Laboratorio di Reti di Calcolatori

I/O Multiplexing.

Paolo D'Arco

Abstract

Scopo della lezione è presentare i vari modelli di I/O, descrivere la funzione *select*(), e costruire un echo client robusto ed un echo server che gestisce più connessioni simultaneamente senza creare processi figli.

1 Gestione contemporanea di diversi input

Qualche lezione fa abbiamo analizzato le varie situazioni d'errore che possono verificarsi durante l'esecuzione di echo client ed echo server. Se ricordate, echo client gestisce contemporaneamente due input: i dati da tastiera e i dati ricevuti da socket. Non appena riceve dati da tastiera (*fgets*() ritorna), li scrive (tramite *reti_writen*()) sul socket e attende (tramite *reti_readline*()) una risposta dal server.

Abbiamo discusso la situazione in cui il server termina la propria esecuzione mentre il client è bloccato in *fgets*(), in attesa di dati dall'utente. Il client non riceve il FIN del server, inviato lungo il socket, fino a quando non giunge input da tastiera, istante in cui il client scrive sul socket e, nel tentativo di leggere la risposta del server, riceve dalla funzione *reti_readline*() immediatamente 0 (EOF), a causa del FIN spedito prima dal server.

Normalmente, come accade per *fgets*(), una funzione di I/O si blocca se non ci sono dati disponibili (nel caso specifico, dati da leggere). Servirebbe un modo per *spostare l'attesa da un canale specifico a più canali*, in modo tale da leggere dal primo che produce dati. Nel nostro esempio, se echo client leggesse prima il FIN dal socket, eviterebbe di attendere dati da tastiera che non potranno più raggiungere il server, non scriverebbe sul socket e terminerebbe l' esecuzione, eventualmente indicando all'utente che la causa della propria terminazione è la terminazione improvvisa del server.

Questa capacità è chiamata *I/O multiplexing* e viene fornita dalle funzioni *select*() e *poll*(). Prima di descriverle ed usarle, cerchiamo di capire meglio come può essere gestito l'I/O.

1.1 Modelli di I/O

In Unix sono disponibili cinque modalità diverse di I/O.

1. Bloccante
2. Non bloccante

3. I/O Multiplexing
4. Guidato dai segnali
5. Asincrono

Per capire le differenze tra i vari modelli, analizziamo un' operazione di lettura. Solitamente sono individuabili due fasi:

- Attesa che i dati siano disponibili all'interno del kernel, pronti per essere letti
- Copia dei dati dal kernel al processo

Relativamente ai socket, la prima fase significa essenzialmente attendere che i dati giungano all'host tramite la rete. La seconda richiede, invece, che i dati dal buffer del kernel vengano copiati nel buffer dell'applicazione.

La modalità più importante è quella bloccante. Normalmente le primitive di I/O sono per default bloccanti. In tutti gli esempi considerati fino ad ora, le primitive sono bloccanti. Graficamente, immaginando di lavorare con socket UDP, il comportamento bloccante di una primitiva quale *recvfrom*(), è il seguente:

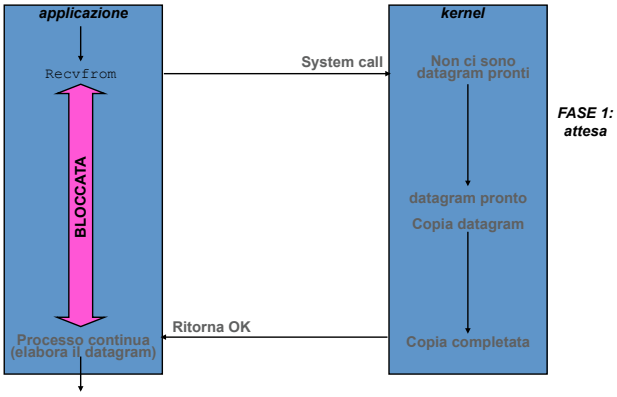


Figure 1: I/O Bloccante

Il processo invoca la chiamata di sistema e la chiamata di sistema non ritorna fino a quando il datagram non arriva e viene copiato nel buffer dell'applicazione o si verifica un errore. Pertanto, il processo è *bloccato* nell'intervallo di tempo che va dall'invocazione della chiamata di sistema al suo ritorno.

Viceversa, quando inizializziamo un socket in modalità non bloccante, è come se dicessimo al kernel "quando l'operazione di I/O richiesta non può esser completata (senza arrestare il processo fino a che i dati non risultino disponibili), non bloccare il processo e restituiscimi un errore". Nell'esempio riportato in Figura 2, alle prime due invocazioni della chiamata di sistema *recvfrom*(), poichè non

ci sono dati disponibili, il kernel restituisce immediatamente l'errore EWOULDBLOCK. Questa fase di interrogazioni ripetute da parte del processo viene detta *polling*. Alla terza invocazione, invece, un datagram è disponibile. Pertanto, il datagram viene copiato nel buffer dell'applicazione, e *recvfrom*() ritorna con successo.

