

Le opzioni dei socket

- L'API socket mette a disposizione due funzioni per gestire il comportamento dei socket

```
int setsockopt(int sockfd, int level, int optname, const void *optval,  
socklen_t optlen);  
  
int getsockopt(int sockfd, int level, int optname, void *optval,  
socklen_t *optlen);
```

- `setsockopt()` per impostare le caratteristiche del socket
- `getsockopt()` per conoscere le caratteristiche impostate del socket
- Entrambe le funzioni restituiscono 0 in caso di successo, -1 in caso di fallimento

Programmazione di applicazioni di rete con socket - parte 2

Funzione `setsockopt()`

• Parametri della funzione `setsockopt()`

`sockfd`: descrittore del socket a cui si fa riferimento

`level`: livello del protocollo (trasporto, rete, ...)

`SOL_SOCKET` per opzioni generiche del socket

`SOL_TCP` per i socket che usano TCP

`optname`: su quale delle opzioni definite dal protocollo si vuole operare (il nome dell'opzione)

`optval`: puntatore ad un'area di memoria contenente i dati che specificano il valore dell'opzione da impostare per il socket a cui si fa riferimento

`optlen`: dimensione (in byte) dei dati puntati da `optval`

Alcune opzioni generiche

• Analizziamo alcune opzioni generiche da usare come valore per `optname`:

`SO_KEEPALIVE`: per controllare l'attività della connessione (in particolare per verificare la persistenza della connessione)

- `optval` è un intero usato come valore logico (on/off)

`SO_RCVTIMEO`: per impostare un timeout in ricezione (sulle operazioni di lettura di un socket)

- `optval` è una struttura di tipo `timeval` contenente il valore del timeout
- utile anche per impostare un tempo massimo per `connect()`

`SO_SNDFTIMEO`: per impostare un timeout in trasmissione (sulle operazioni di scrittura di un socket)

- `optval` è una struttura di tipo `timeval` contenente il valore del timeout

`SO_REUSEADDR`: per riutilizzare un indirizzo locale; modifica il comportamento della funzione `bind()`, che fallisce nel caso in cui l'indirizzo locale sia già in uso da parte di un altro socket

- `optval` è un intero usato come valore logico (on/off)
- Occorre impostare l'opzione prima di chiamare `bind()`

Esempio opzione SO_REUSEADDR

```
...
int reuse = 1;
if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("errore creazione socket");
    exit(1);
}
if (setsockopt(listensd, SOL_SOCKET, SO_REUSEADDR, &reuse,
               sizeof(int)) < 0) {
    perror("errore setsockopt");
    exit(1);
}
if (bind(listensd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
    perror("errore bind");
    exit(1);
}
} ...
```

SD - Valeria Cardellini, A.A. 2008/09

4

Server TCP iterativo (o sequenziale)

- Gestisce una connessione alla volta
 - Mentre è impegnato a gestire la connessione con un determinato client, possono arrivare altre richieste di connessione
 - Il SO stabilisce le connessioni con i client; tuttavia queste rimangono in attesa di servizio nella coda di backlog finché il server non è libero
- Più semplice da progettare, implementare e mantenere (e meno diffuso)
- Adatto in situazioni in cui:
 - il numero di client da gestire è limitato
 - il tempo di servizio per un singolo client è limitato (vedi esempio daytime)

SD - Valeria Cardellini, A.A. 2008/09

5

Struttura di un server TCP iterativo

```
int listensd, connsd;

listensd = socket(AF_INET, SOCK_STREAM, 0);
bind(listensd, ...);
listen(listensd, ...);
for (; ; ) {
    connsd = accept(listensd, ...);
    do_it(connsd); /* serve la richiesta */
    close(connsd); /* chiude il socket di connessione */
}
```

SD - Valeria Cardellini, A.A. 2008/09

6

Server TCP ricorsivo (o concorrente)

- Gestisce più client (connessioni) nello stesso istante
- Utilizza una copia ([processo/thread](#)) di se stesso per gestire ogni connessione
 - Analizziamo l'uso della chiamata di sistema [fork\(\)](#) per generare un processo figlio che eredita una connessione con un client
- I processi server padre e figlio sono eseguiti contemporaneamente sulla macchina server
 - Il processo figlio gestisce la specifica connessione con un dato client
 - Il processo padre può accettare la connessione con un altro client, assegnandola ad un altro processo figlio per la gestione
- Il numero massimo di processi figli che possono essere generati dipende dal SO

