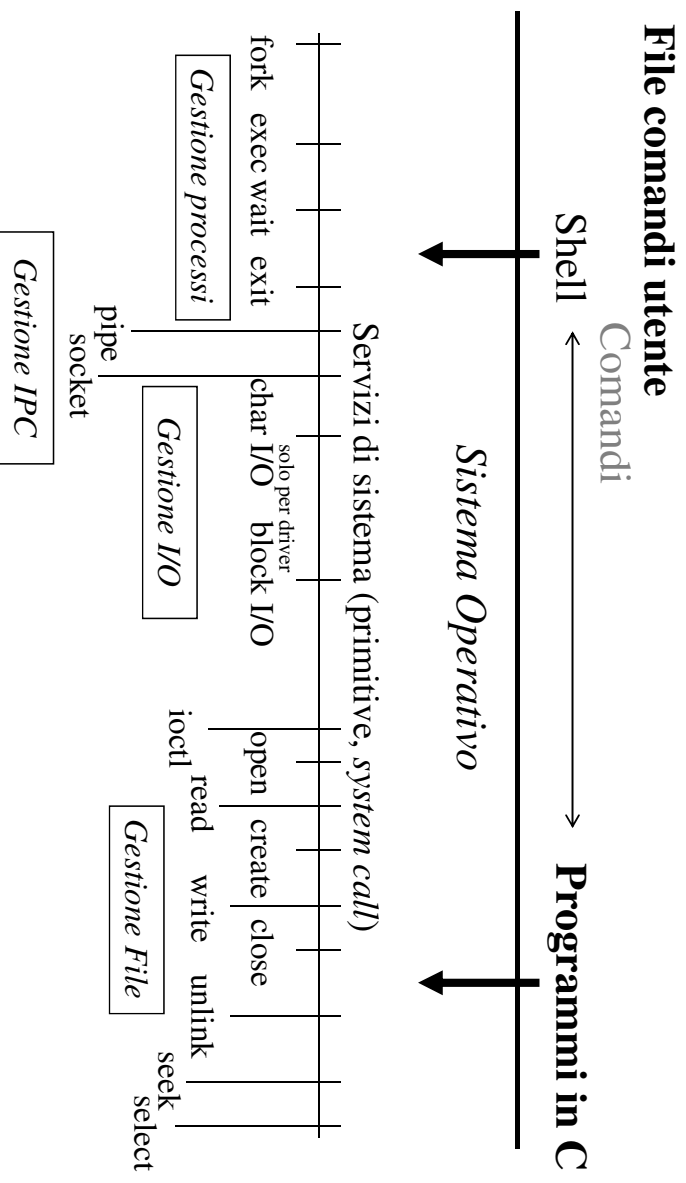


Introduzione

Programmazione di sistema in UNIX



Sis.Op. A - UNIX - Programmazione di sistema

1

Introduzione

Lo spazio di indirizzamento di un processo UNIX

Ogni processo opera in uno spazio di indirizzamento logico e privato

L'immagine di un processo è mappata in un *address space* ed è organizzata in due aree principali

- area utente (codice, dati, stack);
- area di kernel (kernel stack, User area)

entrambe possono essere trasferite sul dispositivo di swap (*swappable*)

Il kernel utilizza inoltre strutture dati residenti (non *swappable*):

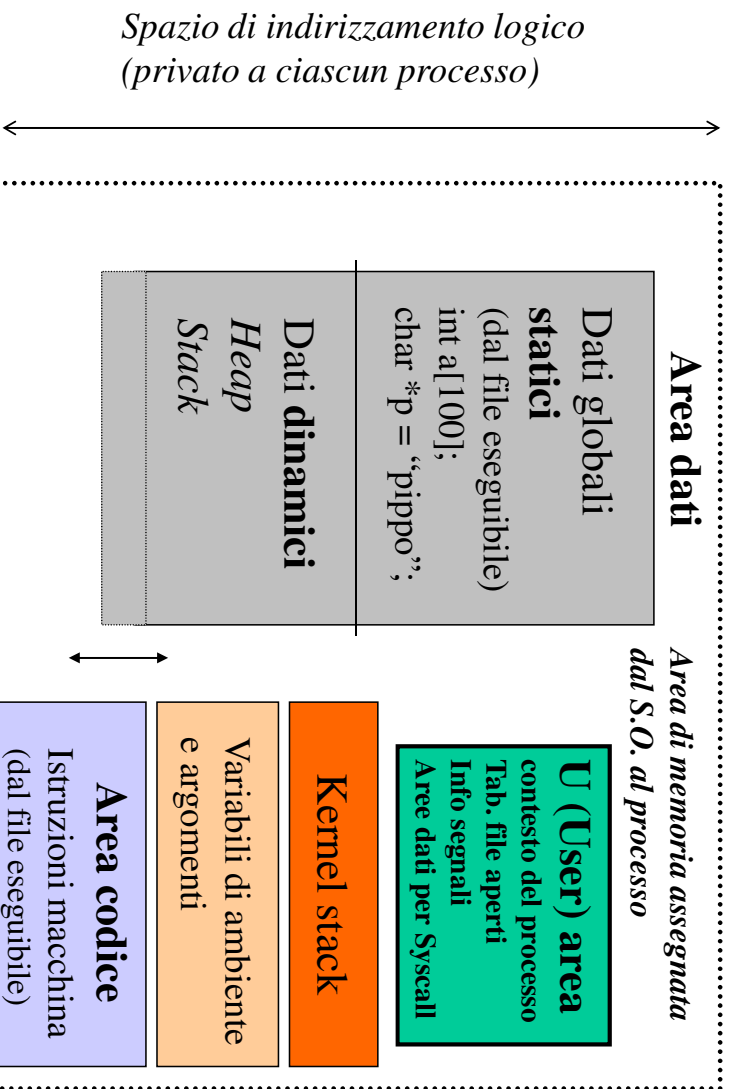
- process table
- text table (tabella dei codici correnti)

Sis.Op. A - UNIX - Programmazione di sistema

2

Introduzione

Immagine di un processo in UNIX



Il processo può riferirsi (generare indirizzi) solo a questa area

Sis.Op. A - UNIX - Programmazione di sistema

3

Introduzione

Lo spazio di indirizzamento di un processo UNIX



Sis.Op. A - UNIX - Programmazione di sistema

4

Introduzione

Esempio ottenuto con pmap in Solaris

```
11368:      -tosh
Address  Kbytes Resident Shared Private Permissions Mapped File
00010000 312      24      24      304      -      24 read/exec      tosh
0006C000 224      24      24      -      -      24 read/write/exec  [ heap ]
00072000 16        8      8      -      -      8 read/write/exec  it.so.2
FF0C0000 16        8      8      -      -      8 read/write/exec  it.so.2
FF0D2000 16        8      8      -      -      8 read/write/exec  it.so.2
FF0E0000 16        8      8      -      -      8 read/write/exec  it.L508859-15.so.2
FF0F2000 16        8      8      -      -      8 read/write/exec  it.L508859-15.so.2
FF100000 648      16      16      600      -      16 read/write/exec  libc.so.1
FF1B0000 40        8      8      -      -      8 read/write/exec  libc.so.1
FF1F0000 24        8      8      16      -      8 read/write/exec  libgen.so.1
FF204000 16        8      8      -      -      8 read/write/exec  libgen.so.1
FF210000 16        8      8      -      -      8 read/write/exec  libmp.so.2
FF222000 8         8      8      -      -      8 read/write/exec  libmp.so.2
FF230000 168      8      8      104      -      8 read/write/exec  libcurses.so.1
FF268000 40        8      8      -      -      8 read/write/exec  libcurses.so.1
FF272000 40        8      8      -      -      8 read/write/exec  [ anon ]
FF280000 512      40      40      440      -      8 read/write/exec  [ anon ]
FF30E000 40        8      8      -      -      8 read/write/exec  libns1.so.1
FF318000 32        8      8      -      -      8 read/write/exec  libns1.so.1
FF330000 16        8      8      -      -      8 read/write/exec  libc_psr.so.1
FF340000 8         8      8      -      -      8 read/write/exec  [ anon ]
FF350000 8         8      8      -      -      8 read/write/exec  libcrypt_i.so.1
FF360000 16        8      8      -      -      8 read/write/exec  libcrypt_i.so.1
FF370000 32        8      8      24      -      8 read/write/exec  libsocket.so.1
FF386000 16        8      8      -      -      8 read/write/exec  libsocket.so.1
FF390000 8         8      8      -      -      8 read/write/exec  libdl.so.1
FF3A0000 8         8      8      -      -      8 read/write/exec  [ anon ]
FF3B0000 120      8      8      112      -      8 read/write/exec  ld.so.1
FF3DC000 8         8      8      -      -      8 read/write/exec  ld.so.1
FFBE4000 48        8      8      -      -      48 read/write/exec  [ stack ]
total kb 2464 2312 1632 680
```

In Linux eseguire : cat `\proc\PIDdelprocesso\maps`

Sis.Op. A - UNIX - Programmazione di sistema

5

Introduzione

Argomenti di un programma

Un programma può accedere agli eventuali argomenti di invocazione attraverso i parametri della funzione principale

main

```
/* mioprogramma.c */
main(int argc, char *argv[])
{int i;
 printf("Numero di argomenti (argc) = %d\n", argc);
 for (i=0;i<argc;i++)
  printf("Argomento %d (argv[%d]) = %s\n", i, argv[i]);
 printf("L'argomento di indice 0 è il nome del programma
 seguito\n");
 }
```

Sis.Op. A - UNIX - Programmazione di sistema

6

Introduzione

Eseguendo il programma

```
$ mioprogramma 1 pippo pluto 4
```

viene visualizzato:

```
Numero di argomenti (argc) = 5
Argomento 0 (argv[0]) = mioprogramma
Argomento 1 (argv[1]) = 1
Argomento 2 (argv[2]) = pippo
Argomento 3 (argv[3]) = pluto
Argomento 4 (argv[4]) = 4
L'argomento di indice 0 è il nome del programma
eseguito
```

Sis.Op. A - UNIX - Programmazione di sistema

7

Introduzione

Variabili di ambiente

Sono accessibili attraverso :

- un terzo parametro char ****envp** della funzione principale **main** (o come **variabile esterna** extern char ****environ** ;)
- mediante le funzioni di utilità **getenv/putenv**

```
/* mioprogramma.c */
main(int argc, char *argv[], char **envp)
{int i;
 printf("Numero di argomenti (argc) = %d\n", argc);
 for (i=0;i<argc;i++)
  printf("Argomento %d (argv[%d]) = %s\n",i,i,argv[i]);
 while (*envp != NULL)
  { printf("%s\n",*envp++);
  }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

8

Introduzione

Le primitive UNIX ritornano sempre un valore intero che esprime il successo ($>=0$) o il fallimento (-1 o comunque <0) della chiamata:

- la descrizione dell'errore viene resa disponibile nella variabile `errno` (non è necessario definirla)
- funzioni di utilità come `perror` o `strerror` permettono di visualizzare o di generare messaggi descrittivi dell'errore

```
if (syscall_N(..., ...) < 0)
{
    perror("Errore nella syscall_N: ");
    /* La descrizione dell'errore viene
    concatenata alla stringa argomento di
    perror */
    exit(-1); /* terminazione del processo con
    errore */
}
```

Sis.Op. A - UNIX - Programmazione di sistema

9

Operazioni sui file

L'uso di file (e di dispositivi) in UNIX si basa su un protocollo di richiesta/rilascio della risorsa

Prologo (richiesta della risorsa)

- primitive `open` o `creat` (o altre per l'accesso a risorse che non siano file)

Uso della risorsa

- primitive `read`, `write` o altre

Epilogo (rilascio della risorsa)

- primitiva `close`

Operazioni sui file

Apertura ed eventuale creazione di un file

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

```
int open (const char *pathname, int flags);
oppure
```

```
fd=int open (const char *pathname, int flags, mode_t mode);
```

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

NB : un file creato con **creat** è accessibile in modalità di sola scrittura

- **pathname** è il nome/percorso del file da aprire o creare
- **flags** contiene il modo di accesso richiesto (uno tra `O_RDONLY`, `O_WRONLY`, `O_RDWR`) più altre eventuali opzioni in OR (ad es. `O_WRONLY | O_CREAT | O_TRUNC` per richiedere la creazione di un nuovo file o per azzerarlo se già esistente)
- **mode** indica i diritti di accesso del nuovo file (ad es. in ottale 0644)

Sis.Op. A - UNIX - Programmazione di sistema

11

Operazioni sui file

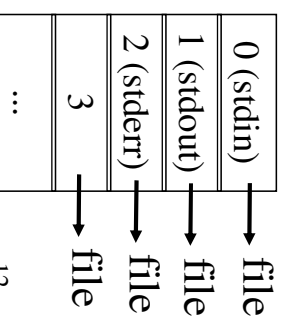
Se la invocazione di una primitiva `open` o `creat` ha successo, viene restituito al processo un valore intero ≥ 0 che costituisce un *file descriptor* (*fd*) per quel file:

- il file descriptor va utilizzato per le successive operazioni su quel file da parte di quel processo (invece del nome/percorso che viene utilizzato solo in `open/creat`)
- numeri successivi vengono associati ai nuovi file aperti (3, 4, ...)

Lo stesso comportamento vale anche per le primitive utilizzate per creare/accedere altri tipi di risorsa (`pipe`, `socket`)

I file descriptor sono indicati per una tabella dei file aperti mantenuta per il processo nella sua User Area

i primi tre file descriptor sono predefiniti (non è necessario crearli ma vanno modificati per ottenere la redirezione)



Sis.Op. A - UNIX - Programmazione di sistema

12

Operazioni sui file

Duplicazione di un file descriptor

```
#include <unistd.h>

int dup(int oldFd);
int dup2(int oldFd, int newFd);
```

*Utilizzati per la redirection
e il piping*

Il valore di ritorno di `dup` o il `newFd` di `dup2` possono essere utilizzati indifferenteemente al posto del file descriptor originale

Chiusura di un file descriptor

```
#include <unistd.h>

int close(int fd);
```

La chiusura di un file descriptor consente di riutilizzarlo per un nuovo file

Se non vi sono altri file descriptor per quell'oggetto, le risorse associate (ad. es. l'I/O pointer) vengono effettivamente liberate

Sis.Op. A - UNIX - Programmazione di sistema

13

Operazioni sui file

Letture e scrittura da un *file descriptor* (file o altre risorse)

