Sistemi di Calcolo

Modulo 2:
Programmazione dei sistemi di calcolo multi-nodo

Primitive C per UNIX System Programming

Ultimo aggiornamento: 25 maggio 2015

Leonardo Aniello, Daniele Cono D'Elia
Dipartimento di Ingegneria Informatica, Automatica e Gestionale "A. Ruberti"
Sapienza Università di Roma

Indice

1. Operazioni su file e descrittori
    1.1 Apertura di un file
    1.2 Lettura da un descrittore
    1.3 Scrittura su un descrittore
    1.4 Chiusura di un descrittore
    1.5 Rimozione di un file
2. Esecuzione concorrente
    2.1 Creazione di un processo figlio
    2.2 Attesa terminazione di un processo figlio
    2.3 Creazione di un thread
    2.4 Uscita da un thread
    2.5 Sincronizzazione tra thread
3. Semafori
    3.1 Apertura e chiusura di un semaforo anonimo
    3.2 Operazioni sui semafori
    3.3 Apertura di un semaforo named
    3.4 Chiusura e distruzione di un semaforo named
4. Socket
    4.1 Socket ed indirizzi
    4.2 Apertura e chiusura di socket
    4.3 Creare una connessione in uscita
    4.4 Accettare connessioni in ingresso
    4.5 Lettura e scrittura su socket
    4.6 Utility per manipolare indirizzi e numeri di porta
5. Pipe e FIFO
    5.1 Apertura e chiusura di pipe
    5.2 Duplicazione di descrittori
    5.3 Apertura e chiusura di FIFO
A. Complementi di programmazione C
    A.1 Gestione memoria
    A.2 Macro per gestire errori
    A.3 Segmentation fault ed uso di gdb

1. Operazioni su file e descrittori

Per operare su file e descrittori, è opportuno includere `<fcntl.h>` ed `<unistd.h>`.

1.1 Apertura di un file

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

La funzione restituisce un descrittore per il file specificato in *pathname*.

Il secondo argomento *flags* deve obbligatoriamente contenere una delle seguenti modalità di accesso: `O_RDONLY` (sola lettura), `O_WRONLY` (sola scrittura), o `O_RDWR` (lettura e scrittura). Si possono specificare ulteriori flag (es. `O_APPEND`, `O_TRUNC`) concatenandoli tramite OR bit a bit. Se `O_CREAT` è specificato, il file viene creato se non è già presente nel sistema; se `O_CREAT` ed `O_EXCL` sono entrambi specificati, viene restituito un errore se il file che si desidera creare è già esistente nel sistema.

I flag descritti sono definiti in `<fcntl.h>`. Quando `O_CREAT` viene specificato, è necessario fornire alla funzione tramite l'argomento *mode* i permessi di creazione per il file (se non già presente nel sistema). Per la specifica dei permessi si può utilizzare la codifica ottale (es. `0660`) oppure una delle macro definite in `<sys/stat.h>`.

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore (es. `EINTR`).

1.2 Lettura da un descrittore

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

La funzione tenta di leggere *count* bytes dal descrittore *fd* e di memorizzarli in *buf*.

In caso di successo, come valore di ritorno viene restituito il numero di byte letti, o `0` nel caso la fine del file sia stata già raggiunta. Si noti come sia possibile che il numero di byte letti sia minore del numero richiesto; questo può accadere per svariate ragioni, ad esempio perché leggendo quei byte si è raggiunta la fine del file, oppure perché al momento dell'operazione solo quel numero di byte era disponibile, o ancora perché l'arrivo di un segnale ha provocato un'interruzione nel mezzo dell'operazione.

In caso di errore, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore (es. `EINTR, EBADF, EIO`).

1.3 Scrittura su un descrittore

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

La funzione tenta di scrivere *count* bytes sul descrittore *fd* leggendoli da *buf*.

In caso di successo, come valore di ritorno viene restituito il numero di byte scritti effettivamente. Si noti come sia possibile che tale numero sia inferiore rispetto al numero richiesto; questo può accadere per svariate ragioni, ad esempio perché lo spazio sul mezzo fisico di destinazione è insufficiente, oppure perché l'arrivo di un segnale ha provocato un'interruzione nel mezzo dell'operazione.

