/}
```



ESECUZIONE DI UN PROGRAMMA

exec (pathname, argomenti)

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



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'nvironment `envp`

LA FAMIGLIA exec

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

```
int execl (  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` (path[1]):

```
const char *filename
```

[1] Fa riferimento alla variabile di shell $PATH

LE PRIMITIVE `wait` E `waitpid`

```
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`**

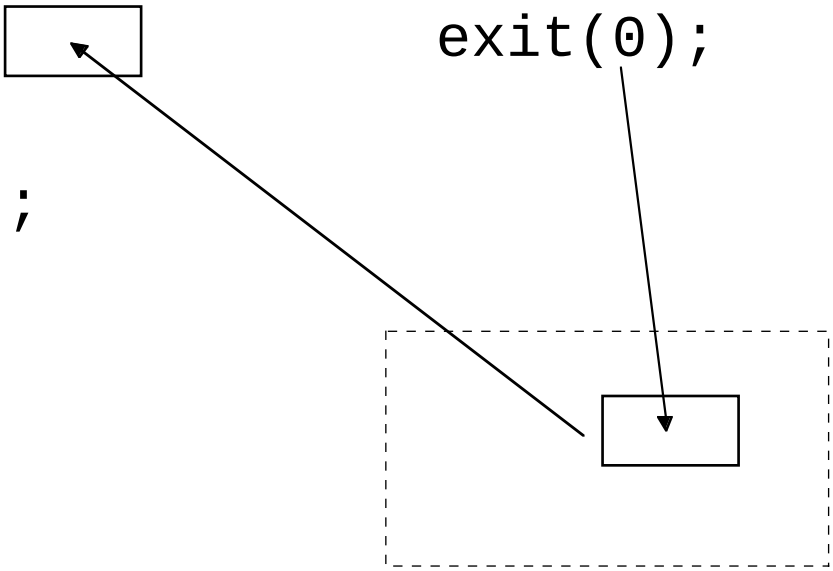
wait E exit

padre

```
...
int st;
...
wait(&st);
...
```

figlio

```
...
exit(0);
```



area del kernel

Nota:

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

PROCESSI ORFANI

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

GRUPPI DI PROCESSI

Ogni processo è membro di un *gruppo*, identificato dal process-id di un processo, detto **leader del gruppo**.

Tale process-id è detto **process group ID**

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 dfferenza di due valori)

4. ESEMPIO:
STRUTTURA DELLA SHELL

STRUTTURA DELLA SHELL

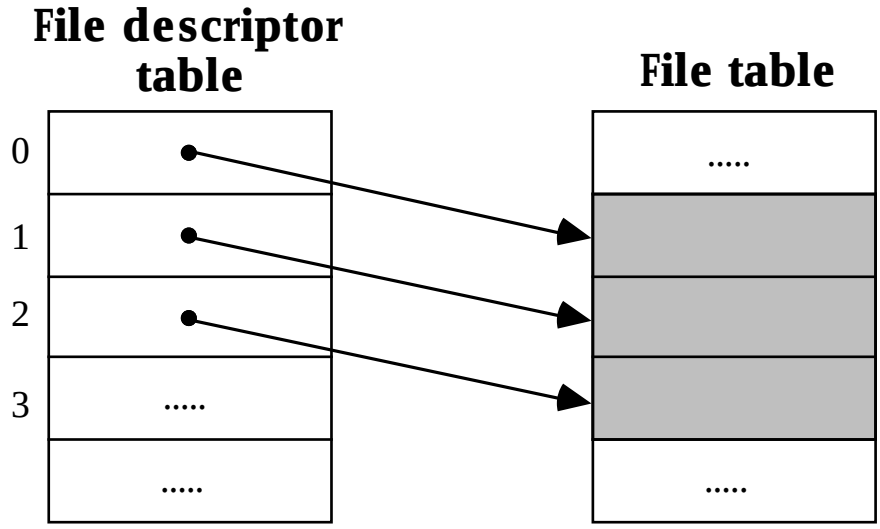
```
write(1, PROMPT);
while ((n=read(0,buffer,80)=!0) {
    /* command line processing */
    if ((pid=fork()) == 0)  {
          /* I/O redirection */
          if (exec(file,args)==-1) exit(1);
    }
    procid=wait(status);
    if (status!=0) write(2,'cmd not found');
    write(1, PROMPT);
}
exit(0);
```

Se il comando è eseguito in background, non viene eseguita la `wait`.

REDIREZIONE DELLO STDOUT: ESEMPIO

$cmd > f1

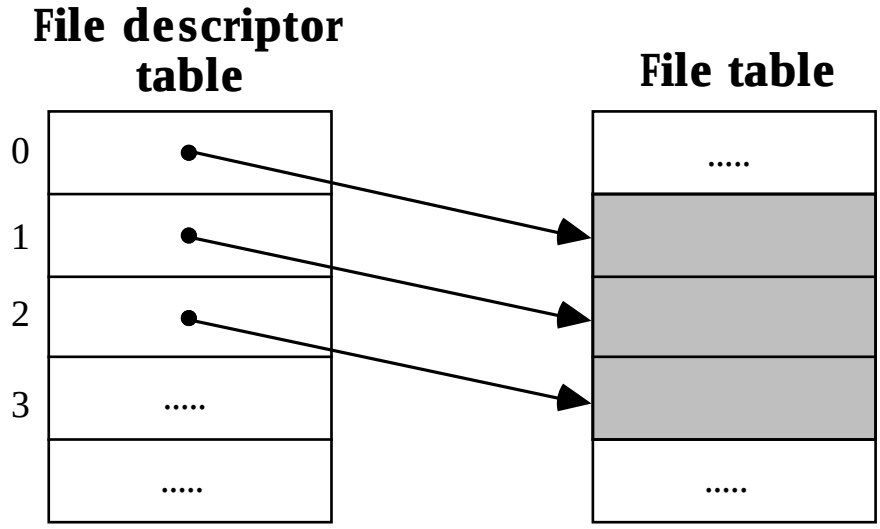
```
if ((pid=fork()) == 0)  {
    /* processo figlio */
    if (output redirection )  {
        fd=creat("f1", ...);
        close(1);
        dup(fd);
        close(fd);
    }
    exec( .... );
}
...
```



REDIREZIONE DELLO STDIN: ESEMPIO

$cmd < f1

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



5. PRIMITIVE DI COMMUNICAZIONE FRA PROCESSI

CLASSI DI PRIMITIVE

Le varie versioni di UNIX mettono a disposizione diversi meccanismi di communicazione fra processi:

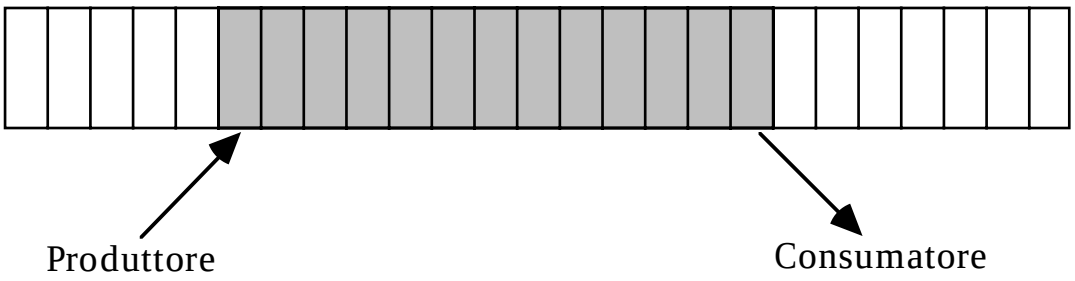
- **pipes**
- **segnali**
- [code di messaggi]
- [semafori]
- [memoria condivisa]
- **sockets**

5.1 Pipes

PIPE

Una pipe è un file di dimensione limitata gestito come una coda FIFO:

- un processo produttore deposita dati (e resta in attesa se la pipe è piena)
- un processo consumatore legge dati (e resta in attesa se la coda è vuota)



Esistono due tipi di pipes:

- **pipe con nome**, create da `mknod` (devono essere aperte con `open`)
- **pipe senza nome**, create e aperte da `pipe` su un "pipe device" definito al momento della configurazione

CREAZIONE DI PIPE SENZA NOME: `pipe`

```
int pipe(int fildes[2]);
```

- crea una “pipe”, la apre in lettura e scrittura
- restituisce l'esito dell'operazione (`0` o `-1`)
- assegna a `fildes[0]` il file descriptor del lato aperto in lettura, e a `fildes[1]` quello del lato aperto in scrittura

PIPE E FILE ORDINARI

Creazione e uso di un file ordinario:

```
int fd;
...
if ((fd=creat(filename,...)<0)err();
...
write(fd, ...);
...
```

Creazione e uso di una pipe (senza nome):

```
int fd[2];
...
if (pipe(fd) < 0) err();
...
write(fd[1], ...);
...
read(fd[0], ...);
...
```

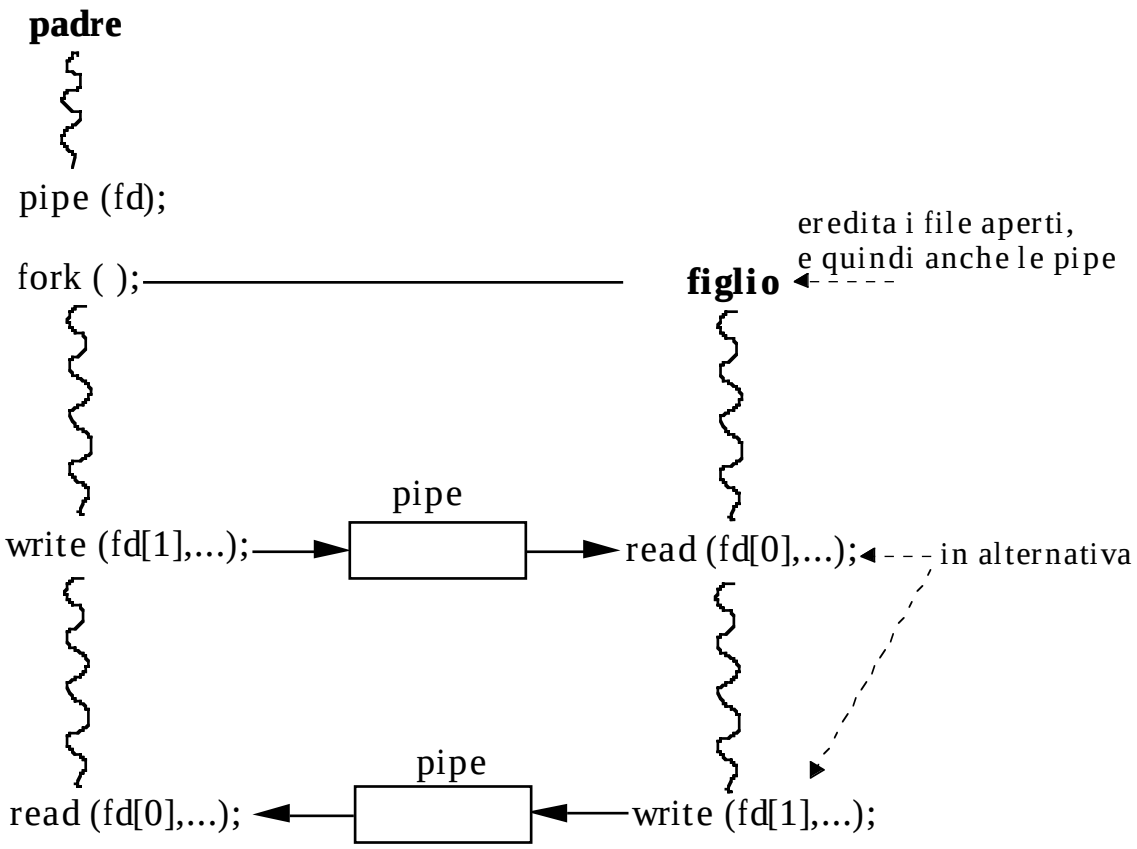
Note:

- la `pipe` è analoga a `creat`, ma non specifica il nome e restituisce due file descriptor
- `read` e `write` sono le stesse nei due casi

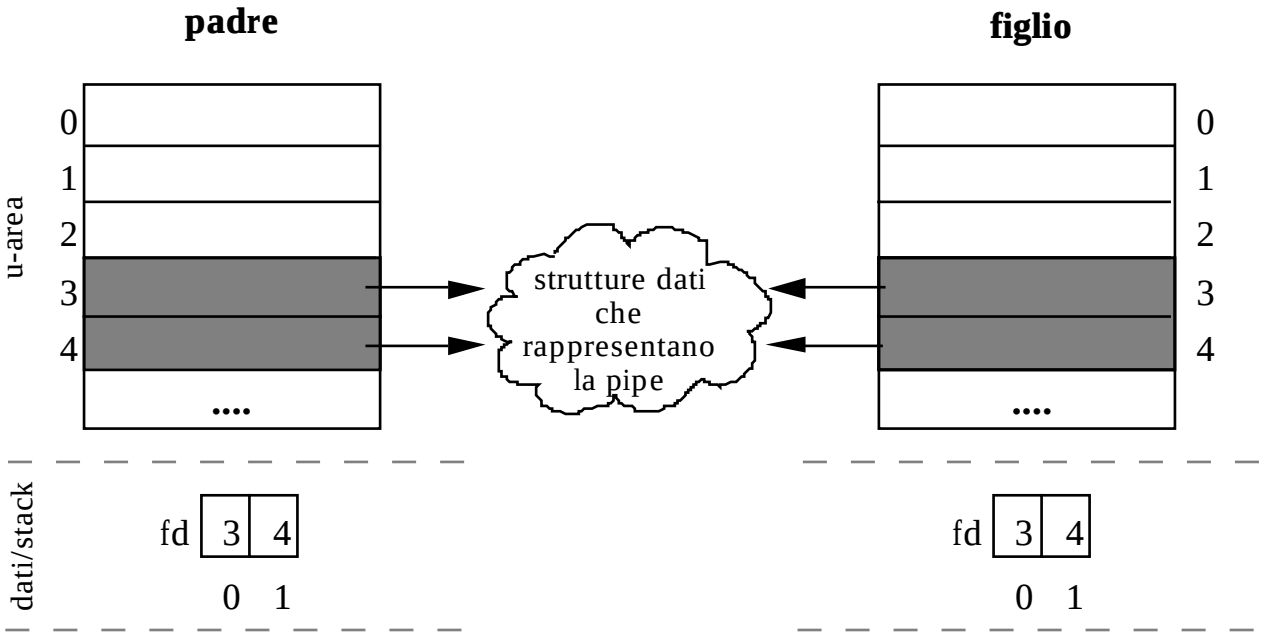
PIPE: USO TIPICO

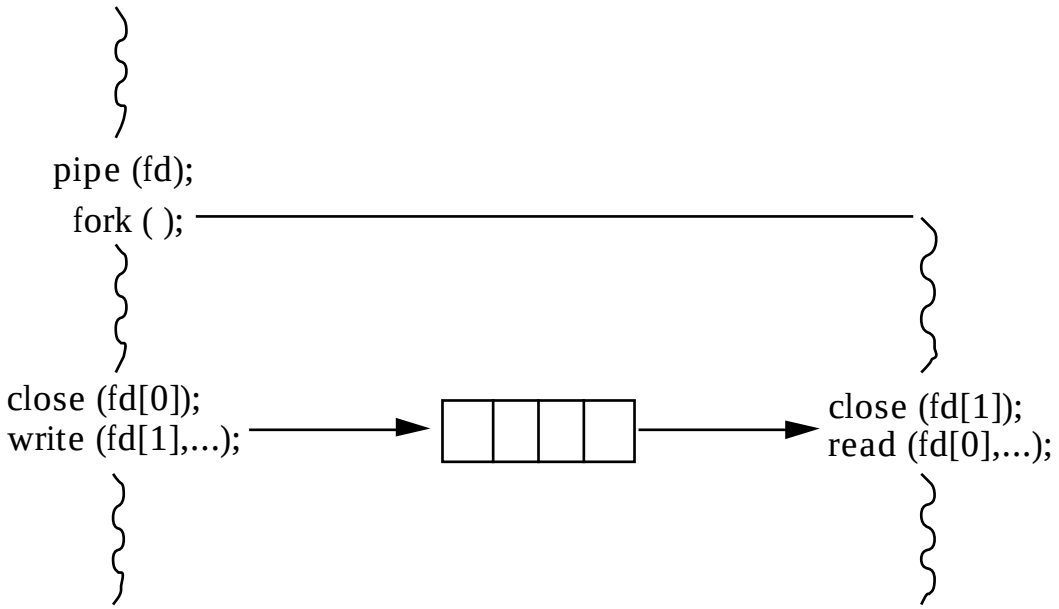
Generalmente una pipe viene usata per far comunicare un processo padre con un suo figlio

Lo schema tipico:



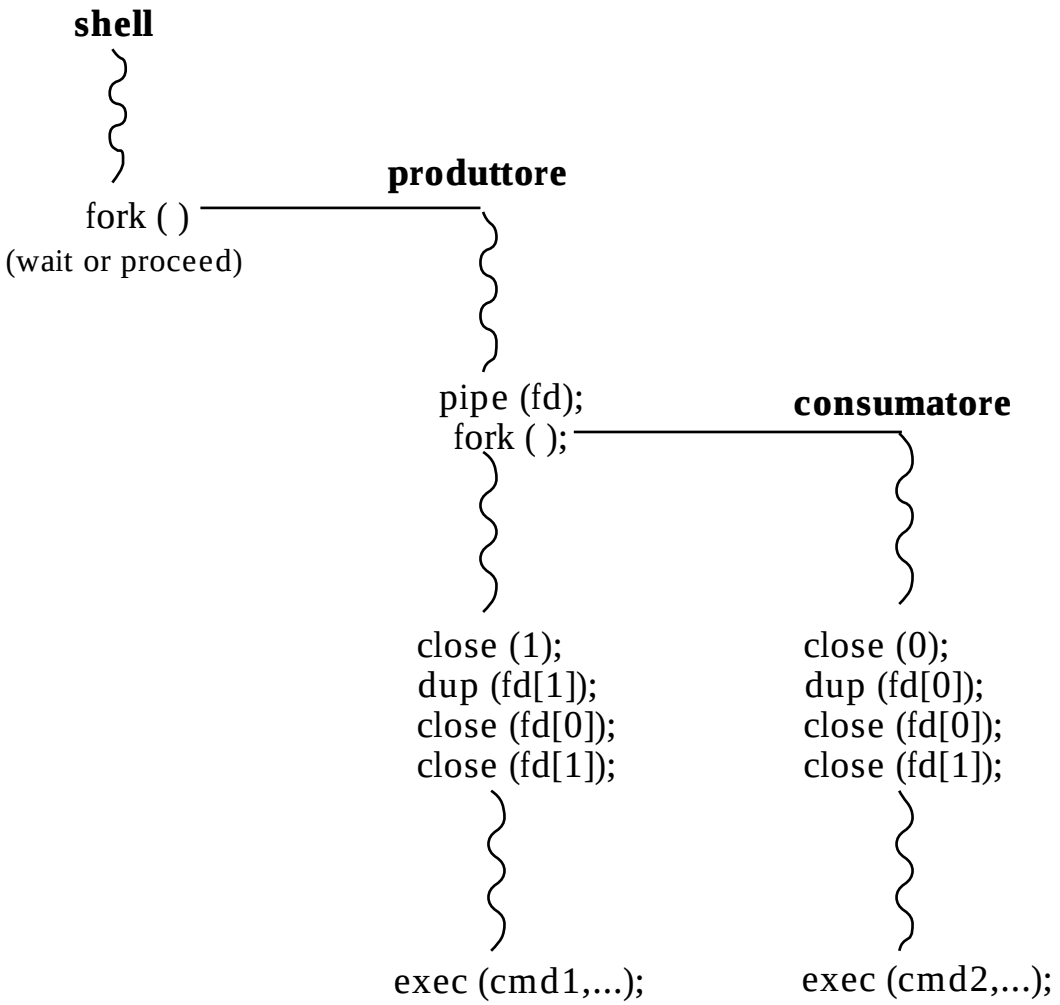
PIPE: USO TIPICO (SEGUE)





REALIZZAZIONE DI UNA PIPELINE

$cmd1|cmd2



REALIZZAZIONE DI UNA PIPELINE (SEGUE)

$cmd1|cmd2

