UNIX: programmazione di sistema

- Per poter utilizzare servizi quali manipolazione di file, creazione di processi e comunicazione tra processi, i programmi di applicazione devono interagire con il sistema operativo.

- Per far ciò devono usare un insieme di routine chiamate `system call`, che costituiscono l'interfaccia *funzionale* del programmatore col nucleo di UNIX.

- Le system call sono simili alle ruotine di libreria ma eseguono una chiamata di subroutine direttamente nel nucleo di UNIX.

- Le chiamate di sistema possono essere raggruppate in tre categorie principali:
  - gestione dei file,
  - gestione degli errori,
  - gestione dei processi.

- La comunicazione tra processi (o *interprocess communication*, IPC) è parte della gestione dei file poiché UNIX tratta i meccanismi per IPC come file speciali.

Gestione dei file

- Le system call per la gestione dei file permettono di manipolare i file regolari, le directory ed i file speciali, compresi:
  - file su disco
  - terminali
  - stampanti
  - meccanismi di IPC, quali pipe e socket.
- *Tipica* sequenza di operazioni su file.

```
int fd; /* dichiara un descrittore di file */
...                 /* apre un file e restituisce il descrittore */
fd = open(fileName, ...);
if (fd == -1) {
  ... /* gestisce l'errore */
};
...
fcntl(fd, ...); /* modifica alcuni flag di I/O */
...
read(fd, ...); /* legge dal file */
...
write(fd, ...); /* scrive nel file */
...
lseek(fd, ...); /* si sposta all'interno del file */
...
close(fd); /* chiude il file, liberando il descrittore */
...
unlink(fileName); /* rimuove il file */
...
```

Gestione dei file

- Per accedere o creare un file si usa la system call `open()`.

  Se `open()` ha successo, restituisce un intero chiamato *descrittore di file* che è usato nelle successive operazioni di I/O sul file. Se fallisce, restituisce il valore `-1`.

- Quando un processo non ha più bisogno di un file aperto, lo chiude invocando `close()`.

  Tutti i file aperti da un processo sono automaticamente chiusi quando il processo termina (quindi la chiusura esplicita potrebbe essere omessa, ma è una buona pratica di programmazione chiudere esplicitamente i file aperti).

  Quando il riferimento ad un file viene chiuso, il descrittore relativo viene rilasciato e può essere riassegnato da una successiva invocazione di `open()`.

- I descrittori dei file sono numerati a partire da 0. Per convenzione, i primi tre hanno un significato particolare:
  - 0: corrisponde allo *standard input*;
  - 1: corrisponde allo *standard output*;
  - 2: corrisponde allo *standard error*.

Descrittore di file

- Molte system call di I/O richiedono un descrittore di file come primo argomento in modo da conoscere su quale file operano.
- Uno stesso file può essere aperto diverse volte e quindi può avere diversi descrittori associati.
- Ogni descrittore possiede un insieme di *proprietà* che non hanno niente a che vedere con quelle associate al file a cui punta:
  - Un *puntatore* al file che registra l'*offset*, rispetto all'inizio, della posizione all'interno del file in cui sta leggendo o scrivendo.

    Quando il descrittore è creato, il puntatore è inizializzato a 0; man mano che il processo effettua operazioni di lettura/scrittura nel file, il puntatore è automaticamente aggiornato di conseguenza.
  - Un flag che indica se il descrittore dev'essere chiuso automaticamente quando viene invocata una system call della famiglia di `exec()` (per creare un nuovo processo).
  - Un flag che indica se l'output sul file deve essere *appeso* in fondo al file.

Descrittore di file

- Inoltre, se il file è speciale, ad esempio è un pipe o un socket, ci sono altre proprietà significative associate al descrittore:
  - Un flag che indica se un processo si deve bloccare quando tenta di leggere dal file ed il file è vuoto.
  - Un numero che indica un identificatore di processo o di gruppo a cui bisogna spedire un segnale `SIGIO` quando l'input sul file diventa disponibile.

- Le proprietà associate ad un descrittore possono essere manipolate tramite le system call `open()` e `fcntl()`.

Esempio: reverse

`reverse -c` [*fileName*]

`reverse` inverte le linee di input lette da *fileName* e le mostra sullo standard output. Se non si specifica alcun file di input, `reverse` inverte lo standard input. Se si usa l'opzione `-c`, `reverse` inverte anche i caratteri su ogni linea.

```
#include <fcntl.h> /* For file mode definitions */
#include <stdio.h>
#include <stdlib.h>

/* Enumerator */
enum { FALSE, TRUE }; /* Standard false and true values */
enum { STDIN, STDOUT, STDERR }; /* Standard I/O channel indices */

/* #define Statements */
#define BUFFER_SIZE    4096   /* Copy buffer size */
#define NAME_SIZE      12
#define MAX_LINES      100000 /* Max lines in file */

/* Globals */
char *fileName = NULL; /* Points to file name */
char tmpName [NAME_SIZE];
int charOption = FALSE; /* Set to true if -c option is used */
int standardInput = FALSE; /* Set to true if reading stdin */
int lineCount = 0; /* Total number of lines in input */
int lineStart [MAX_LINES]; /* Store offsets of each line */
int fileOffset = 0; /* Current position in input */
int fd; /* File descriptor of input */
```

Esempio: reverse

```
/*****************************************************************/
int main(int argc, char *argv[]) {
  parseCommandLine (argc,argv); /* Parse command line */
  pass1 (); /* Perform first pass through input */
  pass2 (); /* Perform second pass through input */
  return (/* EXITSUCCESS */ 0); /* Done */
}
/*****************************************************************/
int parseCommandLine (int argc, char *argv[]) {
  int i;
  for (i= 1; i < argc; i++) {
     if(argv[i][0] == '-')
       processOptions (argv[i]);
     else if (fileName == NULL)
       fileName= argv[i];
     else
       usageError (); /* An error occurred */
  }
  standardInput = (fileName == NULL);
}
/*****************************************************************/
int processOptions (char* str) {
  int j;
  for (j= 1; str[j] != '\0'; j++) {
     switch(str[j]) { /* Switch on command line flag */
        case'c':
          charOption = TRUE;
          break;
        default:
          usageError ();
          break;
     }
  }
}
```

Esempio: reverse

```
/*****************************************************************/
int usageError (void) {
  fprintf (stderr, "Usage: reverse -c [filename]\n");
  exit (/* EXITFAILURE */ 1);
}

/*****************************************************************/
int pass1 (void) { /* Perform first scan through file */
   int tmpfd, charsRead, charsWritten;
   char buffer [BUFFER_SIZE];
   if (standardInput) { /* Read from standard input */
     fd = STDIN;
     sprintf (tmpName, ".rev.%d",getpid ()); /* Random name */
     /* Create temporary file to store copy of input */
     tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
     if (tmpfd == -1) fatalError ();
   }
   else { /* Open named file for reading */
      fd = open (fileName, O_RDONLY);
      if (fd == -1) fatalError ();
   }
   lineStart[0] = 0; /* Offset of first line */
   while (TRUE) { /* Read all input */
      /* Fill buffer */
      charsRead = read (fd, buffer, BUFFER_SIZE);
      if (charsRead == 0) break; /* EOF */
      if (charsRead == -1) fatalError (); /* Error */
      trackLines (buffer, charsRead); /* Process line */
      /* Copy line to temporary file if reading from stdin */
      if (standardInput) {
         charsWritten = write (tmpfd, buffer, charsRead);
         if(charsWritten != charsRead) fatalError ();
      }
   }
```

Esempio: reverse

```
   /* Store offset of trailing line */
   lineStart[lineCount] = fileOffset;
   /* If reading from standard input, prepare fd for pass2 */
   if (standardInput) fd = tmpfd;
}

/*****************************************************************/
int trackLines (char *buffer, int charsRead) {
  /* Store offsets of each line start in buffer */
  int i;
  for (i = 0; i < charsRead; i++) {
     ++fileOffset; /* Update current file position */
     if (buffer[i] == '\n') lineStart[++lineCount] = fileOffset;
  }
}

/*****************************************************************/
int pass2 (void) {
  /* Scan input file again, displaying lines in reverse order */
  int i;
  for (i = lineCount - 1; i >= 0; i--)
    processLine (i);
  close (fd); /* Close input file */
  if (standardInput) unlink (tmpName); /* Remove temp file */
}
```

Esempio: reverse

```
/*****************************************************************/
int processLine (int i) {
  /* Read a line and display it */
  int charsRead;
  char buffer [BUFFER_SIZE];
  lseek (fd, lineStart[i], SEEK_SET); /* Find the line and read it */
  charsRead = read (fd, buffer, lineStart[i+1] - lineStart[i]);
  /* Reverse line if -c option was selected */
  if (charOption) reverseLine (buffer, charsRead);
  write (1, buffer, charsRead); /* Write it to standard output */
}

/*****************************************************************/
int reverseLine (char* buffer, int size) {
  /* Reverse all the characters in the buffer */
  int start = 0, end = size - 1;
  char tmp;
  if (buffer[end] == '\n') --end; /* Leave trailing newline */
  /* Swap characters in a pairwise fashion */
  while (start < end) {
     tmp = buffer[start];
     buffer[start] = buffer[end];
     buffer[end] = tmp;
     ++start; /* Increment start index */
     --end; /* Decrement end index */
  }
}

/*****************************************************************/
int fatalError (void) {
  perror ("reverse: "); /* Describe error */
  exit (1);
}
```

open()

`int open(char *`*fileName*`, int` *mode* `[, int` *permissions*`])`

- Permette di accedere o creare un file per leggere e/o scrivere.
- *fileName* è un pathname assoluto o relativo.
- *mode* è una congiunzione, tramite l'operatore '`|`', di flag di lettura/scrittura con, eventualmente, altri flag.
- *permissions* è un numero che codifica (di solito, in forma ottale) il valore dei flag di permesso del file e dev'essere fornito solo quando il file è creato (sui permessi influisce anche il valore di `umask`).

  I valori predefiniti dei flag sono definiti nei file `fcntl.h`.
- `open()` restituisce un intero non negativo, se ha successo; altrimenti, restituisce il valore `-1`.

open()

- I flag di lettura/scrittura sono:
  - `O_RDONLY`: apertura in sola lettura;
  - `O_WRONLY`: apertura in sola scrittura;
  - `O_RDWR`: apertura in lettura e scrittura.
- Gli altri flag sono:
  - `O_APPEND`: Il puntatore al file è posizionato alla fine del file prima di ogni `write()`.
  - `O_CREAT`: Se il file non esiste viene creato; il valore specificato con `umask` è utilizzato per determinare i permessi iniziali sul file.
  - `O_EXCL`: Se il flag `O_CREAT` è attivo ed il file esiste, allora `open()` fallisce.
  - `O_NONBLOCK` (o `O_DELAY`): Questo flag vale solo per pipe, socket e STREAM. Se attivato, una `open()` in sola lettura termina immediatamente, a prescindere dal fatto che il lato di scrittura sia aperto o no, ed una `open()` in sola scrittura fallisce se il lato di lettura non è aperto. Se non è attivato, una `open()` su un file in sola lettura o in sola scrittura si blocca fino a che l'altro lato non è aperto.
  - `O_TRUNK`: Se il file esiste, è troncato a lunghezza 0.

open(): esempi

- ```
  sprintf(tmpName, ".rev.%d", getpid()); /* Random name */
      /* Create temporary file to store copy of input */
  tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
  if (tmpfd == -1) fatalError ();
  ```

  La funzione `getpid()` è una chiamata di sistema che restituisce il PID del processo. Siccome il PID del processo è unico in tutto il sistema, questo meccanismo costituisce un modo semplice di generare nomi unici per file temporanei.

  Per creare un file, si usa il flag `O_CREAT` e si impostano i permessi iniziali tramite cifre ottali.

- ```
  fd = open (fileName, O_RDONLY);
  if (fd == -1) fatalError ();
  ```

  Per aprire un file esistente si specifica solo il flag di lettura/scrittura.

read()

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

(sintassi da usare per operare con file regolari)

- `read()` copia *count* byte dal file riferito dal descrittore *fd* nel buffer *buf*.
- I byte sono letti a partire dalla posizione corrente del puntatore, che è quindi aggiornato di conseguenza.
- `read()` copia più byte che può, fino ad un massimo di *count*, e restituisce il numero dei byte effettivamente copiati.
- Se invocata dopo che si è già letto l'ultimo byte, restituisce il valore `0`, che indica la fine del file.
- `read()` restituisce il numero dei byte che ha letto, se ha successo; altrimenti, restituisce il valore `-1`.
- La system call `read()` effettua input di basso livello e non ha alcuna delle capacità di formattazione che ha `scanf()`. Il vantaggio dell'uso di `read()` sta nel fatto che essa salta lo strato di buffering addizionale fornito dalle funzioni di libreria del C ed è perciò molto veloce.