SD - Valeria Cardellini, A.A. 2008/09

7

Struttura di un server TCP ricorsivo

```
int listensd, connsd;  
pid_t pid;  
  
listensd = socket(AF_INET, SOCK_STREAM, 0);  
bind(listensd, ...);  
listen(listensd, ...);  
for (; ;){  
    connsd = accept(listensd, ...);  
    if ( (pid = fork()) == 0) { /* processo figlio */  
        close(listensd); /* chiude il socket d'ascolto */  
        do_it(connsd); /* serve la richiesta */  
        close(connsd); /* chiude il socket di connessione */  
        exit(0); /* termina */  
    }  
    close(connsd); /* il padre chiude il socket di connessione */  
}
```

N.B.: nessuna delle due chiamate a `close()` evidenziate in rosso causa l'innesto della sequenza di chiusura della connessione TCP perché il numero di riferimenti al descrittore non si è annullato

fork()

pid_t fork(void);

- Permette di creare un nuovo processo figlio
 - È una copia esatta del processo padre
 - Eredita tutti i descrittori del processo padre
- In caso di successo restituisce un risultato sia al padre che al figlio
 - Al padre restituisce il **pid** (*process id*) del figlio
 - Al figlio restituisce 0
- Il processo figlio è una **copia** del padre
 - Riceve una copia dei segmenti testo, dati e stack
 - Esegue esattamente lo stesso codice del padre
 - La memoria è copiata (non condivisa!): quindi padre e figlio vedono valori diversi delle stesse variabili

Socket e server ricorsivo

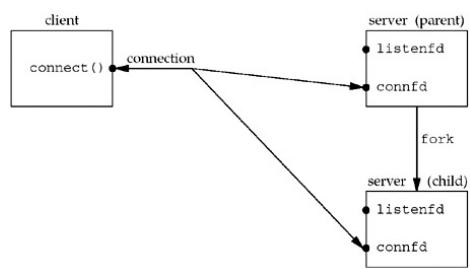
1. Stato del client e del server prima che il server chiami `accept()`



2. Dopo il ritorno di `accept()`

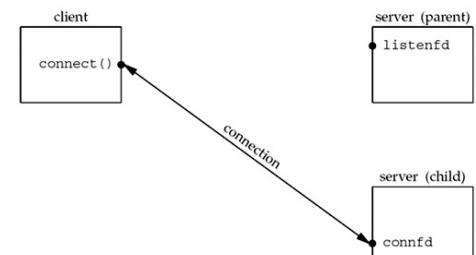


3. Dopo il ritorno di `fork()`



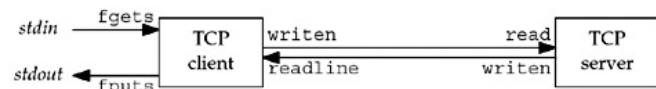
4. Dopo `close()` sui socket opportuni in padre e figlio

- Il padre chiude il socket di connessione
- Il figlio chiude il socket di ascolto



Applicazione echo con server ricorsivo

- Echo: il server replica un messaggio inviato dal client
- Il client legge una riga di testo dallo standard input e la invia al server
- Il server legge la riga di testo dal socket e la rimanda al client
- Il client legge la riga di testo dal socket e la invia allo standard output



SD - Valeria Cardellini, A.A. 2008/09

12

SD - Valeria Cardellini, A.A. 2008/09

13

echo_server.c (2)

```
pid_t pid;
int listenfd, connfd;
struct sockaddr_in servaddr, cliaddr;
int len;

if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("errore in socket");
    exit(1);

    memset((char *)&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    if ((bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr))) < 0) {
        perror("errore in bind");
        exit(1);
    }
}
```

SD - Valeria Cardellini, A.A. 2008/09

14

echo_server.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <time.h>

#define SERV_PORT      5193
#define BACKLOG        10
#define MAXLINE        1024
.....
int main(int argc, char **argv)
{
```

SD - Valeria Cardellini, A.A. 2008/09

13

echo_server.c (3)

```
if (listen(listenfd, BACKLOG) < 0 ) {
    perror("errore in listen");
    exit(1);
}

for ( ; ; ) {
    len = sizeof(cliaddr);
    if ((connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &len)) < 0) {
        perror("errore in accept");
        exit(1);
    }

    if ((pid = fork()) == 0) {
        if (close(listenfd) == -1) {
            perror("errore in close");
            exit(1);
        }
        printf("%s:%d connesso\n", inet_ntoa(cliaddr.sin_addr),
               ntohs(cliaddr.sin_port));
    }
}
```

SD - Valeria Cardellini, A.A. 2008/09

15

echo_server.c (4)