```
#include <unistd.h>

int read(int fd, void *buf, size_t count);
int write(int fd, void *buf, size_t count);
```

- **read** prova a leggere dall'oggetto a cui si riferisce `fd` fino a `count` byte, memorizzandoli a partire dalla locazione `buf`.

- **write** prova a scrivere sull'oggetto a cui si riferisce `fd` fino a `count` byte, letti a partire dalla locazione `buf`

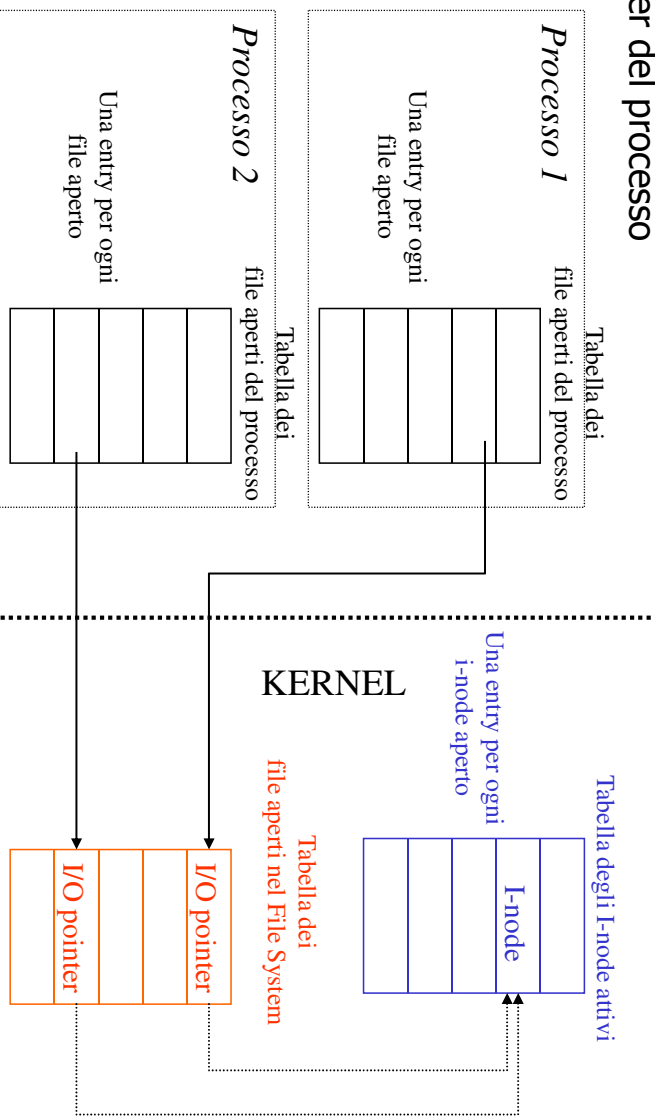
- La lettura e scrittura di un file avvengono a partire dalla posizione corrente del file indicata da uno **I/O pointer** che viene modificato dalla operazione

- Le primitive ritornano il numero di byte effettivamente letti/scritti (una lettura di 0 byte significa che l'I/O pointer punta alla fine del file (EOF))

Operazioni sui file

Ogni processo ha la propria visione dei file aperti

- se più processi aprono lo stesso file, ciascun processo si riferisce ad un proprio I/O pointer distinto da quello degli altri
- le operazioni condotte da un processo su un file modificano il solo I/O pointer del processo



Sis.Op. A - UNIX - Programmazione di sistema

15

Operazioni sui file

Proprietà di `read` e `write`

• Sincronizzazione

- **read** è normalmente sincrona: la primitiva attende la disponibilità dei dati richiesti a meno che non sia stato specificato il flag `O_NONBLOCK` in fase di apertura o creazione
- **write** è normalmente semi-sincrona: la primitiva ritorna subito dopo aver scritto i dati in un buffer di kernel mentre l'effettiva scrittura sul disco viene portata a termine in modo asincrono (a meno che non sia stato specificato il flag `O_SYNC` in fase di apertura o creazione che comporta l'attesa della scrittura fisica sul disco)

• Atomicità

- L'esecuzione di una singola primitiva `write` o `read` non è interrompibile mentre non è garantita l'atomicità di una sequenza di `read` e/o `write`
- l'accesso esclusivo ad un file può essere ottenuto mediante `fcntl` oppure `flock`

Sis.Op. A - UNIX - Programmazione di sistema

16

Operazioni sui file

Esempi di lettura/scrittura : Copia di file (ver. 1)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFFSIZ 4096

main()
{
    char    *f1= "filesorg"; char *f2= "/tmp/filedest";
    int     infile, outfile; /* file descriptor */
    int     nread;
    char    buffer[BUFFSIZ];

    /* apertura file sorgente */
    if ((infile=open(f1,O_RDONLY)) <0)
        { perror("Apertura f1: "); exit(-1); }

    /* creazione file destinazione */
    if ((outfile=open(f2,O_WRONLY|O_CREAT|O_TRUNC, 0644)) <0)
        { perror("Creazione f2: "); exit(-2);}

    /* Ciclo di lettura/scrittura fino alla fine del file sorgente */
    while((nread= read(infile, buffer, BUFFSIZ)) >0)
        if(write(outfile, buffer, nread) != nread)
            { perror("Errore write: "); exit(-3);}
    if(nread < 0) { perror("Errore read: "); exit(-4);}

    close(infile); close(outfile);
    exit(0);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

17

Operazioni sui file

Copia di file (ver. 2 - uso argomenti)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

#define PERM 0644

main(int argc, char *argv[])
{
    int     infile, outfile; /* file descriptor */
    int     nread;
    char    buffer[BUFFSIZ];

    /* Controllo del numero degli argomenti */
    if (argc != 3)
        { fprintf(stderr, "Uso: %s filesorg filedest\n", argv[0]);
          exit(-1); }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

18

Operazioni sui file

Copia di file (ver. 2 - uso argomenti) (cont.)

```
/* apertura file sorgente */
if ((infile=open(argv[1],O_RDONLY)) <0)
    {fprintf(stderr,"Non posso aprire %s: %s\n", argv[1],
    strerror(errno)); exit(-2); }

/* creazione file destinazione */
if ((outfile=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC, PERM)) <0)
    {fprintf(stderr,"Non posso creare %s: %s\n", argv[2],
    strerror(errno)); exit(-3);}

/* Ciclo di lettura/scrittura fino alla fine del file sorgente
*/
while((nread= read(infile, buffer, BUFSIZ)) >0)
    if(write(outfile, buffer, nread) != nread)
        { perror("Errore write: "); exit(-4);}

if (nread < 0) { perror("Errore read: "); exit(-5);}
close(infile); close(outfile);
exit(0);
}
$ls.Op.A - UNIX - Programmazione di sistema
```

19

Operazioni sui file

Copia di file (ver. 3 - richiede la redirectione dell' I/O)

```
/* copyredir.c */
#include <unistd.h>
#define BUFSIZ 4096

main()
{
    int nread;
    char buffer[BUFSIZ];

    /* Ciclo di lettura/scrittura fino alla fine del file sorgente */
    while((nread= read(0, buffer, BUFSIZ)) >0)
        if(write(1, buffer, nread) != nread)
            { perror("Errore write su stdout"); exit(-1);}

    exit(0);
}
```

Esecuzione

```
$copyredir < filesorgente >filedestinazione
```

Operazioni sui file

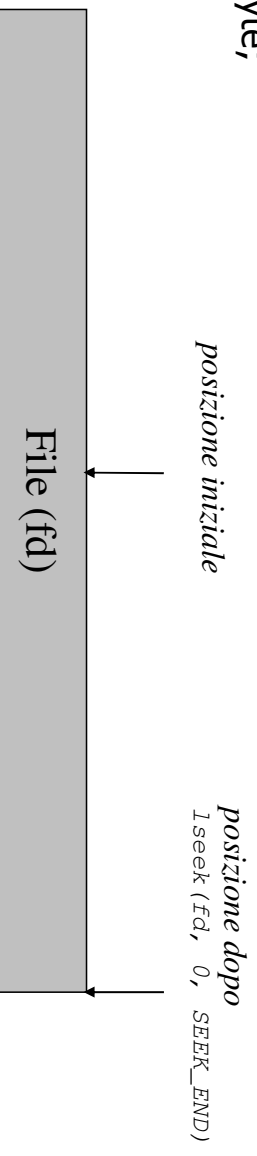
Riposizionamento non sequenziale dell' I/O pointer

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

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

Valori per il parametro whence

- SEEK_SET: l'I/O pointer punta ad offset byte dall'inizio del file;
- SEEK_CUR: l'I/O pointer punta a offset byte oltre la sua posizione corrente;
- SEEK_END: l'I/O pointer punta alla fine del file più offset byte;



Sis.Op. A - UNIX - Programmazione di sistema

21

Operazioni sui file

Informazioni su file (ordinari, speciali, direttori)

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

```
int stat (const char *filename, struct stat *buf);
int fstat (int fd, struct stat *buf);
```

Nella struttura buf vengono riportate le informazioni relative al file:

```
struct stat
{
    dev_t      st_dev;      /* device */
    ino_t      st_ino;     /* inode */
    mode_t     st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device type (if inode device) */
    off_t     st_size;    /* total size, in bytes */
    unsigned long st_blksize; /* blocksz for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Sis.Op. A - UNIX - Programmazione di sistema

22

Operazioni sui file

Cancellazione di file

```
#include <unistd.h>
```

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

Il file viene cancellato solo se:

- si tratta dell'ultimo link al file
- non vi sono altri processi che lo hanno aperto ; il file verrà effettivamente rimosso all'atto dell'ultima close

Nel caso si tratti di un link simbolico viene rimosso il link

Operazioni sui file

Controllo su file e dispositivi

```
#include <unistd.h>  
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd /*, arg */);
```

Vari comandi per il controllo sui file aperti: ad.es. gestione dei lock

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

```
int ioctl(int fd, int cmd /*, arg */);
```

Vari comandi per il controllo della modalità di funzionamento dei dispositivi

I comandi applicabili sono in genere determinati dalla tipologia del dispositivo anche se sono disponibili `ioctl` di carattere generale

```
ioctl(fd, FIONREAD, &available) in available viene restituito il  
numero di byte disponibili per la lettura
```

Operazioni sui file

Esempio di configurazione di un terminale su linea seriale

```
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/termios.h>

struct termios terminal;

...
/* Mancano i controlli sull'esito delle primitive */
fd = open("/dev/ttyS0", O_RDWR);
/* Ottiene la configurazione corrente della linea */
ioctl(fd, TCGETS, &terminal);
/* Modifica la configurazione per una comunicazione a 8 bit */
terminal.c_iflag |= CS8 ;
/* Aggiorna la configurazione corrente della linea */
ioctl(fd, TCSETS, &terminal);
...
```

Sis.Op. A - UNIX - Programmazione di sistema

25

Gestione dei processi

La creazione di un nuovo processo in UNIX

```
#include <unistd.h>

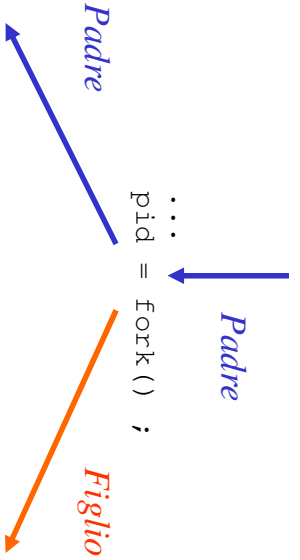
int fork(void);
```

Viene creato un **nuovo** processo (*figlio*) identico (stesso codice, area dati copiata) al processo (*padre*) che ha invocato la `fork`

Solo il valore di ritorno dalla `fork` è diverso per i due processi

```
...
pid = fork();
...
/* pid vale PID_Figlio (l'ID del figlio) */
printf("pid vale %d\n", pid);
...
/* pid vale 0 */
printf("pid vale %d\n", pid);
...

```



Sis.Op. A - UNIX - Programmazione di sistema

26

Gestione dei processi

Schema di generazione

```
#include <unistd.h>
```

I ruoli di padre e figlio sono relativi ad una specifica fork : il figlio può a sua volta generare altri processi

```
if (fork() ==0)
  { /* Codice eseguito dal figlio */
  ...
  }
else
  { /* Codice eseguito dal padre */
  ...
  }
```

Dopo la generazione del nuovo processo, padre e figlio sono processi indipendenti

Il padre può decidere:

- se continuare la propria esecuzione concorrentemente a quella del figlio
- se attendere che il figlio termini (primitiva `wait`)

Gestione dei processi

Il processo figlio :

- utilizza lo stesso codice che sta eseguendo il padre ;
- dispone di una area dati (utente/kernel) che è una copia (spesso *on-write*) di quella che ha il padre all'atto della fork;
⇒ **non vi è nessuna condivisione di memoria tra i processi padri e figlio**
(MOLTO IMPORTANTE)

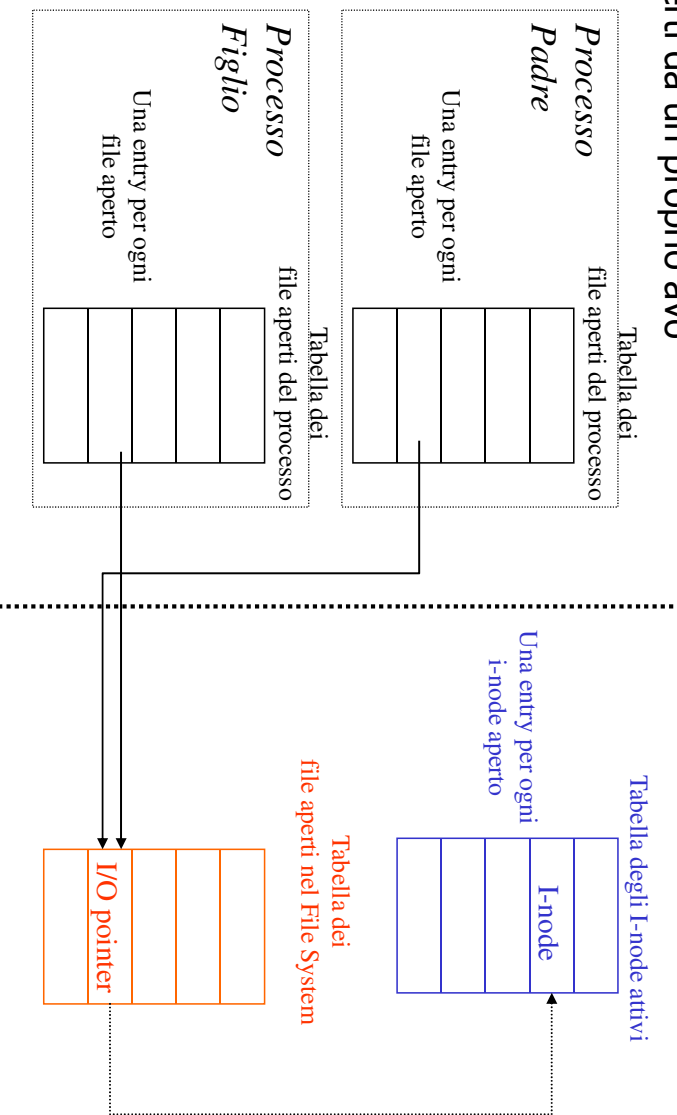
La duplicazione dell'area di kernel ha come effetto che **il processo figlio eredita la gestione dei segnali e la tabella dei file aperti del padre** :

- i file aperti dal padre risultano aperti anche dal figlio con una condivisione degli I/O pointer

⇒ le operazioni su uno stesso file condotte da una famiglia di **processi modificano l'I/O pointer comune**

Operazioni sui file

I processi di una stessa famiglia non hanno una visione indipendente dei file aperti da un proprio avo



È possibile chiudere il file descriptor "ereditato" e riaprire di nuovo il file per avere un accesso indipendente

Sis.Op. A - UNIX - Programmazione di sistema

29

Gestione dei processi

Identificazione dei processi

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

```
pid_t getppid(void);
```

La `getpid` ritorna al processo chiamante il suo identificatore di processo (PID)

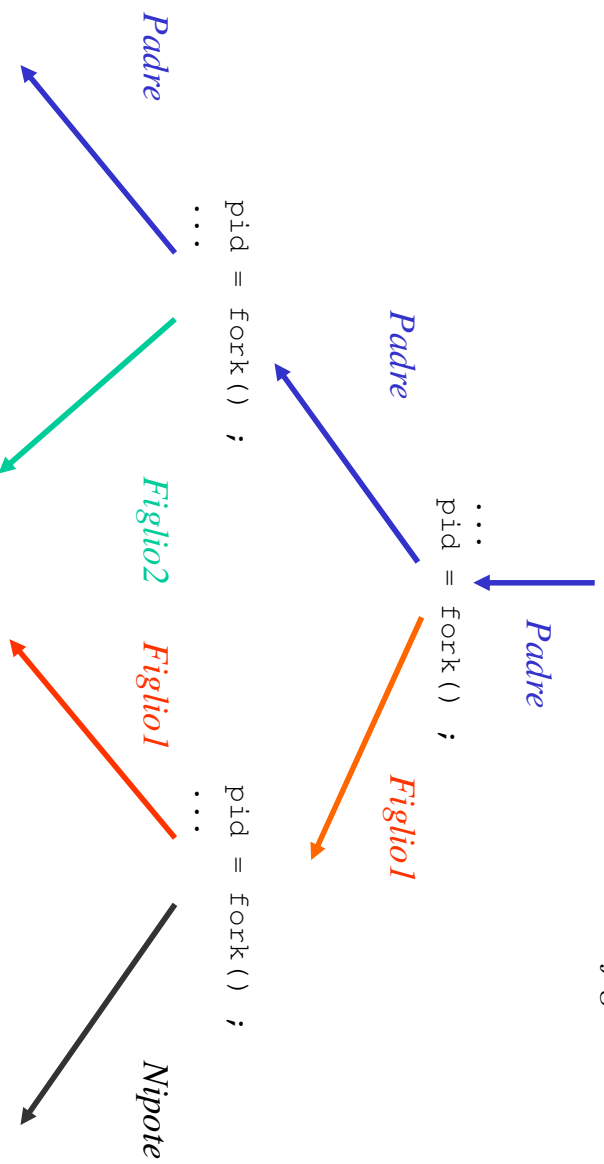
La `getppid` ritorna al processo chiamante l' identificatore di processo del suo processo padre (PID del padre)