In caso di errore, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore (es. `EINTR, EBADF, EIO`).

1.4 Chiusura di un descrittore

```
int close(int fd);
```

Chiude un descrittore di file, così che possa essere riutilizzato dal sistema. Se *fd* è l'ultimo descrittore aperto associato ad uno specifico file, le risorse associate all'apertura del file vengono liberate dal sistema operativo. L'operazione di `close()` si applica a descrittori sia di file ordinari che di file speciali, quali socket e pipe.

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore (es. `EINTR`).

1.5 Rimozione di un file

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

La funzione rimuove il nome *pathname* dal filesystem. Se *pathname* è l'ultimo hardlink[1] esistente al file, il file viene candidato per la rimozione. Se non vi sono descrittori aperti per quel file nei processi in esecuzione, il file viene rimosso immediatamente e le risorse ad esso associate liberate; se invece vi è almeno un processo con un descrittore aperto sul file, la rimozione viene posticipata fino all'avvenuta chiusura dell'ultimo descrittore.

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore (es. `EINTR`).

[1] In Linux un file normale è un hard link ad un inode sul filesystem. Un inode è una struttura dati che contiene le informazioni di base di un oggetto, quali dimensione, posizione fisica su disco e permessi di accesso, ed è possibile creare più hard link che puntano allo stesso inode. Gli hard links hanno proprietà diverse dai symbolic links, in particolare una volta creati i primi sono indistinguibili dall'originale. Per approfondimenti si vedano: http://linuxgazette.net/105/pitcher.html, http://stackoverflow.com/questions/185899/what-is-the-difference-between-a-symbolic-link-and-a-hard-link, http://unix.stackexchange.com/questions/50179/what-happens-when-you-delete-a-hard-link

2. Esecuzione concorrente

2.1 Creazione di un processo figlio

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

Questa funzione crea un nuovo processo duplicando il processo chiamante. Il nuovo processo è detto *figlio*, quello chiamante *padre*. Padre e figlio vengono eseguiti in spazi di memoria separati.

È necessario includere `<unistd.h>` per usarla.

In caso di successo, la funzione restituisce `0` nel processo figlio ed il PID del figlio nel processo padre. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata col codice dell'errore occorso.

2.2 Attesa terminazione di un processo figlio

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

Questa funzione consente ad un processo padre di mettersi in attesa della terminazione dei suoi processi figli. Se `status` è diverso da `NULL`, al termine della chiamata esso conterrà delle informazioni sullo stato del processo figlio che è terminato.

È necessario includere `<sys/wait.h>` per usarla.

In caso di successo, la funzione restituisce il PID del processo figlio terminato. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata col codice dell'errore occorso.

2.3 Creazione di un thread

Nelle applicazioni multithread si includa `<pthread.h>`, ed al momento della compilazione è necessario linkare l'eseguibile prodotto alla libreria `pthread`. Si tenga presente inoltre che in caso di fallimento le funzioni per la gestione dei thread non aggiornano la variabile `errno` per indicarne la causa, ma restituiscono il codice di errore come valore di ritorno!

```
int pthread_create(phtread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

Questa funzione viene usata per creare un nuovo thread, con attributi definitii in *`attr`*, all'interno del processo corrente. Nell'ambito del corso utilizzeremo gli attributi di default[2] specificando `NULL` come valore per l'argomento *`attr`*.

[2] http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_attr_init.html

In caso di successo, l'ID del thread creato viene memorizzato nella porzione di memoria puntata da *thread*. Il thread viene creato eseguendo la funzione *start_routine*, che prenderà come unico argomento il buffer puntato da *arg*. Essa potrà opzionalmente restituire un puntatore tramite `pthread_exit()` o `return`; nel secondo caso, l'effetto è quello di una chiamata implicita a `pthread_exit()` con il valore della `return` come argomento. Maschera e gestori dei segnali saranno ereditati dal thread creante.

In caso di successo, la funzione restituisce `0`, e il nuovo thread erediterà maschera e gestore dei segnali dal thread creante. In caso di fallimento, verrà restituito un codice di errore che ne indicherà la causa (la variabile `errno` non verrà aggiornata).

2.4 Uscita da un thread