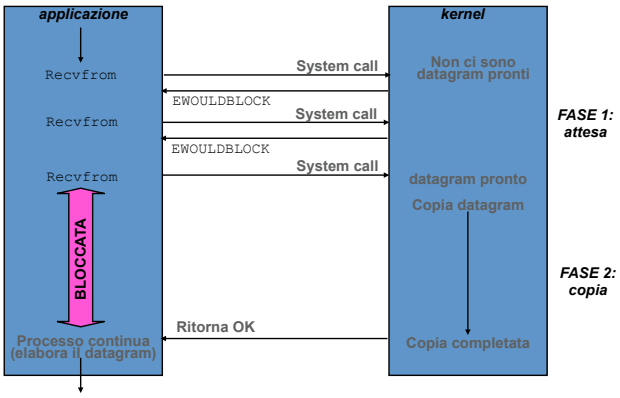


Figure 2: I/O Non bloccante

Utilizzando la modalità multiplexing, in Figura 3, invochiamo la funzione *select*() o la funzione *poll*() e ci blocchiamo in una di queste due chiamate, invece di arrestarci in *recvfrom*(). Quando *select*() ritorna perchè i dati sono disponibili, allora chiamiamo *recvfrom*() per copiare il datagram dal buffer di sistema al buffer dell'applicazione. Tale uso della funzione non sembra fornire alcun vantaggio: invece di attendere in *recvfrom*() attendiamo in *select*(). Tuttavia, utilizzando più canali di input contemporaneamente, i vantaggi appariranno chiari.

Un altro modo per gestire l'I/O sfrutta il meccanismo dei segnali. In breve, possiamo chiedere al kernel di avvisarci quando i dati sono disponibili, inviandoci il segnale SIGIO. Questa forma prende il nome di I/O guidato dai segnali e richiede essenzialmente due azioni:

1. Un'opportuna inizializzazione del socket affinchè SIGIO venga notificato al processo quando il socket può esser letto.

2. L'installazione di un apposito gestore per il segnale SIGIO che o invoca direttamente la chiamata di sistema *recvfrom*(), per leggere il datagram nel buffer dell'applicazione, oppure notifica al programma principale che il datagram è disponibile ed il programma principale si fa poi carico del recupero.

Indipendentemente da come viene organizzata la computazione al punto 2., il vantaggio principale che l' I/O guidato dai segnali porta con sè è che il processo *non* attende e *non* fa polling se il datagram non è disponibile: continua la propria elaborazione, eventualmente fa altro, e quando riceve la notifica sa che il datagram è presente nel buffer del kernel ed è pronto per essere letto e copiato nel buffer dell'applicazione.