```
str_srv_echo(connsd); /* svolge il lavoro del server */

if (close(connsd) == -1) {
    perror("errore in close");
    exit(1);
}
exit(0);
} /* end fork */

if (close(connsd) == -1) {           /* processo padre */
    perror("errore in close");
    exit(1);
}
} /* end for */
}
```

echo_server.c (5)

```
void str_srv_echo(int sockfd)
{
    int     nread;
    char    line[MAXLINE];

    for ( ; ; ) {
        if ((nread = readline(sockfd, line, MAXLINE)) == 0)
            /* readline restituisce il numero di byte letti */
            return; /* il client ha chiuso la connessione e inviato EOF */

        if (writen(sockfd, line, nread)) {
            fprintf(stderr, "errore in write");
            exit(1);
        }
    }
}
```

echo_client.c

```
int main(int argc, char **argv)
{
    int             sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2) {
        fprintf(stderr, "utilizzo: echo_client <indirizzo IP server>\n");
        exit(1);
    }
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("errore in socket");
        exit(1);
    }
    memset((void *)&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0) {
        fprintf(stderr, "errore in inet_ntop per %s", argv[1]);
        exit(1);
    }
}
```

echo_client.c (2)

```
if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    perror("errore in connect");
    exit(1);
}

str_cli_echo(stdin, sockfd);      /* svolge il lavoro del client */

close(sockfd);

exit(0);
}
```

echo_client.c (3)

```
void str_cli_echo(FILE *fd, int sockfd)
{
    char    sendline[MAXLINE], recvline[MAXLINE];
    int     n;

    while (fgets(sendline, MAXLINE, fd) != NULL) {
        if ((n = writen(sockfd, sendline, strlen(sendline))) < 0) {
            perror("errore in write");
            exit(1);
        }

        if ((n = readline(sockfd, recvline, MAXLINE)) < 0) {
            fprintf(stderr, "errore in readline");
            exit(1);
        }

        fputs(recvline, stdout);
    }
}
```

SD - Valeria Cardellini, A.A. 2008/09

20

Analisi applicazione echo

- Il comando **netstat** permette di ottenere informazioni sullo stato delle connessioni instaurate

Opzione **-a**: per visualizzare anche lo stato dei socket non attivi (in stato LISTEN)

Opzione **-Ainet**: per specificare la famiglia di indirizzi Internet

Opzione **-n**: per visualizzare gli indirizzi numerici (invece di quelli simbolici) degli host e delle porte

- Negli esempi seguenti client e server sulla stessa macchina

- Client, processo server padre, processo server figlio

\$ netstat -a -Ainet -n

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:5193	127.0.0.1:1232	ESTABLISHED
tcp	0	0	127.0.0.1:1232	127.0.0.1:5193	ESTABLISHED
tcp	0	0	0.0.0.0:5193	0.0.0.0:*	LISTEN

SD - Valeria Cardellini, A.A. 2008/09

21

Analisi applicazione echo (2)

- Il comando **ps** (*process state*) permette di ottenere informazioni sullo stato dei processi

Opzione **I**: formato lungo

Opzione **w**: output largo

```
$ ps lw
F  UID  PID  PPID  PRI  NI  VSZ RSS WCHAN STAT TTY  TIME COMMAND
000 501  31276 31227 1  0 1080 296 wait_f  S  pts/2  0:00 echo_server
044 501  31308 31276 1  0 1084 360 tcp_re   S  pts/2  0:00 echo_server
000 501  31393 31166 9  0 1084 336 read_c  S  pts/1  0:00 echo_client 127.0.0.1

[wait_for_connect] --- [read_chan]
```

SD - Valeria Cardellini, A.A. 2008/09

22

Analisi applicazione echo (3)

- Il client termina (Control-D)

\$ netstat -a -Ainet -n

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:1232	127.0.0.1:5193	TIME_WAIT
tcp	0	0	0.0.0.0:5193	0.0.0.0:*	LISTEN

\$ ps lw