Dalla `fork` il padre riceve il PID del processo figlio che deve invece invocare `getpid` per conoscere il proprio PID

Gestione dei processi

Attenzione alla fork

```
... pid = fork() ;          Vengono creati 3 processi  
pid = fork() ;          • 2 dal padre  
...                      • 1 da un figlio
```



Sis.Op. A - UNIX - Programmazione di sistema

31

Gestione dei processi

Sincronizzazione tra padre e figlio

```
#include <sys/types.h>  
#include <sys/wait.h>  
  
pid_t wait(int *status);
```

Il processo chiamante rimane bloccato in attesa della terminazione di uno tra i suoi figli (se i processi figli sono già terminati `wait` ritorna immediatamente)

- `status` contiene il valore di uscita del processo figlio: per ottenerlo (`status >> 8`) oppure `WEXITSTATUS(status)`
- se la `wait` ha successo il valore di ritorno è il pid del processo figlio che è terminato

Con la variante `waitpid` è possibile ottenere informazioni sullo stato dei processi figli anche senza bloccare il processo chiamante (opzione `WNOHANG`)

Sis.Op. A - UNIX - Programmazione di sistema

32

Gestione dei processi

Uso della wait

```
int status;
/* NB int *status ; è sbagliato : perché ?
*/
if (fork() ==0)
    { /* Codice eseguito dal figlio */
    ...
    }
else
    { /* Codice eseguito dal padre */
    ...
    /* Attende che il figlio termini */
    wait (&status);
    ...
}
```

Nel caso di più figli può essere necessario attendere la terminazione di uno specifico figlio (pidfiglio):

```
while ( (pid = wait (&status)) != pidfiglio) ;
```

Sis.Op. A - UNIX - Programmazione di sistema

33

Gestione dei processi

Terminazione volontaria di un processo

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

#include <unistd.h>
void _exit(int status)
```

Un processo termina volontariamente invocando la primitiva `_exit` oppure la funzione `exit` della libreria standard I/O del C oppure alla conclusione della funzione `main` (dove viene invocata automaticamente `_exit`);

`status` è il valore di uscita che viene reso disponibile al padre attraverso la `wait` o le altre primitive (`waitpid, ...`)

`_exit` e `exit` chiudono tutti i file aperti del processo

Sis.Op. A - UNIX - Programmazione di sistema

34

Gestione dei processi

Terminazione involontaria di un processo

Un processo termina involontariamente a seguito di:

- azioni non consentite:
 - riferimenti ad indirizzi di memoria non assegnati al processo (SIGSEGV)
 - esecuzione di codici di istruzioni non definite (SIGILL)
-
- segnali generati dall'utente da tastiera
 - ^C (SIGINT)
 - ^\ (SIGQUIT) con generazione del *corefile*
 - segnali inviati da un altro processo
 - mediante la primitiva kill (che vedremo più avanti)
 - mediante il comando kill (ad es. kill -9 pidprocesso)

Sis.Op. A - UNIX - Programmazione di sistema

35

Gestione dei processi

Esempio di uso della wait

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

int main (int argc, char **argv)
{
    int fd,pid, pid_processo;
    int nread,nwrite,status;
    char st1[256];
    char st2[256];

    if (argc != 2) {fprintf(stderr,"Uso: %s nomefile\n",argv[0]); exit(-1); }

    /* Apertura del file in lettura e scrittura */
    if ((fd=open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0644))<0)
        {perror("opening argv[1]: "); exit(-2);}

    if ((pid=fork())<0)
        {
            {perror("fork: "); exit(-3);}
        }
    else
        if (pid==0) /* Processo figlio */
            {
                /* Continua nella trasparenza successiva ... */
            }
        }
```

Sis.Op. A - UNIX - Programmazione di sistema

36

Gestione dei processi

Esempio

```
/* ... continua processo figlio */
printf("Introduci una stringa e premi [Enter]\n");
scanf("%s", st1);
/* Il figlio eredita il descrittore fd dal padre */
nwrite= write(fd, st1, strlen(st1)+1); /* per scrivere anche il '\0' */
/* L' I/O pointer si e' spostato alla fine del file */
exit(nwrite);
}
else
{ /* pid > 0 : Processo padre */

/* Attesa della terminazione del figlio */
pid_processo = wait(&status);
/* Con un solo figlio generato, pid_processo è uguale a pid */
/* Riposizionamento all'inizio del file */
lseek(fd, 0, SEEK_SET);

if( (nread = read(fd, st2, 256)) < 0)
    { perror("read "); exit(-4); }

printf("Il figlio ha letto la stringa %s\n", st2);
close(fd);
exit(0);
}
}
```

Sis.Op. A - UNIX - Programmazione di sistema

37

Gestione dei processi

Esecuzione di un programma

```
#include <unistd.h>

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

Il processo chiamante passa ad eseguire il programma `filename` (file eseguibile o *script*): non è previsto il ritorno della chiamata a meno che non vi sia un errore (-1)

Con `argv` e `envp` è possibile specificare gli argomenti e le variabili di ambiente che il programma riceve

L'ambiente di esecuzione (codice e dati) del processo corrente viene modificato in quello del programma (senza che venga creato un nuovo processo - il PID rimane invariato) :

- la fork crea un nuovo processo identico al padre
- la exec permette di modificare l'ambiente di esecuzione di un processo (in modo da renderlo diverso da quello del padre)

Sis.Op. A - UNIX - Programmazione di sistema

38

Gestione dei processi

Varianti della `execve`

```
#include <unistd.h>

int execl( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execl_e( const char *path, const char *arg , ...,
char * const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
```

La `execlp` e la `execvp` ricercano il `file` nei direttori indicati nella variabile di ambiente `$PATH` (è un comportamento analogo a quello dello `shell` : non è necessario il path completo)

Sis.Op. A - UNIX - Programmazione di sistema

39

Gestione dei processi

Effetti delle primitive `exec`

La gestione dei segnali (vista più avanti) viene alterata:

- se essi venivano **ignorati** rimangono tali
- se erano collegati a **funzioni** (handler) vengono riportati alla gestione di default

Se il file eseguito hai il bit SUID attivo, gli identificatori **effettivi** del processo vengono modificati in quelli del **proprietario del file eseguito**, mentre quelli **reali** rimangono **inalterati**

Si ereditano:

direttorio corrente, maschera dei segnali, terminale di controllo ed altre informazioni

Gestione dei processi

Esempio di uso della execve

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

int main() {
    int status;
    pid_t pid;
    char *env[] = {
        "TERM=vt100",
        "PATH=/bin:/usr/bin",
        (char *) 0 };

    char *args[] = {
        "cat",
        "f1",
        "f2",
        (char *) 0 };
}
```

```
if ( (pid=fork())==0) {
    /* Codice eseguito dal figlio */
    execve ("/bin/cat", args, env);
    /* Si torna qui solo in caso di errore/fallimento della execve */
    perror("execve: ");
    exit(-1);
}
else {
    /* Codice eseguito dal padre */
    wait(&status);
    printf("Il processo %d e' terminato con %d\n", pid, WEXITSTATUS(status));
    exit(0);
}
}
```

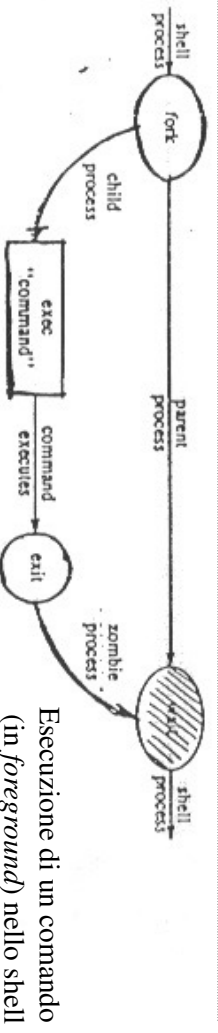
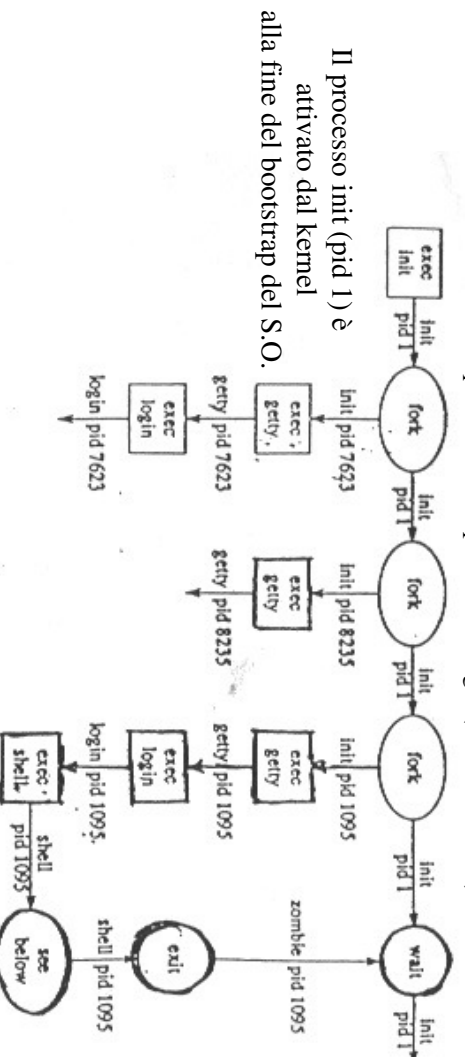
Sis.Op. A - UNIX - Programmazione di sistema

41

Gestione dei processi

La gestione delle sessioni utente

Un processo per ogni dispositivo di linea (terminale) attraverso il quale un utente può fare un login (vedi /etc/inittab)



Sis.Op. A - UNIX - Programmazione di sistema

42

Segnali

Sincronizzazione mediante segnali

Vi sono spesso eventi importanti da notificare ai processi:

- tasti speciali sul terminale (es. ^C)
- eccezioni hardware (es. divisione per 0)
- primitiva/comando kill (es. `kill -9 1221`)
- condizioni software (es. scadenza di un timer)

L'arrivo di tali **eventi asincroni** può richiedere un'immediata gestione da parte del processo (analogamente alle interruzioni hardware)

⇒ **i segnali sono anche detti software interrupts**

Segnali

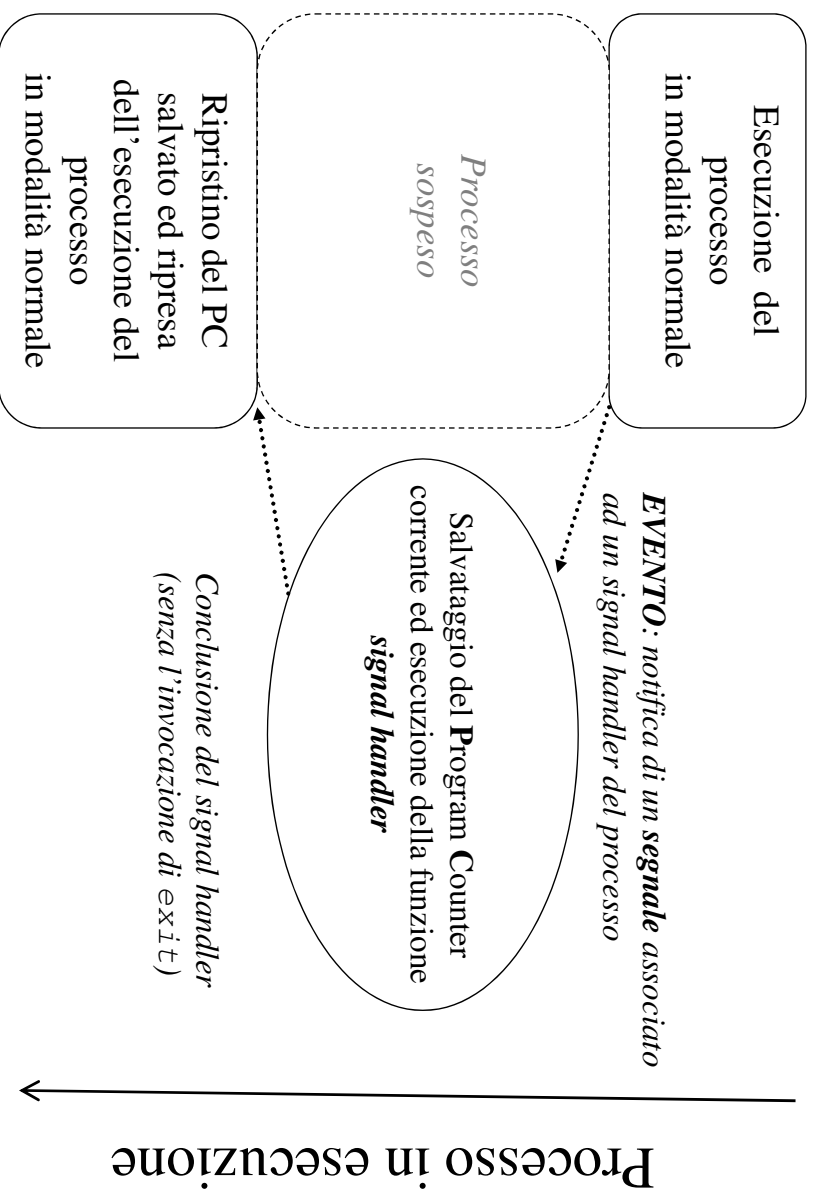
Quali sono le possibilità di gestione di un segnale per un processo ?

1. può decidere di **ignorarlo** (possibile solo per alcuni tipi di segnale)
2. può contare su un'azione di **default**
3. può far eseguire un'**azione** specificata dall'utente (gestore del segnale - *signal handler*)

Per tutta la durata dell'esecuzione del gestore del segnale, l'esecuzione del programma interrotto rimane bloccata :

⇒ al processo UNIX è associato un solo flusso di controllo (un solo Program Counter)

Segnali



Sis.Op. A - UNIX - Programmazione di sistema

45

Segnali

Elenco dei segnali Linux (/usr/include/asm/signal.h)

#define	SIGHUP	1	/* Hangup (POSIX). */
#define	SIGINT	2	/* Interrupt (ANSI). */
#define	SIGQUIT	3	/* Quit (POSIX). */
#define	SIGILL	4	/* Illegal instruction (ANSI). */
#define	SIGTRAP	5	/* Trace trap (POSIX). */
#define	SIGABRT	6	/* Abort (ANSI). */
#define	SIGIOT	6	/* IOT trap (4.2 BSD). */
#define	SIGBUS	7	/* BUS error (4.2 BSD). */
#define	SIGFPE	8	/* Floating-point exception (ANSI). */
#define	SIGKILL	9	/* Kill, unblockable (POSIX). */
#define	SIGUSR1	10	/* User-defined signal 1 (POSIX). */
#define	SIGSEGV	11	/* Segmentation violation (ANSI). */
#define	SIGUSR2	12	/* User-defined signal 2 (POSIX). */
#define	SIGPIPE	13	/* Broken pipe (POSIX). */
#define	SIGALRM	14	/* Alarm clock (POSIX). */
#define	SIGTERM	15	/* Termination (ANSI). */
#define	SIGSTKFLT	16	/* Stack fault. */
#define	SIGCLD	SIGCHLD	/* Same as SIGCHLD (System V). */

Per Solaris 5.x cfr. /usr/include/sys/signal.h : alcuni segnali hanno valori diversi

Sis.Op. A - UNIX - Programmazione di sistema

46

Segnali

Elenco dei segnali (cont.)