```
if ((pid=fork()) == 0)  {
    /* processo figlio */
    if (pipeline)  {
        pipe (fd_pipe);
        if (fork()==0) { /* produttore */
            close(1);
            dup(fd_pipe[1]);
            close(fd_pipe[0]);
            close(fd_pipe[1]);
            exec (cmd1, ...);
        }                   /* consumatore */
        close (0);
        dup (fd_pipe[0]);
        close (fd_pipe[0]);
        close(fd_pipe[1]);
        exec (cmd2, ...);
    }
...
}
```

5.2 Segnali

SEGNALI

Un **segnale** è una notifica a un processo che è occorso un particolare "evento":

- un errore di floating point
- la notifica della morte di un figlio
- una richiesta di terminazione
- ...

I segnali possono essere pensati come degli "interrupts software"

I segnali possono essere inviati:

- da un processo a un altro processo
- da un processo a se stesso
- dal kernel a un processo

Ogni segnale é identificato da un numero intero (associato a un nome simbolico in `<signal.h>`)

SEGNALI IN POSIX

Segnale	Significato	Default
SIGABRT	Abortisce un processo	termina con dump
SIGALRM	Invia un segnale di "sveglia"	termina
SIGCHLD	Lo stato di un figlio è cambiato	ignora
SIGCONT	Continua un processo stoppato	continua o ignora
SIGFPE	Floating Point Error (per esempio divisione per zero)	termina con dump
SIGHUP	Hangup su un terminale	termina
SIGILL	Istruzione di macchina illegale	termina con dump
SIGINT	L'utente ha usato il tasto DEL del terminale per interrompere il processo	termina
SIGKILL	Segnale per terminare un processo (non può essere ignorato)	termina
SIGPIPE	Tentativo di scrivere su una pipe che non ha lettori	termina
SIGQUIT	L'utente ha usato il tasto di quit del terminale	termina con dump
SIGSEGV	Riferimento a un indirizzo di memoria non valido	termina con dump
SIGSTOP	Segnale per stoppare un processo (non può essere ignorato)	stoppa il processo
SIGTERM	Segnale per terminare un processo	termina
SIGTSTP	L'utente ha usato il tasto "suspend" del terminale	stoppa il processo
SIGTTIN	Un processo in background tenta di leggere dal suo terminale di controllo	stoppa il processo
SIGTTOU	Un processo in background tenta di scrivere sul suo terminale di controllo	stoppa il processo
SIGUSR1	Disponibile per scopi definiti dall'applicazione	
SIGUSR2	Disponibile per scopi definiti dall'applicazione	

ESEMPIO

Per terminare o sospendere un processo in fore-ground, l'utente può premere i tasti `CTRL-C` o `CTRL-Z` (rispettivamente)

Tale carattere viene acquisito dal driver del terminale, che notifica al processo il segnale `SIGINT` o `SIGTSTP` (rispettivamente)

Per default, `SIGINT` termina il processo e `SIGTSTP` lo sospende

Nota:

In realtà, tali segnali vengono inviati a tutto il gruppo di processi

GESTIONE DEI SEGNALI

- per inviare un segnale:

  `kill`

- per specificare come trattare un segnale:

  `signal`

Nota:

La primitiva `signal` non é POSIX: la corrispondente primitiva POSIX si chiama `sigaction`

LA PRIMITIVA `kill`