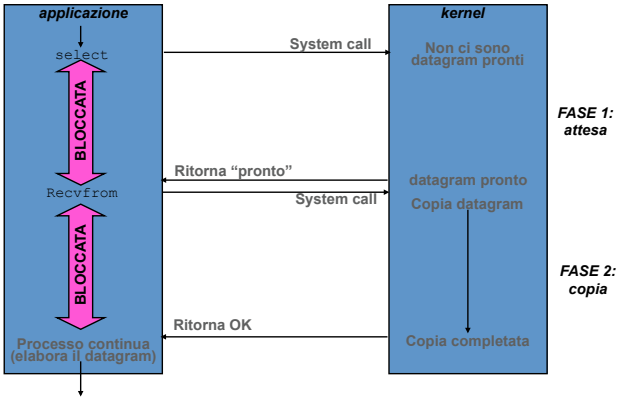


Figure 3: I/O multiplexing

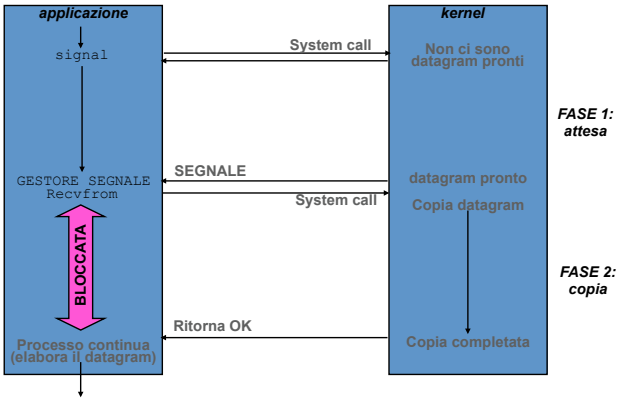


Figure 4: I/O Guidato dai segnali

L'ultima modalità di gestione dell'I/O è la modalità asincrona, riportata in Figura 5. Con essa il processo richiede l'operazione di I/O e riceve una notifica quando l'operazione è stata completata ed il datagram è nel buffer dell'applicazione. Si noti che, seppur simile, esiste una differenza sostanziale tra la modalità di I/O guidata dai segnali e la modalità asincrona. Nel primo caso, la notifica attraverso il segnale I/O dice soltanto che i dati sono disponibili. E' comunque il processo, tramite la chiamata *recvfrom*(), a prelevare i dati e a copiarli nel buffer dell'applicazione o direttamente dall'handler del segnale o dal corpo principale del programma. In altre parole, la notifica dice al processo che l'operazione di lettura *può essere iniziata*. Viceversa, nella modalità asincrona, la notifica indica che l'operazione è stata *completata*.

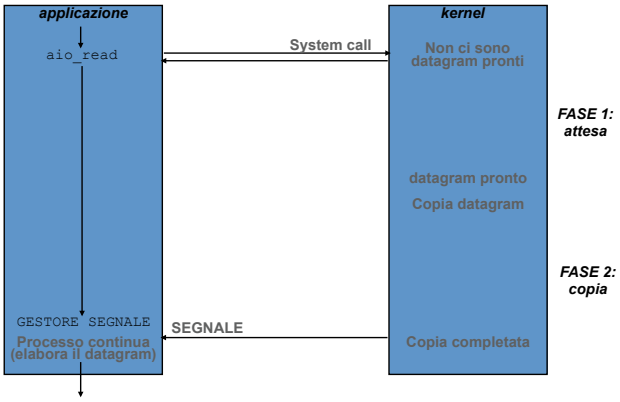


Figure 5: I/O Asincrono

Riguardando le cinque figure, è facile notare che le prime quattro modalità differiscono nella prima fase. La seconda è la stessa. Viceversa, la quinta modalità gestisce entrambe le fasi in un modo unitario.

1.2 Funzione *select*()

Tramite la funzione *select*() un processo chiede al kernel di esser posto in attesa fino a quando *uno o più* tra un insieme di possibili eventi non si sia verificato oppure sia trascorso un tempo massimo stabilito all'atto della richiesta. Per esempio, un processo può usare *select*() per dire al kernel di ritornare solo quando:

- uno qualsiasi dei descrittori nell'insieme {1,4,5} sia pronto per la lettura
- uno qualsiasi dei descrittori nell'insieme {2,6} sia pronto per la scrittura
- una condizione d'errore risulta pendente su uno dei descrittori nell'insieme {1,2}
- sono trascorsi 5 secondi

Quindi, *select*() permette di specificare i descrittori di interesse, eventi quali lettura, scrittura, condizioni di errore, e tempo massimo di attesa. Sintatticamente, è definita come segue:

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

int select(int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptionset, const struct
timeval *timeout);

valore di ritorno: -1 se errore
                   0 se timeout
                   numero descrittori pronti altrimenti
```

Cominciamo dall'ultimo argomento. La struttura timeval è

```
struct timeval {
     long tv_sec;         /* seconds */
     long tv_usec;       /*microseconds */
}
```

Ci sono tre possibilità di usare questo parametro. Possiamo impostare:

1. *Attesa infinita.* La funzione ritorna quando uno dei descrittori è pronto per I/O. Questa condizione può essere specificata passando a *select*() un puntatore nullo NULL.

2. *Attesa finita.* La funzione ritorna quando uno dei descrittori è pronto per I/O oppure quando è trascorso il tempo massimo specificato nella struttura[1].

3. *Attesa nulla.* Se non ci sono descrittori pronti, la funzione ritorna immediatamente. Questa condizione può essere specificata passando a *select*() una struttura timeval con entrambi i campi posti a zero.

Gli argomenti di tipo fd_set rappresentano i descrittori di interesse. Il tipo fd_set è definito come un *vettore di interi*. Ogni intero è una stringa di 32 bit. Ogni bit dell'intero rappresenta un descrittore. Per intenderci, il primo intero rappresenta i descrittori da 0 a 31, il secondo da 32 a 63, e così via. I tre argomenti, **readset*, **writeset* e **exceptionset*, rappresentano, rispettivamente, i descrittori da cui il processo vuole leggere, quelli su cui vuole scrivere, e quelli su cui si aspetta eventuali condizioni di errore (in queste lezioni non ci occuperemo delle condizioni di errore, per cui useremo per questo parametro NULL).

Gli insiemi di descrittori vengono gestiti attraverso le macro che seguono:

```
#include <sys/select.h>
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

[1] Anche se possiamo specificare i microsecondi alcuni sistemi operativi arrotondano a multipli di 10 microsecondi. Inoltre, anche se la specifica usa il qualificatore const per la struttura timeval, alcuni sistemi al ritorno modificano la struttura, in modo tale da contenere il tempo rimanente allo scadere del timeout.