```
void pthread_exit(void *value_ptr);
```

La funzione termina il thread in cui viene invocata, e rende il valore del puntatore *value_ptr* disponibile ad ogni operazione di join che verrà fatta sul thread da terminare. Al momento della terminazione non vengono rilasciate automaticamente risorse che sono visibili nel resto del processo, quali semafori e descrittori di file.

Per tutti i thread diversi dal main thread, un'istruzione `return` equivale ad una chiamata implicita a `pthread_exit()`, usando il valore restituito come argomento della chiamata. Per il main thread, un'istruzione `return` equivale invece ad una chiamata implicita ad `exit()`, usando il valore restituito come exit status. Una chiamata a `pthread_exit()` nel main thread permette invece al processo di proseguire nell'esecuzione finché tutti gli altri thread attivi termineranno spontaneamente.

Una volta che il thread chiamante è terminato, accedere ad indirizzi di variabili locali a quel thread dà luogo ad un comportamento indefinito. Di conseguenza, riferimenti a variabili locali non vanno contemplati come possibile valore per *value_ptr*. In assenza di un valore da restitutire, si può utilizzare `NULL` come argomento.

Non potendo ritornare nel chiamante, la funzione `pthread_exit()` non restituisce alcun valore, né sono previsti codici di errore.

2.5 Sincronizzazione tra thread

Di default i thread sono *joinable*, ossia è possibile attenderne la terminazione in modo esplicito da altri thread e quindi leggerne il valore restituito (se esso è previsto). Per questo motivo l'implementazione sottostante mantiene una serie di strutture dati associate al thread, che restano disponibili anche dopo la terminazione[3]: quando non sono previste operazioni di join su un thread, è buona pratica liberare esplicitamente queste risorse.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

[3] http://www.domaigne.com/blog/computing/joinable-and-detached-threads/

La funzione sospende l'esecuzione del thread chiamante fino alla terminazione del thread *`thread`*, a meno che quest'ultimo non sia già terminato. Quando la chiamata ha successo ed il valore specificato per l'argomento *`value_ptr`* è diverso da `NULL`, il valore passato a `pthread_exit()` nel thread terminante viene scritto nella locazione di memoria puntata da *`value_ptr`*.

In caso di successo, la funzione restituisce `0`. Altrimenti, viene restituito un codice di errore che indica la causa del fallimento.

```
int pthread_detach(pthread_t thread);
```

La funzione viene usata per indicare all'implementazione sottostante che lo storage previsto per il thread *`thread`* potrà essere reclamato quando quel thread terminerà.

In caso di successo, la funzione restituisce `0`. Altrimenti, viene restituito un codice di errore che indica la causa del fallimento.

3. Semafori

Le definizioni dei tipi di dato e i prototipi delle funzioni per la gestione dei semafori sono definite principalmente in `<semaphore.h>`. Al momento della compilazione è essenziale linkare il codice generato alla libreria `pthread`.

3.1 Apertura e chiusura di un semaforo anonimo

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Inizializza il semaforo anonimo puntato da *`sem`* con valore iniziale *`value`*. Nell'ambito del corso assumeremo[4] che il valore scelto per *`pshared`* sia sempre `0`.

In caso di successo, inizializza il semaforo puntato da *`sem`* come richiesto e restituisce `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore.

```
int sem_destroy(sem_t *sem);
```

Distrugge il semaforo anonimo puntato da *`sem`*. Se invocato su un semaforo named, il suo comportamento non è definito.

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore.

3.2 Operazioni sui semafori

```
int sem_wait(sem_t *sem);
```

La funzione decrementa di un'unità il contatore associato al semaforo puntato da *`sem`*.

Se il decremento producesse un valore negativo per il contatore, la chiamata resta bloccata in attesa che un altro thread incrementi il contatore, a meno che essa non venga interrotta dall'arrivo di un segnale.

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore (es. `EINTR`).

```
int sem_post(sem_t *sem);
```

La funzione incrementa di un'unità il contatore associato al semaforo puntato da `sem`.

Se nel processo vi sono thread bloccati in attesa che il semaforo puntato da `sem` venga

[4] http://stackoverflow.com/questions/6847973/do-forked-child-processes-use-the-same-semaphore

incrementato, uno di essi verrà sbloccato e la sua `sem_wait()` ritornerà con successo.

In caso di successo, la `sem_post()` restituisce `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore.