```
int kill(pid_t pid, int sig);
```

- notifica il segnale `sig` al processo/gruppo di processi specificato con `pid` :

  `pid>0`: `pid` indica il process-id

  `pid<0`: `|pid|` indica il process-group-id

- restituisce il risultato dell'operazione (`0` se è stato inviato almeno un segnale, `-1` se errore)

PROTEZIONE

Per motivi di protezione, deve valere almeno una delle seguenti condizioni:

- Il processo che riceve e il processo che invia il segnale devono avere lo stesso owner[2]
- L'owner[3] del processo che invia il segnale éil superuser

[2] Ricevente e inviante devono avere gli stessi Effective-user-id e real-user-id

[3] Effective-user-id

LA PRIMITIVA `signal`

Permette di specificare come dovrà essere trattata, dal processo ricevente, la "prossima" occorrenza di uno specificato segnale

Tre possibilità:

1. il segnale dovrà essere **trattato** da uno specificata funzione ("handler")
2. il segnale dovrà essere **ignorato**
3. il segnale dovrà innescare l'azione di **default** associata al segnale stesso.

Note:

- Le possibili azioni di default sono fisse per ogni segnale e possono essere: terminare il processo (con o senza core dump); ignorare il segnale; sospendere il processo; riattivare il processo
- `SIGKILL` e `SIGSTOP` non possono essere riprogrammati: si attiva sempre l'azione di default

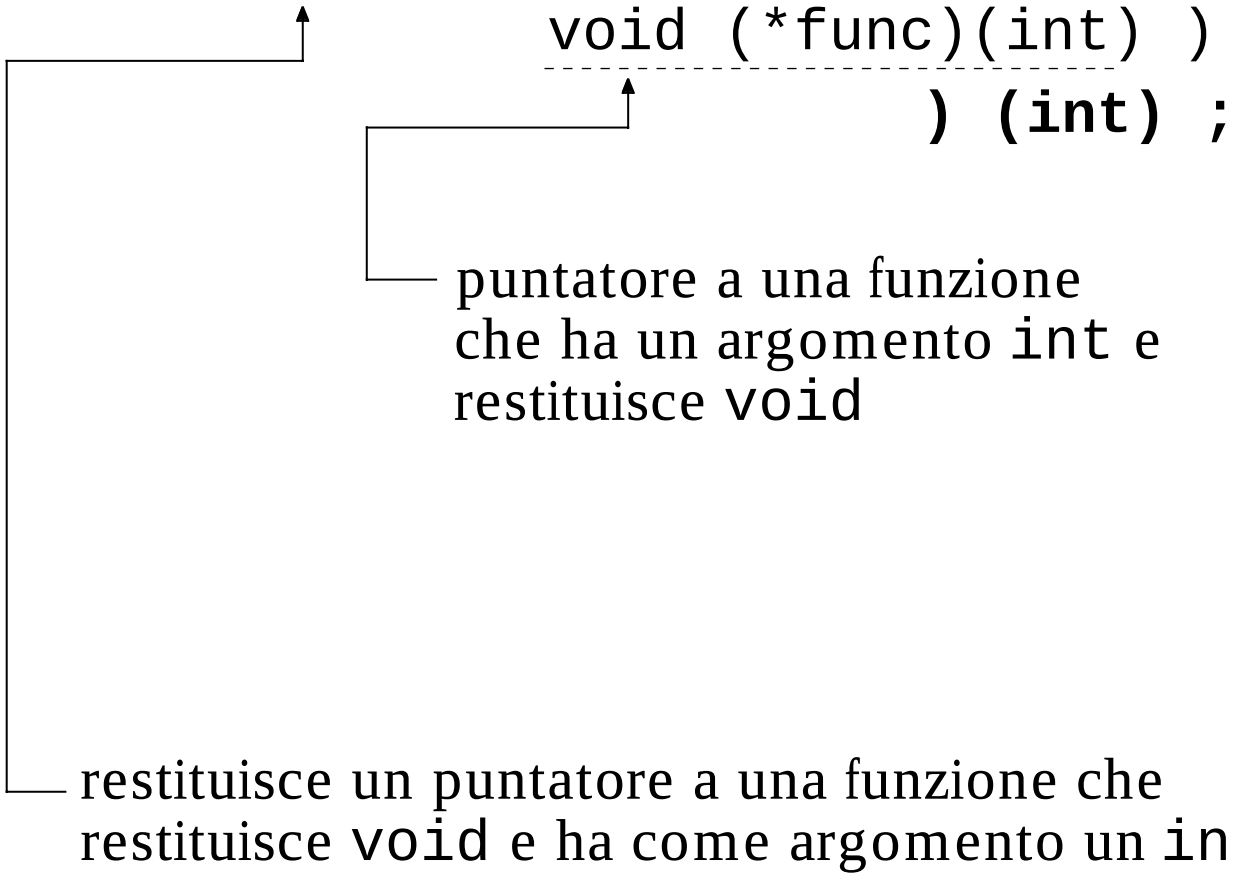
LA PRIMITIVA `signal`: ESEMPIO

```
...
int sig;
void func(int signo)
{ /* corpo dell'handler del segnale.*/ };
signal(sig, func);     /* al ricevimento del
          segnale sig!=SIGKILL, viene
          chiamata func, passandole sig come
          argomento. Al termine di func, il
          controllo ritorna al punto di
          interruzione */
...
signal(sig, SIG_IGN); /* sig dovrà essere
                         ignorato */
...
signal(sig, SIG_DFL); /* all'occorrenza di
                         sig, dovrà essere
                         attivata l'azione di
                         default */
```

N.B. Se ok, `signal` restituisce il puntatore alla funzione precedentemente associata a `sig`

LA PRIMITIVA `signal`: PROTOTIPO

```
void(
      *signal(   int signo,
                 void (*func)(int) )
                              ) (int) ;
```



RESET DI UN SEGNALE

- Dopo aver trattato un segnale, l'azione associata viene "dimenticata".

- Al successivo ricevimento dello stesso segnale verrà eseguita l'azione di default

Esempio:

```
int (*oldHandler);
...
oldHandler=signal(SIGINT, SIG_IGN);
...
/* qui CTRL-C viene ignorato */
...
signal(SIGINT, oldHandler);
...
/* qui CTRL-C viene gestito da
oldHandler, ma solo la prima volta;
poi causa terminazione (default) */
```

LA PRIMITIVA `alarm`

```
unsigned int alarm (unsigned int
                    seconds);
```

- ritorna al chiamante, e dopo `seconds` secondi gli invia il segnale `SIGALRM`

- ci può essere una sola richiesta pendente: la funzione ritorna i secondi mancanti alla terminazione di eventuali `alarm` precedenti

LA PRIMITIVA `sleep`

```
unsigned int sleep (unsigned int
                    seconds);
```

- sospende il chiamante per `seconds` secondi
- restituisce il numero di secondi mancanti a soddisfare la richiesta:

  la sospensione può essere infatti più breve di quanto richiesto, per vari motivi (ad es., un segnale che deve essere trattato)

LA PRIMITIVA pause