Le macro servono per togliere un descrittore dall'insieme (FD_CLR), mettendo il corrispondente bit a zero, controllare che uno specifico descrittore sia pronto (FD_ISSET), cioè che il corrispondente bit sia uguale a uno, dichiarare che uno specifico descrittore è di interesse (FD_SET), cioè porre il corrispondente bit a uno, e porre a zero tutti i bit che rappresentano l'insieme (FD_ZERO), operazione effettuata solitamente in fase di inizializzazione. E' inoltre possibile copiare un insieme di descrittori in un altro attraverso l'operatore di assegnamento = del linguaggio C.

Infine, il primo argomento, rappresenta il numero di descrittori da controllare, calcolato sommando al descrittore massimo il valore uno, visto che il primo descrittore è il bit 0 del primo intero del vettore. Per esempio, se siamo interessati ai descrittori $1, 3, 4, 6, 9$, *maxfdp1* deve valere 10. Nota che la costante FD_SETSIZE, definita in <sys/select.h>, rappresenta il numero massimo di descrittori di un insieme fd_set, ed è in genere uguale a 1024. Passando a *select*() il valore *maxfd*, il sistema evita di controllare l'intero insieme e si limita a quelli effettivamente usati. Pertanto si tratta di un parametro usato per ragioni di *efficienza*.

In gergo, diremo che un socket che può essere letto risulta *leggibile* ed un socket che ha spazio nel buffer per ricevere dati risulta *scrivibile*.

Un socket è leggibile quando

- Nel buffer di lettura c'è almeno un byte. In realtà, esiste un valore soglia (detto *low-water mark*) e il socket risulta leggibile se il numero di byte supera la soglia. Il valore soglia per default è posto a 1, ma può essere modificato usando una opzione dei socket.
- Il socket è stato chiuso in lettura. Per esempio, è stato ricevuto FIN dal server. In questo caso, un'operazione di lettura non provoca l'arresto ma restituisce immediatamente 0 (i.e., EOF).
- Il socket è un socket di ascolto (*listening socket*) e, nella coda delle connessioni complete ci sono delle entry.
- Si è verificata una condizione di errore. In questo caso, un'operazione di lettura ritornerà -1 e la variabile *errno* indicherà la condizione di errore.

Similmente, un socket è scrivibile quando

- Il buffer di scrittura ha più di 2048 byte di spazio disponibile e il socket è connesso oppure il socket non richiede la connessione (e.g., socket UDP). Anche in questo caso, la soglia (*low-water mark*) può essere modificata tramite un'apposita opzione dei socket. L'operazione di scrittura restituisce il numero di byte effettivamente scritti nel buffer.
- Il socket è stato chiuso in scrittura. Un'operazione di scrittura sul socket genera il segnale SIGPIPE.
- Si è verificata una condizione di errore. In questo caso, un'operazione di scrittura ritornerà -1 e la variabile *errno* indicherà la condizione di errore.

1.3 Un esempio: echo client con *select*()

Riscriviamo la funzione che invoca il nostro echo client, chiamandola client_echo_select(), in modo tale da gestire contemporaneamente i due canali di input.

```
void client_echo_select(FILE *fp, int sockfd) {
1.  int     maxfdl, n;
2.  fd_set  rset;
3.  char    sendline[MAXLINE], recvline[MAXLINE];

4.  FD_ZERO(&rset);
5.  for( ; ; ) {
6.  FD_SET(fileno(fp), &rset);
7.  FD_SET(sockfd, &rset);
8.  maxfdl = MAX(fileno(fp), sockfd) + 1;
9.    if( select(maxfdl, &rset, NULL, NULL, NULL) < 0 ){
10.      printf("select error\n"); exit(1);
        }

11.  if( FD_ISSET(sockfd, &rset) ) {
12.     if ( (n = reti_readline(sockfd, recvline, MAXLINE)) < 0) {
13.           if( errno == EPIPE ) {
14.               printf("%s [%d]: server disconnesso\n",__FILE__,__LINE__);
15.               break;
16.           }
17.           else {
18.                printf("readline error\n"); exit(1);
19.               }
20.         }
21.       if (n == 0){
22.         printf("%s [%d]: server disconnesso\n",__FILE__,__LINE__);
23.         exit(1);
24.      }
25.       fputs(recvline, stdout);
26.   }
27.   if( FD_ISSET(fileno(fp), &rset) ) {
28.      if( fgets(sendline, MAXLINE, fp) == NULL)
29.          return;
30.      if( (reti_writen(sockfd, sendline, strlen(sendline))) < 0){
31.         printf("write error\n");
32.         exit(1);
33.       }
34.    }
35.  }
36. }
```