- `read()` non conta tra i caratteri letti effettivamente il carattere EOF eventualmente letto.

read(): esempi

```
charsRead = read (fd, buffer, BUFFER_SIZE);
if (charsRead == 0) break; /* EOF */
if (charsRead == -1) fatalError (); /* Error */
```

`read()` è usata per leggere `BUFFER_SIZE` (che, per motivi di efficenza, è stato scelto in modo che sia un multiplo della dimensione del blocco del disco) caratteri per volta.

write()

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

(sintassi da usare per operare con file regolari)

- `write()` copia *count* byte dal buffer *buf* al file riferito dal descrittore *fd*.
- I byte sono scritti a partire dalla posizione corrente del puntatore, che è quindi aggiornato di conseguenza. Se è stato attivato il flag `O_APPEND` del descrittore *fd*, la posizione del puntatore è spostata alla fine del file prima di ogni scrittura.
- `write()` copia da *buf* più byte che può, fino ad un massimo di *count*, e restituisce il numero dei byte effettivamente copiati. Il processo invocante dovrebbe sempre controllare il valore restituito. Se tale valore non coincide con *count*, allora è probabile che il disco sia pieno e non sia rimasto spazio.
- `write()` restituisce il numero dei byte che ha scritto, se ha successo; altrimenti, restituisce il valore `-1`.
- Valgono per la `write()` (risp. alla `printf()`) le stesse considerazioni fatte per la `read()` (risp. alla `scanf()`).

write(): esempi

```
/* Copy line to temporary file if reading from stdin */
if (standardInput) {
   charsWritten = write (tmpfd, buffer, charsRead);
   if(charsWritten != charsRead) fatalError ();
}
```

lseek()

`off_t lseek(int` *fd*`, off_t` *offset*`, int` *mode*`)`

- Permette di modificare il valore corrente del puntatore al file associato al descrittore e quindi di leggere/scrivere le righe del file in un ordine differente da quello sequenziale.
- *fd* è il descrittore del file, *offset* è un `long int` e *mode* descrive come dev'essere interpretato *offset*.
- I tre possibili valori di *mode* sono definiti nel file `/usr/include/stdio.h` e significano:
  - `SEEK_SET`: *offset* è relativo all'inizio del file;
  - `SEEK_CUR`: *offset* è relativo alla posizione corrente;
  - `SEEK_END`: *offset* è relativo alla fine del file.
- `lseek()` fallisce se si tenta di muovere il puntatore ad un punto che precede l'inizio del file.
- `lseek()` restituisce la posizione corrente, se ha successo; altrimenti, restituisce il valore `-1`.

lseek(): esempi

- ```
  lseek (fd, lineStart[i], SEEK_SET); /* Find the line and read it */
  charsRead = read (fd, buffer, lineStart[i+1] - lineStart[i]);
  ```

  `lseek()` fa puntare il puntatore all'inizio di una linea (la i-esima).

  Quindi viene calcolato il numero dei caratteri da leggere sottraendo il valore dell'offset dell'inizio della linea successiva da quello dell'inizio della linea corrente.

  Infine, viene fatta la lettura di tutti i caratteri in una linea in un colpo solo.

- ```
  currentOffset = lseek (fd, 0, SEEK_CUR);
  ```

  permette di memorizzare nella variabile `currentOffset` il valore corrente dell'offset senza muoversi.

- Se si oltrepassa la fine di un file e quindi si esegue una `write()`, il nucleo del SO automaticamente estende la dimensione del file e tratta l'area intermedia del file come se fosse riempita da caratteri `NULL` (codifica ASCII 0).

  Tuttavia, non viene allocato spazio disco per l'area intermedia e quindi il file occupa meno spazio disco di uno analogo in cui i caratteri `NULL` siano stati inseriti esplicitamente dall'utente.

close()

`int close(int` *fd*`)`

- Libera il descrittore di file *fd*.
- Se *fd* è l'ultimo descrittore associato ad un particolare file aperto, le risorse del nucleo associate al file vengono deallocate.
- Quando un processo termina, tutti i suoi descrittori di file sono automaticamente chiusi, ma sarebbe preferibile chiuderli esplicitamente quando non servono più.
- Se si chiude un descrittore che era già chiuso, si verifica un errore.
- `close()` restituisce il valore `0`, se ha successo; altrimenti, restituisce il valore `-1`.
- Quando un file viene chiuso, non è garantito che i buffer associati al file siano immediatamente scaricati su disco.

- *Esempio*: chiude il descrittore `fd`

```
close(fd);
```

Gestione degli errori: `perror()`

- Per gestire gli errori che possono essere eventualmente originati da system call, i due principali ingredienti da utilizzare sono:
  - `errno`: *variabile globale* che contiene il codice numerico dell'ultimo errore generato da una system call;
  - `perror()`: *subroutine* che mostra una descrizione dell'ultimo errore generato dall'invocazione di una system call.

- Ogni processo contiene una variabile globale `errno` inizializzata a 0 quando il processo è creato.

  Quando si verifica un errore dovuto ad una system call, ad `errno` è assegnato un *codice numerico* corrispondente alla causa dell'errore.

- I file `errno.h` contengono diverse liste di codici di errore predefiniti. Alcuni sono:

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

Gestione degli errori: `perror()`

- Una system call che viene eseguita con successo non influenza in alcun modo il valore corrente di `errno`, mentre una che non ha successo sovrascrive sempre il valore corrente di `errno`.

- La routine

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

  (che effettivamente è una routine di libreria standard del C, non una system call) mostra la stringa `str`, seguita da ': ', seguita da una descrizione in lingua inglese del valore corrente di `errno` (e da un carattere new-line).

  Se non ci sono errori da riportare, viene mostrata la stringa `Error 0` (o, in alcuni sistemi, `Success`).

- Per poter accedere la variabile `errno` ed invocare `perror()` dall'interno di un programma, bisogna includere il file `errno.h`.

- I programmi dovrebbero controllare se il valore restituito da una system call è `-1` e nel qual caso invocare `perror()` per avere una descrizione dell'errore.

Gestione degli errori: `perror()`

Nell'esempio seguente, per illustrare il funzionamento di `perror()` e la gestione di `errno`, vengono forzati un paio di errori di sistema.

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

int main(void) {
  int fd;
     /* Open a non-existent file to cause an error */
  fd = open ("nonexist.txt", O_RDONLY);
  if (fd == -1) {    /* fd == -1 => an error occurred */
     printf ("errno = %d\n", errno);
     perror ("main");
  };
  fd = open ("/", O_WRONLY);    /* Force a different error */
  if (fd == -1) {
    printf ("errno = %d\n", errno);
    perror ("main");
  };
     /* Execute a successful system call */
  fd = open ("nonexist.txt", O_RDONLY | O_CREAT, 0644);
  printf ("errno = %d\n", errno); /* Display after successful call */
  perror ("main");
  errno = 0;   /* Manually reset error variable */
  perror ("main");
  return 0;
}
```

Gestione degli errori: `perror()`

L'output è il seguente.

```
$ showErrno
errno = 2
main: No such file or directory
errno = 21
main: Is a directory
errno = 21
main: Is a directory
main: Error 0
$
```

Creare *hard* link

`int link(const char *`*oldPath*`, const char *`*newPath*`)`

- `link()` crea un nuovo collegamento (*link hard*) *newPath* al file cui è già collegato *oldPath*.
- Il contatore dei link hard associato al file è incrementato di una unità (il suo valore può essere esaminato tramite il comando `ls -l`).
- Se *newPath* esiste, non viene sovrascritto.
- Il nuovo nome può essere usato esattamente come il vecchio per qualsiasi operazione; entrambi fanno riferimento allo stesso file (e quindi hanno gli stessi permessi e possessore) ed è impossibile dire qual'era il nome originale.
- Un file è effettivamente rimosso dal file system solo quando tutti i suoi link hard sono stati cancellati.
- Se *newPath* e *oldPath* risiedono su file system differenti, un hard link non può essere creato e `link()` fallisce; in questo caso bisogna usare `symlink()`.
- `link()` restituisce `0`, se ha successo; altrimenti, restituisce `-1`.

`ln -sf` { *origName* }$^+$ *newLink*

- `ln` è una utility che permette di creare *link hard* o *soft* (*simbolici*) a file esistenti.
- Per creare un link hard ad un file, bisogna specificare il nome del file come *origName* ed il nuovo nome come *newLink*.
- Un *hard link* è un altro nome per un file esistente (non una copia del file); il link e l'originale sono indistinguibili. Tecnicamente parlando, essi condividono lo stesso *inode* e l'inode contiene tutte le informazioni rilevanti sul file (in effetti non è scorretto dire che l'inode *è* il file). Sul file si può operare usando uno qualsiasi dei suoi nomi.
- Il contatore dei link hard associato al file è incrementato di una unità (il suo valore può essere esaminato tramite il comando `ls -l`).
- Il file è effettivamente rimosso dal file system solo quando tutti i suoi link hard sono stati cancellati.
- Se *newLink* è il nome di una directory, allora i link hard sono creati da quella directory a tutti i file in { *origName* }$^+$.
- Su tutte le attuali implementazioni non si possono effettuare hard link a directory ed inoltre non si possono effettuare hard link tra file system differenti (queste restrizioni tuttavia non sono parte dello standard POSIX).
- Se *origName* e *newLink* risiedono su file system differenti, un hard link non può essere creato; in questo caso bisogna usare `ln -s` e creare un link simbolico.

Creare link *simbolici*

`int symlink(const char *`*oldPath*`, const char *`*newPath*`)`

- `symlink()` crea un file speciale, detto collegamento *simbolico* (o soft), *newPath* che contiene la stringa *oldPath*.
- Se *newPath* esiste, non viene sovrascritto.
- I link simbolici sono interpretati a run-time: nel determinare il path da seguire per trovare un file o una directory, il contenuto del link è sostituito al link stesso.
- Un link simbolico può contenere "`..`" che, se usato all'inizio del link, fa riferimento alla directory padre di quella in cui il link risiede.
- Un link simbolico può puntare anche ad un file inesistente; in questo caso è noto come link *pendente*.
- I permessi di un link simbolico sono irrilevanti; il possessore è ignorato quando si segue un link, mentre è considerato quando si tenta di rimuovere o di rinominare un link.
- `symlink()` restituisce `0`, se ha successo; altrimenti, restituisce `-1`.

- I *link simbolici*, o soft, sono un tipo speciale di file (che non tutti i kernel supportano) che fanno effettivamente riferimento ad un file differente tramite lo stesso nome del file.
- I link simbolici sono interpretati a run-time: nel determinare il path da seguire per trovare un file o una directory, il contenuto del link è sostituito al link stesso.
- Quando il link simbolico è passato alle operazioni (apertura, lettura, scrittura, ...), il nucleo automaticamente dereferenzia il link ed opera sul target del link. Tuttavia, alcune operazioni (per esempio, rimozione) agiscono direttamente sul link simbolico.
- Un link simbolico può contenere "`..`" che, se usato all'inizio del link, fa riferimento alla directory padre di quella in cui il link risiede.

Rimuovere link *hard*

`int unlink(const char *`*fileName*`)`

- Rimuove il link hard *fileName* al file corrispondente.
- Se *fileName* è l'ultimo link al file, le risorse assegnate al file vengono deallocate. In questo caso, se qualche descrittore di file di un processo è attualmente associato al file, la entry relativa è immediatamente rimossa dalla directory, ma il file viene deallocato solo dopo che tutti i descrittori sono stati chiusi. Ciò significa che un eseguibile può effettuare un `unlink()` su se stesso durante l'esecuzione e quindi continuare fino al completamento.
- `unlink()` restituisce il valore `0`, se ha successo; altrimenti, restituisce il valore `-1`.

- *Esempio*: rimuove il file temporaneo `tmpName`

```
if (standardInput) unlink (tmpName);
```

Ottenere informazioni sui file

```
int stat(const char *name, struct stat *buf)
int lstat(const char *name, struct stat *buf)
int fstat(int fd, struct stat *buf)
```

- `stat()` riempie il buffer *buf* con informazioni sul file *name*.
- `lstat()` restituisce informazioni sul link simbolico *name* piuttosto che sul file cui fa riferimento.
- `fstat()` opera come `stat()` eccetto per il fatto che prende come primo argomento un descrittore di file anzicché un nome di file.
- Per usarle è necessario includere i file `sys/stat.h` e `unistd.h`.
- Non è necessario avere diritti di accesso al file ma è necessario avere diritti di ricerca (`x`) su tutte le directory nel path del file.
- Restituiscono valore `0`, se hanno successo; altrimenti, restituiscono valore `-1`.

- La *struttura* `stat` contiene informazioni sullo stato di un file. È definita in `/usr/include/sys/stat.h` ed ha i seguenti membri:
  - `st_dev`: numero del dispositivo;
  - `st_ino`: numero di inode;
  - `st_mode`: flag dei permessi;
  - `st_nlink`: numero dei link *hard*;
  - `st_uid`: l'ID dell'utente;
  - `st_gid`: l'ID del gruppo;
  - `st_size`: la dimesione del file;
  - `st_atime`: la data dell'ultimo accesso;
  - `st_mtime`: la data dell'ultima modifica;
  - `st_ctime`: la data dell'ultimo cambiamento di stato.

- I campi riguardanti il tempo possono essere decodificati usando le soubroutine `asctime()` e `localtime()` della libreria C standard.

- In `/usr/include/sys/stat.h` ci sono inoltre alcune macro predefinite che prendono `st_mode` come argomento e restituiscono valore `1` per i seguenti tipi di file:
  - `S_IFDIR`: directory;
  - `S_IFCHR`: dispositivo speciale a caratteri;
  - `S_IFBLK`: dispositivo speciale a blocchi;
  - `S_IFREG`: file regolare;
  - `S_IFFIFO`: pipe.

Ottenere informazioni sulle directory

`int getdents(int` *fd*`, struct dirent *`*buf*`, int` *structSize*`)`

- `getdents()` legge un certo numero di strutture del tipo `struct dirent` dal file della directory riferita da *fd* e le memorizza nell'area di memoria puntata da *buf*.

  structSize è la grandezza dell'area di memoria.

- `getdents()` restituisce il numero di byte letti, se ha successo; altrimenti, restituisce `0` quando l'ultima entry è già stata letta, e `-1` in caso di errore.

- La *struttura* `dirent` contiene informazioni su una entry della directory.

- È definita in `/usr/include/sys/dirent.h` ed ha i seguenti membri:
  - `d_ino`: numero di inode;
  - `d_off`: l'offset della successiva entry;
  - `d_reclen`: la lunghezza della entry attuale;
  - `d_name`: nome del file.

Cambiare possessore e gruppo di un file

`int chown(const char *`*name*`, uid_t` *ownerId*`, gid_t` *groupId*`)`

`int lchown(const char *`*name*`, uid_t` *ownerId*`, gid_t` *groupId*`)`

`int fchown(int` *fd*`, uid_t` *ownerId*`, gid_t` *groupId*`)`

- `chown()` modifica gli ID del possessore e del gruppo del file *name* in *ownerId* e *groupId*, rispettivamente.

  Un valore `-1` in un argomento, significa che il valore associato deve restare inalterato.

- `lchown()` modifica possessore e gruppo del link simbolico *name* piuttosto che quelli relativi al file a cui il link fa riferimento.

- `fchown()` opera come `chown()` ma prende come argomento un descrittore di file aperto.

- Solo un *superutente* può modificare il possessore di un file; un utente può modificare il gruppo in un altro a cui l'utente appartiene.

- Restituiscono valore `0`, se hanno successo; altrimenti, restituiscono `-1`.

Cambiare i permessi di un file

`int chmod(const char *`*name*`, int` *mode*`)`

`int fchmod(int` *fd*`, int` *mode*`)`

- `chmod()` modifica i permessi sul file *name* in *mode*.

  mode di solito è una stringa di cifre ottali.

- Solo il possessore del file o un *superutente* può modificare i permessi sul file.

- `fchmod()` opera come `chmod()` ma prende come argomento un descrittore di file aperto anzicché il nome di un file.

- Restituiscono valore `0`, se hanno successo; altrimenti, restituiscono `-1`.

Duplicare descrittori di file

`int dup(int` *oldFd*`)`

`int dup2(int` *oldFd*`, int` *newFd*`)`

- `dup()` trova il più piccolo descrittore di file non utilizzato e lo fa riferire allo stesso file a cui fa riferimento *oldFd*.
- `dup2()` chiude il descrittore *newFd* se esso è attualmente attivo e quindi lo fa riferire allo stesso file a cui fa riferimento *oldFd*.
- In entrambi i casi, il descrittore di file originale e quello copiato condividono lo stesso puntatore interno al file e le stesse modalità di accesso.
- Restituiscono il nuovo descrittore, se hanno successo; altrimenti, restituiscono `-1`.

Duplicare descrittori: esempio

Il programma `mydup.c`