```
F  UID  PID  PPID  PRI  NI  VSZ RSS WCHAN STAT TTY  TIME COMMAND
000 501  31276 31227 1  0 1080 296 wait_f  S  pts/2  0:00 echo_server
044 501  31308 31276 1  0 1084 360 do_exit  Z  pts/2  0:00 [echo_server <defu
[僵死进程] --- [僵尸进程]
```

SD - Valeria Cardellini, A.A. 2008/09

23

I segnali

- I segnali sono interruzioni software inviate ad un processo
- Permettono di notificare ad un processo l'occorrenza di qualche evento asincrono
 - Inviati dal kernel o da un processo
 - Usati dal kernel per notificare situazioni eccezionali (ad es. errori di accesso, eccezioni aritmetiche)
 - Usati anche per notificare eventi (ad es. terminazione di un processo figlio)
 - Usati anche come forma elementare di IPC
- Ogni segnale ha un nome, che inizia con SIG; ad es.:
 - SIGCHLD: inviato dal SO al processo padre quando un processo figlio è terminato o fermato
 - SIGALRM: generato quando scade il timer impostato con la funzione alarm()
 - SIGKILL: per terminare immediatamente (kill) il processo
 - SIGSTOP: per fermare (stop) il processo
 - SIGUSR1 e SIGUSR2: a disposizione dell'utente per implementare una forma di comunicazione tra processi

SD - Valeria Cardellini, A.A. 2008/09

24

Gestione dei segnali

- Un processo può decidere quali segnali gestire
 - Ovvero le notifiche di segnali che accetta
 - Per ogni segnale da gestire, deve essere definita un'apposita funzione di gestione (**signal handler**)
- Il segnale viene consegnato al processo quando viene eseguita l'azione per esso prevista
- Per il tempo che intercorre tra la generazione del segnale e la sua consegna al processo, il segnale rimane **pendente**
- Alcuni segnali non possono essere ignorati e vengono gestiti sempre (SIGKILL e SIGSTOP)
- Alcuni segnali vengono ignorati per default (es. SIGCHLD)
 - Tale comportamento può tuttavia essere modificato

SD - Valeria Cardellini, A.A. 2008/09

25

Gestione dei segnali (2)

- Per tutti i segnali non aventi un'azione specificata che è fissa, il processo può decidere di:
 - Ignorare il segnale
 - Catturare il segnale
 - Accettare l'azione di default propria del segnale
- La scelta riguardante la gestione del segnale può essere specificata mediante le funzioni signal() e sigaction()
- Per approfondimenti vedere GaPiL (Guida alla Programmazione in Linux)

SD - Valeria Cardellini, A.A. 2008/09

26

Segnale SIGCHLD

- Quando un processo termina (evento asincrono) il kernel manda un **segnale SIGCHLD** al padre ed il figlio diventa zombie
 - Mantenuto dal SO per consentire al padre di controllare il valore di uscita del processo e l'utilizzo delle risorse del figlio
 - Per default, il padre ignora il segnale SIGCHLD ed il figlio rimane zombie finché il padre non termina
- Per evitare di riempire di zombie la tabella dei processi bisogna fornire un handler per SIGCHLD
- Il processo zombie viene rimosso quando il processo padre chiama le funzioni **wait()** o **waitpid()**

SD - Valeria Cardellini, A.A. 2008/09

27

wait() e waitpid()

- ```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```
- Definite in sys/wait.h
  - Restituiscono il pid e il valore di uscita del processo figlio che è terminato
    - Consentono al processo zombie di essere rimosso
  - Caratteristiche di wait()
    - Sospende il padre finché non termina un qualunque figlio
    - Non accoda i segnali ricevuti durante l'esecuzione
      - Alcuni processi restano zombie
  - Caratteristiche di waitpid()
    - Non bloccante (se opzione WNOHANG)
    - Permette di specificare quale figlio attendere sulla base del valore dell'argomento pid
      - WAIT\_ANY (oppure -1) per il primo che termina
    - Chiamata all'interno di un ciclo, consente di catturare tutti i segnali

SD - Valeria Cardellini, A.A. 2008/09

28

## Handler per SIGCHLD

```
#include <signal.h>
#include <sys/wait.h>
void sig_chld_handler(int signum)
{
 int status;
 pid_t pid;

 while ((pid = waitpid(WAIT_ANY, &status, WNOHANG)) > 0)
 printf ("child %d terminato\n", pid);
 return;
}
```

waitpid ritorna 0 quando non c'è nessun figlio di cui non è stato ancora ricevuto dal padre lo stato di terminazione

- Quando un figlio termina e lancia il segnale SIGCHLD, waitpid() lo cattura e restituisce il pid; il processo figlio può essere rimosso