Il client dichiara le variabile di cui ha bisogno (linee 1. - 3.), azzera l'insieme dei descrittori (linea 4.), ed entra in ciclo (linea 5.). All'interno del ciclo setta i descrittori di interesse (linee 6. e 7.) ed invoca *select*() (linee 8. - 10.) A questo punto, durante l'esecuzione, il client viene arrestato e riprende non appena uno dei due socket, *stdin* o *sockfd* risulti leggibile. Alla ripresa, controlla se è leggibile il socket (linea 11.). In tal caso, legge (linea 12.) e stampa a video (linea 25.) gestendo le eventuali situazioni di errore (linee 13. - 24.) Successivamente, controlla se è leggibile *stdin* (linea 27.). Se così, legge (linea 28.) e scrive sul socket (linea 30.), gestendo eventuali situazioni di errore.

Graficamente, il nostro client gestisce contemporaneamente le condizioni riportate in Figura 6

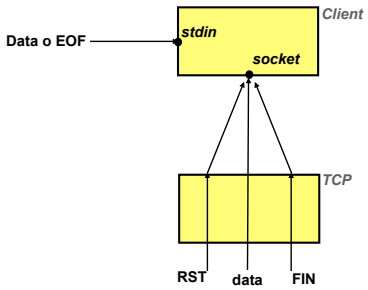


Figure 6: Condizioni gestite da *select*() in lettura

A parte l'input da tastiera, il client gestisce tre condizioni relativamente al socket:

- La disponibilità di dati nel buffer del socket.
- La presenza di un FIN, inviato dall'altra estremità della comunicazione.
- La presenza di un RST, inviato a seguito di crash o reboot dell'altra estremità della comunicazione.

In tutti e tre i casi il socket diventa *leggibile.*

1.4 Shutdown della connessione

Il nostro client ancora non risulta robusto. Per capire il perchè, si analizzi per un attimo la vecchia versione, che non utilizza *select*(). Il client opera secondo la modalità *stop-and-wait*, cioè invia dati, si ferma e attende la risposta del server. Come vedrete nel corso di teoria, il tempo d'attesa, detto RTT (round-trip-time), equivale al tempo che mediamente un pacchetto impiega per andare da una estremità all'altra. Graficamente, una interazione client-server può essere pertanto rappresentata come in Figura 7. Una richiesta viene inviata dal client al tempo 0, raggiunge il server al tempo 3, viene elaborata (assumiamo che questo tempo sia idealmente nullo), e viene spedita la risposta al client, che giunge al tempo 7.

L'esempio mostra chiaramente che stiamo sfruttando soltanto l'ottava parte della capacità trasmissiva del canale di comunicazione. Per input interattivi, la nostra applicazione client-server che usa la funzione *select*() funziona bene. Tuttavia, se usiamo la *redirezione* di input ed output su file che in Unix si ottiene usando il comando

echocli 127.0.0.1 9999 < file_input > file_output

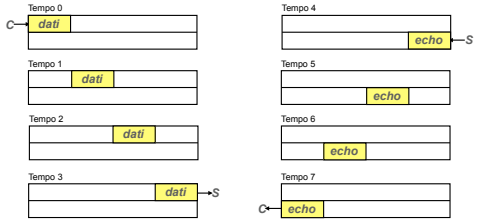


Figure 7: Esecuzione secondo la modalità Stop and Wait

si può notare che file_output è sempre più corto di file_input. Cosa accade? Usando la redirezione inviamo linee tra le parti alla massima velocità e sfruttiamo *pienamente* il canale di comunicazione. Graficamente, cioè, l'interazione può essere rappresentata come in Figura 8. L'ultima richiesta

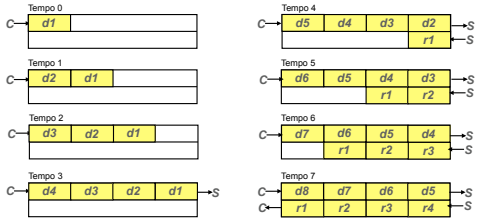


Figure 8: Esecuzione con redirezione di input e output

viene inviata al tempo 7 ma *non possiamo* chiudere il socket perchè ci sono ancora dati nel canale, inviati dal server ma non ancora ricevuti dal client. Purtroppo, il nostro client, non appena riceve EOF dalle operazioni di lettura da *stdin*, termina la propria esecuzione e, di conseguenza, chiude anche il socket. I dati *in viaggio* non raggiungeranno mai il client. Ecco la ragione del file di output più corto.