```
int pause (void);
```

- sospende il chiamante fino a quando esso riceve un segnale (che non deve essere ignorato)

- restituisce `-1` in caso di errore

5.3. Sockets

Riferimenti:

- W.R.Stevens, Unix Network Programming, Prentice Hall, 1990

SOCKETS: CHE COSA SONO

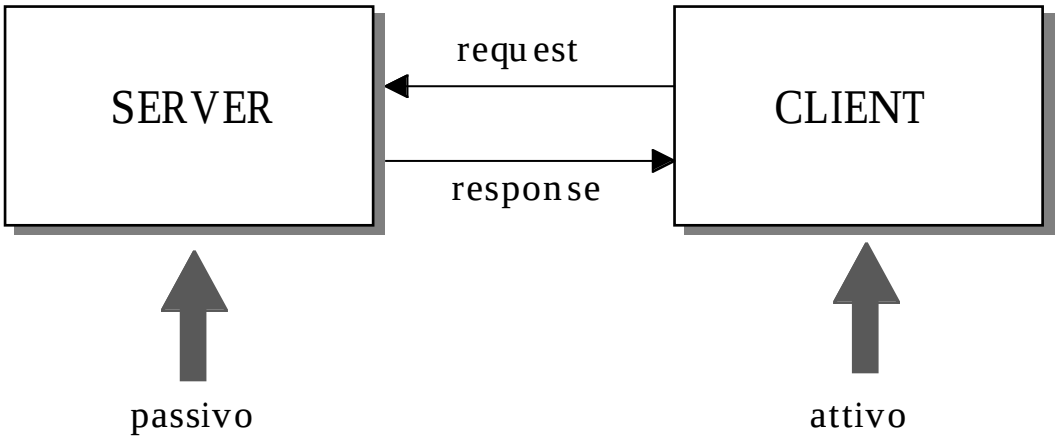
Un insieme di **primitive per la comunicazione fra processi**, residenti sulla stessa macchina o su macchine diverse ...

...realizzate per Unix BSD, ...

...e poi divenute lo standard di fatto per realizzare applicazioni secondo il **modello client-server**

MODELLO CLIENT-SERVER

Modello logico di comunicazione fra processi, residenti sulla stessa macchina o su macchine diverse in rete, di tipo asimmetrico:



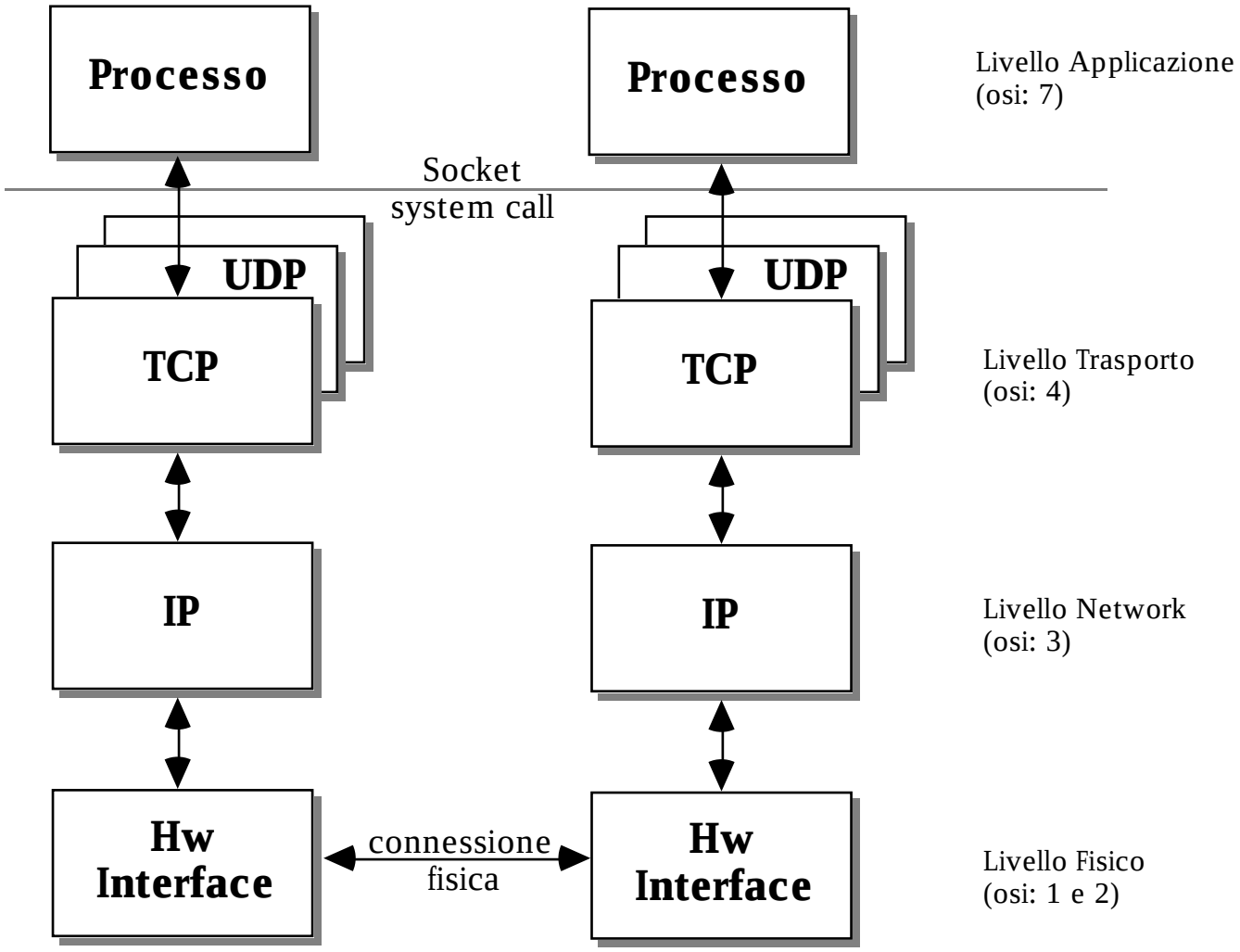
ESEMPI

Una situazione frequente:

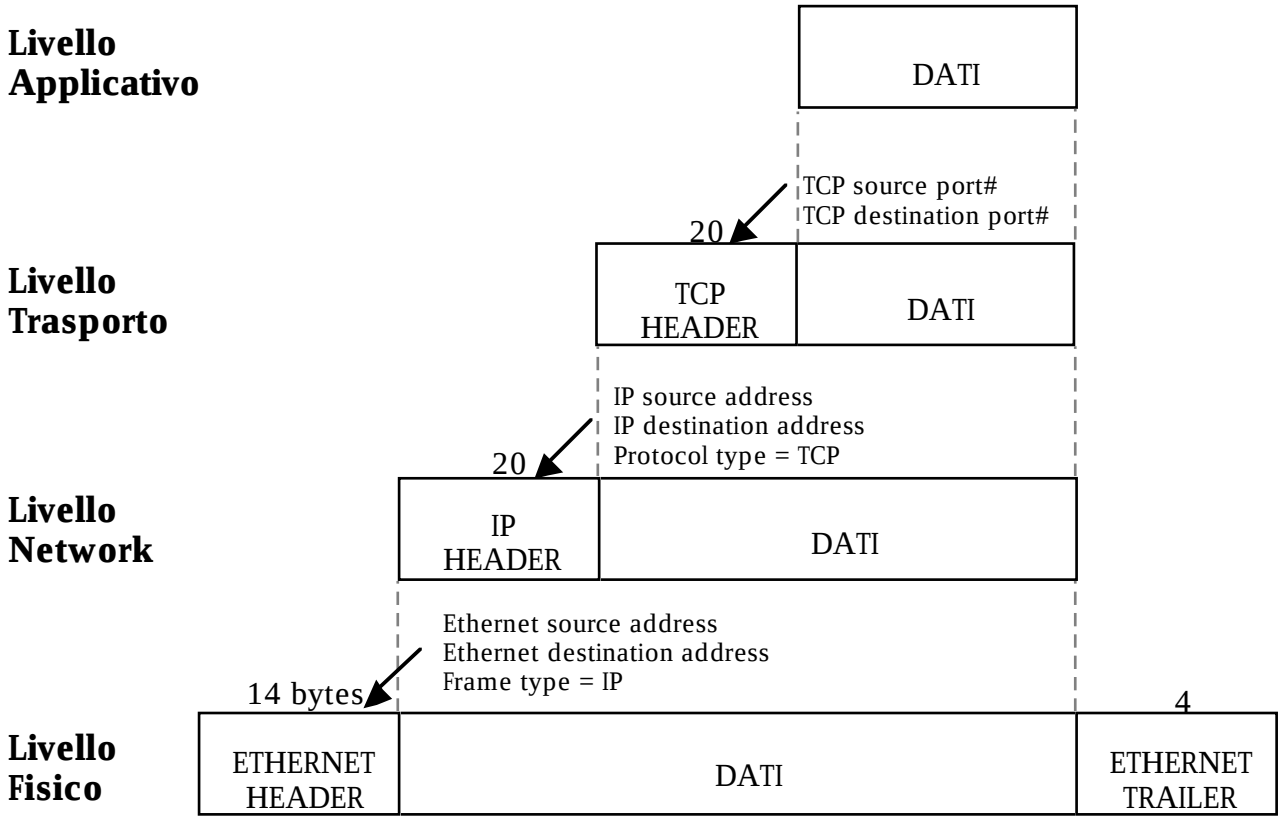
Ma altre situazioni sono possibili!

Richiami sui protocolli

LIVELLI DEL PROTOCOLLO TCP/IP



TCP/IP: INCAPSULAZIONE DEI MESSAGGI



LIVELLI DEL PROTOCOLLO TCP/IP (SEGUE)

Livello network:

IP (Internet Protocol)

- connectionless
  invia "datagrammi" fra loro indipendenti a un host (IP address)
- unreliable
  non c'é garanzia che i datagrammi vengano recapitati, o vengano recapitati in una sequenza specificata

Livello trasporto:

TCP (Transmission Control Protocol)

- connection-oriented
  realizza un byte stream fra processi (porte)
- reliable
  garantisce la completezza e sequenza dei dati

UDP (Universal Datagram Protocol)

- connectionless
- unreliable

INDIRIZZI IP

È composto da 32 bits (= 4 interi da 1 byte)

Specifica:

- network ID
- host ID all'interno del network

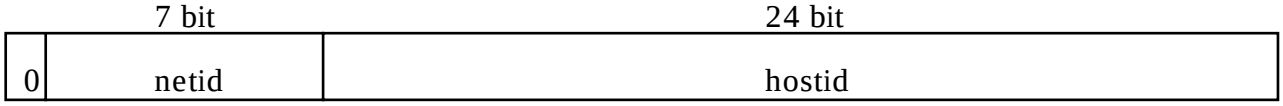
È assegnato da un organismo centrale (a livello di formato e network ID)

Esempio:

191. 45. 3. 58

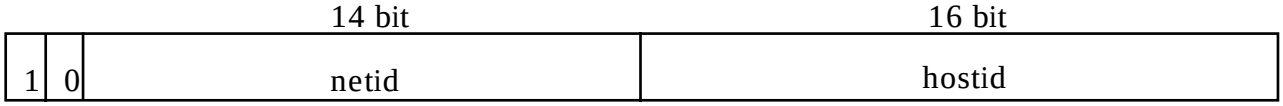
CLASSI DI INDIRIZZI IP

Classe A



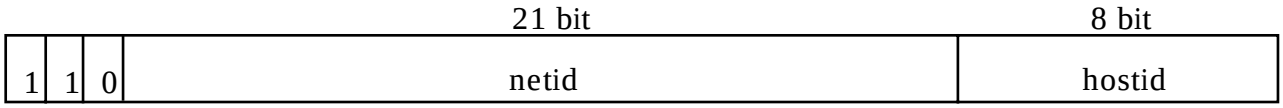
128 reti di oltre 16 milioni di host ciascuna

Classe B



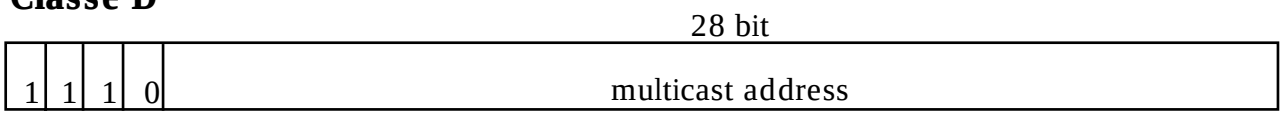
circa 16.000 reti di oltre 65.000 host ciascuna

Classe C



circa 2 milioni di reti di 256 host ciascuna

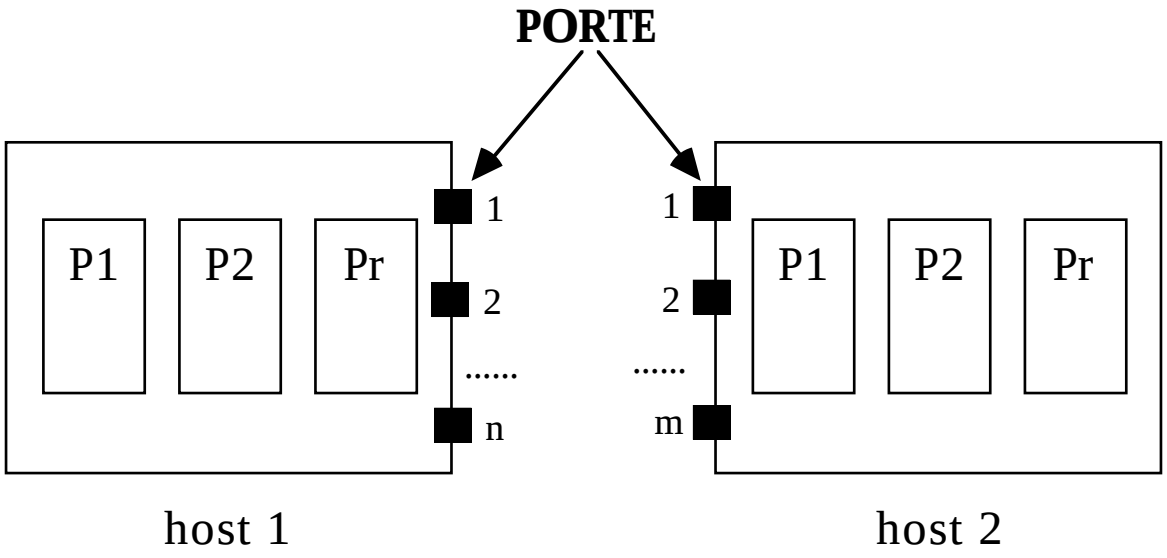
Classe D



PORTE

Su ogni host ho più processi: come indirizzarli?

I processi sono entità dinamiche, i cui nomi sono noti solo localmente --> non indirizzo il processo, ma introduco una nuova astrazione: la "**porta**"



La porta è lo "sportello" al quale il processo client e il processo server si "incontrano"

Port number: intero di 16 bit

PORTE NOTE

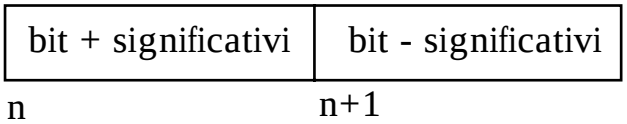
Sono porte il cui numero è prefissato, che corrispondono a servizi standard, ad esempio:

- porta 21: FTP
- porta 80: http

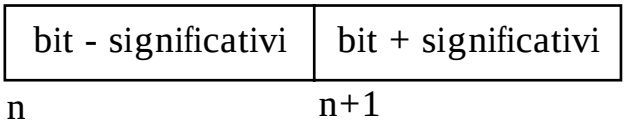
Esempio:

ORDINAMENTO DEI BYTE

Macchine diverse possono usare rappresentazioni diverse:

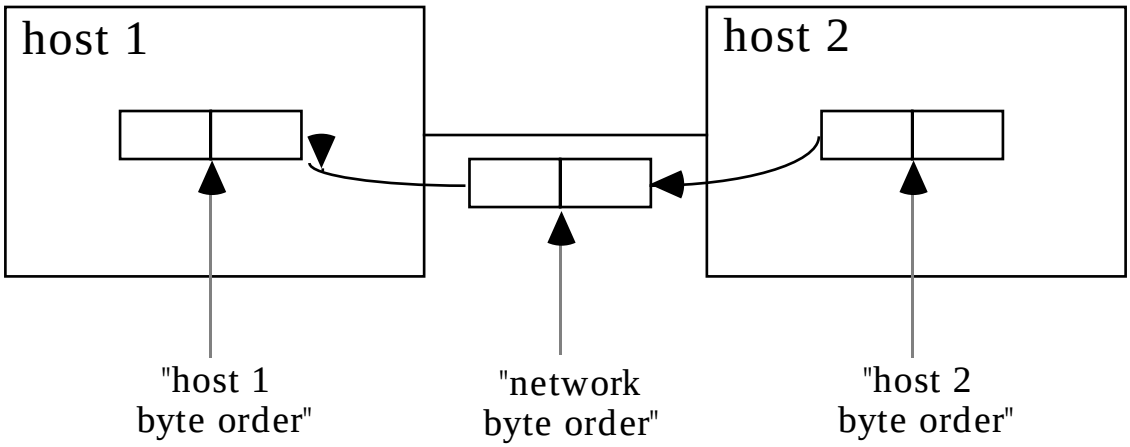


big endian



little endian

Si standardizza il "network byte order":



... e si effettuano le conversioni opportune:

`ntoh()` network to host byte order

`hton()` host to network byte order

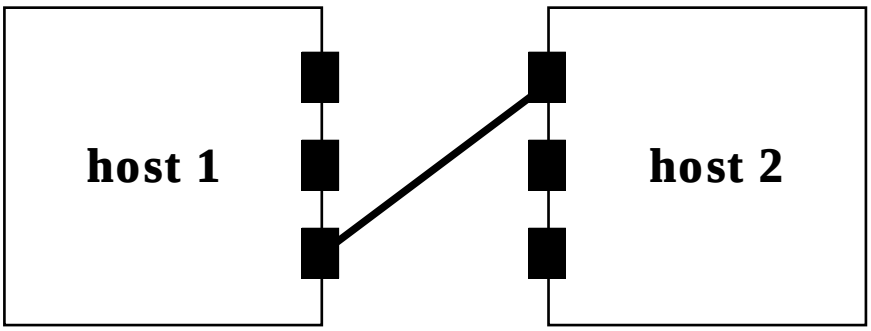
Socket API

IL CONCETTO DI SOCKET

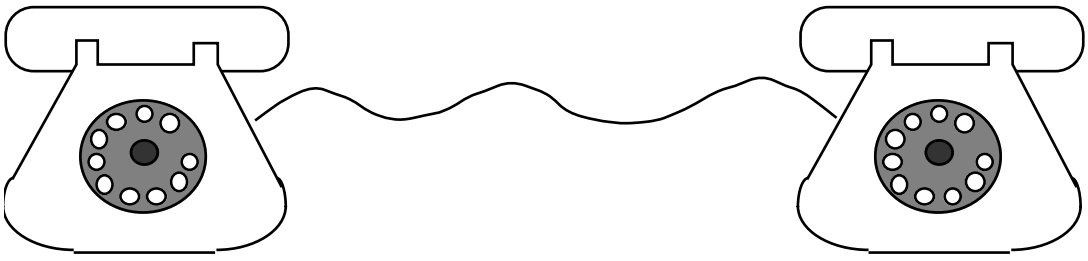
Una **socket** è un punto terminale della comunicazione fra processi, individuata da:

- il **protocollo** usato;
- l'indirizzo IP della **macchina**
- il numero della **porta**

Una **connessione** è univocamente identificata da una coppia di socket:

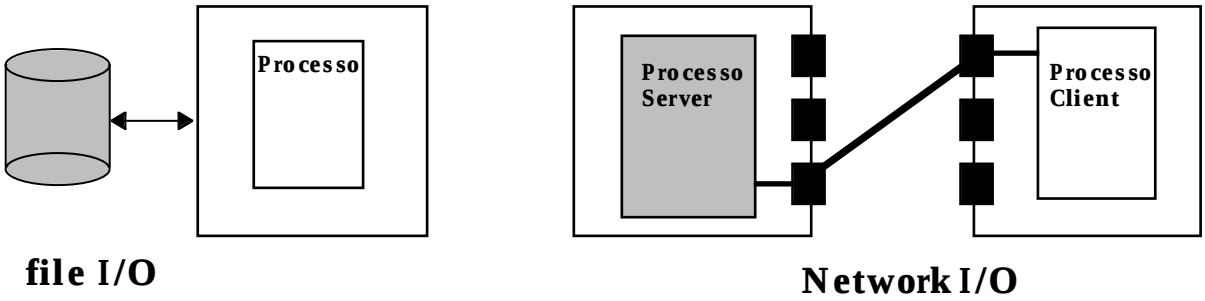


Un'analogia:



<n° telefono> <n° telefono>

FILE I/O E NETWORK I/O: DIFFERENZE



file I/O

Network I/O

Analogie:

- ci sono due interlocutori che si scambiano messaggi
- la relazione non è simmetrica: uno chiede, l'altro risponde
- chi chiede conosce il nome di chi risponde
- se la comunicazione client-server è connection oriented, i due modelli sono molto simili

SOCKET API: L'IDEA

Seguire il modello delle primitive per la gestione dell'I/O su file:

```
fd=open(name,...);
read(fd, ...);
write(fd, ...);
close(fd);
```

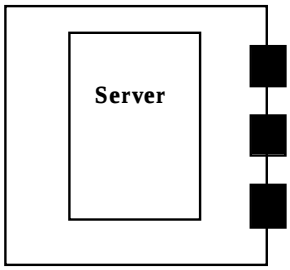
più o meno così:

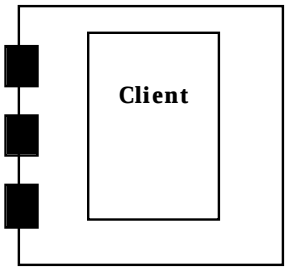
```
sd=creasocket(...);
read(sd, ...);
write(sd, ...);
close(sd);
```

Si avrebbero i seguenti vantaggi:

- trattamento uniforme di file I/O e network I/O
- trasparenza (un'applicazione può non sapere da dove legge o dove scrive)

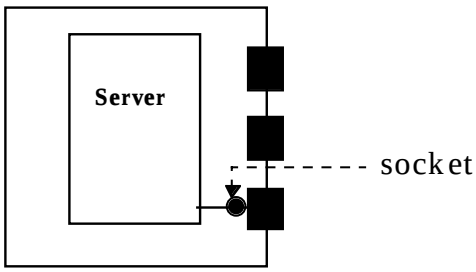
SCHEMA DI COMUNICAZIONE CONNECTION ORIENTED

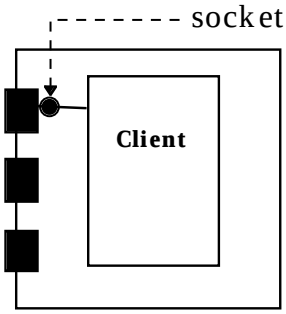