```
#include <stdio.h>
#include <fcntl.h>
int main (void) {
  int fd1, fd2, fd3;
  fd1 = open ("test.txt", O_CREAT | O_RDWR, 0600);
  printf ("fd1 = %d\n", fd1);
  write (fd1, "what's", 6);
  fd2 = dup (fd1); /* Make a copy of fd1 */
  printf ("fd2 = %d\n", fd2);
  write (fd2, " up", 3);
  close (0); /* Close standard input */
  fd3 = dup (fd1); /* Make another copy of fd1 */
  printf ("fd3 = %d\n", fd3);
  write (0, " doc", 4);
  dup2 (3, 2); /* Duplicate channel 3 to channel 2 */
  write (2, "?\n", 2);
  return 0;
}
```

produce il seguente output

```
$ mydup
fd1 = 3
fd2 = 4
fd3 = 0
$ cat test.txt
wht's up doc?
$
```

- Il programma nell'esempio crea il file *test.txt* e lo accede in scrittura tramite quattro differenti descrittori:
  - il primo è quello originale (allocato in 3);
  - il secondo è una copia del primo (allocato in 4);
  - il terzo è una copia del primo allocato in 0 (preventivamente liberato da `close(0)`);
  - il quarto è una copia del terzo allocato in 2.

Operare su descrittori di file

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

- `fcntl()` esegue l'operazione *cmd* sul file associato al descrittore *fd*. *arg* è un argomento opzionale per *cmd*.
- I valori più comuni per *cmd* sono i seguenti
  - `F_SETFD`: assegna il bit meno significativo di *arg* al flag close-on-exec.
  - `F_GETFD`: restituisce un numero il cui bit meno significativo è `1` o `0` a seconda che il flag close-on-exec sia attivo o no.
  - `F_GETFL`: restituisce un numero che corrisponde ai flag correnti del file ed ai modi di accesso.
  - `F_SETFL`: pone ad *arg* i flag dello stato corrente del file.
  - `F_GETOWN`: restituisce l'identificatore del processo o il gruppo di processo che è al momento destinato a ricevere i segnali `SIGIO/SIGURG`. Se il valore è positivo si riferisce all'ID di un processo. Se è negativo, il suo valore assoluto si riferisce a un gruppo di processo.
  - `F_SETOWN`: pone ad *arg* l'identificatore del processo o il gruppo di processo che dovrebbe ricevere i segnali `SIGIO/SIGURG` (vale quanto detto per `F_GETOWN`).
- `fcntl()` restituisce `-1` se non ha successo.

Controllare i dispositivi

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

- `ioctl()` esegue l'operazione *cmd* sul file associato al descrittore *fd*. *arg* è un argomento opzionale per *cmd*.
- I valori validi per *cmd* dipendono dal dispositivo a cui *fd* fa riferimento e sono tipicamente documentati nel manuale d'istruzioni del dispositivo.
- `ioctl()` restituisce `-1` se non ha successo.

Gestione dei processi

- In UNIX ogni processo ha i seguenti attributi:
  - un qualche codice,
  - alcuni dati,
  - una pila,
  - un numero identificativo unico (*PID*).

- Quando il SO comincia la sua esecuzione, c'è un solo processo visibile chiamato `init` che ha PID `1`.

- Poiché l'unico modo di creare un nuovo processo in UNIX è quello di duplicare un processo esistente (tramite una `fork()`), `init` è l'antenato comune di tutti i processi esistenti in un dato momento nel sistema.

- Quando un processo è duplicato, il processo *padre* ed il processo *figlio* sono virtualmente identici (eccetto per alcuni aspetti quali PID, PPID e risorse a run-time): il codice, i dati e la pila del figlio sono una copia di quelli del padre ed il processo figlio continua ad eseguire lo stesso codice del padre.

Gestione dei processi

- Un processo figlio tuttavia può rimpiazzare il suo codice con quello di un altro file eseguibile (tramite una `exec()`), perciò differenziandosi dal padre.

  Esempio: `init()` crea i processi `getty()` responsabili di gestire i *login* degli utenti.

- Solitamente un processo padre si sospende (tramite una `wait()`) in attesa della terminazione del figlio.

  Esempio: una shell che esegue un comando in foreground.

- Quando un processo figlio termina (tramite una `exit()`), la sua *terminazione* è comunicata al padre (tramite un *segnale*) e questi si comporta di conseguenza.

Creare un nuovo processo: `fork()`

```
pid_t fork (void)
```

- `fork()` permette ad un processo di duplicarsi.

- Il processo figlio è *quasi* la copia esatta del processo padre da cui eredita una copia del codice, dei dati, della pila, dei descrittori di file aperti e della tabella dei segnali.

  Tuttavia, ha numeri PID e PPID differenti.

- Se `fork()` ha successo, restituisce il PID del figlio al padre ed il valore `0` al figlio; altrimenti, restituisce `-1` al padre e non crea alcun figlio.

- La particolarità della `fork()` sta nel fatto che è invocata da un processo ma restituisce due processi. Entrambi i processi continuano ad eseguire lo stesso codice concorrentemente ma hanno spazi dati e pile separate.

Ottenere il PID: `getpid()` e `getppid()`

```
pid_t getpid (void)

pid_t getppid (void)
```

- `getpid()` restituisce il PID del processo invocante.
- `getppid()` restituisce il PPID (cioè il PID del padre) del processo invocante.
- Esse hanno sempre successo
- Il PPID del processo con PID `1` è `1`.

Esempio: `myfork.c`

Il seguente programma semplicemente si sdoppia e mostra PID e PPID dei due processi componenti.

```
$ cat myfork.c
#include <stdio.h>

int main (void) {
  int pid;
  printf ("I'm the original process with PID %d and PPID %d.\n",
            getpid (), getppid ());
  pid = fork (); /* Duplicate. Child and parent continue from here */
  if (pid != 0) { /* pid is non-zero, so I must be the parent */
     printf ("I'm the parent process with PID %d and PPID %d.\n",
               getpid (), getppid ());
     printf ("My child's PID is %d.\n", pid);
  }
  else { /* pid is zero, so I must be the child */
     printf ("I'm the child process with PID %d and PPID %d.\n",
               getpid (), getppid ());
  }
  printf ("PID %d terminates.\n", getpid () );
                                    /* Both processes execute this */
  return 0;
}
$ myfork
I'm the original process with PID 724 and PPID 572.
I'm the parent process with PID 724 and PPID 572.
I'm the child process with PID 725 and PPID 724.
PID 725 terminates.
My child's PID is 725
PID 724 terminates.
$
```

- In questo esempio, il padre non aspetta la terminazione del figlio per terminare a sua volta.

- Se un padre termina prima di un suo figlio, il figlio diventa *orfano* e viene automaticamente adottato dal processo `init()`.

Terminazione di un processo

A meno che non riceva un segnale o che il sistema vada in crash, un processo in esecuzione può terminare in uno tra 3 modi differenti:

- invoca la system call `exit()`
- durante l'esecuzione della funzione `main`, invoca `return`
- termina *implicitamente* al completamento dell'esecuzione della funzione `main`.

Terminazione di un processo: `exit()`

`void exit (int` *status*`)`

- `exit()` chiude tutti i descrittori di file di un processo, dealloca il suo codice, i suoi dati e la sua pila, e quindi fa terminare il processo.

- Quando un figlio termina, spedisce al padre un segnale `SIGCHLD` ed aspetta che il suo codice di terminazione *status* sia accettato.

  Solo gli 8 bit meno significativi di *status* sono utilizzati, così i valori del codice vanno da `0` a `255`.

- Un processo accetta il codice di terminazione di un figlio eseguendo una `wait()`.

- Il codice di terminazione di un processo figlio può essere usato dal processo padre per vari scopi.

  Esempio: le shell possono accedere il codice di terminazione dell'ultimo processo figlio terminato tramite una variabile speciale (la C-shell usa `status`).

- `exit()` non restituisce mai il controllo al chiamante.

Terminazione di un processo: `exit()`

- Se un padre termina prima di un suo figlio, il figlio diventa *orfano*.

  Il nucleo del SO assicura che tutti i processi *orfani* siano adottati da `init()` ed assegna loro PPID `1`. `init()` accetta sempre i codici di terminazione dei suoi figli.

- Un processo che termina non può lasciare il sistema fino a che il padre non avrà accettato il suo codice di terminazione. Un processo che sta aspettando che il padre accetti il suo codice di terminazione è chiamato processo *zombi*.

- Se il padre non è ancora terminato ma non esegue mai una `wait()`, il codice di terminazione non sarà mai accetto ed il processo è destinato a restare uno zombi.

- Uno zombi non ha codice, dati o pila, quindi non usa molte risorse di sistema ma continua ad avere una tabella di processo di grandezza fissa (la presenza di molti processi zombi potrebbe costringere l'amministratore ad intervenire).

Attendere la terminazione: `wait()`

```
pid_t wait (int *status)
```

- `wait()` sospende un processo fino a che uno dei suoi figli termina.
- Se `wait()` ha successo, restituisce il PID del figlio che è terminato e gestisce *status* (passato per riferimento) nel modo seguente:
  - se il byte meno significativo di *status* è `0`, il byte alla sua sinistra conterrà gli 8 bit meno significativi del valore restituito dalla `exit()` o dalla `return()` eseguita dal figlio (cioè il suo codice di terminazione);
  - altrimenti, i 7 bit meno significativi di *status* sono posti al numero del segnale che ha causato la terminazione del figlio, ed il bit rimanente del byte meno significativo è posto ad 1 se il figlio ha prodotto un *core dump*.
- Se un processo esegue una `wait()` e non ha figli, la `wait()` termina immediatamente con valore `-1`. Se un processo esegue una `wait()` ed uno o più dei suoi figli sono già zombi, `wait()` termina immediatamente con lo stato di uno degli zombi.

Differenziare un processo: `exec()`

```
int execl (const char *path, const char *arg0, ...,
           const char *argn, NULL)
int execv (const char *path, const char *argv[])
int execlp (const char *path, const char *arg0, ...,
           const char *argn, NULL)
int execvp (const char *path, const char *argv[])
```

- La famiglia di system call `exec()` rimpiazza il codice, i dati e la pila del processo invocante con quelli dell'eseguibile il cui pathname è immagazzinato in *path*.

  In effetti sono tutte routine di libreria C che invocano la vera system call `execve()`.

- `execl()` è identica a `execlp()` e `execv()` è identica a `execvp()` a parte per il fatto che `execl()` e `execv()` richiedono che sia fornito il pathname assoluto o quello relativo del file eseguibile, mantre `execlp()` e `execvp()` usano la variabile d'ambiente `PATH` per trovare l'eseguibile *path*.

Differenziare un processo: `exec()`

- `execl()` e `execlp()` invocano l'eseguibile con gli argomenti *arg1*, ..., *argn*. *arg0* dev'essere il nome dello stesso eseguibile e la lista degli argomenti dev'essere terminata da `NULL`.

- `execv()` e `execvp()` invocano l'eseguibile con gli argomenti *arg[1]*, ..., *arg[n]*. *arg[0]* dev'essere il nome dello stesso eseguibile e *arg[n+1]* dev'essere `NULL`.

- Se l'eseguibile non viene trovato, viene restituito `-1`; altrimenti, il processo invocante rimpiazza il suo codice, i suoi dati e la sua pila con quelli dell'eseguibile e comincia ad eseguire il nuovo codice.

  Una `exec()` che ha successo non restituisce mai il controllo al chiamante (quindi, quando `exec()` termina, termina anche il chiamante).

Differenziare un processo: esempio

Il programma seguente mostra un messaggio e quindi rimpiazza il suo codice con quello dell'eseguibile `ls` (con opzione `-l`). Si noti che nessuna `fork()` è invocata.

Si noti che la `execl()` è stata eseguita con successo e perciò non restituisce il controllo all'invocante (difatti il secondo `printf()` non è mai eseguito).