Occorrerebbe un modo per chiudere *a metà* una connessione TCP: in questo caso, chiudere il socket solo in scrittura, lasciandolo aperto in lettura. Tale operazione è possibile invocando la funzione *shutdown*(), definita come segue:

```
#include <sys/socket.h>
int shutdown (int sockfd, int howto);

valore di ritorno: -1 se errore
                    0 se OK
```

Il parametro *howto*, può assumere tre valori:

- SHUT_RD - Il socket viene chiuso in lettura. Nessun ulteriore dato può essere ricevuto sul socket e qualsiasi dato non ancora letto viene eliminato.
- SHUT_WR - Il socket viene chiuso in scrittura. Nel caso si stia usando il protocollo TCP, questa opzione definisce la cosiddetta *mezza chiusura.* I dati già presenti nel buffer del socket vengono spediti. Nessuna operazione di scrittura è più ammessa.
- SHUT_RDWR - Il socket viene chiuso in lettura e scrittura. Equivale ad invocare due volte *shutdown*, prima con l'opzione SHUT_RD e poi SHUT_WR.

In sintesi, le differenze fondamentali tra *close*() e *shutdown*() sono due:

- *close*() decrementa il contatore dei riferimenti e chiude il socket se raggiunge il valore zero. *shutdown*() inizia la sequenza di chiusura della connessione, indipendentemente dal valore del contatore di riferimenti
- *close*() chiude il socket in lettura e scrittura. *shutdown*() permette di chiudere solo in un verso.

Utilizzando la funzione *shutdown*(), possiamo riscrivere il nostro echo client come segue:

```
void client_echo_select(FILE *fp, int sockfd) {
1.  int     maxfdl, n, stdineof;
2.  fd_set  rset;
3.  char    sendline[MAXLINE], recvline[MAXLINE];

4.  FD_ZERO(&rset);
5.  for( ; ; ) {
6.  FD_SET(fileno(fp), &rset);
7.  FD_SET(sockfd, &rset);
8.  maxfdl = MAX(fileno(fp), sockfd) + 1;
9.    if( select(maxfdl, &rset, NULL, NULL, NULL) < 0 ){
10.      printf("select error\n"); exit(1);
        }
11.  if( FD_ISSET(sockfd, &rset) ) {
12.     if ( (n = reti_readline(sockfd, recvline, MAXLINE)) < 0) {
13.           if( errno == EPIPE ) {
14.               printf("%s [%d]: server disconnesso\n",__FILE__,__LINE__);
15.               break;
16.           }
17.           else {
18.                printf("readline error\n"); exit(1);
19.               }
20.         }
21.       if (n == 0){
22.         if( stdineof == 1 )
23.              return;
24.         else{
25.         printf("%s [%d]: server disconnesso\n",__FILE__,__LINE__);
26.         exit(1);
27.      }
28.       fputs(recvline, stdout);
29.   }
30.   if( FD_ISSET(fileno(fp), &rset) ) {
31.    if( fgets(sendline, MAXLINE, fp) == NULL) {
32.    stdineof = 1;
33.    shutdown(sockfd, SHUT_WR);
34.    FD_CLR(fileno(fp), &rset);
35.    continue;
36.     }
37.   if( (reti_writen(sockfd, sendline, strlen(sendline))) < 0){
38.     printf("write error\n");
39.     exit(1);
40.    }
41.   }
42. }
43.}
```

Echo client, quando non riceve più input da tastiera (*fgets*() restituisce il puntatore NULL), setta la variabile stdineof ad uno e invoca la funzione *shutdown*() per chiudere il socket in scrittura

(linee 31.-33). Inoltre rimuove stdin dall'insieme dei descrittori che la *select*() deve controllare per leggibilità (linea 34.). Successivamente, quando non riceve più dati dal socket, se *readline*() ha restituito zero e non giungeranno più dati da tastiera (stdineof è uguale a 1), allora il server torna in attesa di nuove connessioni.