SD - Valeria Cardellini, A.A. 2008/09

29

## Attivazione dell'handler

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

typedef void Sigfunc(int);
Sigfunc *signal(int signum, Sigfunc *func)
{
 struct sigaction act, oldact;

 act.sa_handler = func;
 sigemptyset(&act.sa_mask);
 act.sa_flags = 0;
 if (signum != SIGALRM)
 act.sa_flags |= SA_RESTART;
 if (sigaction(signum, &act, &oldact) < 0)
 return(SIG_ERR);
 return(oldact.sa_handler);
}
```

void (\*signal (int signo, void (\*func) (int))) (int);  
signal() attiva l'handler: prende in ingresso il numero del segnale ed il puntatore all'handler

la struttura sigaction memorizza informazioni riguardanti la manipolazione del segnale

insieme di segnali bloccati durante l'esecuzione dell'handler

la funzione sigaction() prende in ingresso una struttura con il puntatore all'handler, una maschera di segnali da mascherare e vari flag e installa l'azione per il segnale

flag SA\_RESTART per far ripartire le chiamate di sistema "lente" interrotte dal segnale

SD - Valeria Cardellini, A.A. 2008/09

30

## echo\_server con gestione SIGCHLD

```
...
if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
 fprintf(stderr, "errore in socket");
 exit(1);
}

if ((bind(listensd, (struct sockaddr *)&servaddr, sizeof(servaddr))) < 0) {
 fprintf(stderr, "errore in bind");
 exit(1);
}
if (listen(listensd, QLEN) < 0) {
 fprintf(stderr, "errore in listen");
 exit(1);
}
if (signal(SIGCHLD, sig_chld_handler) == SIG_ERR) {
 fprintf(stderr, "errore in signal");
 exit(1);
}
...
```

SD - Valeria Cardellini, A.A. 2008/09

31

## echo\_server con gestione EINTR

```
...
for (; ;) {
 len = sizeof(cliaddr);
 if ((connfd = accept(listensd, (struct sockaddr *)&cliaddr, &len)) < 0) {
 if (errno == EINTR)
 continue; /* riprende da for */
 else {
 perror("errore in accept");
 exit(1);
 }
 }
 ...
}
```

## Gestione SIGALRM

- Per evitare che un client UDP o un server UDP rimangono indefinitamente bloccati su recvfrom() si può usare il segnale di allarme SIGALRM
- E' il segnale del timer dalla funzione alarm()  
    unsigned int alarm(unsigned int seconds);
  - alarm() predispone l'invio di SIGALRM dopo *seconds* secondi, calcolati sul tempo reale trascorso (il clock time)
  - Restituisce il numero di secondi rimanenti all'invio dell'allarme programmato in precedenza
  - alarm(0) per cancellare una programmazione precedente del timer

## Client UDP daytime con SIGALRM

```
#define TIMEOUT 20
void sig_alm_handler(int signo)
{
}
...
int main(int argc, char *argv[])
{
 ...
 struct sigaction sa;
 ...
 sa.sa_handler = sig_alm_handler; /* installa il gestore del segnale */
 sa.sa_flags = 0;
 sigemptyset(&sa.sa_mask);
 if (sigaction(SIGALRM, &sa, NULL) < 0) {
 fprintf(stderr, "errore in sigaction");
 exit(1);
 }
 if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { /* crea il socket */
 fprintf(stderr, "errore in socket");
 exit(1);
 }
```

## Client UDP daytime con SIGALRM (2)

```
...
/* Invia al server il pacchetto di richiesta*/
if (sendto(sockfd, NULL, 0, 0, (struct sockaddr *) &servaddr,
 sizeof(servaddr)) < 0) {
 fprintf(stderr, "errore in sendto");
 exit(1);
}
alarm(TIMEOUT);
n = recvfrom(sockfd, recvline, MAXLINE, 0 , NULL, NULL);
if (n < 0) {
 if (errno != EINTR) alarm(0);
 fprintf(stderr, "errore in recvfrom");
 exit(1);
}
alarm(0);
...
```

## Funzioni bloccanti e soluzioni

- La funzione accept() e le funzioni per la gestione dell'I/O (ad es., read() e write()) sono **bloccanti**
  - Ad es., read() e recv() rimangono in attesa finché non vi sono dati da leggere disponibili sul descrittore del socket
- Server ricorsivo tradizionale:
  - Il server si blocca su accept() aspettando una connessione
  - Quando arriva la connessione, il server effettua fork(), il processo figlio gestisce la connessione ed il processo padre si mette in attesa di una nuova richiesta
- Soluzioni possibili:
  - Usare le opzioni dei socket per impostare un timeout
  - Usare un socket non bloccante tramite la funzione fcntl() nel modo seguente