```
$ cat myexec.c
#include <stdio.h>

int main (void) {
  printf("I'm process %d and I'm about to exec an ls -l\n", getpid());
  execl("/bin/ls", "ls", "-l", NULL); /* Execute ls -l */
  printf("This line should never be executed\n");
}
$ myexec
I'm process 797 and I'm about to exec an ls -l
total 3324
drwxr-xr-x   3 rossi   users        4096 May 16 19:14 Glass
-rwxr-xr-x   1 rossi   users       22199 Mar 17 18:34 lez1.tex
-rwxr-xr-x   1 rossi   users       25999 May 21 17:14 Lez20.tex
$
```

Esempio: elaborazione in background

Il programma seguente usa `fork()` e `exec()` per eseguire un programma in background.

Si noti che la lista degli argomenti è passata da `main()` ad `execvp()` tramite il secondo argomento `&argv[1]` e che l'uso di `execvp()` permette di usare la variabile `PATH` per trovare l'eseguibile.

```
$ cat background.c
#include <stdio.h>
int main (int argc, char *argv[]) {
  if (fork () == 0) {      /* Child */
     execvp (argv[1], &argv[1]); /* Execute other program */
     fprintf (stderr, "Could not execute %s\n", argv[1]);
  }
}
$ background cc myexec.c -o myexec
$ ps
  PID TTY          TIME CMD
  572 pts/0    00:00:00 bash
  593 pts/0    00:00:28 xemacs
  602 pts/0    00:00:01 xdvi.bin
  746 pts/0    00:00:01 xemacs
  822 pts/0    00:00:00 xterm
  925 pts/0    00:00:01 xemacs
  964 pts/0    00:00:00 cc
  965 pts/0    00:00:00 gcc.colorgcc
  969 pts/0    00:00:00 ps
  970 pts/0    00:00:00 collect2
  971 pts/0    00:00:00 ld
$
```

Questo funziona perché, in un programma C, vale sempre che `argv[argc+1] = NULL`.

Cambiare directory: `chdir()`

```
int chdir (const char *pathname)

int fchdir (int fd)
```

- Ogni processo ha una *directory di lavoro corrente* che è usata per interpretare pathname relativi.

  Un processo figlio eredita la directory di lavoro corrente del padre.

- `chdir()` modifica la directory di lavoro corrente di un processo in *pathname*.

  `fchdir()` opera come `chdir()` ma prende come argomento un descrittore di file aperto *fd*.

- Affinché `chdir()` abbia successo, il processo invocante deve avere permessi di esecuzione (`x`) in tutte le directory nel path della directory. In questo caso, `chdir()` restituisce `0`; altrimenti, restituisce `-1`.

Modificare le priorità: `nice()`

`int nice (int` *delta*`)`

- `nice()` aggiunge *delta* al valore corrente della priorità del processo invocante.

  La priorità di un processo influenza la quantità di tempo di CPU che è allocata al processo.

- I valori legali della priorità vanno da `-20` a `+19`; se viene specificato un *delta* che porta oltre questi limiti il valore è troncato al limite.

  In generale, più piccolo è il valore, più veloce è il processo.

- Solo i processi del nucleo del SO o di un superutente possono avere priorità negativa.

  Le shell di login cominciano con una priorità pari a `0`.

- Se `nice()` ha successo, restituisce il nuovo valore della priorità; altrimenti restituisce `-1` (che può causare ambiguità con la corrispondente priorità).

Esempio: uso del disco

L'esempio seguente conta il numero dei file che non sono directory all'interno di una gerarchia.

Il programma prende come argomento il nome della directory da cui cominciare a cercare, crea un processo per ogni entry della directory e termina restituendo la somma totale dei codici di uscita dei figli. Ogni processo figlio o termina con valore `1`, se la entry corrispondente non è una directory, o, altrimenti, ripete lo stesso comportamento del padre.

Svantaggi di questa tecnica: crea molti processi e può contare fino ad un massimo di 255 file per ogni directory (usa rappresentazioni ad 8 bit).

Esempio: uso del disco

```
$ cat count.c
#include <stdio.h>
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>

long processFile (char *);
long processDirectory (char *);

int main (int argc, char *argv[]) {
  long count;
  count = processFile(argv[1]);
  printf ("Total number of non-directory files is %ld\n", count);
  return (/* EXIT_SUCCESS */ 0);
}

long processFile (char *name) {
  struct stat statBuf; /* To hold the return data from stat () */
  mode_t mode;
  int result;
  result = stat (name, &statBuf); /* Stat the specified file */
  if (result == -1) return (0); /* Error */
  mode = statBuf.st_mode; /* Look at the file's mode */
  if (S_ISDIR (mode)) /* Directory */
      return (processDirectory(name));
  else
      return (1); /* A non-directory file was processed */
}
```

Esempio: uso del disco

```
long processDirectory (char *dirName) {
  int fd, children, i, charsRead, childPid, status;
  long count, totalCount;
  char fileName [100];
  struct dirent dirEntry;
  fd = open (dirName, O_RDONLY); /* Open directory for reading */
  children = 0; /* Initialize child process count */
  while (1) { /* Scan directory */
     charsRead = getdents (fd, &dirEntry, sizeof (struct dirent));
     if (charsRead == 0) break; /* End of directory */
     if (strcmp (dirEntry.d_name, ".") != 0 &&
         strcmp (dirEntry.d_name, "..") != 0) {
         if (fork () == 0) { /* Create a child to process dir. entry */
             sprintf (fileName, "%s/%s", dirName, dirEntry.d_name);
             count = processFile (fileName);
             exit (count);
         } else
           ++children; /* Increment count of child processes */
     }
     lseek (fd, dirEntry.d_off, SEEK_SET); /* Jump to next dir.entry */
  }
  close (fd); /* Close directory */
  totalCount = 0; /* Initialize file count */
  for (i = 1; i <= children; i++) { /* Wait for children to terminate*/
     childPid = wait (&status); /* Accept child's termination code */
     totalCount += (status >> 8); /* Update file count */
  }
  return (totalCount); /* Return number of files in directory */
}
$ count ..
Total number of non-directory files is 213
$
```

Esempio: redirezione

Il programma seguente quando invocato con un nome di file come primo parametro e con un comando ed i suoi argomenti come parametri successivi ridirige lo standard output del programma nel file passato come parametro.

```
$ cat redirect.c
#include <stdio.h>
#include <fcntl.h>

int main (int argc, char *argv[]) {
  int fd;
  /* Open file for redirection */
  fd = open (argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
  dup2 (fd, 1); /* Duplicate descriptor to standard output */
  close (fd); /* Close original descriptor to save descriptor space */
  execvp (argv[2], &argv[2]); /* Invoke program; will inherit stdout */
  perror ("main"); /* Should never execute */
}
$ redirect lista ls -l
-rw-------   1 pugliese users          0 May 22 18:02 lista
-rwxr-xr-x   1 pugliese users       4314 May 22 18:02 redirect
-rw-r--r--   1 pugliese users        432 May 22 09:12 redirect.c
-rwxr-xr-x   1 pugliese users       6477 May 19 17:52 reverse
-rw-r--r--   1 pugliese users       5125 May 19 17:52 reverse.c
$
```

Thread

- A volte, per eseguire un determinato task, non è necessario creare un processo con un insieme di risorse separato ma è sufficente creare un processo che condivide spazio di memoria e dispositivi aperti con altri processi.

- Un *thread* (o light-weight process) è un'astrazione che permette di creare *thread di controllo multipli* in un unico spazio di processi (una sorta di "processo dentro un processo").

- Le implementazioni dei thread variano notevolmente ma tutte forniscono le seguenti funzionalità
  - *create*: per creare thread;
  - *join*: per permettere ad un thread di sospendersi ed aspettare che un thread creato termini;
  - *terminate*: per permettere ad un thread di rilasciare le sue risorse al sistema quando ha finito (richiede un *join*);
  - *detach*: per permettere ad un thread di rilasciare le sue risorse al sistema quando ha finito e non richiede un *join*.

Thread

- Per la *sincronizzazione* dei thread, oltre ai meccanismi di IPC standard di UNIX, molte implementazioni mettono anche a disposizione primitive specifiche.

- Per gestire la mutua esclusione tra thread, si può usare un oggetto *mutex*. Gli oggetti mutex possono essere creati, distrutti, bloccati e sbloccati. Gli attributi di un tale oggetto sono condivisi tra thread e sono usati per permettere ad un altro thread di conoscere lo stato del thread che l'oggetto mutex descrive.

- Gli oggetti mutex possono essere usati congiuntamente alle *variabili condizionali* che conservano dei valori che permettono una gestione più precisa della sincronizzazione tra thread.

- Una libreria di funzioni è detta *thread-safe* quando è scritta in modo da poter essere utilizzata da un programma multithread.

Segnali

- I programmi talvolta devono fronteggiare eventi inaspettati o impredicibili, quali:
  - mancanza di corrente elettrica
  - terminazione di un processo figlio
  - richiesta di terminazione da parte di un utente (Ctrl-C)
  - richiesta di sospensione da parte di un utente (Ctrl-Z).

- Tali eventi sono chiamati *interrupt* poiché, per essere elaborati, si deve interrompere il flusso regolare di un programma.

  Quando UNIX si rende conto che un tale evento si è verificato, spedisce un *segnale* al processo corrispondente.

- C'è un unico segnale numerato per ogni possibile evento.

- Un processo può spedire un segnale ad ogni altro processo, purché abbia i permessi per farlo.

Segnali

- I segnali sono definiti nei file `signal.h`

- Un programmatore può fare in modo che un particolare segnale sia ignorato oppure che sia elaborato da un codice speciale chiamato *signal handler*.

- In quest'ultimo caso, il processo che riceve il segnale sospende il suo flusso di esecuzione corrente, esegue il signal handler, e quindi riprende il flusso di controllo originario quando il signal handler termina.

- Il signal handler può essere definito dal programmatore o può essere fornito dal kernel.

  L'handler di default di solito esegue una delle azioni seguenti:

  – termina il processo e genera un file core (*dump*)
  – termina il processo senza generare un file core (*quit*)
  – ignora il segnale e lo cancella (*ignore*)
  – sospende il processo (*suspend*)
  – riprende l'esecuzione del processo.

Segnali

Alcuni segnali predefiniti.

```
Macro      #  Default Description
SIGHUP     1  quit    hang up
SIGINT     2  quit    interrupt
SIGQUIT    3  dump    quit
SIGILL     4  dump    illegal instruction
SIGTRAP    5  dump    trace trap
SIGABRT    6  dump    abort
SIGEMT     7  dump    emulator trap instruction
SIGFPE     8  dump    arithmetic execption
SIGKILL    9  quit    kill (cannot be caught, blocked or ignored)
SIGBUS    10  dump    bus error (bad format address)
SIGSEGV   11  dump    segmentation violation (out-of-range address)
SIGSYS    12  dump    bad argument to system call
SIGPIPE   13  quit    write on a pipe or other socket with no one
                                                        to read it
SIGALRM   14  quit    alarm clock
SIGTERM   15  quit    software termination signal
SIGUSR1   16  quit    user signal 1
SIGUSR2   17  quit    user signal 2
SIGCHLD   18  ignore  child status changed
SIGPWR    19  ignore  power fail or restart
SIGWINCH  20  ignore  window size change
SIGURG    21  ignore  urgent socket condition
SIGPOLL   22  exit    pollable event
SIGSTOP   23  quit    stopped (signal)
SIGSTP    24  quit    stopped (user)
SIGCONT   25  ignore  continued
SIGTTIN   26  quit    stopped (tty input)
SIGTTOU   27  quit    stopped (tty output)
SIGVTALRM 28  quit    virtual timer expired
SIGPROF   29  quit    profiling timer expired
SIGXCPU   30  dump    CPU time limit exceeded
SIGXFSZ   31  dump    file size limit exceeded
```

- Il modo più semplice per inviare un segnale ad un processo in foreground è premere Ctrl-C o Ctrl-Z dalla tastiera.

- Quando il driver di un terminale, cioè il software che supporta il terminale, riconosce che Ctrl-C (Ctrl-Z) è stato premuto, spedisce un segnale `SIGINT` (`SIGSTP`) a tutti i processi nel gruppo del processo corrente in foreground.

- Alcune corrispondenze
  - `SIGINT`: l'utente ha digitato Ctrl-C;
  - `SIGCHLD`: inviato al padre da un figlio che termina;
  - `SIGSTOP`: sospensione da dentro un programma;
  - `SIGSTP`: l'utente ha digitato Ctrl-Z;
  - `SIGCONT`: riprende l'esecuzione di un programma dopo una sospensione.

  - I segnali `SIGKILL` e `SIGSTP` non possono essere riprogrammati.

Richiedere un segnale di allarme: `alarm()`

`unsigned int alarm (unsigned int` *count*`)`

- `alarm()` istruisce il nucleo a spedire il segnale `SIGALRM` al processo invocante dopo *count* secondi.

- Se un `alarm()` è già stato schedulato, viene sovrascritto col nuovo.

- Se *count* è `0`, nessun nuovo `alarm()` è schedulato (quello eventualmente già schedulato è cancellato).

- `alarm()` restituisce il numero dei secondi che restano fino all'invio del segnale corrispondente oppure `0` se nessun `alarm()` è schedulato.

Gestire i segnali: `signal()`

```
void (*signal(int signum, void (*handler)(int)))(int)
```

o, equivalentemente,

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int signum, sighandler_t handler)
```

- `signal()` permette di installare un nuovo signal handler, *handler*, per il segnale con numero *signum*.

- L'handler può essere o l'indirizzo di una funzione definita dall'utente o uno dei seguenti valori:
  - `SIGN_IGN`: che indica che il segnale dev'essere ignorato;
  - `SIGN_DFL`: che indica che deve essere usato l'handler di default fornito dal nucleo.

- La funzione *handler* prende un argomento `int` (il numero del segnale); in questo modo, si può utilizzare lo stesso handler per segnali differenti.

- `signal()` restituisce il precedente l'handler associato con *signum*, se ha successo; altrimenti, restituisce `-1`.