#define	SIGCHLD	17	/* Child status has changed (POSIX). */
#define	SIGCONT	18	/* Continue (POSIX). */
#define	SIGSTOP	19	/* Stop, unblockable (POSIX). */
#define	SIGTSTP	20	/* Keyboard stop (POSIX). */
#define	SIGTTIN	21	/* Background read from tty (POSIX). */
#define	SIGTTOU	22	/* Background write to tty (POSIX). */
#define	SIGURG	23	/* Urgent condition on socket (4.2 BSD). */
#define	SIGXCPU	24	/* CPU limit exceeded (4.2 BSD). */
#define	SIGXFSZ	25	/* File size limit exceeded (4.2 BSD). */
#define	SIGVTALRM	26	/* Virtual alarm clock (4.2 BSD). */
#define	SIGPROF	27	/* Profiling alarm clock (4.2 BSD). */
#define	SIGWINCH	28	/* Window size change (4.3 BSD, Sun). */
#define	SIGPOLL	29	/* Pollable event occurred (System V). */
#define	SIGIO	29	/* I/O now possible (4.2 BSD). */
#define	SIGPWR	30	/* Power failure restart (System V). */
#define	SIGSYS	31	/* Bad system call. */
#define	SIGUNUSED	31	

Sis.Op. A - UNIX - Programmazione di sistema

47

Segnali

L'interfaccia **signal** per la gestione dei segnali

```
#include <signal.h>
void (*signal( int signo, void (*func)(int))) (int) ;
```

Si specifica quale segnale (`signo`) e come deve essere trattato (`func`)

Valori ammessi per il parametro `func` :

1. **SIG_IGN** (ignora il segnale - solo per alcuni segnali)
2. **SIG_DFL** (azione di default per quel segnale)
3. indirizzo (ovvero in C il nome) di una funzione del programma (*signal handler* o gestore del segnale)

Segnali

`signal` fornisce un'interfaccia semplificata presente in ANSI C

- in SVR4 si ottengono segnali *inaffidabili* :
 - alla notifica del segnale, dopo l'esecuzione del gestore, l'azione torna ad essere quella di default (il gestore deve quindi installare di nuovo se stesso richiamando la `signal`)
 - segnali inviati possono andare perduti
- ⇒ deve essere usata la `sigaction` (più avanti)

- in BSD4.3+ e in LINUX si ottengono invece segnali *affidabili* perchè la `signal` è implementata attraverso la `sigaction` (più avanti)

Sis.Op. A - UNIX - Programmazione di sistema

49

Segnali

Uso della `signal`

```
...
/* Il processo richiede che una eventuale notifica
di un segnale SIGINT venga gestita dalla propria
funzione sigint_handler */
signal(SIGINT, sigint_handler);
...
void sigint_handler(int signo)
{
/* In SVR4 e' necessario installare di nuovo il gestore
per la successiva notifica di SIGINT */
signal(SIGINT, sigint_handler);
...
}
```

Un gestore di segnale può decidere di far continuare il processo oppure di farlo terminare (`exit`)

Sis.Op. A - UNIX - Programmazione di sistema

50

Segnali

Esempio di gestione di segnale con signal

```
#include <signal.h>                                     Per terminare l'esecuzione di questo processo:
                                                         \ oppure kill -9 pidprocesso dallo shell
void catchint(int);

main()
{
    int i;

    /* La notifica di un segnale SIGINT deve avviare il gestore
    catchint: si dice comunemente che il processo "intercetta" o
    "aggrancia" il segnale */
    signal(SIGINT, catchint);

    while(1) /* Ciclo senza fine */
        for(i=0 ; i< 10000 ; i++)
            printf("i vale %d\n", i);
}

void catchint(int signo)
{
    signal(SIGINT, catchint);printf("catchint: signo=%d\n", signo);
    /* Non viene invocata la exit : ritorno al segnalato */
}

Sis.Op. A - UNIX - Programmazione di sistema
```

51

Segnali

Altre primitive per i segnali

```
#include <sys/types.h>
#include <signal.h>

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

Invia il segnale `sig` al processo `pid`

- il processo mittente e quello destinatario del segnale devono appartenere allo stesso utente
- solo root può inviare segnali a processi di altri utenti

La primitiva `kill` è asincrona (non bloccante):

⇒ il processo mittente prosegue immediatamente mentre al destinatario viene notificato il segnale

Segnali

Invio temporizzato di segnali

```
#include <unistd.h>
unsigned int alarm(unsigned int nseconds);
```

Dopo *nseconds* secondi il processo chiamante riceve un segnale SIGALRM inviato dal S.O (alarm(0) annulla il countdown)

Attesa di un segnale

```
#include <unistd.h>
int pause(void);
```

Il processo chiamante rimane bloccato fino a quando non viene eseguito un gestore di segnale per l'arrivo di un qualunque segnale

La pause ritorna sempre con -1 e con errno che vale EINTR

Sis.Op. A - UNIX - Programmazione di sistema

53

Segnali

Sospensione temporizzata

```
unsigned int sleep(unsigned int nseconds);
```

Sospende il processo chiamante per il tempo specificato

L'arrivo di un segnale mentre il processo è sospeso interrompe l'attesa :

- in questo caso `sleep` ritorna con -1 e con `errno` che vale EINTR

Alcune implementazioni della `sleep` usano SIGALRM per cui non è possibile utilizzare contemporaneamente `sleep` e `alarm`

E' preferibile l'utilizzo della `nanosleep` che in caso di interruzione restituisce l'intervallo di attesa residuo

Segnali

nanosleep e gettimeofday

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Sospende il processo chiamante per l'intervallo temporale specificato nella struttura req :

```
struct timespec {          time_t tv_sec;          /* seconds */
                          long tv_nsec;         /* nanoseconds */
};
```

In uscita nella struttura rem è riportato il tempo che resta da attendere dopo l'uscita anticipata della primitiva a causa della notifica di un segnale.

Il processo può quindi richiamarla indicando il tempo residuo di attesa.

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

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Permette di conoscere l'ora e la data correnti come numero di secondi dal 1 gennaio 1970.

Sis.Op. A - UNIX - Programmazione di sistema

55

Segnali

Esempio di interazione tra processi mediante segnali inaffidabili

```
#include <signal.h>
#include <unistd.h>

void catcher(int signo)
{
    static int ntimes = 0;
    printf("Processo %d ricevuto #%d volte\n", getpid(), ++ntimes);
}

int main()
{
    int pid, ppid;
    signal(SIGUSR1, catcher); /* il figlio erediterà questa politica di
                               gestione del segnale SIGUSR1 */

    if ((pid=fork()) < 0)
    {
        perror("fork error");
        exit(1);
    }
    else
    if (pid == 0)
    {
        /* Processo figlio - continua ... */
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

56

Segnali

```
/* ... continua processo figlio */
ppid = getpid();
printf("figlio: mio padre e' %d\n", ppid);
for (;;)
{
    sleep(1);
    kill(ppid, SIGUSR1);
    pause();
}
else
{
    /* Processo padre */
    printf("padre: mio figlio e' %d\n", pid);
    for (;;)
    {
        pause();
        sleep(1);
        kill(pid, SIGUSR1);
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

57

Segnali

Problemi con i segnali inaffidabili (*unreliable*)

1) Reset del gestore dopo una notifica del segnale (SVR4)

```
...
signal(SIGINT, sig_int);
...
void sig_int()
{
    <= = = =====
    signal(SIGINT, sig_int);
    ...
}
```

⇒ Esiste una finestra temporale in cui la notifica di un secondo segnale fa terminare il processo (N.B. prima di eseguire la nuova signal è tornata attiva la gestione di default !)

Sis.Op. A - UNIX - Programmazione di sistema

58

Segnali

Problemi con i segnali inaffidabili (*unreliable*)

2) Attesa di un segnale

```
...
int segnale_arrivato=0;
...
signal(SIGINT, sig_int);
...
if (! segnale_arrivato)
    <== == == =====
    pause(); /*Il processo attenderà un ulteriore
    segnale */
...
void sig_int ()
{
    signal(SIGINT, sig_int);
    segnale_arrivato= 1;
}
```

⇒ Esiste una finestra temporale in cui la notifica del segnale può provocare un deadlock

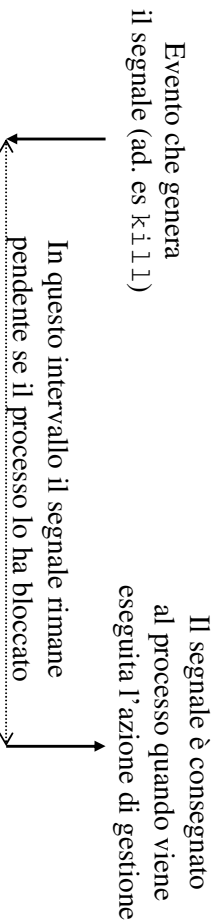
Sis.Op. A - UNIX - Programmazione di sistema

59

Segnali

La gestione affidabile dei segnali

Un processo può bloccare temporaneamente la consegna di un segnale (non il `SIGKILL`) che rimane pendente



Il segnale rimane pendente fino a che:

- il processo lo sblocca
- oppure
- il processo sceglie di ignorare quel segnale

⇒ un processo può cambiare l'azione da eseguire prima della consegna del segnale

Sis.Op. A - UNIX - Programmazione di sistema

60

Segnali

`sigset_t` è un nuovo tipo di dato che rappresenta un insieme di segnali (*signal set*)

Funzioni di utilità per la manipolazione dei *signal set*

```
#include <signal.h>
int sigemptyset (sigset_t *set);           azzera / rende vuoto un sigset
int sigfillset (sigset_t *set);           mette tutti i segnali nel sigset
int sigaddset (sigset_t *set, int signo); aggiunge un segnale al sigset
int sigdelset (sigset_t *set, int signo); rimuove un segnale dal sigset
int sigismember (sigset_t *set, int signo); valuta se quel segnale è presente nel sigset
```

Sis.Op. A - UNIX - Programmazione di sistema

61

Segnali

Un processo può esaminare e/o modificare la propria *signal mask* che è l'insieme dei segnali che sta attualmente bloccando

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

dove *how* può valere:

- SIG_BLOCK** ⇒ la nuova *signal mask* diventa l'OR binario di quella corrente con quella specificata da *set*
- SIG_UNBLOCK** ⇒ i segnali indicati da *set* sono rimossi dalla *signal mask*
- SIG_SETMASK** ⇒ la nuova *signal mask* diventa quella specificata da *set*

se *oset* è diverso da `NULL` la *signal mask* precedente viene restituita in *oset*

```
int sigpending(sigset_t *set);
restituisce in set il sottoinsieme dei segnali bloccati che sono
```

attualmente pendenti

Sis.Op. A - UNIX - Programmazione di sistema

62

Segnali

La `sigaction` è la primitiva fondamentale per la gestione dei segnali affidabili

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
const struct sigaction *oact);
```

Permette di esaminare e/o modificare l'azione associata ad un segnale

- `signo` identifica il segnale del quale si vuole esaminare e/o modificare l'azione
- `se act` non è NULL si modifica l'azione
- `se oact` non è NULL viene restituita la precedente azione

Sis.Op. A - UNIX - Programmazione di sistema

63

Segnali

Definizione della struttura `sigaction`

```
struct sigaction {
void (*sa_handler)(); /* indirizzo del gestore o SIG_IGN o
SIG_DFL */
void (*sa_sigaction)(int, siginfo_t *, void *);
/* indirizzo del gestore che riceve informazioni aggiuntive
sul segnale ricevuto (utilizzato al posto di sa_handler se
sa_flags contiene SA_SIGINFO) */
sigset_t sa_mask; /* segnali aggiuntivi da
bloccare prima dell'esecuzione del gestore */
int sa_flags; /* opzioni aggiuntive */
};
```

Notifiche (eventualmente multiple) dello stesso segnale durante l'esecuzione dell'azione sono bloccate fino al termine del gestore (a meno che `sa_flags` valga `SA_NODEFER`) quando ne viene notificata comunque una sola

Con la `sigaction` l'azione rimane permanentemente installata fino a quando non viene modificata

Sis.Op. A - UNIX - Programmazione di sistema

64

Segnali

Attesa di un segnale con la gestione affidabile

```
int sigsuspend(const sigset_t *sigmask)
```

permette **l'attivazione** della *signal mask* specificata (sbloccando alcuni o tutti gli eventuali segnali pendenti) e **l'attesa di un qualunque segnale** in modo **atomico**:

- se il segnale è pendente viene immediatamente eseguita l'azione corrispondente: successivamente, se non viene invocata la `exit`, la `sigsuspend` ritorna ;
- se il segnale non è pendente la `sigsuspend` attende fino alla notifica di un qualunque segnale che causa l'immediata esecuzione dell'azione corrispondente: se non viene invocata la `exit` la `sigsuspend` ritorna ;

`sigsuspend` ritorna sempre -1 con `errno` che vale `EINTR`

Sis.Op. A - UNIX - Programmazione di sistema

65

Segnali

L'uso di `sigsuspend` è indispensabile per un processo che intende sospendersi in attesa di uno specifico segnale (gestione sincrona) : inizialmente va bloccato il segnale di interesse, per poi sbloccarlo e attenderlo atomicamente con la `sigsuspend` quando necessario

```
...
sigaction(...);
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
/* Da qui il segnale SIGINT e' bloccato per il processo */
...
/* Sblocco di tutti i segnali pendenti
e attesa di un qualunque segnale */
sigsuspend(&zeromask);
```

```
sigprocmask(SIG_SETMASK, &oldmask, NULL);
/* sigsuspend rimette la maschera attiva prima
della sua chiamata: se necessario va rimessa
quella ancora precedente (oldmask) */
```

••• Sis.Op. A - UNIX - Programmazione di sistema

66

Segnali

Esempio di interazione tra processi mediante segnali affidabili

```
#include <signal.h>
#include <unistd.h>

void catcher(int signo)
{
    static int ntimes = 0;
    printf("Processo %d: SIGUSR1 ricevuto #%d volte\n", getpid(), ++ntimes);
}

int main()
{
    int pid, ppid;
    struct sigaction sig, osig;
    sigset_t sigmask, oldmask, zeromask;

    sig.sa_handler= catcher;
    sigemptyset( &sig.sa_mask);
    sig.sa_flags= 0;

    sigemptyset( &zeromask);

    sigemptyset( &sigmask);
    sigaddset( &sigmask, SIGUSR1);
    sigprocmask(SIG_BLOCK, &sigmask, &oldmask);

    sigaction(SIGUSR1, &sig, &osig); /* il figlio la erediterà */
}
```

```
/* Continua */
Sis.Op.A - UNIX - Programmazione di sistema
```

67

Segnali

```
/* ..continua */
if ((pid=fork()) < 0) {
    perror("fork error");
    exit(1);
}
else
    if (pid == 0) {
        /* Processo figlio */
        ppid = getppid();
        printf("figlio: mio padre e' %d\n", ppid);
        while(1) {
            sleep(1);
            kill(ppid, SIGUSR1);
            /* Sblocca il segnale SIGUSR1 e lo attende */
            sigsuspend(&zeromask);
        }
    }
    else {
        /* Processo padre */
        printf("padre: mio figlio e' %d\n", pid);
        while(1) {
            /* Sblocca il segnale SIGUSR1 e lo attende */
            sigsuspend(&zeromask);
            kill(pid, SIGUSR1);
        }
    }
}
```