```
sd=creasocket(info);
```

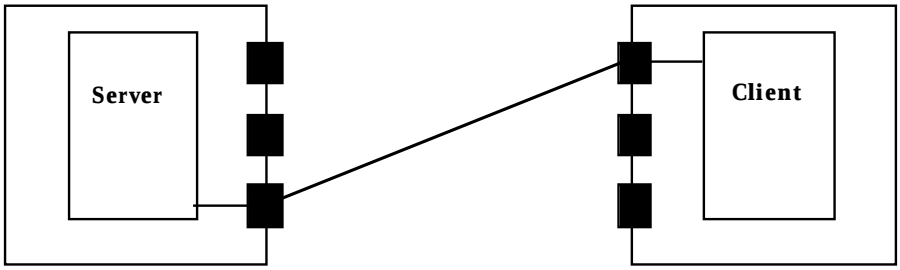
```
sd=creasocket(info);
```






```
accept(sd,da_chi);
```

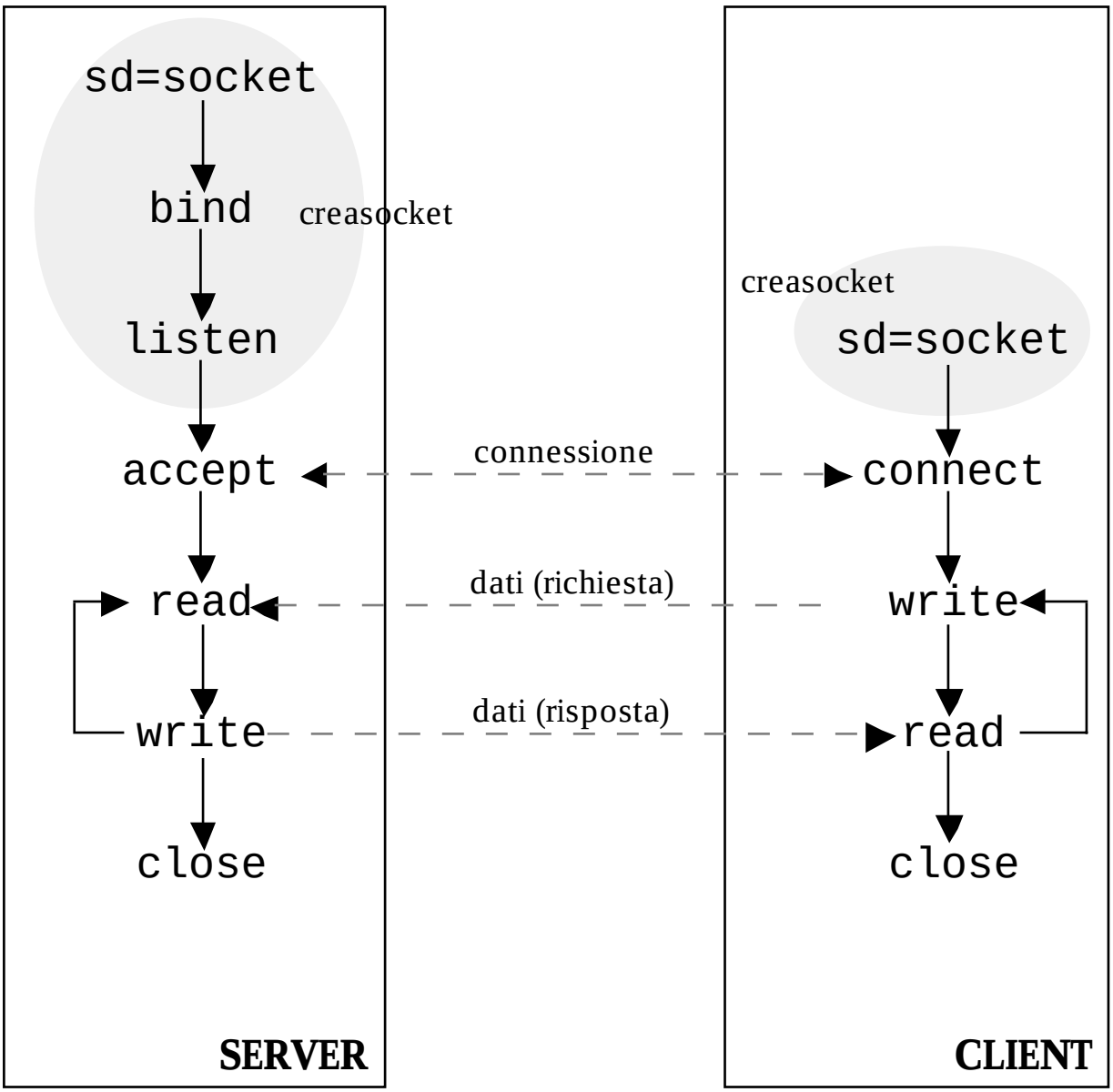
```
connect(sd,a_chi);
```




```
read(sd, ...);
...
write(sd, ...);
...
close(sd);
```

```
write(sd, ...)
...
read(sd, ...);
...
close(sd);
```

SCHEMA DI COMUNICAZIONE CONNECTION-ORIENTED (SEGUE)



LA PRIMITIVA `socket`

```
int socket (int family, int type,
            int protocol);
```

- crea una socket, del genere specificato, e ne restituisce il socket descriptor

`family`	`AF_UNIX`	comunicazione fra processi sulla stessa macchina
	`AF_INET`	protocolli Internet
	`...`	
`type`	`SOCK_STREAM`	connection (TCP)
	`SOCK_DGRAM`	connectionless (UDP)
	`...`	
`protocol`	normalmente `0`	

LA PRIMITIVA bind