```
int sem_getvalue(sem_t *sem, int *sval);
```

Memorizza nell'indirizzo *`sval`* il valore del contatore associato al semaforo puntato da *`sem`*, senza alterare lo stato del semaforo.

Se la coda di thread in attesa non è vuota, nei sistemi Linux il valore scritto in *`sval`* sarà zero (su altri sistemi il valore potrà invece corrispondere al numero di thread in coda).

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore.

3.3 Apertura di un semaforo named

Per i semafori named si includa `<fcntl.h>` per le costanti da usare per il parametro `oflag` (es. `O_CREAT`, `O_EXCL`), ed eventualmente `<sys/stat.h>` per usare le costanti previste per argomenti `mode_t` in luogo di una maschera dei permessi in codifica ottale.

```
sem_t* sem_open(const char* name, int oflag);
sem_t* sem_open(const char* name, int oflag,
                mode_t mode, unsigned int value);
```

La funzione crea un semaforo named o ne apre uno esistente.

Il semaforo è identificato univocamente da *`name`*, che per convenzione è una stringa del tipo `/nomeSemaforo`; ossia, una stringa null-terminated composta da massimo 251 caratteri, di cui un simbolo backslash `/` iniziale, seguito da uno o più caratteri, nessuno dei quali può essere un backslash[5].

Il parametro *`oflag`* specifica i flag (definiti in `<fcntl.h>`) che controllano il modo in cui deve operare la chiamata. In assenza di flag da specificare, il valore da utilizzare è `0`.

Se il flag `O_CREAT` è specificato, il semaforo viene creato se non è già presente nel sistema. Se `O_CREAT` ed `O_EXCL` sono specificati contemporaneamente (tramite OR bit a bit: `O_CREAT | O_EXCL`), viene restituito un errore qualora il semaforo per il *`name`* specificato sia già esistente.

Quando `O_CREAT` è specificato, sono necessari due ulteriori argomenti per la funzione. Il parametro *`mode`* specifica i permessi di creazione per il semaforo, analogamente a quanto avviene per i file. Per operare correttamente, un processo ha bisogno di poter accedere al semaforo sia in lettura che in scrittura. Per la specifica dei permessi si può utilizzare la codifica ottale (es. `0660`) oppure una delle macro definite in

[5] http://man7.org/linux/man-pages/man7/sem_overview.7.html

<sys/stat.h>. Il parametro *value* specifica il valore iniziale per il semaforo da creare. Si tenga a mente che qualora il semaforo named sia già presente nel sistema, i parametri mode e value saranno ignorati.

In caso di successo, `sem_open()` restituisce l'indirizzo del nuovo semaforo, che potrà essere usato come argomento per operazioni quali `sem_wait()`, `sem_getvalue()` e `sem_post()`. L'implementazione alloca la struttura `sem_t` in una memoria condivisa gestita dal kernel, per cui il puntatore continua ad essere valido in caso di `fork()`.

In caso di errore, `sem_open()` restituisce `SEM_FAILED`, e la variabile `errno` indicherà la causa dell'errore (es. `EEXIST`).

3.4 Chiusura e distruzione di un semaforo named

```
int sem_close(sem_t *sem);
```

Chiude il semaforo named puntato da *sem*, permettendo che le risorse allocate dal sistema operativo per il processo che ha aperto il semaforo vengano liberate.

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore.