1.5 Echo server con *select*

La funzione *select*() permette di riscrivere anche il nostro echo server. Al posto di creare un processo figlio per ogni una nuova connessione, echo server usa *select*() per gestire *contemporaneamente tutte le connessioni*. A tal fine, echo server usa le seguenti strutture di dati (vedi Figura 9):

- Un array *rset*, che contiene i descrittori di file e socket utilizzati (i.e., *stdin, stdout, stderr*, il socket di ascolto e i socket di connessione)
- Un array di interi *client*, che contiene il descrittore del socket di connessione, per ogni client connesso. Inizialmente, tutte le entry del vettore sono uguali a -1.

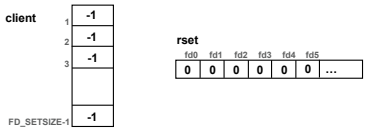


Figure 9: Strutture usate da echo server

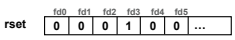


Figure 10: Array rset aggiornato

Il server crea il socket di ascolto, invocando *listen*(). Supponendo che *stdin, stdout, stderr* siano aperti (il processo viene eseguito interattivamente, non in background), il primo descrittore libero per il socket è 3. Pertanto, l'array *rset* viene aggiornato come in Figura 10. Il valore *maxfdp1*, primo parametro di *select*() vale 4.

Non appena il primo client stabilisce una connessione con il server, il socket di ascolto diventa leggibile ed il server invoca *accept*(). In accordo alle assunzioni di questo esempio, il nuovo descrittore restituito da *accept*() è 4. Il server memorizza il nuovo socket connesso, memorizzando il descrittore nella prima posizione libera dell'array *client* e aggiunge il socket all'insieme dei descrittori *rset* da controllare tramite *select*(). All'arrivo di un secondo client, lo stesso processo si ripete: il nuovo descrittore viene memorizzato nella prima posizione disponibile di *client* ed *rset* viene aggiornato. La Figura 11 illustra l'arrivo dei due client appena descritto e le modifiche alla configurazione di Figura 10. Quando un client termina la propria esecuzione, la corrispondente entry in *client* viene riposta a -1 e il bit associato al descrittore in *rset* viene rimesso a zero. Nel nostro esempio, se termina il primo client, le strutture vengono aggiornate come in Figura 12.

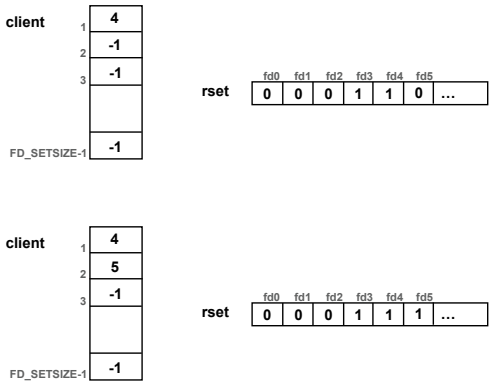


Figure 11: Aggiornamento di *client*() ed *rset*.



Figure 12: Rimozione di un *client*().

In conclusione, l'arrivo di un client comporta la memorizzazione del descrittore del nuovo socket di connessione nella prima posizione libera di *client* e l'impostazione ad 1 del corrispondente bit in *rset*, al fine di gestirlo tramite *select*(). La terminazione, viceversa, comporta la liberazione della posizione occupata dal descrittore di connessione nell'array *client* e la riposizione a 0 del corrispondent bit in *rset*.

Il codice del server, diviso per comodità di presentazione in due parti, è il seguente:

```
#include        "basic.h"

int main(int argc, char **argv) {
1. int                      listenfd, connfd, sockfd;
2. int                      i, maxi, maxfd;
3. int                      ready, client[FD_SETSIZE];
4. char                     buff[MAXLINE];
5. fd_set                   rset, allset;
6. ssize_t                  n;
7. struct sockaddr_in       servaddr, cliaddr;
8. socklen_t                cliaddr_len;

9.   if (argc != 2){
10.  printf("Usage: echosrv-slct <PORT> \n"); exit(1);
11.  }
12.  if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
13.   printf("socket error\n"); exit(1);
14.  }
15.  bzero(&servaddr, sizeof(servaddr));
16.  servaddr.sin_family      = AF_INET;
17.  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
18.  servaddr.sin_port        = htons(atoi(argv[1]));

19.  if( (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0){
20.   printf("bind error\n"); exit(1);
21.  }
22.  if( listen(listenfd, BACKLOG) < 0 ){
23.   printf("listen error\n"); exit(1);
24.  }
25.  maxfd = listenfd;   /* numero di descrittori */
26.  maxi = -1;
27.  for ( i = 0; i < FD_SETSIZE; i++)
28.     client[i] = -1; /* inizializza l'array client[] */
29.  FD_ZERO(&allset);   /* azzera tutti i descrittori */
30.  FD_SET(listenfd, &allset); /* setta il descrittore di ascolto */
```