- Quando in una dichiarazione di parametro si ha una funzione, essa viene interpretata dal compilatore come un puntatore.

  In pratica, questo vuol dire che l'intestazione della `signal()` è equivalente a quella senza `*`.

Gestire i segnali: `signal()`

- I segnali `SIGKILL` e `SIGSTP` non possono essere riprogrammati.

- In seguito ad una `fork()`, il processo figlio eredita le politiche di gestione dei segnali del padre. Quando il figlio esegue una `exec()`, i segnali precedentemente ignorati continuano ad essere ignorati, ma quelli trattati da handler ridefiniti vengono ora trattati dagli handler di default.

- Ad eccezione di `SIGCHLD`, i segnali non sono accodati; ciò implica che se più segnali arrivano contemporaneamente ad un processo, solo uno viene trattato.

Gestire i segnali: esempio

Il seguente programma mostra come proteggere pezzi di codice critici da interruzioni dovute a Ctrl-C (segnale `SIGINT`) o altri segnali simili.

In questi casi, di solito si salva il precedente valore dell'handler in modo da poterlo ripristinare quando il codice critico è stato eseguito.

```
$ cat critical.c
#include <stdio.h>
#include <signal.h>

int main (void) {
  void (*oldHandler) (int); /* To hold old handler value */
  printf ("I can be Control-C'ed\n");
  sleep (3);
  oldHandler = signal (SIGINT, SIG_IGN); /* Ignore Control-C */
  printf ("I'm protected from Control-C now\n");
  sleep (3);
  signal (SIGINT, oldHandler); /* Restore old handler */
  printf ("I can be Control-C'ed again\n");
  sleep (3);
  printf ("Bye!\n");
  return 0;
}
$ critical
I can be Control-C'ed
I'm protected from Control-C now
I can be Control-C'ed again
Bye!
$
```

Inviare segnali: `kill()`

`int kill(pid_t` *pid*`, int` *signum*`)`

- `kill()` spedisce il segnale con valore *signum* al processo con PID *pid*.
- `kill()` ha successo ed il segnale viene spedito allorquando almeno una delle due seguenti condizioni è soddisfatta:
  - i processi mittente e destinatario hanno lo stesso proprietario;
  - il mittente ha per proprietario un superutente.

Inviare segnali: `kill()`

- Ci sono alcune variazioni sul modo in cui `kill()` opera:
  - se *pid* è positivo, il segnale è inviato al processo *pid*;
  - se *pid* è `0`, il segnale è inviato a tutti i processi nel gruppo di processi del mittente;
  - se *pid* è `-1` ed il mittente ha per proprietario un superutente, il segnale è inviato a tutti i processi, mittente incluso;
  - se *pid* è `-1` ed il mittente non ha per proprietario un superutente, il segnale è inviato a tutti quei processi appartenenti allo stesso proprietario del mittente, con esclusione del processo mittente;
  - se *pid* è negativo, ma non è `-1`, il segnale è inviato a tutti i processi nel gruppo di processi il cui numero identificativo è il valore assoluto di *pid*.

- `kill()` restituisce valore `0`, se invia con successo almeno un segnale; altrimenti, restituisce `-1`.

Esempio: terminazione di un figlio

- Il seguente programma definisce un handler per il segnale `SIGCHLD` e permette all'utente di limitare il tempo impiegato da un comando per l'esecuzione.

- Il primo parametro di `limit.c` è il massimo numero di secondi concesso per l'esecuzione, i rimanenti parametri costituiscono il comando da eseguire.

- N.B. Il codice del programma contiene dichiarazione ed installazione dell'handler, ma nessuna invocaziuone esplicita.

Esempio: terminazione di un figlio

```
$ cat limit.c
#include <stdio.h>
#include <signal.h>

int delay;
void childHandler (int);

int main (int argc, char *argv[]) {
  int pid;
  signal (SIGCHLD, childHandler); /* Install death-of-child handler */
  pid = fork (); /* Duplicate */
  if (pid == 0) { /* Child */
    execvp (argv[2], &argv[2]); /* Execute command */
    perror ("limit"); /* Should never execute */
  }
  else { /* Parent */
    sscanf (argv[1], "%d", &delay); /* Read delay from command line */
    sleep (delay); /* Sleep for the specified number of seconds */
    printf ("Child %d exceeded limit and is being killed\n", pid);
    kill (pid, SIGINT); /* Kill the child */
  }
  return 0;
}

void childHandler (int sig) {   /* Executed if the child dies */
  int childPid, childStatus;    /* before the parent */
  childPid = wait (&childStatus); /* Accept child's termination code */
  printf ("Child %d terminated within %d seconds\n", childPid, delay);
  exit (/* EXITSUCCESS */ 0);
}
```

Esempio: terminazione di un figlio

```
$ limit 3 find / -name pippo -print
find: /tmp/kfm-cache-503: Permission denied
find: /tmp/kfm-cache-504: Permission denied
find: /tmp/nscomm40-bettini/1902: Permission denied
...
Child 13754 exceeded limit and is being killed
$

...

$ ./limit 1 ls ~/public_html/
didattica.html     lez13.ps           lez2.ps         lez8.ps
foto.jpg           lez14.ps           lez20.ps        lez9.ps
index.html         lez15.ps           lez21.ps        progett1.ps
lez1.ps            lez16.ps           lez3.ps         progett2.ps
lez10.ps           lez17.ps           lez5.ps         publications.html
lez11.ps           lez18.ps           lez6.ps
lez12.ps           lez19.ps           lez7.ps
Child 27059 terminated within 1 seconds
$
```

Esempio: sospensione e ripresa

Il seguente programma crea due figli che entrano in un ciclo infinito e mostrano un messaggio ogni secondo.

Il programma padre aspetta 3 secondi e quindi sospende il primo figlio. Il secondo figlio continua come al solito.

Dopo altri 3 secondi il padre riesuma il primo figlio, aspetta altri 3 secondi e quindi uccide entrambi i figli.

```
$ cat pulse.c
#include <signal.h>
#include <stdio.h>

int main (void) {
  int pid1;
  int pid2;
  pid1 = fork ();
  if (pid1 == 0) { /* First child */
    while (1) { /* Infinite loop */
        printf ("pid1 is alive\n");
        sleep (1);
      }
  }
  pid2 = fork ();
  if (pid2 == 0) { /* Second child */
    while (1) { /* Infinite loop */
        printf ("pid2 is alive\n");
        sleep (1);
      }
  }
```

Esempio: sospensione e ripresa

```
  sleep (3);
  kill (pid1, SIGSTOP); /* Suspend first child */
  sleep (3);
  kill (pid1, SIGCONT); /* Resume first child */
  sleep (3);
  kill (pid1, SIGINT); /* Kill first child */
  kill (pid2, SIGINT); /* Kill second child */
  return 0;
}
$ pulse
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid2 is alive
pid2 is alive
pid2 is alive
pid1 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
$
```

Gruppi di processi e terminali di controllo

- Ogni processo è membro di un *gruppo di processi*. Ad uno stesso gruppo possono appartenere molti processi. I figli ereditano il gruppo di appartenenza del padre. Per cambiare gruppo di appartenenza, un processo può invocare `setpgid()`.

- Ad ogni processo può essere associato un *terminale di controllo*, che è tipicamente il terminale da cui il processo è lanciato. I figli ereditano il terminale di controllo del padre. Anche se un processo esegue una `exec()`, il terminale di controllo non cambia.

- Ad ogni terminale è associato un solo *processo di controllo*. Quando un metacarattere tipo Ctrl-C è individuato, il terminale spedisce il segnale appropriato a tutti i processi nel gruppo di processi del suo processo di controllo.

- Se un processo tenta di leggere dal suo terminale di controllo e non è membro del gruppo del processo che controlla quel terminale, il processo riceve un segnale `SIGTTIN` che, normalmente, lo sospende.

Ecco come sono usate tali caratteristiche.

- Quando una shell interattiva comincia la sua esecuzione, essa è il processo di controllo di un terminale ed ha quel terminale come terminale di controllo.

- Quando una shell esegue un comando in foreground, la shell figlia si mette in un gruppo di processi differente prima di eseguire il comando e quindi assume il controllo del terminale. Qualsiasi segnale generato dal terminale viene così indirizzato al comando e non alla shell originaria. Quando il comando termina, la shell originaria riprende il controllo del terminale.

- Quando una shell esegue un comando in background, la shell figlia si mette in un gruppo di processi differente prima di eseguire il comando, ma non assume il controllo del terminale. Qualsiasi segnale generato dal terminale continua ad essere indirizzato alla shell originaria. Se il comando in background tenta di leggere dal suo terminale di controllo, viene sospeso da un segnale `SIGTTIN`.

Cambiare il gruppo: `setpgid()`

`int setpgid(pid_t` *pid*`, pid_t` *pgrpId*`)`

- `setpgid()` pone al valore *pgrpId* l'ID del gruppo di processi a cui appartiene il processo con PID *pid*.

  Se *pid* è `0`, al posto del processo con PID *pid* usa il processo invocante.

  Se *pgrpId* è `0`, al posto di *pgrpId* usa l'ID del gruppo del processo con PID *pid*.

- Affinché `setpgid()` abbia successo, deve verificarsi almeno una delle due condizioni seguenti:
  - il processo invocante ed il processo specificato come primo argomento hanno lo stesso proprietario;
  - il proprietario del processo invocante è un superutente.

- Quando un processo vuole dare origine ad un proprio gruppo di processi distinto dagli altri gruppi nel sistema, tipicamente passa il proprio PID come secondo argomento a `setpgid()`.

- `setpgid()` restituisce `0` se ha successo; altrimenti, restituisce `-1`.

Ottenere il gruppo: `getpgid()`

`pid_t getpgid(pid_t` *pid*`)`

- `getpgid()` restituisce il gruppo di processi a cui appartiene il processo con PID *pid*.
- Se *pid* è `0`, `getpgid()` restituisce il gruppo di processi a cui appartiene il processo invocante.
- `getpgid()` non fallisce mai.

Esempio: gruppo del processo di controllo

Il programma seguente mostra che un terminale invia i segnali ad ogni processo appartenente al gruppo di processi del suo processo di controllo.

Poiché il figlio eredita il gruppo di processi del padre, sia il padre che il figlio catturano il segnale `SIGINT`.

```
$ cat pgrp1.c
#include <signal.h>
#include <stdio.h>

void sigHandler (int sig) {
  printf ("Process %d got a %d signal \n", getpid (), sig);
}

int main (void) {
  signal (SIGINT, sigHandler); /* Handle Control-C */
  if (fork () == 0)
    printf ("Child PID %d PGRP %d waits\n", getpid (), getpgid (0));
  else
    printf ("Parent PID %d PGRP %d waits\n", getpid (), getpgid (0));
  pause (); /* Wait for a signal */
}
$ pgrp1
Parent PID 24444 PGRP 24444 waits
Child PID 24445 PGRP 24444 waits
<^C> Process 24445 got a 2 signal
Process 24444 got a 2 signal
$
```

- `SIGINT` viene automaticamente passato a `sigHandler`.

- `pause()` sospende l'invocante in attesa dell'arrivo di un segnale.

Esempio: gruppo non di controllo

Il programma seguente mostra che se un processo lascia il gruppo di processi a cui appartiene il processo che controlla il terminale, allora non riceverà più segnali dal terminale.

Nell'esempio, il figlio non sarà influenzato da Ctrl-C.

```
$ cat pgrp2.c
#include <signal.h>
#include <stdio.h>
void sigHandler (int sig) {
  printf ("Process %d got a SIGINT\n", getpid ());
  exit (1);
}
int main (void) {
  int i;
  signal (SIGINT, sigHandler); /* Install signal handler */
  if (fork () == 0)
    setpgid (0, getpid ()); /* Place child in its own process group */
  printf ("Process PID %d PGRP %d waits\n", getpid (), getpgid (0));
  for (i = 1; i <= 3; i++) { /* Loop three times */
    printf ("Process %d is alive\n", getpid ());
    sleep(2);
  }
  return 0;
}
$ pgrp2
Process PID 24535 PGRP 24535 waits
Process 24535 is alive
Process PID 24536 PGRP 24536 waits
Process 24536 is alive
<^C> Process 24535 got a SIGINT
$ Process 24536 is alive
Process 24536 is alive
<return>
$
```

Esempio: segnale SIGTTIN

Il programma seguente mostra che se un processo lascia il gruppo di processi a cui appartiene il processo che controlla il terminale e quindi tenta di leggere dal terminale, gli viene inviato un segnale SIGTTIN che ne provoca la sospensione.

Nell'esempio, l'handler per SIGTTIN (21) viene riprogrammato.

```
$ cat pgrp3.c
#include <signal.h>
#include <stdio.h>
#include <sys/termio.h>
#include <fcntl.h>
void sigHandler (int sig) {
  printf ("%d Attempted inappropriate read from control terminal\n",
           sig);
  exit (1);
}
int main (void) {
  int status;
  char str [100];
  if (fork () == 0) { /* Child */
    signal (SIGTTIN, sigHandler); /* Install handler */
    setpgid (0, getpid ()); /* Place myself in a new process group */
    printf ("Enter a string: ");
    scanf ("%s", str); /* Try to read from control terminal */
    printf ("You entered %s\n", str);
  } else { /* Parent */
    wait (&status); /* Wait for child to terminate */
  }
}
$ pgrp3
Enter a string: 21 Attempted inappropriate read from control terminal
$
```

- La metavariabile `%s` richiede la lettura dei caratteri fino al primo carattere di spazio.

IPC: Interprocess Communication

- Termine generico che indica lo scambio di informazioni tra due o più processi.

- I processi che comunicano possono risiedere sulla stessa macchina o su macchine diverse (sebbene alcuni meccanismi, quali segnali e pipe, possono essere usati solo per la comunicazione tra processi che risiedono sulla stessa macchina).