```
int sem_unlink(const char *name);
```

Rimuove il semaforo named identificato univocamente da *name*.

Il nome associato al semaforo viene immediatamente reso libero nel sistema, mentre il semaforo viene distrutto non appena gli altri processi che hanno aperto il semaforo l'avranno chiuso.

La funzione è MT-Safe: può essere eseguita in modo safe in programmi multi-thread[6].

In caso di successo, viene restituito `0`. Altrimenti, viene restituito `-1` e la variabile `errno` indicherà la causa dell'errore.

[6] http://man7.org/linux/man-pages/man7/attributes.7.html

4. Socket

4.1 Socket ed indirizzi

Una socket è un canale di comunicazione bidirezionale tra due endpoint. Ogni endpoint è identificato da una coppia indirizzo IP, porta. Per rappresentare un endpoint viene usata la struttura dati `struct sockaddr_in`, i cui campi sono:

- `sin_family`: la famiglia della socket (`AF_INET` nell'ambito del corso, vedi 4.2)
- `sin_addr.s_addr`: l'indirizzo IP, rappresentato con la struttura dati `struct in_addr`
  - per indicare che ci si vuole mettere in ascolto su tutte le interfacce locali, usare la macro `INADDR_ANY` (definita in `<netinet/in.h>`)
- `sin_port`: numero di porta
- `sin_zero`: padding di 8 byte (vanno sempre inizializzati a `0`)[7]

4.2 Apertura e chiusura di socket

Per la creazione di una socket, da usare sia per accettare nuove connessioni che per connettersi ad uno specifico endpoint, bisogna usare la funzione `socket()` descritta di seguito. Per la chiusura, fare riferimento a 1.4.

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

Questa funzione crea un endpoint di comunicazione e ne ritorna il descrittore.

Il parametro *`domain`* specifica il dominio di comunicazione, ossia seleziona la famiglia di protocolli da usare. Tali famiglie sono definite in `<sys/socket.h>`. Nell'ambito del corso useremo solo la famiglia `AF_INET`, corrispondente al protocollo IPv4.

Il parametro *`socket`* specifica la specifica la semantica di comunicazione. Nell'ambito del corso useremo solo il tipo `SOCK_STREAM` per indicare che vogliamo usare connessioni TCP.

Il parametro *`protocol`* specifica un protocollo particolare da usare per la socket. Normalmente per un tipo specifico di socket esiste un solo protocollo, nel qual caso il parametro va impostato a `0`. Nell'ambito del corso non avremo bisogno di usare protocolli particolari, quindi potremo sempre settare *`protocol`* a `0`.

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna il descrittore della nuova socket. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

[7] http://stackoverflow.com/questions/15608707/why-is-zero-padding-needed-in-sockaddr-in

4.3 Creare una connessione in uscita

```
int connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

Effettua un tentativo di connessione verso l'endpoint specificato come secondo parametro, usando il descrittore *socket* indicato come primo parametro.

Il parametro *address* è un puntatore ad una struttura dati `struct sockaddr`. Nell'ambito del corso useremo strutture dati `struct sockaddr_in` ed effettueremo il cast a `struct sockaddr`.

Il parametro *address_len* specifica la dimensione della struttura dati puntata da `address`.

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna `0`. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

4.4 Accettare connessioni in ingresso

Per accettare connessioni in ingresso, sono necessarie tre funzioni da chiamare in sequenza: la `bind()` per collegare una socket ad un endpoint, la `listen()` per segnalare che la socket può essere usata per accettare connessioni, ed infine la `accept()` per attendere l'arrivo di connessioni ed accettarle.

```
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Collega il descrittore *socket* ad un indirizzo *address*.

Il parametro *address* è un puntatore ad una struttura dati `struct sockaddr`. Nell'ambito del corso useremo strutture dati `struct sockaddr_in` ed effettueremo il cast a `struct sockaddr`.

Il parametro *address_len* specifica la dimensione della struttura dati puntata da *address*.

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna `0`. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

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

Segnala che la socket identificata dal descrittore *sockfd* sarà usata per accettare richieste di connessione.

Il parametro *backlog* indica la dimensione massima della coda delle richieste pendenti per la socket. Si noti che tale dimensione può essere interpretata soltanto come un *hint* all'implementazione sottostante[8].

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna `0`. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