```
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

    - Polling del socket per sapere se ci sono informazioni da leggere

## Funzioni bloccanti e soluzioni (2)

- Soluzione alternativa:
  - Usare la **funzione select()** che permette di esaminare più canali di I/O contemporaneamente e realizzare quindi il **multiplexing dell'I/O**
- Nel caso del server
  - Invece di avere un processo figlio per ogni richiesta, c'è un solo processo che effettua il multiplexing tra le richieste, servendo ciascuna richiesta il più possibile
  - Vantaggio: il server può gestire tutte le richieste tramite un singolo processo
    - No memoria condivisa e primitive di sincronizzazione
  - Svantaggio: il server non può agire come se ci fosse un unico client, come avviene con la soluzione del server ricorsivo che utilizza fork()
- Nel caso del client
  - Può gestire più input simultaneamente
    - Ad es., il client echo gestisce due flussi di input: lo standard input ed il socket
    - Usando select(), il primo dei due canali che produce dati viene letto

## Funzione select()

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

- Header file
  - sys/time.h, sys/types.h, unistd.h
- Permette di controllare contemporaneamente lo stato di uno o più descrittori degli insiemi specificati
- Si blocca finché:
  - non avviene un'attività (**lettura** o **scrittura**) su un descrittore appartenente ad un dato insieme di descrittori
  - non viene generata un'**eccezione**
  - non scade un **timeout**
- Restituisce
  - -1 in caso di errore
  - 0 se il timeout è scaduto
  - Altrimenti, il numero totale di descrittori pronti

## Parametri della funzione select()

- Insiemi di descrittori da controllare
  - **readfds**: pronti per operazioni di lettura
    - Es: un socket è pronto per la lettura se c'è una connessione in attesa che può essere accettata con accept()
  - **writefds**: pronti per operazioni di scrittura
  - **exceptfds**: per verificare l'esistenza di eccezioni
    - un'eccezione non è un errore (ad es., l'arrivo di dati urgenti fuori banda, caratteristica specifica dei socket TCP)
- **readfds**, **writefds** e **exceptfds** sono puntatori a variabili di tipo **fd\_set**
  - **fd\_set** è il tipo di dati che rappresenta l'insieme dei descrittori (è una bit mask implementata con un array di interi)
- **numfds** è il numero massimo di descrittori controllati da **select()**
  - Se **maxd** è il massimo descrittore usato, **numfds = maxd + 1**
  - Può essere posto uguale alla costante **FD\_SETSIZE**

## Timeout della funzione select()

- timeout specifica il valore massimo che la funzione select() attende per individuare un descrittore pronto  

```
struct timeval {
 long tv_sec; /* numero di secondi */
 long tv_usec; /* numero di microsecondi */
};
```
- Se impostato a NULL (timeout == NULL)
  - si blocca indefinitamente fino a quando è pronto un descrittore
- Se impostato a zero (timeout->tv\_sec == 0 && timeout->tv\_usec == 0 )
  - non si attende affatto; modo per effettuare il polling dei descrittori senza bloccare
- Se diverso da zero (timeout->tv\_sec != 0 || timeout->tv\_usec != 0 )
  - si attende il tempo specificato
  - select() ritorna se è pronto uno (o più) dei descrittori specificati (restituisce un numero positivo) oppure se è scaduto il timeout (restituisce 0)

## Operazioni sugli insiemi di descrittori

- Macro utilizzate per manipolare gli insiemi di descrittori  
`void FD_ZERO(fd_set *set);`
  - Inizializza l'insieme di descrittori di *set* con l'insieme vuoto`void FD_SET(int fd, fd_set *set);`
  - Aggiunge *fd* all'insieme di descrittori *set*, mettendo ad 1 il bit relativo a *fd*`void FD_CLR(int fd, fd_set *set);`
  - Rimuove *fd* dall'insieme di descrittori *set*, mettendo ad 0 il bit relativo a *fd*`int FD_ISSET(int fd, fd_set *set);`
  - Al ritorno di select(), controlla se *fd* appartiene all'insieme di descrittori *set*, verificando se il bit relativo a *fd* è pari a 1 (restituisce 0 in caso negativo, un valore diverso da 0 in caso affermativo)

## Descrittori pronti in lettura

- La funzione select() rileva i descrittori pronti
  - Significato diverso per i tre gruppi (lettura, scrittura, eccezione)
- Un descrittore è **pronto in lettura** nei seguenti casi:
  - Nel buffer di ricezione del socket sono arrivati dati in quantità sufficiente (soglia minima per default pari a 1, modificabile con opzione del socket SO\_RCVLOWAT)
  - Per il lato in lettura è stata chiusa la connessione
    - select() ritorna con quel descrittore di socket pari a "pronto per la lettura" (a causa di EOF)
    - Quando si effettua read() su quel socket, read() restituisce 0
  - Si è verificato un errore sul socket
  - Se un socket è nella fase di listening e ci sono delle connessioni completate
    - E' possibile controllare se c'è una nuova connessione completata ponendo il descrittore del socket d'ascolto nell'insieme readfds