Sis.Op.A - UNIX - Programmazione di sistema

68

Segnali

Implementazione della `signal` mediante la `sigaction`

```
#include <signal.h>
typedef void Sigfunc(int);
Sigfunc *signal(int signo, Sigfunc *func)
{
    struct sigaction    act,oact;
    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags= 0;

    /* Il segnale SIGALRM viene normalmente utilizzato per avere un
    timeout sulle primitive bloccanti (read, connect, ...) */
    if(signo == SIGALRM)
    #ifdef SA_INTERRUPT /* in SunOS si avrebbe di default il RESTART */
        act.sa_flags |= SA_INTERRUPT;
    #endif
    } else {
    #ifdef SA_RESTART /* in SVR4/4.3+BSD di default niente RESTART */
        act.sa_flags |= SA_RESTART;
    #endif
    }
    if(sigaction(signo, &act, &oact) < 0 ) return (SIG_ERR);
    return (oact.sa_handler);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

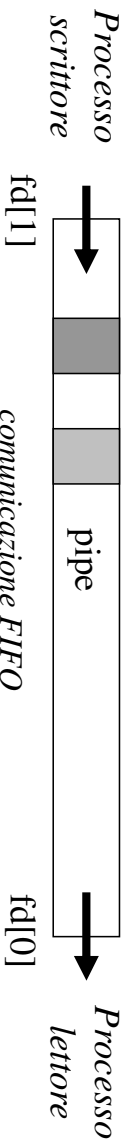
69

Pipe

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

Le pipe sono canali di comunicazione unidirezionali che costituiscono un primo strumento di comunicazione (con diverse limitazioni), basato sullo scambio di messaggi, tra processi UNIX

La creazione di una *pipe* mediante la primitiva omonima restituisce in `fd` due descrittori: `fd[0]` per la lettura e `fd[1]` per la scrittura



Letture e scrittura sulla *pipe* sono ottenute mediante le normali primitive `read` e `write`

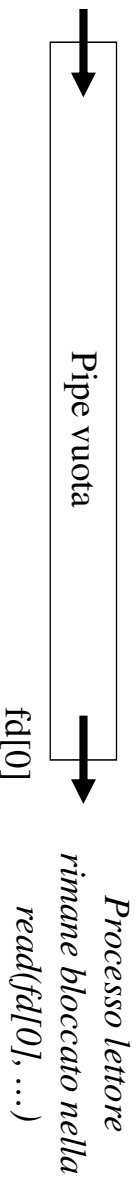
Sis.Op. A - UNIX - Programmazione di sistema

70

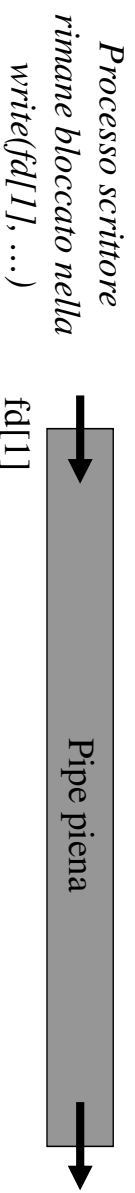
Pipe

L'accesso alle pipe è regolato da un meccanismo di sincronizzazione:

- la lettura da una pipe da parte di un processo è bloccante se la pipe è vuota (in attesa che arrivino i dati)



- la scrittura su una pipe da parte di un processo è bloccante se la pipe è piena



Sis.Op. A - UNIX - Programmazione di sistema

71

Pipe

La sincronizzazione è legata alla implementazione delle pipe:

ad ogni pipe viene associato un buffer circolare nel kernel (e quindi inaccessibile direttamente ai processi) di dimensione prefissata (ad es. 4 Kb)

- la lettura da una pipe vuota è bloccante per proteggere dall' underflow del buffer
- la scrittura su una pipe piena è bloccante per proteggere dall' overflow del buffer

Pipe

Le pipe sono anche dette pipe anonime (*unnamed pipe*) perché non sono associate ad alcun nome nel File System:

⇒ solo i processi che possiedono i descrittori possono comunicare attraverso una pipe

La tabella dei file aperti di un processo (contenente i descrittori della pipe) viene duplicata per i processi figli:

⇒ la comunicazione attraverso una pipe anonima è quindi possibile solo per processi in relazione di parentela che condividono un progenitore

Sis.Op. A - UNIX - Programmazione di sistema

73

Pipe

Esempio di comunicazione su pipe

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

#define N_MESSAGGI 10

int main()
{
    int pid, j,k, piped[2];

    /* Apre la pipe creando due file descriptor,
       uno per la lettura e l'altro per la scrittura
       (vengono memorizzati nei due elementi dell'array piped[]) */
    if (pipe(piped) < 0)
        exit(1);

    if ((pid = fork()) < 0)
        exit(2);
    else if (pid == 0) /* Il figlio eredita una copia di piped[] */
    {
        /* Il figlio e' il lettore dalla pipe: piped[1] non gli serve */
        close(piped[1]);

        /* Continua ... */
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

74

Pipe

```
/* ... continua */

for (j = 1; j <= N_MESSAGGI; j++)
{
    read(piped[0], &k, sizeof (int));
    printf("Figlio: ho letto dalla pipe il numero %d\n", k);
}
exit(0);
}
else {
    /* Processo padre */
    /* Il padre e' scrittore sulla pipe: piped[0] non gli serve */
    close(piped[0]);
    for (j = 1; j <= N_MESSAGGI; j++)
    {
        write(piped[1], &j, sizeof (int));
    }
    wait(NULL);
    exit(0);
}
}
```

Sis.Op. A - UNIX - Programmazione di sistema

75

Pipe

Implementazione del *ping* di comandi

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

int join(char* com1[], char *com2[])
{
    int status, pid;
    int piped[2];

    switch (Fork())
    {
        case -1: /* errore */ return 1;
        case 0: /* figlio */ break; /* esce dal case */
        default: /* padre: attende il figlio */ wait(&status); return
            status;
    }

    /* il figlio crea la pipe e un proprio figlio (nipote del primo
    processo)*/
    if (pipe(piped) < 0)
        return 2;
    if ((pid = fork()) < 0)
        return 3;
    else if (pid == 0)
        { /* Nipote: continua ... */
```

Sis.Op. A - UNIX - Programmazione di sistema

76

Pipe

```
/* ... continua il nipote */
/* Lo stdin corrente non interessa a com2 (lettore): chiude il
descrittore 0 */
close(0);
/* Si richiede un nuovo descrittore per la lettura dalla pipe:
il primo descrittore libero e' lo 0 (stdin) che viene
associato alla lettura dalla pipe */
dup(piped[0]);

/* Gli altri descrittori delle pipe non servono piu':
il lettore usa stdin per leggere dalla pipe */
close(piped[0]);
close(piped[1]);

/* Esecuzione del comando 2 (che ha la pipe come stdin) */
execvp(com2[0], com2);
error("exec com2"); return 4;
}
else { /* Figlio */
/* Lo stdout corrente non interessa a com1 (scrittore): chiude
il descrittore 1 */
close(1);
/* Si richiede un nuovo descrittore per la scrittura sulla pipe:
il primo descrittore libero e' 1 (stdout) che viene
associato alla scrittura sulla pipe */
dup(piped[1]); /* Figlio continua ... */
}
```

Sis.Op. A - UNIX - Programmazione di sistema

77

Pipe

```
/* ... continua figlio */
/* Gli altri descrittori delle pipe non servono piu':
lo scrittore usa stdout per scrivere sulla pipe */
close(piped[0]);
close(piped[1]);

/* Esecuzione del comando 1 (che ha la pipe come stdout) */
execvp(com1[0], com1);
error("exec com1"); return 5;
}
}

int main(int argc, char **argv)
{
int integri, i, j;
char *temp1[10], *temp2[10];

/* si devono fornire nella linea di comando due comandi distinti
runpiping cmd1 [arg11...arg1n] | cmd2 [arg21...arg2n]

utilizzando \l oppure '|' o "|" per indicare il piping : occorre
evitare che il simbolo "|" venga direttamente interpretato dallo shell
come una pipe) */
/* main continua ... */
```

Sis.Op. A - UNIX - Programmazione di sistema

78

Pipe

```
/* ... continua */

if (argc > 2) {
    for (i = 1; i < argc) && (strcmp(argv[i], "|") != 0); i++)
        temp1[i-1] = argv[i];
    temp1[i-1] = (char *)0;
    i++;
    for (j = 1; i < argc; i++, j++)
        temp2[j-1]=argv[i];
    temp2[j-1] = (char *)0;
}
else {
    printf("errore");
    exit(-2);
}
integi = join(temp1, temp2);
exit(integi);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

79

Pipe

Altri usi delle pipe

- Uno scrittore con più lettori potenziali

Poiché la read è atomica, un lettore q caso riesce a leggere, l'altro si blocca



read(&M) ⇒ M

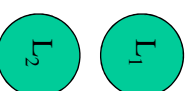


- Uno scrittore vuole scrivere ad un determinato lettore



```
read(&M) ;
while(M.PID != PID_L1)
{ write(&M);read(&M);}
```

read () **bloccante**



Il destinatario può essere indicato nel messaggio (↑) oppure inviando un segnale (↓)



```
sigsuspend() ;
read(&M) ⇒ M
```



```
Kill(PID_L1, SIGUSR1) ;
write(&M) // o viceversa
```

```
sigsuspend() ;bloccante
read (&M)
```

Sis.Op. A - UNIX - Programmazione di sistema

80

Esercizi

Uno schema generale di soluzione per gli esercizi UNIX

5 fasi principali con un preciso ordinamento:

1 Controllo degli argomenti di invocazione

2 Selezione della politica di gestione dei segnali

- per il processo padre (main) e/o per i figli che la ereditano

3 Creazione pipe (ed eventuale inizializzazione del contenuto)

- le pipe saranno così accessibili ai figli e ai discendenti

4 Creazione dei processi figli

5 Esecuzione dei figli confinata in specifiche funzioni

- nel caso alcuni processi abbiano lo stesso comportamento si ottiene una maggiore compattezza del codice:
⇒ le funzioni vanno scritte in modo parametrico rispetto ai propri argomenti e al PID del processo corrente

Sis.Op. A - UNIX - Programmazione di sistema

81

Esercizi

Esercizio n. 1

```
/******  
Un processo padre crea N (N numero pari) processi figli.  
Ciascun processo figlio Pi e' identificato da una variabile intera i  
(i=0,1,2,3...,N-1). Due casi:  
  
1. Se argv[1] è uguale ad 'a' ogni processo figlio Pi con i pari  
   manda un segnale (SIGUSR1) al processo i+1;  
  
2. Se argv[1] è uguale a 'b' ogni processo figlio Pi con i < N/2  
   manda un segnale (SIGUSR1) al processo i + N/2.  
*****/  
  
#include <stdio.h>  
#include <ctype.h>  
#include <signal.h>  
  
#define N2 5  
#define N N2*2  
  
int pg[2];  
int tabpid[N];  
char arg1;
```

/* Continua */
Sis.Op. A - UNIX - Programmazione di sistema

82

Esercizi

Esercizio n. 1 (cont.)

```
void handler(int signo)
{
    printf("Sono il processo %d e ho ricevuto il segnale %d\n",
        getpid(), signo);
}

/* Funzione eseguita da ciascun figlio: ne definisce il
comportamento a regime */
int body_proc(int id)
{
    printf("Sono il processo %d con id=%d\n", getpid(), id);

    if (argl=='a')
        /* % è l'operatore modulo, il resto della divisione intera */
        if (id % 2) pause(); /* id dispari */
        else
            /* id pari */
            read(pg[0], tabpid, sizeof tabpid);
            write(pg[1], tabpid, sizeof tabpid);
            kill(tabpid[id+1], SIGUSR1);
        }
    }
else
    {
        /* Continua ... */
        Sis.Op.A - UNIX - Programmazione di sistema
    }
}
```

83

Utilizzo di una pipe come deposito di informazioni condiviso tra i processi: l'atomicità della read garantisce un accesso mutuamente esclusivo al contenuto

Il processo che completa la read consuma il messaggio dalla pipe che va quindi reintegrato con una write

Esercizi

Esercizio n. 1 (cont.)

```
if (id >= N/2) pause();
else
    {
        read(pg[0], tabpid, sizeof tabpid);
        write(pg[1], tabpid, sizeof tabpid);
        kill(tabpid[id+N/2], SIGUSR1);
    }
}
return(0);
}

main (int argc, char* argv[])
{
    int i, status;

    /* 1) Controllo argomenti */

    if(argc != 2)
        {
            fprintf(stderr, "Uso:  %s a\n(oppure)\n%s b \n", argv[0], argv[0]);
            exit(-1);
        }
    /* Continua ... */
}
```

Esercizi

Esercizio n. 1 (cont.)

```
argv= argv[1][0]; /* primo carattere del secondo argomento */
/* 2) Gestione dei segnali che i figli erediteranno */
signal(SIGUSR1,handler);
/* 3) Creazione della pipe di comunicazione tra padre e figli */
if (pipe(pg)<0)
{
    perror("creazione pipe"); exit(-1);
}
/* 4) Creazione dei processi figli */
for (i=0;i<N;i++)
{
    if ((tabpid[i]=fork())<0)
    {
        perror("fork");
        exit(-1);
    }
    else
    if (tabpid[i]==0)
    { /* 5) Esecuzione dei figli confinata all'interno di una
        funzione specifica */
        status= body_proc(i);
        /* Continua ... */
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

85

Esercizi

Esercizio n. 1 (cont.)

```
/* 5b) Terminazione dei figli: si evita che i figli
rimanendo all'interno del ciclo for generino a loro
volta altri processi
*/
    exit(status);
}
}
/* Il padre pone la tabella (che contiene tutti gli N pid dei figli)
nella pipe */
printf("Sono il padre e scrivo sulla pipe la tabella dei pid\n");
write(pg[1],tabpid,sizeof tabpid);
```

La write permette l'inizializzazione del contenuto della pipe da parte del processo padre

```
    exit(0);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

86

Esercizi

Esercizio n. 2

```
*****
Si progetti in ambiente Unix/C l'interazione di tre
processi P1,P2 e Pa mediante segnali e una pipe pa :
- P1 e P2 inviano segnali SIGUSR1(P1)/SIGUSR2(P2) a Pa,
  attendendo per un intervallo di durata casuale (massimo 5 secondi)
  tra un segnale e l'altro;
- il numero di segnali che ciascun processo deve inviare viene
  determinato dall'unico argomento di invocazione del programma main;
- al termine della spedizione dei segnali, ciascun processo P1/P2
  invia mediante la pipe pa un messaggio a Pa contenente il proprio pid
  e il numero di segnali spediti;
- alla ricezione dei due messaggi Pa deve visualizzare il numero di
  segnali spediti e il numero di segnali effettivamente ricevuti.

Si utilizzino le primitive per la gestione affidabile dei segnali.