```
int accept(int socket, struct sockaddr *address, socklen_t
*address_len);
```

Accetta una nuova connessione sulla socket identificata dal descrittore *socket*.

Il parametro *address* è un puntatore ad una struttura dati `struct sockaddr` che verrà impostata dalla funzione stessa con le informazioni sull'endpoint remoto.

Il parametro *address_len* è un puntatore ad un valore che verrà impostato dalla funzione stessa con la dimensione della struttura dati puntata da *address*.

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna il descrittore della socket relativa alla connessione accettata. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

4.5 Lettura e scrittura su socket

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Trasmette un messaggio via socket.

Il parametro *sockfd* indica il descrittore di socket da usare per l'invio.

Il parametro *buf* è un puntatore all'area di memoria contenente messaggio da inviare.

Il parametro *len* indica il numero massimo di byte da inviare.

Nell'ambito del corso, il parametro *flags* va impostato a 0. Così facendo, la `send()` diventa equivalente alla `write()` (vedi 1.3).

[8] A tale proposito si vedano http://stackoverflow.com/questions/5145392/setting-listen-backlog-to-0 e http://stackoverflow.com/questions/5111040/listen-ignores-the-backlog-argument

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna il numero di byte effettivamente inviati. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Riceve un messaggio via socket.

Il parametro *`sockfd`* indica il descrittore di socket da usare per la ricezione.

Il parametro *`buf`* è un puntatore all'area di memoria dove copiare il messaggio ricevuto.

Il parametro *`len`* indica il numero massimo di byte da leggere.

Nell'ambito del corso, il parametro *`flags`* va impostato a 0. Così facendo, la `recv()` diventa equivalente alla `read()` (vedi 1.2).

Per usarla bisogna includere `<sys/socket.h>`.

In caso di successo, la funzione ritorna il numero di byte effettivamente ricevuti. Se la connessione viene chiusa dall'altro lato della comunicazione, la funzione ritorna `0`. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

4.6 Utility per manipolare indirizzi e numeri di porta

I numeri di porta sono *unsigned short integer* rappresentati in *network byte order*. Per manipolarli correttamente vengono messe a disposizione le seguenti funzioni per le conversioni da network byte order a *host byte order*. Per usare queste funzioni va inclusa `<arpa/inet.h>`.

```
uint16_t htons(uint16_t hostshort);
```

Converte l'unsigned short integer *`hostshort`* da host byte order a network byte order.

```
uint16_t ntohs(uint16_t netshort);
```

Converte l'unsigned short integer *`netshort`* da network byte order a host byte order.

Gli indirizzi IPv4 sono rappresentati con la struttura dati `struct in_addr`, che include il campo `s_addr` di tipo `in_addr_t` dove l'indirizzo IP è memorizzato in network byte order. Per la corretta conversione del valore contenuto in questo campo vengono messe a disposizione le seguenti funzioni (definite in `<arpa/inet.h>`).

```
in_addr_t inet_addr(const char *cp);
```

Converte un indirizzo IPv4 dalla forma dotted (x.y.z.w) al network byte order.

Il parametro *cp* è un puntatore alla stringa che rappresenta in forma dotted l'indirizzo da convertire in network byte order.

```
const char *inet_ntop(int af, const void *src, char *dst,
socklen_t size);
```

Converte l'indirizzo di rete *src* della address family *af* in una stringa di lunghezza *size* e la copia in *dst*.

Ritorna un puntatore a *dst* in caso di successo, NULL in caso di errore.

Le macro AF_INET e INET_ADDRSTRLEN possono essere usate rispettivamente per il primo e l'ultimo argomento.

5. Pipe e FIFO

5.1 Apertura e chiusura di pipe

Le pipe vengono create con la funzione `pipe()`, descritta di seguito. Per la chiusura, vedi 1.4.

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

Crea una pipe, un canale di comunicazione unidirezionale che può essere usato per comunicazioni inter-processo. L'array *`pipefd`* è usato per restituire i due descrittori relativi alla pipe: *`pipefd`*`[0]` per la lettura e *`pipefd`*`[1]` per la scrittura.

Per usarla, bisogna includere `<unistd.h>`.

In caso di successo, la funzione ritorna `0`. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

5.2 Duplicazione di descrittori

```
int dup(int oldfd);
```

Crea una copia del descrittore *`oldfd`*. Per il nuovo descrittore, viene scelto il primo valore di descrittore non utilizzato (quello con valore minimo nella tabella dei descrittori del processo).

Per usarla, bisogna includere `<unistd.h>`.

In caso di successo, la funzione ritorna il nuovo descrittore. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