- La comunicazione può coinvolgere lo scambio di dati tra i processi che cooperano attivamente alla elaborazione dei dati o semplicemente lo scambio di informazioni di sincronizzazione per aiutare due processi indipendenti, ma correlati, a schedulare le loro attività in modo da non sovrapporsi.

Pipe

- Tale meccanismo è frequentemente usato all'interno delle shell per connettere l'output di un comando all'input di un altro (*pipeline*).

  ```
  $ who | sort
  ```

- È importante tener presente che i due processi connessi da un pipeline sono eseguiti concorrentemente. Un pipe automaticamente memorizza l'output dello scrittore in un buffer e sospende lo scrittore se il buffer diventa pieno. Allo stesso modo, se il buffer è vuoto, il pipe sospende il lettore fino a che dell'output diventa disponibile.

- Tutte le versioni di UNIX supportano i pipe *anonimi*, che sono il tipo di pipe che usano le shell.

  Le versioni di UNIX System V supportano anche pipe con nome.

Pipe anonimi: `pipe()`

- Un pipe anonimo è un canale di comunicazione unidirezionale che automaticamente memorizza il suo input in un buffer (la grandezza massima varia con le implementazioni ma è approssimativamente intorno ai 5K) e può essere creato invocando la system call `pipe()`.

- Ciascun lato di un pipe ha associato un descrittore di file.

- Il lato di scrittura può essere scritto invocando `write()`, quello di lettura può essere letto invocando `read()`.

- Quando un processo ha finito di utilizzare il descrittore di file del pipe, lo chiude invocando `close()`.

Pipe anonimi: `pipe()`

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

- `pipe()` crea un pipe anonimo e restituisce due descrittori di file: il descrittore associato con il lato di lettura del pipe è memorizzato in *fd[0]*, quello associato col lato di scrittura è memorizzato in *fd[1]*.

- Ai processi che leggono da un pipe si applicano le seguenti regole:
  - se un processo tenta di leggere da un pipe il cui lato di scrittura è stato chiuso, la `read()` restituisce `0` che indica la fine dell'input;
  - se un processo tenta di leggere da un pipe vuoto il cui lato di scrittura è ancora aperto, si sospende fino a che qualche input diventa disponibile;
  - se un processo tenta di leggere da un pipe più byte di quelli effettivamente presenti nel buffer associato, tutti i byte contenuti vengono letti e `read()` restituisce il numero dei byte effettivamente letti.

Pipe anonimi: `pipe()`

- Ai processi che scrivono in un pipe si applicano le seguenti regole:
  - se un processo scrive in un pipe il cui lato di lettura è già stato chiuso, la scrittura fallisce ed allo scrivente è inviato un segnale `SIGPIPE`, la cui azione di default è di far terminare il ricevente;
  - se un processo scrive meno byte di quelli che un pipe può contenere, la `write()` viene effettuata in maniera *atomica* (non possono avvenire interleaving dei dati scritti da processi diversi sullo stesso pipe);
  - se un processo scrive più byte di quelli che un pipe può contenere, non c'è nessuna garanzia di atomicità.

- `lseek()` non ha senso quando applicato ad un pipe.

- `pipe()` fallisce e restituisce `-1` quando il nucleo del SO non può allocare abbastanza spazio per un nuovo pipe; altrimenti restituisce `0`.

- Poiché l'accesso ad un pipe anonimo avviene tramite il meccanismo dei descrittori di file, solo il processo creante ed i suoi discendenti possono usare il pipe.

Pipe anonimi: `pipe()`

- La tipica sequenza di eventi è
  - il processo originario crea un pipe anonimo (`pipe()`);
  - il processo originario crea un figlio (`fork()`);
  - lo scrittore chiude il suo lato di lettura del pipe ed il lettore chiude il suo lato scrittura (`close()`);
  - i procesi comunicano usando `write()` e `read()`;
  - ogni processo chiude (`close()`) il suo descrittore quando ha finito.
- La comunicazione bidirezionale è possibile solo usando due pipe.

Esempio: scambio di un messaggio

Il programma seguente usa un pipe per permettere al padre di leggere un messaggio inviato dal figlio.

L'inclusione del carattere `NULL` facilita il lettore nel determinare la fine del messaggio.

```
$ cat talk.c
#include <stdio.h>
#define READ   0       /* The index of the read end of the pipe */
#define WRITE  1       /* The index of the write end of the pipe */
char *phrase = "Stuff this in your pipe";

int main (void) {
  int fd [2], bytesRead;
  char message [100]; /* Parent process' message buffer */
  pipe (fd);                /* Create an unnamed pipe */
  if (fork () == 0) {       /* Child, writer */
     close(fd[READ]);       /* Close unused end */
     write (fd[WRITE], phrase, strlen (phrase) + 1); /* include \0 */
     close (fd[WRITE]);     /* Close used end */
  } else { /* Parent, reader*/
     close (fd[WRITE]);     /* Close unused end */
     bytesRead = read (fd[READ], message, 100);
     printf ("Read %d bytes: %s\n", bytesRead, message);
     close (fd[READ]);      /* Close used end */
  }
}
$ ./talk
Read 24 bytes: Stuff this in your pipe
$
```

- Quando uno scrittore spedisce diversi messaggi di lunghezza variabile tramite un pipe, deve usare un protocollo per indicare al lettore la fine di ogni singolo messaggio.

- I metodi più comunemente usati per far ciò sono:
  - spedire la lunghezza del messaggio prima del messaggio stesso;
  - terminare un messaggio con un carattere speciale come `NULL` o un new-line.

Esempio: realizzazione di una pipeline

Il seguente programma esegue due comandi concorrentemente e connette l'output di uno all'input dell'altro.

Il programma assume che nessuno dei due comandi sia invocato con opzioni e che i nomi dei comandi siano passati come argomenti del programma principale.

```
$ cat connect.c
#include <stdio.h>
#define READ   0
#define WRITE  1

int main (int argc, char *argv []) {
  int fd [2];
  pipe (fd); /* Create an unamed pipe */
  if (fork () != 0) { /* Parent, writer */
     close (fd[READ]); /* Close unused end */
     dup2 (fd[WRITE], 1); /* Duplicate used end to stdout */
     close (fd[WRITE]); /* Close original used end */
     execlp (argv[1], argv[1], NULL); /* Execute writer program */
     perror ("connect");  /* Should never execute */
   } else { /* Child, reader */
     close (fd[WRITE]); /* Close unused end */
     dup2 (fd[READ], 0); /* Duplicate used end to stdin */
     close (fd[READ]); /* Close original used end */
     execlp (argv[2], argv[2], NULL); /* Execute reader program */
     perror ("connect"); /* Should never execute */
   }
}
$ connect ls sort
...
$
```

Pipe con nome

- I pipe con nome, chiamati anche *FIFO*, hanno meno restrizioni di quelli anonimi ed offrono i seguenti vantaggi:
  - esistono come file *speciali* nel file system;
  - esistono fino a che non sono esplicitamente rimossi;
  - il loro uso non è limitato a processi con antenati comuni; un pipe con nome può essere usato da qualunque processo che conosca il nome del file corrispondente.

- Inoltre essi hanno capacità di buffer fino a 40K (è il valore della macro `PIPE_BUF` definita in `limits.h`); tuttavia, sono supportati solo da varianti del System V.

- I pipe con nome sono file speciali e possono essere creati in uno dei seguenti due modi:
  - usando l'utility UNIX `mknod`;
  - usando la system call `mknod()`.

Utility `mknod`

`mknod [-m` *mode*`]` *name* `p`

- Il comando `mknod` permette ad un superutente di creare diversi tipi di file speciali.

- La sintassi sopra specificata, permette ad un utente qualsiasi di creare un file speciale di tipo pipe con nome (opzione `p`) individuato dal nome *name*.

- I diritti di accesso ad un pipe possono essere assegnati o contestualmente alla creazione del pipe con l'opzione `-m`, dove *mode* è la specifica dei diritti in cifre ottali, o successivamente alla creazione del pipe con il comando `chmod`, come per i file regolari.

- Un comando equivalente è `mkfifo [-m` *mode*`]` *name*.

Utility mknod

- ```
  $ mknod -m 0660 prova p
  $ ls -l prov*
  prw-rw----   1 rossi  users    0   May 28 10:35 prova
  $
  ```

  crea un pipe `prova` con diritti di lettura e scrittura per l'utente che impartisce il comando ed il suo gruppo.

- Una volta creato un pipe con nome si può operare su di esso scrivendo e leggendo dati.

- ```
  $  ls -l > prova &
  [3] 827
  $ more < prova
  total 292
  -rwxr-xr-x    1 pugliese users        4230 May 22  2000 background
  -rw-r--r--    1 pugliese users         220 May 22  2000 background.c
  -rwxr-xr-x    1 pugliese users        5203 May 29  2000 chef
  -rw-r--r--    1 pugliese users        1916 May 29  2000 chef.c
  -rwxr-xr-x    1 pugliese users        4482 May 25  2000 connect
  -rw-r--r--    1 pugliese users         786 May 25  2000 connect.c
  -rwxr-xr-x    1 pugliese users        4854 Jun  1  2000 cook
  -rw-r--r--    1 pugliese users        1569 Jun  1  2000 cook.c
  ...
  -rwxr-xr-x    1 pugliese users        4591 May 29  2000 writer
  -rw-r--r--    1 pugliese users         669 May 29  2000 writer.c
  [1]+  Done                    ls $LS_OPTIONS -l >prova
  $
  ```

  realizza il comando `ls -l | more` (il primo comando dev'essere lanciato in background poiché altrimenti l'apertura in scrittura del pipe si bloccherebbe per mancanza di lettori).

System call `mknod()`

`int mknod (const char *`*pathname*`, mode_t` *mode*`, dev_t` *dev*`)`

- La system call `mknod()` permette ad un superutente di creare un nodo del file system (file regolare, file speciale di dispositivo, pipe con nome).
- Un utente qualsiasi può invocare `mknod()` solo per creare un pipe con nome. In questo caso
  - *pathname* rappresenta il nome di file utilizzato per identificare il pipe con nome;
  - *mode* specifica i di diritti di accesso da assegnare al pipe, con le stesse convenzioni seguite per `open()`, ed il tipo di file da creare; in pratica è una congiunzione tramite `|` dei flag relativi ai diritti di accesso con il flag `S_IFIFO`, che indica che il nodo da creare è un pipe con nome.
  - *dev* è un parametro che viene ignorato nel caso dei pipe con nome, e quindi posto a `0`.
- `mknod()` restituisce `0` se ha successo; altrimenti, restituisce `-1`.
- La funzione di libreria

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

  permette di ottenere lo stesso risultato.
- I pipe con nome sono file, quindi, diversamente dai pipe anonimi, per poterli utilizzare vanno prima aperti come tutti i file.

- I flag di permesso del file sono:
  - `O_RDONLY`: apertura in sola lettura;
  - `O_WRONLY`: apertura in sola scrittura.

- I permessi sono modificati da `umask` nel solito modo: i permessi effettivi del file creato sono `mode & ~ umask`.

Pipe con nome: `open()`

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

- Per i FIFO, i parametri della `open()` tipicamente sono:
  - *pathname*: il nome del pipe che si vuole aprire (se non esiste, si ottiene l'errore `E_NOENT`);
  - *flags*: indicano il tipo di accesso al pipe
    - `O_RDONLY`: sola lettura;
    - `O_WRONLY`: sola scrittura;
    - `O_NONBLOCK`: apertura in modalità non bloccante: l'apertura in sola lettura avrà successo anche se il FIFO non è stato ancora aperto in scrittura, mentre l'apertura in sola scrittura fallirà (con errore `ENXIO`) a meno che il lato di lettura non sia già stato aperto.
- Normalmente, l'apertura di un FIFO in lettura blocca il processo fino a che il FIFO non viene aperto anche in scrittura, e viceversa.
- Se un processo tenta di scrivere su un FIFO che non è aperto in lettura, riceve un segnale `SIGPIPE`.
- `open()` restituisce un descrittore di file, se ha successo; altrimenti, restituisce `-1`.

Pipe con nome: `read()` e `write()`

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

- La lettura da un pipe con nome avviene come per i file regolari.
- Tuttavia, l'operazione è *bloccante*: qualora non fossero già stati scritti nel pipe i caratteri richiesti, la richiesta di lettura verrebbe sospesa fino al momento in cui i caratteri richiesti diventassero disponibili.
- Se non si vuole che la `read()` sia bloccante, e si vuole invece che la chiamata restituisca il controllo (con l'errore `EAGAIN`) anche nel caso che non ci siano sufficenti caratteri da leggere, bisogna specificare il flag `O_NONBLOCK` in fase di apertura del pipe con nome.

`ssize_t write (int` *fd*`, const void *`*buf*`, size_t` *count*`)`

- La scrittura su un pipe con nome avviene come per i file regolari.

Pipe con nome: `close()` e `unlink()`

`int close (int` *fd*`)`

- Un volta terminate le operazioni su di un pipe con nome, il suo descrittore *fd* viene chiuso il pipe con `close()` (come per i file regolari).

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

- Come per i file regolari, `unlink()` rimuove il link hard da *filename* al file relativo. Se *filename* è l'ultimo link al file, anche le risorse del file sono deallocate.

Pipe con nome: esempio

L'esempio seguente, composto da un processo lettore e due processi scrittori, mostra l'uso dei pipe con nome.

`reader.c` crea un pipe con nome chiamato `aPipe` e lo apre in lettura; quindi legge e stampa sullo schermo delle linee che terminano con `\0` fino a che il pipe è chiuso da tutti i processi scrittori.

`writer.c` apre in scrittura il pipe chiamato `aPipe`, vi scrive tre messaggi e quindi chiude il pipe e termina. Se quando `writer.c` tenta di aprire `aPipe` il file non esiste, `writer.c` ci riprova dopo un secondo fino a che non ha successo.

- Il lettore attende di leggere fino a che un processo qualsiasi non ha aperto il FIFO: a questo punto il lettore legge quello che di volta in volta viene scritto e capisce che non c'è più niente da leggere quando legge `EOF`. A quel punto il lettore chiude il suo descrittore e rimuove il FIFO.