## Descrittori pronti in scrittura

- Un descrittore è **pronto in scrittura** nei seguenti casi:
  - Nel buffer di invio del socket è disponibile uno spazio in quantità sufficiente (soglia minima per default pari a 2048, modificabile con opzione del socket SO\_SNDLOWAT) ed il socket è già connesso (TCP) oppure non necessita di connessione (UDP)
  - Per il lato in scrittura è stata chiusa la connessione (segnale SIGPIPE generato dall'operazione di scrittura)
  - Si è verificato un errore sul socket

## Multiplexing dell'I/O nel server

- Il multiplexing dell'I/O può essere usato sul server per ascoltare su più socket contemporaneamente
  - Un unico processo server (iterativo) ascolta sul socket di ascolto e sui socket di connessione
- Struttura generale di un server che usa select()
  - riempire una struttura fd\_set con i descrittori dai quali si intende leggere
  - riempire una struttura fd\_set con i descrittori sui quali si intende scrivere
  - chiamare select() ed attendere finché non avviene qualcosa
  - quando select() ritorna, controllare se uno dei descrittori ha causato il ritorno. In questo caso, servire il descrittore in base al tipo di servizio offerto dal server (ad es., lettura della richiesta per una risorsa Web)
  - ripetere il ciclo forever

## Client TCP echo con select

- Il client deve controllare due diversi descrittori in lettura
  - Lo standard input, da cui legge il testo da inviare al server
  - Il socket connesso con il server, su cui scriverà il testo e dal quale riceverà la risposta
- L'implementazione con I/O multiplexing consente al client di accorgersi di errori sulla connessione mentre è in attesa di dati immessi dall'utente sullo standard input
- La fase iniziale in cui viene stabilita la connessione è analoga al caso precedente (vedere codice `client_echo.c`)

## Client TCP echo con select (2)

```
void str_cli_echo_sel(FILE *fd, int sockfd)
{
 int maxd, n;
 fd_set rset;
 char sendline[MAXLINE], recvline[MAXLINE];

 FD_ZERO(&rset); /* inizializza a 0 il set dei descrittori in lettura */
 for (;) {
 FD_SET(fileno(fd), &rset); /* inserisce il descrittore del file (stdin) */
 FD_SET(sockfd, &rset); /* inserisce il descrittore del socket */
 maxd = (fileno(fd) < sockfd) ? (sockfd + 1): (fileno(fd) + 1);
 if (select(maxd, &rset, NULL, NULL, NULL) < 0) { /* attende descrittore pronto
 in lettura */
 perror("errore in select");
 exit(1);
 }
 }
}
```

## Client TCP echo con select (3)

```
/* Controlla se il file (stdin) è leggibile */
if (FD_ISSET(fileno(fd), &rset)) {
 if (fgets(sendline, MAXLINE, fd) == NULL)
 return; /* non vi sono dati perché si è concluso l'utilizzo del client */
 if ((writen(sockfd, sendline, strlen(sendline))) < 0) {
 fprintf(stderr, "errore in write");
 exit(1);
 }
}
```

## Client TCP echo con select (4)

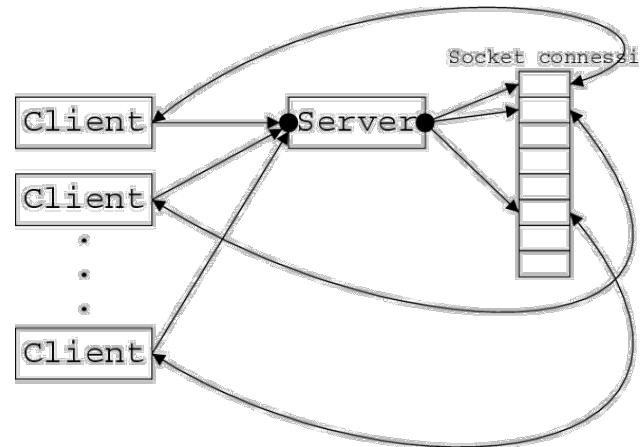
```
/* Controlla se il socket è leggibile */
if (FD_ISSET(sockfd, &rset)) {
 if ((n = readline(sockfd, recvline, MAXLINE)) < 0) {
 fprintf(stderr, "errore in lettura");
 exit(1);
 }
 if (n == 0) {
 fprintf(stdout, "str_cli_echo_sel: il server ha chiuso la connessione");
 return;
 }
 /* Stampa su stdout */
 recvline[n] = 0;
 if (fputs(recvline, stdout) == EOF) {
 perror("errore in scrittura su stdout");
 exit(1);
 }
}
}
```