```
int dup2(int oldfd, int newfd);
```

Crea una copia del descrittore *`oldfd`*. Per il nuovo descrittore, viene scelto il valore *`newfd`* passato come secondo parametro. Se necessario, *`newfd`* viene prima chiuso.

Per usarla, bisogna includere `<unistd.h>`.

In caso di successo, la funzione ritorna il nuovo descrittore. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

5.3 Apertura e chiusura di FIFO

Le named pipe (FIFO) vanno prima create con la funzione `mkfifo()`, poi aperte con la `open()`. Quando tutti i processi hanno effettuato la `close()`, è possibile rimuovere la FIFO definitivamente usando la funzione `unlink()`.

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

Crea una FIFO con nome *`pathname`*. Il parametro *`mode`* specifica i permessi della FIFO stessa (es. `0660`).

Per usarla, bisogna includere `<sys/types.h>` e `<sys/stat.h>`.

In caso di successo, la funzione ritorna `0`. In caso di fallimento, viene restituito `-1` ed `errno` viene impostata con un codice che caratterizza l'errore avvenuto.

Per l'apertura e la chiusura di una FIFO, fare riferimento a 1.1 (segnatura con due parametri) e 1.4, rispettivamente. Per la rimozione, vedere 1.5.

A. Complementi di programmazione C

A.1 Gestione memoria

```
void *malloc(size_t size);
```

Alloca un'area di memoria di dimensione *size* byte e ritorna un puntatore a tale area.

Per usarla, bisogna includere `<stdlib.h>`.

In caso di successo, la funzione ritorna un puntatore all'area di memoria allocata. In caso di fallimento, viene restituito `NULL`.

```
void *calloc(size_t nmemb, size_t size);
```

Alloca memoria per un array di *nmemb* elementi, ognuno di dimensione *size* byte, e ritorna un puntatore alla memoria allocata.

Per usarla, bisogna includere `<stdlib.h>`.

In caso di successo, la funzione ritorna un puntatore all'area di memoria allocata. In caso di fallimento, viene restituito `NULL`.

```
void *realloc(void *ptr, size_t size);
```

Cambia la dimensione dell'area di memoria puntata da *ptr* impostandola a *size* byte. Il contenuto della memoria rimarrà invariato nel range compreso tra l'inizio dell'area di memoria ed il minimo tra la vecchia dimensione e la nuova. Se la nuova dimensione è più grande della vecchia, la memoria aggiunta non verrà inizializzata. Se *ptr* è `NULL`, la chiamata è equivalente ad una `malloc(`*size*`)`, per qualsiasi valore di *size*. Se *size* è `0` e *ptr* non è `NULL`, la chiamata è equivalente ad una `free(`*ptr*`)`. Il puntatore *ptr* deve essere stato ritornato da una precedente chiamata a `malloc()`, `calloc()` o `realloc()`, a meno che non sia `NULL`.

Per usarla, bisogna includere `<stdlib.h>`.

In caso di successo, la funzione ritorna un puntatore all'area di memoria reallocata. In caso di fallimento, viene restituito `NULL`.

```
void free(void *ptr);
```

Rilascia l'area di memoria puntata da *ptr*. È possibile rilasciare aree di memoria che siano state allocate tramite `malloc()`, `calloc()` e `realloc()`.

Per usarla, bisogna includere `<stdlib.h>`.

Se *`ptr`* è `NULL`, non viene eseguita alcuna operazione.

```
void *memset(void *s, int c, size_t n);
```

Riempie i primi *`n`* byte dell'area di memoria puntata da *`s`* con il valore costante del byte *`c`*.

Per usarla, bisogna includere `<string.h>`.

Ritorna un puntatore all'area di memoria *`s`*.

```
void *memcpy(void *dest, const void* src, size_t n);
```

Copia i primi `n` byte dell'area di memoria puntata da *`src`* nel blocco puntato da *`dest`*.

Per usarla, bisogna includere `<string.h>`.

Ritorna un puntatore all'area di memoria `dest`.

A.2 Macro per gestire errori

Per semplificare la gestione dei valori di ritorno per le funzioni descritte in questa dispensa, è possibile attingere alla collezione di macro qui riportata:

```
#include <stdio.h>  // fprintf()
#include <stdlib.h> // macro EXIT_FAILURE
#include <string.h> // funzione strerror()
#include <errno.h>  // variabile errno

#define GENERIC_ERROR_HELPER(cond, errCode, msg) do { \
    if (cond) { \
        fprintf(stderr, "%s: %s\n", msg, strerror(errCode)); \
        exit(EXIT_FAILURE); \
    } \
  } while(0)