*****
```

Sis.Op. A - UNIX - Programmazione di sistema

87

Esercizi

Esercizio n. 2

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

int pa[2];
int numsegnali;
int sigusr1cnt;
int sigusr2cnt;

/* contatore segnali SIGUSR1 ricevuti */
/* contatore segnali SIGUSR2 ricevuti */

/* Gestore segnale SIGUSR1 per il padre Pa */
void usr1_handler()
{
  ++sigusr1cnt;
  printf("%d riceve SIGUSR1: %d\n", getpid(), sigusr1cnt);
}

/* Gestore segnale SIGUSR2 per il padre Pa */
void usr2_handler()
{
  ++sigusr2cnt;
  printf("%d riceve SIGUSR2: %d\n", getpid(), sigusr2cnt);
}

/* Continua ... */
```

Sis.Op. A - UNIX - Programmazione di sistema

88

Esercizi

Esercizio n. 2 (cont.)

```
main(int argc, char *argv[])
{
    int pid;
    struct sigaction act;

    /* 1) Controllo argomenti */
    if(argc !=2)
    {
        fprintf(stderr, "Uso: %s numsegnali\n", argv[0]);
        exit(-1);
    }
    numsegnali=atoi(argv[1]);
    if(numsegnali <=0 || numsegnali>100) /* Sanity checking */
        numsegnali=10;

    /* 2) Gestione dei segnali del processo Pa */
    act.sa_handler= usr1_handler; /* Gestore del SIGUSR1 */
    sigemptyset (&act.sa_mask);
    sigaddset (&act.sa_mask, SIGUSR2); /* Blocca SIGUSR2 nel gestore */
    act.sa_flags= SA_RESTART; /* Restart automatico di una primitiva di
    lettura interrotta dal segnale SIGUSR1
    (ad.es. la read dalla pipe pa) */

    sigaction(SIGUSR1, &act, NULL);
    /* Continua ... */
}

Sis.Op. A - UNIX - Programmazione di sistema
```

89

Esercizi

Esercizio n. 2 (cont.)

```
act.sa_handler= usr2_handler; /* Gestore del SIGUSR2 */
sigemptyset (&act.sa_mask);
sigaddset (&act.sa_mask, SIGUSR1); /* Blocca SIGUSR1 nel gestore */
act.sa_flags= SA_RESTART; /* Restart automatico di una primitiva di
lettura interrotta dal segnale SIGUSR2
(ad.es. la read dalla pipe pa) */

sigaction(SIGUSR2, &act, NULL);

/* 3) Creazione della pipe di comunicazione tra figli e padre */
if(pipe(pa) <0)
{ perror("creazione pipe"); exit(-1);
}

/* 4) Creazione dei processi figli */
if((pid=fork())==0)
{ /* 5) Esecuzione dei figli confinata all'interno di una
funzione specifica */
    body_figlio(SIGUSR1);
    exit(0);
}
else
/* Continua ... */
```

Esercizi

Esercizio n. 2 (cont.)

```
if ((pid=fork())==0)
    { /* 5) Esecuzione dei figli confinata all'interno di una
      funzione specifica */
      body_figlio(SIGUSR2);
      exit(0);
    }
else
    body_padre();
}
/* Funzione eseguita da ciascun figlio: ne definisce il comportamento
a regime */
body_figlio(int signo) /* segnale che questo figlio manda al padre */
{
    int i,mesg[2];
    for (i=0; i<numsegnali; i++)
        {
            kill(getppid(),signo);
            printf("Processo %d invia %s\n",getpid(),
                (signo==SIGUSR1)? "SIGUSR1":"SIGUSR2");
            sleep(1+rand()%5); /* Attesa tra 1 e 5 secondi */
        }
}
/* Continua ... */
Sis.Op. A - UNIX - Programmazione di sistema
```

91

Esercizi

Esercizio n. 2 (cont.)

```
/* Preparazione messaggio da inviare sulla pipe pa al padre */
mesg[0]= getpid(); /* indicazione del mittente */
mesg[1]= numsegnali; /* messaggio */
write(pa[1], mesg, sizeof(mesg));
}
body_padre()
{
    int mesga[2],mesgb[2];
read(pa[0],mesga,sizeof(mesga));
read(pa[0],mesgb,sizeof(mesgb));
printf("Processo %d ha mandato %d segnali\n",mesga[0],mesga[1]);
printf("Processo %d ha mandato %d segnali\n",mesgb[0],mesgb[1]);
printf("Ricevuti %d SIGUSR1 e %d SIGUSR2\n\n", sigusr1cnt, sigusr2cnt);
}
}
```

Esercizi

Esercizio n. 3

```

/*****
Si progetti in ambiente Unix/C la seguente interazione di processi:

- il sistema consiste di 3 processi: un processo padre (P1) che
  provvede alla creazione di 2 processi figli (P11 e P12) e di due pipe
  pa e pb ;

- P1 provvede a generare la sequenza dei primi N interi,
  scrivendoli nella pipe pa;

- P11 inizialmente preleva i numeri pari da pa e li riscrive in pb ;

- P12 preleva l (con l > N/4 ) interi da pb e li scrive
  su un primo file;

- dopo aver letto l messaggi, P12 invia un segnale
  SIGUSR1 a P11 per effetto del quale esso passa a prelevare
  i numeri dispari da pa e a scriverli in pb ;

- P12 preleva i rimanenti interi da pb e li scrive su un secondo file.
*****/
```

Sis.Op. A - UNIX - Programmazione di sistema

93

Esercizi

Esercizio n. 3 (cont.)

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <ctype.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ioctl.h>

#include <stdio.h>

#define FILENAME1 "prova-1.txt"
#define FILENAME2 "prova-2.txt"

#define PARI 0
#define DISPARI 1

pid_t pid11, pid12;
int pa[2],pb[2];
int n,l;

int pari_o_dispari=0; /* 0 PARI --- 1 DISPARI */
/* Continua ... */

Sis.Op. A - UNIX - Programmazione di sistema
```

94

Esercizi

Esercizio n. 3 (cont.)

```
void sigusr1_handler(int signo)
{
    if(pari_o_dispari==PARI)
        pari_o_dispari=DISPARI ;
    else
        pari_o_dispari=PARI ;

    printf("Il processo p11 ha ricevuto SIGUSR1: ora sono selezionati
i %s\n", (pari_o_dispari==PARI)? "pari":"dispari" );
}

int body_p11()
{
    int i,value;

    printf("Sono il processo p11\n");
    pari_o_dispari=0;

    while(1) /* Il processo viene terminato con il SIGKILL da p12 */
    {
        read(pa[0], &value, sizeof(value));
        if(value % 2) /* dispari */
            if(pari_o_dispari != PARI)
                /* Continua ... */
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

95

Esercizi

Esercizio n. 3 (cont.)

```
printf("%d dispari ->\n",value);
write(pb[1], &value, sizeof(value));
/* inviati a p12 */
}
else /* pari */
{
    if(pari_o_dispari == PARI)
    {
        printf("%d pari ->\n", value);
        write(pb[1], &value, sizeof(value));
        /* inviati a p12 */
    }
    sleep(1);
}
}

int body_p12()
{
    int i,fd1,fd2;
    int avaiable, timeout;
    char number[10];

    printf("Sono il processo p12\n");

    if ((fd1=open(FILENAME1, O_WRONLY | O_CREAT |O_TRUNC, 0644))<0)
        /* Continua ... */
}
```

Sis.Op. A - UNIX - Programmazione di sistema

96

Esercizi

Esercizio n. 3 (cont.)

```
    perror("open");
    exit(-1);
}
if ((fd2=open(FILENAME2, O_WRONLY | O_CREAT | O_TRUNC, 0644))<0)
{
    perror("open");
    exit(-1);
}
for (i=0;i<1;i++)
{
    read(pb[0], &i, sizeof(i));
    printf("->%d in %s\n", i, FILENAME1);
    sprintf(number, "%d\n", i); /*Per scrivere numeri come testo*/
    write(fd1, number, strlen(number));
    sleep(1);
}
kill(pid1, SIGUSR1);

timeout =0 ;
while(timeout < 4)
{
    ioctl(pb[0], FIONREAD, &available);
    if(available >0)
        /* Continua ... */
}
Sis.Op. A - UNIX - Programmazione di sistema

97
```

Esercizi

Esercizio n. 3 (cont.)

```
    timeout=0;
    read(pb[0], &i, sizeof(i));
    printf("->%d in %s\n", i, FILENAME2);
    sprintf(number, "%d\n", i); /*Per scrivere numeri come testo*/
    write(fd2, number, strlen(number));
}
else
    timeout++;
    sleep(1);
}
printf("Timeout in lettura per il processo p12 che termina p11\n");
kill(pid1, SIGKILL);
return(0);
}
main (int argc, char* argv[])
{
    int i, status;
    struct sigaction act;

    /* 1) Controllo argomenti */
    if(argc != 3)
        /* Continua ... */
}
Sis.Op. A - UNIX - Programmazione di sistema

98
```

Esercizi

Esercizio n. 3 (cont.)

```
    fprintf(stderr, "Uso:  %s N l\n", argv[0]);
    exit(-1);
}

n = atoi(argv[1]);
l = atoi(argv[2]);

if (l <= n/4)
{
    fprintf(stderr, "l deve essere maggiore di N/4\n");
    exit(-2);
}

/* 2) Gestione dei segnali che i figli erediteranno */
act.sa_handler= sigusr1_handler;
sigemptyset (&act.sa_mask);
act.sa_flags= SA_RESTART; /* Per la read di p11 - vedi es. n. 2 */
if (sigaction (SIGUSR1, &act, NULL) < 0)
{
    perror ("sigaction");
    exit (-2);
}

/* 3) Creazione delle pipe di comunicazione */
if (pipe (pa) < 0)
{ /* Continua ... */
    Sis.Op.A - UNIX - Programmazione di sistema
}

99
```

Esercizi

Esercizio n. 3 (cont.)

```
    perror ("pipe error");
    exit (-3);
}
if (pipe (pb) < 0)
{
    perror ("pipe error");
    exit (-4);
}

/* 4) Creazione dei processi figli */
if ((pid11=fork ()) < 0)
{
    perror ("fork error");
    exit (-5);
}
else
if (pid11==0)
{ /* 5) Esecuzione del figlio in una funzione specifica */
    status= body_p11 ();
    exit(status);
}

if ((pid12=fork ()) < 0)
{
    perror ("fork error");
    /* Continua ... */
}

Sis.Op.A - UNIX - Programmazione di sistema

100
```

Esercizi

Esercizio n. 3 (cont.)

```
        exit(-5);
    }
    else
        if (pid12==0)
            { /* 5) Eseecuzione del figlio in una funzione specifica */
              status= body_pl2();
              exit(status);
            }
        /* Padre */
        for (i=0;i<n;i++)
            write(pa[i],&i, sizeof(i));
        /* Attende la terminazione di entrambi i figli */
        wait(NULL);
        wait(NULL);
        exit(0);
    }
```

Sis.Op. A - UNIX - Programmazione di sistema

101

FIFO

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

int mkfifo ( const char *pathname, mode_t mode );
```

Sono anche dette pipe con nome (le pipe classiche sono anonime)

Viene creata una entità nel filesystem (`pathname`) accessibile anche da processi non in relazione di parentela

Vanno utilizzate le normali SysCall che si usano per file e pipe (`open`, `read/write`, `close`, ...)

Una FIFO deve essere aperta con `open` dopo che è stata creata con `mkfifo`

Sis.Op. A - UNIX - Programmazione di sistema

102

FIFO

Effetto del flag `O_NONBLOCK` :

- in sua assenza una open in sola lettura blocca il processo fino a quando un altro processo apre la FIFO in scrittura (lo stesso per la sola scrittura);
- se `O_NONBLOCK` viene specificato, l'apertura in sola lettura ritorna immediatamente ma quella in sola scrittura ritorna con errore che vale `ENXIO` se non ci sono processi lettori

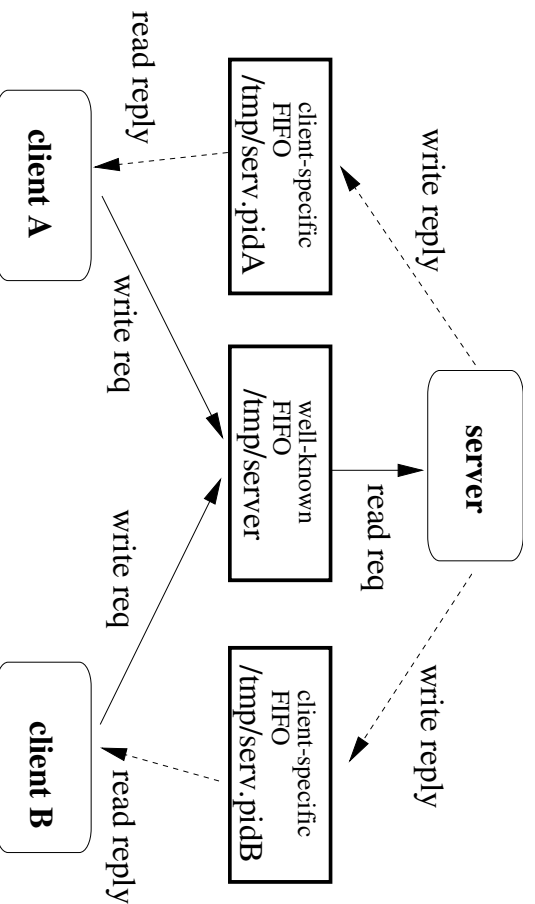
E' normale avere piu' scrittori su una FIFO (le `write` sono atomiche se riguardano meno di `PIPE_BUF` byte)

FIFO

Le FIFO sono adatte per applicazioni client-server in locale:

- il server apre una FIFO con un nome noto (ad es. `/tmp/server`)
- i client aprono la FIFO e scrivono le proprie richieste

Per le risposte il server utilizza una FIFO per ogni cliente



Socket

Una socket è un punto estremo di un canale di comunicazione accessibile mediante un file descriptor

Le socket costituiscono un fondamentale strumento di comunicazione, basato sullo scambio di messaggi, tra processi locali e/o remoti (sia UNIX che di altri sistemi operativi) :

- ⇒ vengono superate le limitazioni delle pipe e delle FIFO (comunicazione locale, con le pipe ristretta ai processi di uno stesso utente, discendenti di uno stesso avo)

Una socket va creata all'interno di un dominio di comunicazione che determina i protocolli utilizzati :

⇒ le socket sono l'elemento di base per la programmazione di applicativi e servizi di rete (ad es. utilizzando i protocolli TCP/IP - Internet)

Socket

Una socket è un oggetto con un tipo, determinato dal sottoinsieme delle seguenti proprietà che quel tipo di socket garantisce:

- 1) **consegna ordinata dei messaggi** (l'ordine di ricezione dei messaggi è uguale all'ordine di trasmissione)
- 2) **consegna non duplicata** (lo stesso messaggio non può essere consegnato due volte)
- 3) **consegna affidabile** (i messaggi inviati non possono andare persi)
- 4) **preservamento dei confini dei messaggi** (i messaggi inviati non vengono frazionati nella comunicazione)
- 5) **supporto per i messaggi *out-of-band*** (messaggi prioritari che superano quelli ordinari nella coda di ricezione)
- 6) **comunicazione orientata alla connessione** (più avanti)

Le pipe (che non sono socket) garantiscono le proprietà 1, 2 e 3

Socket

Alcuni tipi predefiniti di socket

- **SOCK_STREAM**
orientata alla connessione, trasferisce byte stream con proprietà 1, 2, 3, 5, 6 ma non 4
- **SOCK_DGRAM**
trasferisce datagram con proprietà 4 ma non 1, 2, 3, 5, 6
- **SOCK_SEQPACKET**
trasferisce datagram con proprietà 1, 2, 3, 4, 5, 6 (non è implementata nel dominio di comunicazione Internet)

- **SOCK_RAW**
permette l'accesso diretto ai protocolli di rete sottostanti (ad es. Ethernet)

Sis.Op. A - UNIX - Programmazione di sistema

107

Socket

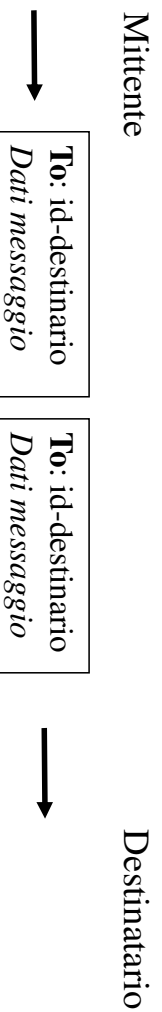
Le modalità di comunicazione *byte stream* e *datagram*

- **byte stream (SOCK_STREAM / protocollo TCP)**



Nella connessione il flusso di dati trasmesso è uguale a quello ricevuto (ma il protocollo può suddividere i messaggi i cui confini non sono quindi preservati)

- **datagram (SOCK_DGRAM / protocollo UDP)**



Ogni messaggio reca l'indicazione del destinatario ed è singolarmente inviato, trasferito e ricevuto

Sis.Op. A - UNIX - Programmazione di sistema

108

Socket