Pipe con nome: esempio

```
$ cat reader.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>           /* For S_IFIFO */
#include <fcntl.h>

int readLine (int fd, char *str) {
/* Read a single '\0'-terminated line into str from fd */
/* Return 0 when the end-of-input is reached and 1 otherwise */
  int n;
  do {   /* Read characters until '\0' or end-of-input */
     n = read (fd, str, 1); /* Read one character */
  } while (n > 0 && *str++ != '\0');
  return (n > 0); /* Return false if end-of-input */
}

int main (void) {
  int fd;
  char str[100];
  unlink("aPipe"); /* Remove named pipe if it already exists */
  mknod ("aPipe", S_IFIFO, 0); /* Create named pipe */
  chmod ("aPipe", 0660); /* Change its permissions */
  fd = open ("aPipe", O_RDONLY); /* Open it for reading */
  while (readLine (fd, str)) /* Display received messages */
     printf ("%s\n", str);
  close (fd); /* Close pipe */
  unlink("aPipe"); /* Remove used pipe */
}
```

Pipe con nome: esempio

```
$ cat writer.c
#include <stdio.h>
#include <fcntl.h>
int main (void) {
  int fd, messageLen, i;
  char message [100];
  /* Prepare message */
  sprintf (message, "Hello from PID %d", getpid ());
  messageLen = strlen (message) + 1;
  do { /* Keep trying to open the file until successful */
     fd = open ("aPipe", O_WRONLY); /* Open named pipe for writing */
     if (fd == -1) sleep (1); /* Try again in 1 second */
  } while (fd == -1);
  for (i = 1; i <= 3; i++) { /* Send three messages */
     write (fd, message, messageLen); /* Write message down pipe */
     sleep (3); /* Pause a while */
  }
  close (fd); /* Close pipe descriptor */
  return 0;
}
$ reader & writer & writer &
[5] 1268
[6] 1269
[7] 1270
$ Hello from PID 1269
Hello from PID 1270
Hello from PID 1269
Hello from PID 1270
Hello from PID 1269
Hello from PID 1270
<return>
[5]   Done                    reader
[6]-  Done                    writer
[7]+  Done                    writer
$
```

Socket

- I *socket* sono uno dei più potenti meccanismi di comunicazione tra processi disponibili in ambiente UNIX; difatti essi consentono la comunicazione bidirezionale anche tra processi residenti su macchine diverse.

- Alcuni usi comuni dei socket sono:
  - collegamento remoto di un utente su un'altra macchina (`rlogin`);
  - stampa di un file su una macchina da un'altra macchina;
  - trasferimento di file da una macchina all'altra.

- Originariamente disponibili solo nelle versioni BSD, attualmente i socket sono supportati da tutte le varianti di UNIX disponibili sul mercato.

- Da un punto di vista del programmatore, i socket possono essere visti come file sui quali è possibile fare operazioni differenti a seconda della modalità di apertura e dei protocolli utilizzati.

Socket

- La comunicazione tramite i socket si basa sul modello *client-server*.
  - Un processo, il *server*, crea un socket il cui nome è noto anche ai processi *client*.
  - I client possono comunicare col server tramite una connessione al suo socket.
  - Per far ciò un client prima crea un socket anonimo e quindi chiede che sia connesso al socket del server.
  - Una connessione che ha successo restituisce un descrittore di file al client ed uno al server, che possono essere entrambi usati per leggere e scrivere.
  - Una volta creata la connessione col client, solitamente il server crea un processo figlio che si occupa della gestione della connessione, mentre il processo originario continua ad accettare altre connessioni da client.

- Esempio tipico di questo scenario è un server di stampa remoto: il processo server accetta un client che desidera spedire un file in stampa, quindi crea un figlio per eseguire il trasferimento del file mentre il padre attende altre richieste di stampa da client.

Socket: attributi principali

- *Dominio*: indica dove risiedono il server ed il client.

  Alcuni domini sono:

  - `AF_UNIX`: client e server sono sulla stessa macchina;
  - `AF_INET`: client e server sono ovunque in Internet.

- *Tipo*: determina il tipo di comunicazione che può essere instaurata tra server e client. I due tipi principali sono:

  - `SOCK_STREAM`: affidabile, preserva l'ordine di invio dei dati, basata su stream di byte di lunghezza variabile;
  - `SOCK_DGRAM`: inaffidabile, senza una connessione fissa, basata su messaggi di lunghezza variabile.

- *Protocollo*: specifica i mezzi di basso livello con cui il tipo di socket è implementato.

  Ogni socket può adottare protocolli diversi per la comunicazione: protocolli *datagram*, come UDP, che non assicurano l'affidabilità della comunicazione, o protocolli *stream*, come TCP/IP, che assicurano la ricezione dei dati spediti preservando anche l'ordine di spedizione.

  Tipicamente, le system call che si aspettano un parametro per specificare un protocollo accettano `0` per indicare "il protocollo più opportuno".

I socket hanno 3 attributi principali.

I due tipi principali sono:

- `SOCK_STREAM`: affidabile, preserva l'ordine di invio dei dati, basata su stream di byte di lunghezza variabile;

  Un socket viene creato ed esiste durante una sessione in cui coppie di processi comunicano tra loro. Alla fine della sessione il socket viene chiuso e, di fatto, rimosso dal sistema.

- `SOCK_DGRAM`: inaffidabile, senza una connessione fissa, basata su messaggi di lunghezza variabile.

  Il socket viene creato per una singola comunicazione tra processi e non esiste nè prima nè dopo. Una eventuale seconda comunicazione tra gli stessi processi avviene mediante un nuovo socket.

Noi vedremo solo il primo.

Socket `SOCK_STREAM`

Il server

- crea un socket anonimo con `socket()`
- gli assegna un indirizzo con `bind()`
- dichiara quante richieste di connessioni su quel socket è disposto ad accodare con `listen()`
- accetta una connessione con `accept()`
- opera sul socket con `read()` e `write()`.

Il client

- crea un socket anonimo con `socket()`
- lo collega all'indirizzo del socket del server con `connect()`
- opera sul socket con `read()` e `write()`.

Sia il client che il server, una volta terminate le operazioni col socket, lo chiudono con `close()`.

File di intestazione

- Un programma che usa i socket deve includere i file header `sys/types.h` e `sys/socket.h`.

- Inoltre, deve includere il file `sys/un.h`, se usa socket del dominio `AF_UNIX`, ed i file `netinet/in.h`, `arpa/inet.h` e `netdb.h`, se usa socket del dominio `AF_INET`.

Creazione di un socket: `socket()`

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

- *domain* specifica il dominio in cui si vuole utilizzare il socket. Useremo solo i valori `AF_UNIX` e `AF_INET`.

- *type* specifica il tipo di comunicazione. Useremo solo il valore `SOCK_STREAM`.

- *protocol* specifica il protocollo da utilizzare per la trasmissione dei dati. Di solito tale parametro viene posto a `0` per indicare che si vuole utilizzare il protocollo di default per la coppia domain/type in questione.

- `socket()` restituisce un descrittore di file associato al nuovo socket creato, se ha successo; altrimenti, restituisce `-1`.

Assegnazione di un indirizzo: `bind()`

`int bind (int` *fd*`, const struct sockaddr *`*addr*`, int` *addrlen*`)`

- `bind()` associa al socket anonimo riferito dal descrittore di file *fd* l'indirizzo (locale o di rete, a seconda del dominio) memorizzato nella struttura puntata da *addr*.

- Tipo e valore dell'indirizzo dipendono dal dominio del socket. Al momento di invocare `bind()`, l'indirizzo della struttura dati contenente l'indirizzo del socket viene convertito nel tipo `struct sockaddr *`.

```
struct sockaddr {
  unsigned short sa_family;    /* address family, AF_xxxx */
  char           sa_data[14];  /* 14 bytes of protocol address */
}
```

- *addrlen* deve contenere la lunghezza della struttura dati usata per memorizzare l'indirizzo.

- `bind()` restituisce `0`, se ha successo; altrimenti, restituisce `-1`.

Assegnazione di un indirizzo: AF_UNIX

- Gli indirizzi sono *nomi di file*. Qualsiasi file può essere utilizzato, purché si abbiano i *diritti di scrittura* sulla directory che lo contiene. Per connettersi ad un socket, è sufficente avere *diritti di lettura* sul file relativo.

- La struttura dati che contiene l'indirizzo è così definita (nel file `sys/un.h`)

```
struct sockaddr_un {
  unsigned short sun_family;     /* AF_UNIX */
  char sun_path[108];            /* pathname */
}
```

  I suoi campi dovrebbero assumere i seguenti valori:

  – `sun_family`: `AF_UNIX`,

  – `sun_path`: il pathname completo o relativo del socket.

- La lunghezza che costituisce il terzo parametro di `bind()` va calcolata come somma della lunghezza della componente `sun_family` con la lunghezza della stringa contenuta nella componente `sun_path`.

- Se si tenta di creare un socket e già ne esiste uno con lo stesso nome, si verifica un errore (per cui è più prudente invocare prima un `unlink()` di quel nome).

Assegnazione di un indirizzo: AF_UNIX

Esempio: creare un socket del dominio AF_UNIX ed associargli un nome.

```
#include <stddef.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>  /* For AF_UNIX sockets */

int make_named_socket (const char *filename) {
  struct sockaddr_un name;
  int sock;

  sock = socket (AF_UNIX, SOCK_STREAM, 0);
  if (sock < 0) {
     perror ("socket");
     exit (EXIT_FAILURE);
  }

  name.sun_family = AF_UNIX;
  strcpy (name.sun_path, filename);

  if (bind (sock, (struct sockaddr *) &name, sizeof(name)) < 0) {
     perror ("bind");
     exit (EXIT_FAILURE);
  }

  return sock;
}
```

Assegnazione di un indirizzo: `connect()`

`int connect (int` *fd*`, const struct sockaddr *`*addr*`, int` *addrlen*`)`

- `connect()` tenta di connettere un socket anonimo creato da un client riferito dal descrittore di file *fd* ad un socket il cui indirizzo è memorizzato nella struttura puntata da *addr*.

  Dominio, tipo e protocollo del socket del client devono accordarsi con quelli del socket del server.

- *addrlen* deve contenere la lunghezza della struttura dati usata per memorizzare l'indirizzo.

- Se `connect()` ha successo, *fd* può essere usato per comunicare con il socket del server.

- Il tipo di struttura a cui punta *addr* deve seguire le stesse regole viste nel caso di `bind()`.

- `connect()` restituisce valore `0`, se ha successo (la connessione è instaurata); altrimenti (il socket del server non esiste o la sua coda è piena), restituisce `-1`.

Numero di connessioni: `listen()`

`int listen (int` *fd*`, int` *queuelength*`)`

- `listen()` specifica il numero massimo *queuelength* di richieste di connessioni pendenti che possono essere accodate sul socket corrispondente al descrittore *fd*.

- Se un client tenta una connessione ad un socket la cui coda è piena, la connessione gli viene negata e gli viene notificato l'errore `ECONNREFUSED`.

Accetazione di una connessione: `accept()`

`int accept (int` *fd*`, struct sockaddr *`*addr*`, int *`*adLen*`)`

- `accept()` estrae la prima richiesta di connessione sulla coda delle connessioni pendenti del socket riferito da *fd*, crea un nuovo socket con gli stessi attributi di quello originario, lo connette al socket del client e restituisce un nuovo descrittore di file. Il socket originario può essere usato per accettare altre connessioni.
- Normalmente la `accept()` blocca l'invocante fino a quando non si è verificata una connessione.
- La struttura puntata da *addr* è riempita con l'indirizzo del client ed è solitamente usata solo con connessioni Internet.
- *adLen* inizialmente punta ad un intero che fornisce la lunghezza della struttura puntata da *addr*; quando una connessione è stabilita, l'intero a cui punta contiene la lunghezza effettiva, in byte, di *addr*.
- `accept()` restituisce un descrittore di file che può essere utilizzato per comunicare con il client, se ha successo; altrimenti, restituisce `-1`.

Operazioni sui socket

- Una volta realizzata una connessione, il socket del client e quello del server possono essere utilizzati in tutto e per tutto come descrittori di file sui quali si opera mediante `read()` e `write()`.

- Una volta terminate le operazioni su un socket, si può chiudere il relativo descrittore tramite una `close()`.

Socket `AF_UNIX`: esempio

L'esempio mostra l'uso di un socket nel dominio `AF_UNIX`.

È formato da due programmi:

- `chef.c`, il server, crea un socket chiamato `recipe` e tramite questo fornisce una ricetta a tutti i client che la richiedono. La ricetta è una sequenza di stringhe di lunghezza variabile terminate dal carattere `'\0'`.
- `cook.c`, il client, si connette al socket chiamato `recipe` e legge la ricetta fornita dal server. Man mano che la legge, `cook.c` mostra la ricetta sullo standard output e quindi termina.