#define ERROR_HELPER(ret, msg) GENERIC_ERROR_HELPER((ret < 0), \
                                                    errno, msg)
#define PTHREAD_ERROR_HELPER(ret, msg) GENERIC_ERROR_HELPER( \
                                          (ret != 0), ret, msg)
```

Un lettore dotato di una certa dimestichezza col C noterà che l'azione della `fprintf` nel blocco è del tutto analoga ad una chiamata alla routine `perror` (non trattata nel corso[9]).

A.3 Segmentation fault ed uso di gdb

Gli errori di tipo `Segmentation fault` sono all'ordine del giorno nella programmazione C, e determinarne l'origine può essere dispendioso in termini di tempo. Quando tali errori si verificano in modo sistematico, esistono strumenti per risalire con ragionevole certezza all'ultima istruzione eseguita prima del verificarsi del problema.

In particolare, il debugger `gdb` è uno strumento molto potente e diffuso, nonché ben documentato in rete[10]. Affinché un debugger fornisca informazioni sufficientemente dettagliate, è importante che i sorgenti siano compilati senza ottimizzazioni[11] (`-O0`) e con i simboli di debug abilitati (`-g`), opzioni presenti di default in tutti i Makefile di esercitazioni e tracce d'esame previste per questo corso.

Supponiamo per semplicità che un programma `prova` produca sistematicamente un segmentation fault quando invocato con argomenti `25` e `100`:

```
$ ./prova 25 100
Segmentation fault
```

Proviamo ora ad eseguire il programma dentro il debugger `gdb`:

```
$ gdb ./prova
[...]
(gdb) run 25 100
```

Il comando `run` va fatto seguire quindi dai valori da passare come argomenti al programma da analizzare. Ad un certo punto si verificherà[12] un accesso non valido alla memoria e l'errore verrà riportato dal debugger. Possiamo quindi ispezionare la sequenza di metodi presenti sulla stack in quel momento utilizzando `bt` (o `backtrace`):

```
[...]
(gdb) bt
#0  0x4007fc13 in _IO_getline_info () from /lib/libc.so.6
#1  0x4007fb6c in _IO_getline () from /lib/libc.so.6
#2  0x4007ef51 in fgets () from /lib/libc.so.6
#3  0x80484b2 in main (argc=3, argv=0xbffffaf4) at prova.c:10
#4  0x40037f5c in __libc_start_main () from /lib/libc.so.6
```

Il frame identificato con `#0` corrisponde allo stack frame dell'ultimo metodo eseguito, che in questo caso fa parte della `libc`. Di norma è lecito assumere che le funzioni di libreria siano ben testate e che quindi difficilmente contengono bug, per cui scorriamo la lista dei frame ed individuiamo quello più in alto associato ad un metodo tra quelli da noi scritti. Nel

[9] Per approfondimenti risorse utili sono http://man7.org/linux/man-pages/man3/perror.3.html e http://stackoverflow.com/questions/12102332/when-i-should-use-perror-and-fprintfstderr

[10] Per iniziare si vedano ad esempio http://www.unknownroad.com/rtfm/gdbtut/gdbsegfault.html e https://www.cs.cmu.edu/~gilpin/tutorial/

[11] Si pensi ad esempio alle considerazioni fatte per il primo modulo sull'uso del registro EBP.

[12] In linea di principio ciò non è sempre vero: http://en.wikipedia.org/wiki/Heisenbug

nostro esempio lo stack frame `#3` è associato al metodo `main`, eseguito con i valori riportati per gli argomenti `argc` e `argv` e che ha effettuato una chiamata a `fgets()` in corrispondenza della linea `10` del sorgente `prova.c` che lo contiene.

Avere informazioni di questo tipo è un ottimo punto di partenza per individuare cause di errori, che possono naturalmente essere localizzate in altri punti del programma (ad esempio un metodo che fa ricerca su una lista può causare un segmentation fault perché il metodo che inizializza la lista in partenza contiene un bug).

Per ulteriori approfondimenti su un corretto uso della memoria nella programmazione in C, si raccomanda di tenere in considerazione `valgrind`, altro tool prezioso nel debugging[13].

[13] http://valgrind.org/docs/manual/quick-start.html