```
#include <sys/types.h>
#include <sys/socket.h>
```

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

Una socket viene creata nel dominio di comunicazione domain (detto anche *protocol family* o *address family*)

Domini principali:

- PF_UNIX dominio per una comunicazione locale
- PF_INET dominio per una comunicazione su TCP/IP (IPv4)
- PF_INET6 dominio per una comunicazione su TCP/IP (IPv6)

`type` indica il tipo di socket che si vuole creare (ad. es. SOCK_STREAM oppure SOCK_DGRAM); `protocol` indica lo specifico protocollo utilizzato tra quelli disponibili nel dominio (se ne esiste uno solo vale zero)

Viene restituito un descrittore da utilizzare sia per la lettura (ricezione) che la scrittura (invio) di messaggi

Sis.Op. A - UNIX - Programmazione di sistema

109

Socket

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t
  addrLen);
```

Una socket può venire legata ad un indirizzo (nome)

`bind` assegna un nome ad una socket per renderla designabile (indirizzabile) da parte di un processo intenzionato a comunicare con il processo che ha creato la socket

L'interfaccia è generica (`my_addr`, `addrLen`) in quanto i diversi domini di comunicazione prevedono indirizzi di forma diversa:

- PF_UNIX indirizzo = un percorso nel file system (ad es. /tmp.X11-unix/X0)
- PF_INET indirizzo = (indirizzo IP del nodo, numero porta)

Socket

Nel dominio PF_INET

indirizzo socket = (indirizzo IP del nodo, numero porta)

Ad esempio

indirizzo socket = (172.28.14.253, 22)

Indirizzo IP del nodo Porta assegnata
darkstar.cedi.unipr.it al servizio sshd

È l'indirizzo della socket utilizzata dal servizio sshd su darkstar

Se l'indirizzo specificato in una `bind` è già assegnato ad un'altra socket (la porta non è libera) la `bind` fallisce

Se il numero di porta contenuto nell'indirizzo specificato nella `bind` è zero, viene restituita una porta a caso tra quelle libere

- numeri di porta < 1024 riservati ai servizi di rete di sistema (root)
- numeri di porta >= 1024 utilizzabili liberamente dagli utenti

Sis.Op. A - UNIX - Programmazione di sistema

111

Socket

Porte riservate ai servizi di rete di sistema (/etc/services)

```
tcpmux 1/tcp # TCP port service multiplexer
echo 7/tcp
echo 7/udp
discard 9/tcp
discard 9/udp
sysstat 11/tcp
daytime 13/tcp
daytime 13/udp
netstat 15/tcp
gold 17/tcp
msp 18/tcp
msp 18/udp
chargen 19/tcp
chargen 19/udp
ftp-data 20/tcp
ftp 21/tcp
ftp 21/udp
fsp 21/udp
ssh 22/tcp
ssh 22/udp
telnet 23/tcp
# 24 - private
smtp 25/tcp
# 26 - unassigned
time 37/tcp
time 37/udp
rtp 39/udp
nameserver 42/tcp
whois 43/tcp
re-mail-ck 50/tcp
re-mail-ck 50/udp
domain 53/tcp
...

sink null
sink null
users

quote
# message send protocol
# message send protocol

tyst source
tyst source
# SSH Remote Login Protocol
# SSH Remote Login Protocol

fspd
mail
mail
timserver
timserver
resource
resource
# resource location
# IEN 116
nicname
# Remote Mail Checking Protocol
# Remote Mail Checking Protocol
nameserver # name-domain server
```

Sis.Op. A - UNIX - Programmazione di sistema

112

Socket

Nel dominio AF_INET le strutture dati utilizzate da bind e dalle altre primitive sono:

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    u_int16_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};
```

Network byte order corrisponde alla convenzione big-endian : i byte più significativi sono all'indirizzo più basso

⇒ utilizzare sempre la macro htons (numeroporta) che effettua (se necessario) la conversione

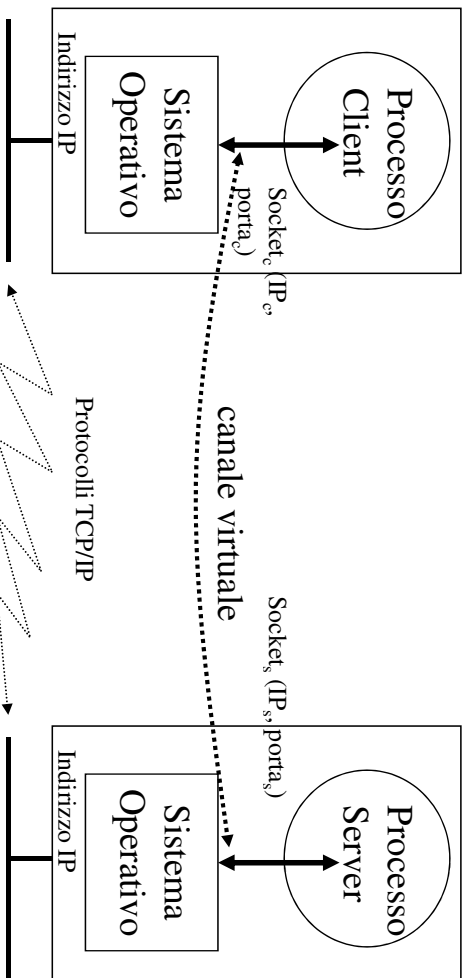
```
/* Internet address. */
struct in_addr {
    u_int32_t s_addr; /* address in network byte order */
};
```

L'indirizzo Internet può essere ricavato dal nome simbolico del nodo (ad es. darkstar.cedi.unipr.it) utilizzando la funzione gethostbyname

Socket

SOCK_STREAM nel dominio AF_INET

Prima di effettuare il trasferimento dati deve essere creata una connessione (protocollo TCP/IP)



Completata con successo la fase di creazione della connessione (socket "connessa") è sufficiente inoltrare i messaggi lungo la connessione perché raggiungano la destinazione

Socket

Creazione della connessione

Un **cliente** inizia una connessione sulla propria socket specificando l'indirizzo della socket del server:

```
connect (int cli_sockfd, ...);  
/* bloccante */
```

sincronizzazione

Il **server** dichiara al S.O. la sua disponibilità a ricevere connessioni sulla propria socket:

```
listen ( int serv_sockfd, ... ) ;  
/* non bloccante */
```

Il **server** attende richieste di connessioni sulla propria socket e riceve un nuovo descrittore (conn_sockfd) per ogni nuova connessione:

```
conn_sockfd=accept (int serv_sockfd,...);  
/* bloccante */
```

Su socket connesse

```
write (cli_sockfd, ... ) ;  
  
read (cli_sockfd, ... ) ;
```

```
o1 write (conn_sockfd, ... ) ;
```

Se la **read** ritorna 0 significa che la connessione è stata chiusa dal partner

115

Sis.Op. A - UNIX - Programmazione di sistema

Socket

```
#include <sys/types.h>  
#include <sys/socket.h>
```

```
int connect(int sockfd, const struct sockaddr *serv_addr,  
socklen_t addrlen);
```

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

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

backlog specifica la dimensione massima della coda delle richieste di connessione pendenti (non ancora accettate)

```
#include <sys/types.h>  
#include <sys/socket.h>
```

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

s è il descrittore della socket di controllo, in addr (se non è NULL viene memorizzato l'indirizzo del cliente che si è connesso). Un valore di uscita positivo identifica il descrittore della socket connessa che è utilizzata per comunicare con il cliente

Socket

Server concorrente

Normalmente un server su SOCK_STREAM (cfr. i servizi TCP, ad es. ftp) crea un nuovo server figlio per gestire una nuova connessione da un cliente mentre il server padre continua ad attendere nuove connessioni

```
do
{ /* Attesa di una connessione */
  if((msgsock= accept(sock,(struct sockaddr *)&client,(socklen_t *)&len))<0) {
    perror("accept"); exit(-1); }
  else {
    if(fork()==0) {
      /* Server figlio */
      printf("Serving connection from %s, port %d\n",
            inet_ntoa(client.sin_addr), ntohs(client.sin_port));
      close(sock); /* Non interessa la socket di controllo */
      myservice(msgsock); /* Servizio specifico del server attraverso
                           la server connessa */
      close(msgsock); /* La socket connessa può essere rimossa */
      exit(0);
    }
    else /* Server padre */
      close(msgsock); /* Non interessa la socket connessa : si
                       ritorna in accept */
  }
}
while(1);
```

Sis.Op. A - UNIX - Programmazione di sistema

117

Socket

Datagram

Non vi alcun stato di connessione (protocollo UDP di TCP/IP)

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
```

Invio di un messaggio con designazione esplicita del destinatario
(indirizzo specificato in to)

```
int recvfrom(int s, void *buf, size_t len, int flags, struct
sockaddr *from, socklen_t *fromlen);
```

Ricezione di un messaggio

L'indirizzo del mittente del messaggio viene posto in from (se diverso da
NULL)

Sis.Op. A - UNIX - Programmazione di sistema

118

Socket

Esempio n. 1

```

/*****
                                     Semplice interazione su socket di tipo STREAM
                                     S   E   R   V   E   R
Utilizzo:      server [numeroporta]
*****/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

#define DEFAULTPORT 1520
#define BYTES_NR 8192
#define MSG_NR 64
```

Sis.Op. A - UNIX - Programmazione di sistema

119

Socket

Esempio n. 1 (server cont.)

```

main(int argc, char *argv[])
{
    int sock,length;
    struct sockaddr_in server;
    int s,msgsock,rval;
    struct hostent *hp,*gethostbyname();
    char buf[BYTES_NR];
    int myport;

    if(argc == 2)
        myport = atoi(argv[1]);
    else
        myport = DEFAULTPORT;

    if((myport < 1024 && myport != 0)|| myport > 65535)
    {
        fprintf(stderr, "Il numero di porta puo essere 0 (per ottenere
una libera)\noppure deve essere compresa tra 1024 e 65535\n");
        exit(-1);
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

120

Socket

Esempio n. 1 (server cont.)

```
/* Crea la socket STREAM */
sock= socket (AF_INET, SOCK_STREAM, 0);
if(sock<0)
{
    perror("creazione stream socket");
    exit(1);
}

server.sin_family = AF_INET;
/* La socket viene legata a tutti gli indirizzi IP del server
(un indirizzo per ciascuna interfaccia di rete):
il server otterrà i pacchetti ricevuti su qualunque interfaccia.
Con INADDR_ANY non serve conoscere l'indirizzo IP del nodo su cui
esegue il server */
server.sin_addr.s_addr= INADDR_ANY;
server.sin_port = htons(myport);

if (bind(sock, (struct sockaddr *)&server, sizeof server)<0)
{
    perror("bind su stream socket");
    exit(1);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

121

Socket

Esempio n. 1 (server cont.)

```
/* Chiede conferma dell'indirizzo assegnato alla socket */
length= sizeof server;
if(getsockname(sock, (struct sockaddr *)&server, &length)<0)
{
    perror("getsockname"); exit(-1);
}

printf("Porta (della socket) del server =
      #d\n", ntohs(server.sin_port));

/* Il server e' pronto ad accettare connessioni */
if(listen(sock, 2) <0)
{
    perror("listen"); exit(-1);
}

do
{
    /* Attesa di una richiesta di connessione: l'indirizzo della socket
del mittente viene in questo caso ignorato (NULL) */
    if((msgsock= accept(sock, (struct sockaddr *)NULL, (int *)NULL)) < 0)
    {
        perror("accept");
        exit(-2);
    }
}
else
```

Sis.Op. A - UNIX - Programmazione di sistema

122

Socket

Esempio n. 1 (server cont.)

```
do
{
    s = 0;
    /* Ricezione del messaggio */
    do {
        /* Il messaggio può essere stato frammentato dal protocollo TCP :
        il numero di read potrà essere superiore al numeri di write.
        Occorre effettuare read multiple fino a quando saranno BYTES_NR
        i byte letti (che sono stati inviati con un' unica write) */

        if((rval = read(msgsock, &buf[s], sizeof buf))<0)
            {
                perror("read su stream message");
                exit(-3);
            }
            s+= rval;
            printf("r=%d) byte letti\n", rval,s);
        }
        while ((s!=BYTES_NR) && rval !=0);

        if(rval == 0)
            printf("Termine della connessione\n");
        else
```

Sis.Op. A - UNIX - Programmazione di sistema

123

Socket

Esempio n. 1 (server cont.)

```
/* Invio della risposta */
if((rval = write(msgsock,buf, sizeof buf))<0)
{
    perror("writing on stream socket");
    exit(-4);
}

printf("w=%d byte scritti\n",rval);
}

} while(rval !=0);
close (msgsock);
}

while(1); /* ^C o un altro segnale per terminare il server */

exit(0);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

124

Socket

Esempio n. 1

```
*****
```

Semplice interazione su socket di tipo STREAM

C L I E N T

Utilizzo: client nomeserver portaserver

```
*****
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

#define BYTES_NR      8192
#define MSG_NR       64
```

Sis.Op. A - UNIX - Programmazione di sistema

125

Socket

Esempio n. 1 (client cont.)

```
main(int argc, char *argv[])
{
    int      i,s,sock,rval;
    struct   sockadr_in server;
    struct   hostent *hp,*gethostbyname();
    char     buf[BYTES_NR];

    if(argc != 3)
    {
        fprintf(stderr, "Uso: %s nomeserver portaserver\n\n", argv[0]);
        exit(-1);
    }

    /* Crea una socket di tipo STREAM per il dominio TCP/IP */
    if((sock= socket(AF_INET, SOCK_STREAM, 0)) <0)
    {
        perror("creazione stream socket");
        exit(1);
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

126

Socket

Esempio n. 1 (client cont.)

```
/* Ottiene l'indirizzo IP del server */
server.sin_family= AF_INET;
hp= gethostbyname(argv[1]);

if (hp==0)
{
    fprintf(stderr, "%s: server sconosciuto", argv[1]);
    exit(2);
}

memcpy( (char *)&server.sin_addr, (char *)hp->h_addr , hp->h_length);

/* La porta è sulla linea di comando */
server.sin_port= htons(atoi(argv[2]));

/* Tenta di realizzare la connessione */
printf("Connessione in corso...\n");
if(connect(sock, (struct sockaddr *)&server, sizeof server) <0)
{
    perror("connect su stream socket");
    exit(1);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

127

Socket

Esempio n. 1 (client cont.)

```
printf("...connesso.\n");

rval=BYTES_NR;

/* Invio/Ricezione di MSG_NR messaggi */
for (i=0; i<MSG_NR; i++)
{
    if((rval = write(sock, buf, sizeof buf))<0)
        perror("write su stream socket");

    printf("w=%d byte scritti\n", rval);

    s=0 ;
    do
    { /* Il messaggio puo` essere stato frammentato dal protocollo TCP */
        if((rval = read(sock, &buf[s], sizeof buf))<0)
            perror("reading stream message");

        s+= rval;
        printf("r=%d(%d) byte letti\n", rval, s);
    }
    while((s!=BYTES_NR) && rval !=0);
}

close(sock);
exit(0);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

128

Socket

Esempio n. 2 (header file)

```

/*****
      FTP-like CLIENT-SERVER su socket di tipo STREAM
*****/
ftpdefines.h
#define BYTES_NR      8192
#define MSG_NR       64
#define SOL_TCP      pp->p_proto
#define RICHMSG_MAXPATHNAME 256

typedef struct _RICHIESTA_MSG {
    char filename[RICHMSG_MAXPATHNAME];
} RICHIESTA_MSG;

typedef struct _RISPOSTA_MSG {
    int result;
    char errmsg[512];
    int filesize;
} RISPOSTA_MSG;

```

Sis.Op. A - UNIX - Programmazione di sistema

129

Socket

Esempio n. 2 (server)

```

/*****
      FTP-like CLIENT-SERVER su socket di tipo STREAM
*****/
      S E R V E R
*****/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <sys/timeb.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include "ftpdefines.h"

char    buf[BYTES_NR];

```

```

/*****
Utilizzo:      ftpserver # opera alla porta 1520
*****/

```

Sis.Op. A - UNIX - Programmazione di sistema

130

Socket

Esempio n. 2 (server - cont.)

```
main()
{
    int          sock,length;
    struct       sockaddr_in  server,client;
    char         buff[512];
    int         s,msgsock,rval,rval2,i;
    struct       hostent *hp,*gethostbyname();

    /* Crea la socket STREAM */
    sock= socket (AF_INET,SOCK_STREAM,0);
    if(sock<0)
        { perror("opening stream socket");
          exit(1);
        }
    server.sin_family = AF_INET;
    server.sin_addr.s_addr= INADDR_ANY;
    server.sin_port = htons(1520);
    if (bind(sock,(struct sockaddr *)&server,sizeof server)<0)
        {
            perror("binding stream socket");
            exit(1);
        }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

131

Socket

Esempio n. 2 (server - cont.)