Socket AF_UNIX: esempio

```
$ cat chef.c
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>  /* For AFUNIX sockets */

#define DEFAULT_PROTOCOL   0

int writeRecipe (int fd) {
  static char* line1 = "spam, spam, spam, spam,";
  static char* line2 = "spam, and spam.";
  write (fd, line1, strlen (line1) + 1); /* Write first line */
  write (fd, line2, strlen (line2) + 1); /* Write second line */
}
```

Socket AF_UNIX: esempio

```
int main (void) {
  int serverFd, clientFd, serverLen, clientLen;
  struct sockaddr_un serverUNIXAddress;/* Server address */
  struct sockaddr_un clientUNIXAddress; /* Client address */
  struct sockaddr* serverSockAddrPtr; /* Ptr to server address */
  struct sockaddr* clientSockAddrPtr; /* Ptr to client address */

  /* Ignore death-of-child signals to prevent zombies */
  signal (SIGCHLD, SIG_IGN);

  serverSockAddrPtr = (struct sockaddr*) &serverUNIXAddress;
  serverLen = sizeof (serverUNIXAddress);
  clientSockAddrPtr = (struct sockaddr*) &clientUNIXAddress;
  clientLen = sizeof (clientUNIXAddress);

  /* Create a UNIX socket, bidirectional, default protocol */
  serverFd = socket (AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
  serverUNIXAddress.sun_family = AF_UNIX; /* Set domain type */
  strcpy (serverUNIXAddress.sun_path, "recipe"); /* Set name */
  unlink ("recipe"); /* Remove file if it already exists */
  bind (serverFd, serverSockAddrPtr, serverLen); /* Create file */
  listen (serverFd, 5); /* Maximum pending connection length */

  while (1) { /* Loop forever */     /* Accept a client connection */
     clientFd = accept (serverFd, clientSockAddrPtr, &clientLen);
     if (fork () == 0) { /* Create child to send receipe */
        writeRecipe (clientFd); /* Send the recipe */
        close (clientFd); /* Close the socket */
        exit (/* EXIT_SUCCESS */ 0); /* Terminate */
     } else
        close (clientFd); /* Close the client descriptor */
  }
}
$
```

Socket AF_UNIX: esempio

```
$ cat cook.c
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>        /* For AF_UNIX sockets */

#define DEFAULT_PROTOCOL    0

int readRecipe (int fd) {
  char str[200];

  while (readLine (fd, str)) /* Read lines until end-of-input */
     printf ("%s\n", str); /* Echo line from socket */
}

int readLine (int fd, char *str) {
/* Read a single '\0'-terminated line into str from fd */
/* Return 0 when the end-of-input is reached and 1 otherwise */
  int n;
  do { /* Read characters until '\0' or end-of-input */
     n = read (fd, str, 1); /* Read one character */
  } while (n > 0 && *str++ != '\0');
  return (n > 0); /* Return false if end-of-input */
}
```

Socket AF_UNIX: esempio

```
int main (void) {
  int clientFd, serverLen, result;
  struct sockaddr_un serverUNIXAddress;
  struct sockaddr* serverSockAddrPtr;

  serverSockAddrPtr = (struct sockaddr*) &serverUNIXAddress;
  serverLen = sizeof (serverUNIXAddress);

  /* Create a UNIX socket, bidirectional, default protocol */
  clientFd = socket (AF_UNIX, SOCK_STREAM, DEFAULT_PROTOCOL);
  serverUNIXAddress.sun_family = AF_UNIX; /* Server domain */
  strcpy (serverUNIXAddress.sun_path, "recipe"); /* Server name */

  do { /* Loop until a connection is made with the server */
     result = connect (clientFd, serverSockAddrPtr, serverLen);
     if (result == -1) sleep (1); /* Wait and then try again */
  }   while (result == -1);
  readRecipe (clientFd); /* Read the recipe */
  close (clientFd); /* Close the socket */
  exit (/* EXIT_SUCCESS */ 0); /* Done */
}
$
```

Socket AF_UNIX: esempio

- Il processo `chef` gira in background.

- Ogni volta che un nuovo processo client `cook` si collega al server, `chef` si duplica: il processo figlio gestisce il trasferimento della ricetta al client, il processo padre accetta altre connessioni.

- Un semplice output dell'esempio.

```
$ chef &
[1] 1476
$ cook
spam, spam, spam, spam,
spam, and spam.
$ cook
spam, spam, spam, spam,
spam, and spam.
$ jobs
[1]+  Running                 chef &
$ kill %1
$
```

Socket `AF_INET`

- La principale differenza tra un socket del dominio `AF_UNIX` ed uno del dominio `AF_INET` è l'indirizzo.

- L'indirizzo di un socket del dominio `AF_UNIX` è un pathname valido nel filesystem.

- L'indirizzo di un socket del dominio `AF_INET` è specificato da due valori:
  - un indirizzo IP a 32 bit, che specifica un unico *host* Internet,
  - un numero di porta a 16 bit, che specifica una particolare *porta* dell'host.

  Entrambi vanno rappresentati in un formato indipendente dalle singole macchine (*network byte order*).

Assegnazione di un indirizzo: AF_INET

- La struttura dati che contiene l'indirizzo è così definita (nel file `netinet/in.h`)

```
struct sockaddr_in {
  short int          sin_family;  /* Address family */
  struct in_addr     sin_addr;    /* Internet address */
  unsigned short int sin_port;    /* Port number */
}
```

  I suoi campi dovrebbero assumere i seguenti valori:

  – `sin_family`: `AF_INET`,
  – `sin_addr`: l'indirizzo Internet della macchina host,
  – `sin_port`: il numero di porta del socket.

- Quando si invoca la `bind()`, il terzo parametro passatogli dovrebbe essere il risultato di

  `sizeof (struct sockaddr_in)`.

Indirizzi Internet e host name

- In Internet ogni computer ha uno o più *indirizzi Internet*, numeri (es. `150.217.14.144`) che identificano univocamente il computer in tutta la rete.

  In particolare, su qualsiasi macchina ci si trovi, l'indirizzo (di **loopback**) `127.0.0.1` fa sempre riferimento alla stessa macchina.

- Ogni computer ha anche uno o più *host name*, stringhe (es. `rap.dsi.unifi.it`) identificative del computer.

  Il vantaggio rispetto al precedente è che un indirizzo in questo formato è più semplice da ricordare.

  Per aprire una connessione, però, un indirizzo in questo formato va *convertito* nel precedente.

Indirizzi Internet

- Gli indirizzi Internet in alcuni contesti sono rappresentati come interi (`unsigned long int`) in altri sono "impaccati" in una struttura di tipo `struct in_addr` che comunque contiene un campo intero (`unsigned long int`).

- Definizioni utili (file `netinet/in.h`):
  - `struct in_addr`: struttura dati che contiene un campo chiamato `s_addr` di tipo `unsigned long int` che contiene l'indirizzo numerico dell'host.
  - `INADDR_LOOPBACK`: costante (di tipo `unsigned long int`), definita uguale al valore `127.0.0.1` (talvolta chiamato `localhost`) dell'indirizzo di loopback, che significa "l'indirizzo di questa macchina".
  - `INADDR_ANY`: costante (di tipo `unsigned long int`), di solito usata quando si invoca una `accept()`, che significa "qualsiasi indirizzo richiedente".

Indirizzi Internet

Nel file `arpa/inet.h` sono dichiarate alcune funzioni utili per manipolare indirizzi Internet.

Eccone alcune.

- `int inet_aton (const char *`*name*`, struct in_addr *`*addr*`)`

  converte l'indirizzo Internet *name* dalla notazione standard con numeri e punti in formato binario e lo memorizza nella struttura puntata da *addr*. Restituisce valore diverso da `0`, se l'indirizzo è valido; `0` altrimenti.

- `unsigned long int inet_addr (const char *`*name*`)`

  converte l'indirizzo Internet *name* dalla notazione standard con numeri e punti in network byte order. Restituisce `-1` se l'indirizzo non è valido.

- `char *inet_ntoa (struct in_addr` *addr*`)`

  converte l'indirizzo Internet *addr* dato in network byte order nella notazione standard con numeri e punti.

Host name

- Ogni computer ha un *database* (di solito il file `/etc/hosts`) in cui mantiene le associazioni tra host name conosciuti ed indirizzi Internet.
- Le funzioni e gli altri simboli per manipolare il database sono definite nel file `netdb.h`.
- Una *entry* nel database ha il seguente tipo:

```
struct hostent {
  char *h_name         /* host official name */
  char **h_aliases     /* host alternative names */
  int h_addrtype;      /* AF_INET */
  int h_length         /* length in byte of address */
  char **h_addr_list   /* array of host adresses, NULL terminated */
  char *h_addr         /* synonym for h_addr_list[0] */
}
```

- In particolare, `*h_addr` fornisce l'indirizzo Internet dell'host a cui una data entry corrisponde.
- In una struttura del tipo `struct hostent` gli indirizzi Internet di un host sono sempre in formato network byte order.

Host name

- Alcune utili funzioni di ricerca nel database:
  - `struct hostent *gethostbyname (const char *`*name*`)`

    restituisce la entry relativa all'host con host name *name*. Se il nome *name* non compare in `/etc/hosts`, restituisce `NULL`.
  - `struct hostent *gethostbyaddr (const char *`*addr*`,`
    `int` *length*`, int` *format*`)`

    restituisce la entry relativa all'host con indirizzo *addr* (*length* è la lunghezza in byte dell'indirizzo puntato da *addr* e *format* è `AF_INET`).

- Altre funzioni utili:
  - `int gethostname (char *`*name*`, size_t` *length*`)`

    inserisce il nome dell'host locale (seguito dal carattere `NULL`) nell'array di caratteri *name* di lunghezza *length*.
  - `void bzero (void *`*buffer*`, size_t` *length*`)`

    riempie l'array *buffer* di lunghezza *length* con il carattere ASCII `NULL`. Tale funzione è utilizzata per "ripulire" il contenuto di una struttura `struct sockaddr` prima di assegnare valori significativi ai suoi campi.

`bzero()` è disponibile nelle versioni UNIX Berkeley; la sua corrispondente in System V è

`void memset (void *`*buffer*`, int` *value*`, size_t` *length*`)`

che riempie l'array *buffer* di lunghezza *length* con il valore di *value*.

Per riempire i campi relativi all'indirizzo di un socket, risultano utili le funzioni che operano sulle stringhe.

Alcune sono:

- `char *strcpy (char *`*dst*`, const char *`*src*`)`

  `strcpy()` copia la stringa puntata da *src* (incluso il carattere di terminazione `NULL`) nell'array puntato da *dst*. Le due stringhe non si devono sovrapporre e *dst* dev'essere sufficentemente lunga.

- `char *strncpy (char *`*dst*`, const char *`*src*`, size_t` *n*`)`

  `strncpy()` opera in maniera simile a `strcpy()` ma copia al più *n* byte (il risultato potrebbe quindi non essere terminato da `'\0'`). Se *src* è più corta di *n*, il resto di *dst* sarà riempita da caratteri `'\0'`.

- `void *memcpy (void *`*dst*`, const void *`*src*`, size_t` *n*`)`

  copia *n* byte dall'area di memoria *src* all'area di memoria *dst* (le due aree non si devono sovrapporre) e restituisce un puntatore a *dst*.

Numeri di porta

- I numeri di porta sono `unsigned short int`, con valori compresi tra `0` e `65.535`.
- Esistono delle porte "riservate" per usi ben specifici; il file `/etc/services` contiene una lista di associazioni tra servizi e porte (e protocolli).

  Esempi: *ftp* 21, *ssh* 22, *telnet* 23, *www* 80 ...
- I numeri di porta utilizzabili dall'utente non devono interferire con quelli riportati in `/etc/services` per i quali esistono dei "gestori" di sistema che si occupano di implementare i relativi servizi.
- Ci sono due macro intere (definite in `netinet/in.h`) che hanno a che fare con i numeri di porta:
  - `IPPORT_RESERVED`: i numeri di porta minori di questo valore sono riservati ai superutenti;
  - `IPPORT_USERRESERVED`: i numeri di porta non minori di questo valore sono riservati per un uso esplicito e non sono mai allocati automaticamente.

Quando si usa un socket senza specificarne l'indirizzo, il sistema genera automaticamente un numero di porta per il socket che è compreso tra `IPPORT_RESERVED` e `IPPORT_USERRESERVED`.

Network byte order

- È un formato di rappresentazione dei dati che è indipendente dalle singole piattaforme.

- Per creare una connessione ad un socket Internet, bisogna assicurarsi che i valori nei campi `sin_addr` e `sin_port` di un dato di tipo `struct sockaddr_in` siano rappresentati in network byte order (stesso discorso vale per dati di tipo intero trasmessi tramite un socket).

- Le seguenti quattro funzioni (dichiarate in `netinet/in.h`) permettono di effettuare la conversioni necessarie.

  `in_addr_t htonl (in_addr_t` *hostLong*`)`

  `in_port_t htons (in_port_t` *hostShort*`)`

  `in_addr_t ntohl (in_addr_t` *networkLong*`)`

  `in_port_t ntohs (in_port_t` *networkShort*`)`

  Le funzioni `htonl` e `htons` trasformano un intero, long e short ripettivamente, in uno in formato network byte order.

  Le funzioni `ntohl` e `ntohs` effettuano le conversioni opposte.

- Le funzioni `htonl` e `ntohl` sono usate con valori che rappresentano numeri di host (`unsigned long int`), mentre `htons` e `ntohs` sono usate con valori che rappresentano numeri di porta.

Assegnazione di un indirizzo: AF_INET

Esempio: creare un socket del dominio AF_INET ed associargli un nome.

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>  /* For AF_INET sockets */

int make_socket (unsigned short int port) {
  struct sockaddr_in name;
  int sock;

  sock = socket (AF_INET, SOCK_STREAM, 0);
  if (sock < 0) {
     perror (``socket'');
     exit (EXIT_FAILURE);
  }

  name.sin_family = AF_INET;
  name.sin_port = htons (port);
  name.sin_addr.s_addr = htonl (INADDR_LOOPBACK);

  if (bind (sock, (struct sockaddr *) &name, sizeof (name)) < 0) {
     perror (``bind'');
     exit (EXIT_FAILURE);
  }

  return sock;
}
```

Assegnazione di un indirizzo: AF_INET

Esempio: creare un socket del dominio AF_INET.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>  /* For AF_INET sockets */
#include <netdb.h>

void init_sockaddr (struct sockaddr_in *name,
                    const char *hostname,
                    unsigned short int port) {
  struct hostent *hostinfo;

  name->sin_family = AF_INET;
  name->sin_port = htons (port);
  hostinfo = gethostbyname (hostname);

  if (hostinfo == NULL) {
     fprintf (stderr, ‘‘Unknown host %s\n’’, hostname);
     exit (EXIT_FAILURE);
  }

  name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
}
```