SD - Valeria Cardellini, A.A. 2008/09

48

## Server TCP echo con select

- Schema del server TCP echo basato sull'I/O multiplexing



SD - Valeria Cardellini, A.A. 2008/09

49

## Server TCP echo con select (2)

```
#include "basic.h"
#include "echo_io.h"

int main(int argc, char **argv)
{
 int listensd, connsd, socksd;
 int i, maxi, maxd;
 int ready, client[FD_SETSIZE];
 char buff[MAXLINE];
 fd_set rset, allset;
 ssize_t n;
 struct sockaddr_in servaddr, cliaddr;
 int len;

 if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
 perror("errore in socket");
 exit(1);
 }
}
```

SD - Valeria Cardellini, A.A. 2008/09

50

## Server TCP echo con select (3)

```
memset((void *)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

if ((bind(listensd, (struct sockaddr *)&servaddr, sizeof(servaddr))) < 0) {
 perror("errore in bind");
 exit(1);
}

if (listen(listensd, QLEN) < 0) {
 perror("errore in listen");
 exit(1);
}

/* Inizializza il numero di descrittori */
maxd = listensd; /* maxd è il valore massimo dei descrittori in uso */
maxi = -1;
```

SD - Valeria Cardellini, A.A. 2008/09

51

## Server TCP echo con select (4)

```

/* L'array di interi client contiene i descrittori dei socket connessi */
for (i = 0; i < FD_SETSIZE; i++)
 client[i] = -1;

FD_ZERO(&allset); /* Inizializza a zero l'insieme dei descrittori */
FD_SET(listensd, &allset); /* Inserisce il descrittore di ascolto */

for (; ;) {
 rset = allset; /* Imposta il set di descrittori per la lettura */
 /* ready è il numero di descrittori pronti */
 if ((ready = select(maxd+1, &rset, NULL, NULL, NULL)) < 0) {
 perror("errore in select");
 exit(1);
 }
 /* Se è arrivata una richiesta di connessione, il socket di ascolto
 è leggibile: viene invocata accept() e creato un socket di connessione */
 if (FD_ISSET(listensd, &rset)) {
 len = sizeof(cliaddr);

```

SD - Valeria Cardellini, A.A. 2008/09

52

## Server TCP echo con select (6)

```

/* Altrimenti inserisce connsd tra i descrittori da controllare
 ed aggiorna maxd */
FD_SET(connsd, &allset);
if (connsd > maxd) maxd = connsd;
if (i > maxi) maxi = i;
if (--ready <= 0) /* Cicla finché ci sono ancora descrittori da controllare */
 continue;
}
/* Controlla i socket attivi per controllare se sono leggibili */
for (i = 0; i <= maxi; i++) {
 if ((socksd = client[i]) < 0)
 /* Se il descrittore non è stato selezionato, viene saltato */
 continue;
 if (FD_ISSET(socksd, &rset)) {
 /* Se socksd è leggibile, invoca la readline */
 if ((n = readline(socksd, buff, MAXLINE)) == 0) {
 /* Se legge EOF, chiude il descrittore di connessione */
 if (close(socksd) == -1) {

```

SD - Valeria Cardellini, A.A. 2008/09

54

## Server TCP echo con select (5)

```

if ((connsd = accept(listensd, (struct sockaddr *)&cliaddr, &len)) < 0) {
 perror("errore in accept");
 exit(1);
}

/* Inserisce il descrittore del nuovo socket nel primo posto libero di client */
for (i=0; i<FD_SETSIZE; i++) {
 if (client[i] < 0) {
 client[i] = connsd;
 break;
 }
}
/* Se non ci sono posti liberi in client, errore */
if (i == FD_SETSIZE) {
 fprintf(stderr, "errore in accept");
 exit(1);
}

```

SD - Valeria Cardellini, A.A. 2008/09

53

## Server TCP echo con select (7)

```
 perror("errore in close");
 exit(1);
}
/* Rimuove socksd dalla lista dei socket da controllare */
FD_CLR(socksd, &allset);
/* Cancella socksd da client */
client[i] = -1;
}
else /* echo */
{
 if (writen(socksd, buff, n) < 0) {
 fprintf(stderr, "errore in write");
 exit(1);
 }
 if (--ready <= 0) break;
}
}
}
```

SD - Valeria Cardellini, A.A. 2008/09

55