```
int bind (int sockfd,
struct sockaddr *myaddr,
    int addrlen);
```

- associa alla socket di descrittore `sockfd` uno specifico indirizzo, puntato da `myaddr`, di lunghezza `addrlen`

Nota:

l'indirizzo della socket può essere di formato diverso, a seconda del protocollo usato:

```
struct sockaddr {
   u_short sa_family;    /*address family*/
   char    sa_data[14];
      /*up to 14 bytes of protocol specific
      address; e.g. 16 bit port number and
      32 bit IP address */
};
```

LA PRIMITIVA `listen`

```
int listen(int sockfd, int backlog);
```

- usata da un server connection-oriented per indicare che desidera ricevere connessioni sulla socket locale `sockfd`

`backlog:` numero massimo di richieste di connessione che possono essere messe in coda

LA PRIMITIVA `accept`

```
int accept (int sockfd,
struct sockaddr *peer,int *addrlen);
```

- usata dal server per attendere una richiesta di connessione sulla socket locale `sockfd`
- il server rimuove la richiesta di connessione dalla coda, e carica l'indirizzo del client in `*peer` (e la sua lunghezza in `*addrlen`)
- crea un'altra socket con le stesse proprietà di `sockfd`, connessa con il client, e ne ritorna il socket descriptor (o ritorna `-1` se errore)
- se non esistono richieste di connessione pendenti, il processo è sospeso

LA PRIMITIVA connect

```
int connect (int sockfd,
struct sockaddr *servaddr,
        int addrlen);
```

- usata da un client per connettere la socket locale `sockfd` alla socket di indirizzo specificato (puntato da `servaddr`, e di lunghezza `addrlen`)
- invia al server l'indirizzo locale a cui inviare la risposta, assegnando una porta libera locale
- la funzione è sospensiva, e ritorna quando la connessione è stata stabilita
- restituisce `0` in caso di connessione, o `-1` in caso di errore (la socket del server non esiste, o la sua coda è piena)

LE PRIMITIVE `read e write`

Sono quelle già note, e servono per leggere e scrivere una **stream** (comunicazione connection-oriented):

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

```
ssize_t write(int sockfd,
const void *buf, size_t nbyte);
```

LA PRIMITIVA `close`

```
int close(int sockfd);
```

- chiude la socket `sockfd` (è la primitiva già nota per i files)

ESEMPIO: CODICE DEL SERVER (CONNECTION-ORIENTED, ITERATIVO)

```
int sockfd, newsockfd;
...
if    (  (sockfd = socket(...) ) < 0)
           err_sys("socket error");
if    (  (bind(sockfd, my_addr, my_ll) < 0)
           err_sys("bind error");
if    (  (listen(sockfd, 5) < 0)
           err_sys("listen error");

for( ; ; )  {
  newsockfd=accept(sockfd,cli_addr,cli_ll);
   if  (newsockfd < 0)
          err_sys("accept error");
   /* process request on newsockfd */
          read(newsockfd, ...);
          write(newsockfd, ...);
   close (newsockfd);
}
```

Nota:

`accept` restituisce in `cli_addr` l'indirizzo del client di cui accetta la richiesta di connessione; pertanto, `newsockfd` é completa: protocollo, `my_addr`, `cli_addr`

ESEMPIO: CODICE DEL SERVER (CONNECTION-ORIENTED, CONCORRENTE)

```
int sockfd, newsockfd;
...
if    (  (sockfd = socket(...) ) < 0)
           err_sys("socket error");
if    (  (bind(sockfd, myaddr, myll) < 0)
           err_sys("bind error");
if    (  (listen(sockfd, 5) < 0)
           err_sys("listen error");

for( ; ; )  {
   newsockfd=accept(sockfd,cliaddr,clill);
   if  (newsockfd < 0)
         err_sys("accept error");
   if  (fork() == 0) { /* child */
         close(sockfd);
         /* process request on newsockfd */
            read(newsockfd, ...);
            write(newsockfd, ...);
         exit (0);
   }
   close (newsockfd);  /* parent */
}
```

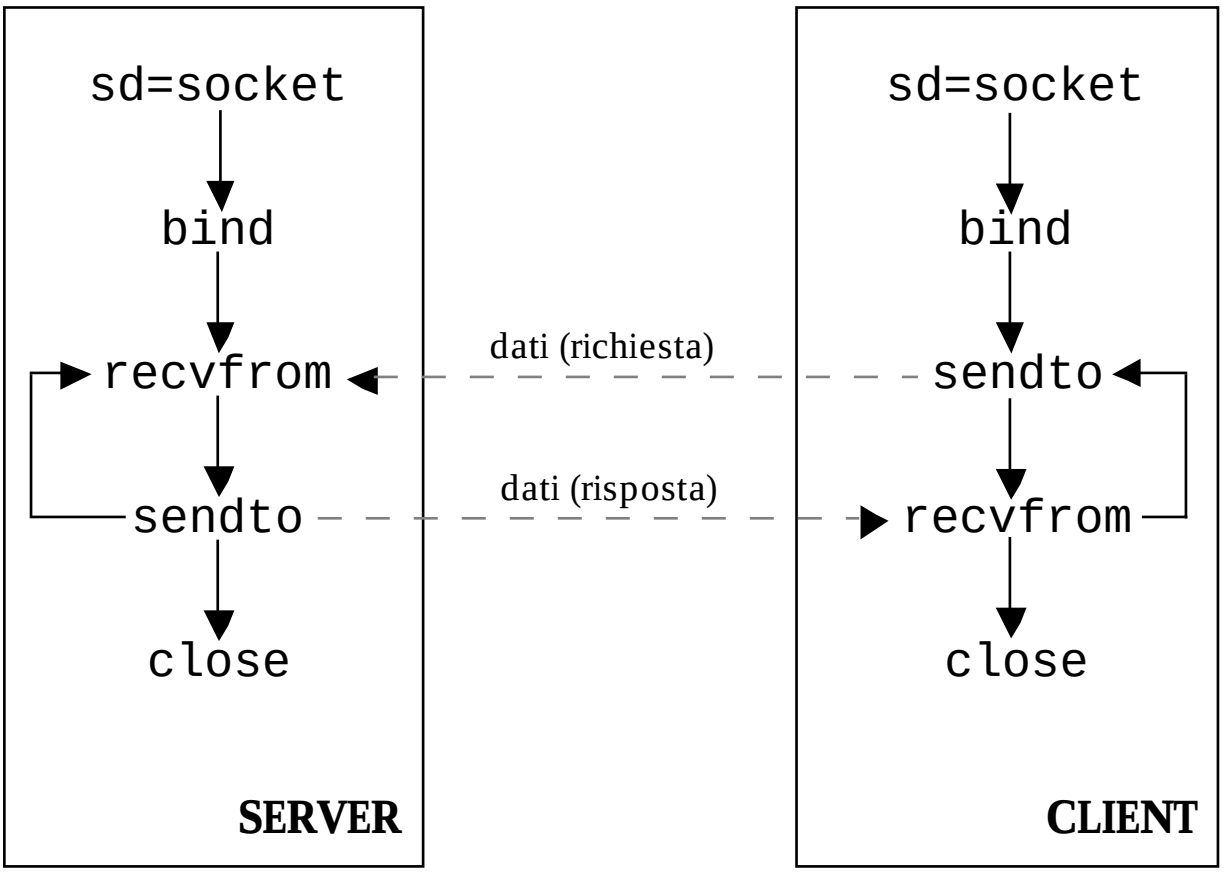
ESEMPIO: CODICE DEL CLIENT (CONNECTION-ORIENTED)

```
int sockfd;
...
if (  (sockfd=socket(...))<0)
         err_sys("socket error");
if (  (connect(sockfd,servaddr,servll)<0)
         err_sys("connect error");

/* process request to server */
   write(sockfd, ...);
   read(sockfd, ...);

close(sockfd);
```

SCHEMA DI COMUNICAZIONE CONNECTIONLESS



LE PRIMITIVE `recvfrom` e `sendto`

Servono per ricevere e inviare un **datagramma** (comunicazione connectionless):

```
int recvfrom(int sockfd, char *buf,
    int nbyte, int flags, struct
   sockaddr *from, int *addrlen);
```

(restituisce in `*from` e `*addrlen` l'indirizzo del mittente)

```
        int sendto(int sockfd,
  char *buf, int nbyte, int flags,
 struct sockaddr *to, int addrlen);
```