```
length= sizeof server;
if(getsockname(sock,(struct sockaddr *)&server,&length)<0)
    {
        perror("getting socket name");
        exit(1);
    }
printf("Socket port %#d\n",ntohs(server.sin_port));

/* Pronto ad accettare connessioni */
listen(sock,2);

do {
    /* Attesa di una connessione */
    msgsock= accept (sock,(struct sockaddr *)&client,(int *)&length);
    if(msgsock ==-1)
        { perror("accept"); exit(-1);
        }
    else
        {
            if(fork()==0) {
                printf("Servng connection from %s, port %d\n",
                    inet_ntoa(client.sin_addr), ntohs(client.sin_port));
            }
        }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

132

Socket

Esempio n. 2 (server - cont.)

```
    close(sock);
    ftpserv(msgsock);
    close(msgsock);
    exit(0);
}
else
    close(msgsock);
}
} while(1);
}

ftpserv(int sock)
{
    int s,rval,nread,fd;
    struct stat fbuf;
    RICHIESTA_MSG rich_msg;
    RISPOSTA_MSG risp_msg;

    s = 0;
    /* Ricezione del comando GET */

    if((rval=read(sock,&rich_msg,sizeof(RICHIESTA_MSG))<0)
        {
            perror("reading client request");
        }
```

Sis.Op. A - UNIX - Programmazione di sistema

133

Socket

Esempio n. 2 (server - cont.)

```
    exit(-1); }

if((fd= open(rich_msg.filename,O_RDONLY))<0)
    { fprintf(stderr,"Non riesco ad aprire il file %s (%s)...uscita
      !\n",rich_msg.filename,strerror(errno));
      risp_msg.result = -1;
      strcpy( risp_msg.errmsg,strerror(errno));
    }
    else {
        risp_msg.result= 0;
        fstat(fd,&fbuf); /* Ottiene la dimensione del file */
        risp_msg.filesize= fbuf.st_size;
    }
    /* Invio della risposta */
    if((rval = write(sock,&risp_msg,sizeof(RISPOSTA_MSG))<0)
        perror("writing on stream socket");
    if( risp_msg.result != 0) return -1;

do {
    if((nread = read(fd,buf,sizeof buf))<0)
        perror("reading from file");
    if (nread >0)
        if((rval = write(sock,buf,nread))<0)
            perror("writing on stream socket");
    }
    while(nread > 0);
}
Sis.Op. A - UNIX - Programmazione di sistema
```

134

Socket

Esempio n. 2 (client)

```
*****  
/***** CLIENT-SERVER su socket di tipo STREAM *****  
C L I E N T  
*****  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <netdb.h>  
#include <stdio.h>  
#include <string.h>  
#include <sys/timeb.h>  
#include <fcntl.h>  
  
#include "ftplib.h"  
  
struct protoent *pp;  
  
char buf[BYTES_NR];  
  
/*****  
Utilizzo: ftpclient nomeserver portaserver nomefile  
*****/  
Sis.Op. A - UNIX - Programmazione di sistema
```

135

Socket

Esempio n. 2 (client - cont.)

```
main(argc,argv)  
int argc;char *argv[];  
{  
int i,s,fd,sock,rval,rval2;  
struct sockaddr_in server;  
struct hostent *hp,*gethostbyname();  
char copiafilenome[RICHMSG_MAXPATHNAME+16];  
char rich_mesg;  
RISPOSTA_MSG risp_mesg;  
  
if(argc != 4) {  
fprintf(stderr,"Uso: %s servername porta nomefile\n\n",argv[0]);  
exit(-1);  
}  
  
/* Crea una socket di tipo STREAM per il dominio TCP/IP */  
sock= socket(AF_INET,SOCK_STREAM,0);  
  
if(sock<0)  
{  
perror("opening stream socket");  
exit(1);  
}
```

Socket

Esempio n. 2 (client - cont.)

```
/* Ottiene l'indirizzo del server */
server.sin_family= AF_INET;

hp= gethostbyname(argv[1]);
if(hp==0) {
    fprintf(stderr, "%s: unknown host", argv[1]);
    exit(2);
}

memcpy( (char *)&server.sin_addr, (char *)hp->h_addr ,hp->h_length);

/* La porta e' sulla linea di comando */
server.sin_port= htons(atoi(argv[2]));

/* Tenta di realizzare la connessione */
printf("Connecting to the server %s...\n", argv[1]);
if(connect(sock, (struct sockaddr *)&server, sizeof server) <0)
{
    perror("connecting stream socket");
    exit(1);
}

printf("Connected to the server.\n");
```

Sis.Op. A - UNIX - Programmazione di sistema

137

Socket

Esempio n. 2 (client - cont.)

```
strncpy(rich_msg.filename, argv[3], RICHMSG_MAXPATHNAME);

/* Invio comando RICHIESTA (GET) */
write(sock, &rich_msg, sizeof(RICHIESTA_MSG));

/* Riceve la RISPOSTA dal server */
if((rval = read(sock, &risp_msg, sizeof(RISPOSTA_MSG))) <0)
    perror("reading server answer");

if(risp_msg.result !=0) {
    fprintf(stderr, "OOPS il server risponde %d (%s) ...uscita
!\n", risp_msg.result, risp_msg.errmsg);
    close(sock);
    exit(0);
}

strcpy(copiafilename, "copia.");
strcat(copiafilename, rich_msg.filename);

if((fd= open(copiafilename, O_WRONLY|O_CREAT|O_TRUNC, 0644)) <0) {
    fprintf(stderr, "Non posso aprire il file copia %s ...uscita
!\n", copiafilename);
    close(sock);
}

Sis.Op. A - UNIX - Programmazione di sistema
```

138

Socket

Esempio n. 2 (client - cont.)

```
        exit(0);
    }
    s=0;
    do
    {
        if((rval = read(sock,buf,sizeof buf))<0)
            perror("reading stream message");

        if(rval >0)
            {
                write(fd,buf,rval);
                putchar('.');
                s += rval;
            }
        while (rval !=0);

        printf("\nTrasferimento completato di %s completato - ricevuti %d
        byte (dimensione sul server
        %d\n",rich_mesg.filename,s, risp_mesg.filesize);

        close(sock);
        close(fd);
    }
    exit(0);
}
,
Sis.Op.A - UNIX - Programmazione di sistema
```

139

Socket

Esempio n. 3

```

/*****
Semplice ping-pong su socket di tipo DATAGRAM
S E R V E R
Utilizzo:      server [numeroporta]

*****
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <sys/timeb.h>
#include <string.h>

#define BYTES_NR      512
#define MSG_NR       32

#define DEFAULTPORT  3001
char buf[BYTES_NR];

```

Socket

Esempio n. 3 (server - cont)

```
main(argc, argv)
int argc;char *argv[];
{
    int      sock, length;
    struct   sockaddr_in  sockaddr_in      server, client;
    int      msgsock, rval, i;
    struct hostent *hp, *gethostbyname();
    int myport;

    if(argc >2)
    {
        fprintf(stderr, "Uso: %s [portaserver] \n", argv[0]);
        exit(-1);
    }

    if(argc == 2)
        myport = atoi(argv[1]);
    else
        myport = DEFAULTTPORT;

    if((myport < 1024  && myport != 0)|| myport > 65535)
    {
        fprintf(stderr, "Il numero di porta puo essere 0 (per ottenerne
una libera) \noppure deve essere compresa tra 1024 e 65535\n");
        exit(-1);
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

141

Socket

Esempio n. 3 (server - cont)

```
/* Crea la socket DGRAM */
sock=  socket(AF_INET, SOCK_DGRAM, 0);
if(sock<0)
{
    perror("open sulla socket dgram");
    exit(1);
}

/* Name socket using wildcards */
server.sin_family =  AF_INET;
server.sin_addr.s_addr= INADDR_ANY;
server.sin_port = htons(myport);

if (bind(sock, (struct sockaddr *)&server, sizeof server)<0)
{
    perror("bind sulla socket dgram");
    exit(1);
}

/* Find out assigned port and print out */
length= sizeof server;
if(getsockname(sock, (struct sockaddr *)&server, &length)<0)
{
    perror("getsockname");
    exit(1);
}
}
```

Sis.Op. A - UNIX - Programmazione di sistema

142

Socket

Esempio n. 3 (server - cont)

```
printf("Server attivo sulla porta #%d\n", ntohs(server.sin_port));

while(1)
{
    bzero(buf, sizeof buf);
    if((rval = recvfrom(sock, buf, sizeof buf, 0, (struct sockaddr* )
        &client, (socklen_t *)&length ))<0)
    {
        perror("recvfrom sulla socket dgram");
        exit(-3);
    }

    printf("Messaggio ricevuto dal client IP=%s porta=%d: rispedisco il
messaggio\n", inet_ntoa(client.sin_addr), client.sin_port);
    strcat(buf, "*");
    if(sendto(sock, buf, sizeof buf, 0, (struct sockaddr *)&client, sizeof
client)<0)
    {
        perror("recvfrom sulla socket dgram");
        exit(-4);
    }
} /* ^C o un altro segnale per farlo terminare */
}
```

Sis.Op. A - UNIX - Programmazione di sistema

143

Socket

Esempio n. 3 (client)

```
/*
*****
          Semplice ping-pong su socket di tipo DATAGRAM
*****
C L I E N T
*/
```

Utilizzo: client nomeserver portaserver

```
*****
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

#include <sys/time.h>
#include <unistd.h>

#define BYTES_NR      512
#define MSG_NR       32

Sis.Op. A - UNIX - Programmazione di sistema
```

144

Socket

Esempio n. 3 (client - cont)

```
char buf[BYTES_NR];
char buf2[BYTES_NR];

char msg[MSG_NR][BYTES_NR];
char ans[MSG_NR][BYTES_NR];

struct timeval xstime[MSG_NR];
struct timeval xftime[MSG_NR];

main(argc, argv)
int argc;char *argv[];
{
    int i,sock,rval,length;
    unsigned long delay;
    struct sockaddr_in server,client;
    struct hostent *hp,*gethostbyname();

    if(argc !=3)
    {
        fprintf(stderr,"Uso: %s nomeserver portaserver\n",argv[0]);
        exit(-1);
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

145

Socket

Esempio n. 3 (client - cont)

```
/* Prepara i messaggi da inviare al server */
for (i=0;i<MSG_NR;i++)
    {
        sprintf(&msg[i][0],"%d",i);
    }

/* Crea la socket DGRAM */
sock= socket(AF_INET,SOCK_DGRAM,0);
if(sock<0)
    {
        perror("opening stream socket");
        exit(1);
    }

client.sin_family= AF_INET;
client.sin_addr.s_addr = INADDR_ANY;
client.sin_port = htons(0);

if (bind(sock,(struct sockaddr *)&client,sizeof client) <0)
    {
        perror("bind su socket dgram");
        exit(1);
    }
}
```

Sis.Op. A - UNIX - Programmazione di sistema

146

Socket

Esempio n. 3 (client - cont)

```
length= sizeof client;
if(getsockname(sock, (struct sockaddr *)&server, &length) < 0)
{
    perror("getsockname");
    exit(1);
}

/* Ottiene l'indirizzo IP del server */
hp = gethostbyname(argv[1]);
if (hp == 0)
{
    fprintf(stderr, "%s : nodo sconosciuto", argv[1]);
    exit(2);
}
bcopy( (char *)hp ->h_addr, (char *)&server.sin_addr, hp ->h_length);
server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[2]));

for (i=0; i<MSG_NR; i++)
{
    strcpy(buf, msg[i]);
    /* Ottiene il tempo corrent (prima della send) */
    gettimeofday(&xsstime[i], NULL);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

147

Socket

Esempio n. 3 (client - cont)

```
if(sendto(sock, buf, sizeof buf, 0, (struct sockaddr
    *)&server, sizeof server) < 0)
    perror("sendto sulla socket dgram");
if((rval = recvfrom(sock, buf2, sizeof buf2, 0, (struct sockaddr
    *)NULL, (socklen_t *)NULL)) < 0)
{
    perror("recvfrom sulla socket dgram");
    exit(-3);
}
strcpy(answ[i], buf2);
/* Ottiene il tempo corrente (dopo la recvfrom) */
gettimeofday(&xsftime[i], NULL);
}
close(sock);

printf("Ping-pong con %s con datagrammi da %d
    bytes\n", inet_ntoa(server.sin_addr), BYTES_NR);
for (i=0; i<MSG_NR; i++)
{
    /* Calcolo del ritardo */
    delay= (xsftime[i].tv_sec-xstime[i].tv_sec)
    *1000000.+(xsftime[i].tv_usec-xstime[i].tv_usec);
    printf("msg n.%d [%s]: %0.3f ms\n", i, answ[i], delay/1000.);
}
exit(0);
}
```

Sis.Op. A - UNIX - Programmazione di sistema

148

Select

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

La primitiva `select` permette di attendere una variazione di stato per i file descriptor (riferiti a file, pipe, socket, ...) all'interno di tre distinti insiemi di file descriptor (tipo `fd_set`):

- si attende la disponibilità di dati in lettura per i fd contenuti in `readfds`
- si attende la possibilità di scrittura immediata sui fd contenuti in `writefds`
- si attende la presenza di eccezioni per i fd contenuti in `exceptfds`

`n` è il valore del descrittore più alto nei tre insiemi, aumentato di 1

L'attesa può essere limitata superiormente da un `timeout`

Select

Il valore di uscita è il numero di descrittori che sono variati di stato (zero significa che è scaduto l'intervallo di `timeout`)

Gli `fd_set` sono modificati in uscita dalla `select` in modo che contengano i soli fd che hanno variato di stato

Macro utili per la manipolazione di variabili `fd_set`

`FD_ZERO(fd_set *set)` azzera un `fd_set`

`FD_CLR(int fd, fd_set *set)` rimuove un fd da un `fd_set`

`FD_SET(int fd, fd_set *set)` inserisce un fd in un `fd_set`

`FD_ISSET(int fd, fd_set *set)` predicato che verifica se un certo fd è membro di un `fd_set`

Select

Esempio: processo lettore su due pipe

```
int          max_fd;
fd_set      rpipe_fds;
struct timeval tv;
...
if(pipe(pipea) < 0) ...
if(pipe(pipeb) < 0) ...

if(fork() == 0) {
    /* I descrittori di lettura delle pipe sono inseriti nel
    fd_set di lettura) */

    FD_ZERO(&rpipe_fds);
    FD_SET(pipea[0], &rpipe_fds);
    FD_SET(pipeb[0], &rpipe_fds);

    /* Attesa al massimo di 5 secondi */
    tv.tv_sec= 5 ; tv.tv_nsec= 0 ;

    max_fd = pipeb[0]+1;
}
```

Sis.Op. A - UNIX - Programmazione di sistema

151

Select

Esempio

```
/* Il processo si blocca in attesa della lettura su pipea e/o
su pipeb o del timeout */
retval= select(max_fd, &rpipe_fds, NULL, NULL, &tv) ;

if(retval>0) /* C'e' almeno un descrittore pronto per la
lettura : rpipe_fds ora contiene i descrittori pronti */
{
    /* Occorre verificare quale descrittore sia pronto */
    if(FD_ISSET(pipea[0], &rpipe_fds)
        read(pipea[0], buffer, sizeof message);

    if(FD_ISSET(pipeb[0], &rpipe_fds)
        read(pipeb[0], buffer, sizeof message);
    }
else if(!retval)
    { /* retval vale 0 */
    printf("Timeout !\n");
    }
else {perror("errore select: "); exit(-1);
}
}
```

Sis.Op. A - UNIX - Programmazione di sistema

152