La prima parte del server non presenta molte novità, a parte la dichiarazione di variabili e strutture (linee 1. - 8.) che saranno usate nel seguito e l'inizializzazione dell'array client[] e dell'insieme di descrittori (linee 25. - 30). Viceversa, la seconda parte, ha una struttura sostanzialmente diversa. Le computazioni sono gestite da *select*(), invocata all'interno del loop (linea 33.). Select, quando ritorna restituisce il numero di descrittori leggibili (nready). Se è leggibile il socket d'ascolto (linea

36.) viene creato un nuovo socket di connessione (linee 37. - 40), viene salvato il descrittore nella prima posizione libera dell'array client[] (linee 41. - 44) e, in caso non ve ne fossero (linea 46.) viene fornita indicazione d'errore (linea 47.).

```
31. for ( ; ; ) {
32.   rset = allset; /* assegnamento di strutture */
33.   if( (ready = select(maxfd+1, &rset, NULL, NULL, NULL)) < 0 ){
34.   printf("select error\n"); exit(1);
35.   }
36.   if( FD_ISSET(listenfd, &rset) ) { /* nuovo client */
37.    cliaddr_len = sizeof(cliaddr);
38.    if( (connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len)) < 0){
39.    printf("accept error\n"); exit(1);
40.  }
41.  for(i = 0; i < FD_SETSIZE; i++)
42.    if( client[i] < 0 ) {
43.        client[i] = connfd;  /* salva il descrittore */
44.       break;
45.     }
46.     if( i == FD_SETSIZE ){
47.        printf("troppi client\n"); exit(1);
48.     }
49.     FD_SET(connfd, &allset); /* aggiunge connfd all'insieme */
50.       if( connfd > maxfd )
51.          maxfd = connfd;   /* per la select() */
52.       if( i > maxi )
54.           maxi = i;  /* massimo indice in client[] */
55.       if( --ready <= 0 )
56.          continue;
57.  }
58.  for( i = 0; i <= maxi; i++ ) {  /* controlla tutti i socket attivi */
59.    if( (sockfd = client[i]) < 0 )
60.     continue;
61.   if ( FD_ISSET(sockfd, &rset) ) {
62.                      /* se sockfd \'e leggibile invoca la readline */
63.       if ( (n = reti_readline(sockfd, buff, MAXLINE)) == 0) {
64.                           /* connessione chiusa dal client */
65.          close(sockfd);
66.          FD_CLR(sockfd, &allset); /*rimuove sockfd dall'insieme*/
67.          client[i] = -1; /* cancella sockfd da client[] */
68.       }
69.        else  reti_writen(sockfd, buff, n);
70.        if ( --ready <= 0 )
71.              break; /* non ci sono piu' socket leggibili */
        }
      }
   }
}
```

Inoltre, il descrittore viene aggiunto aggiunto all'insieme dei descrittori che *select*() controlla (linea 49.) e viene aggiornato il valore di maxfd, da usare per la chiamata a *select*() e del valore della variabile maxi, che tiene conto del descrittore più grande nell'array client[] (linee 50. - 54). Infine, se non ci sono altri descrittori leggibili, il server riprende l'esecuzione dall'inizio del loop (linea 32.). Viceversa, se ci sono, l'esecuzione continua con il loop successivo (linea 58.): il server, per ogni descrittore leggibile, legge dal socket (linea 63.) Se ottiene dati, li spedisce al client (linea 70.) mentre se legge EOF, chiude il socket e aggiorna l'array client e l'insieme di descrittori che *select*() deve gestire (linee 65. - 67.). Infine, quando non ci sono più descrittori leggibili, riprende l'esecuzione dal loop principale.

1.6 Denial of Service

Il server presentato non è ancora robusto: un client può bloccarne l'esecuzione per sempre. In tal modo, nessun altro client potrà beneficiare dei servizi del server. Una situazione di questo tipo sono in gergo dette *attacchi di denial of service* (letteralmente significa mancanza di servizio).

Per capire, si supponga che:

- Un client, una volta connesso, invii un byte al server (diverso dal carattere di newline, che client e server hanno stabilito convenzionalmente come terminatore dei record di informazione che si scambiano) e non faccia più nulla.
- Il server invoca *readline*() che legge il singolo byte e poi si blocca in attesa di altri byte e/o del newline.

In tal modo, il server non può servire nessun'altra richiesta. Il server non dovrebbe bloccarsi nel servire un singolo client. Soluzioni possibili sono: l'uso di un processo diverso per ogni client, l'utilizzo di un timeout sulle operazioni di I/O, oppure usare primitive di I/O in modalità non bloccante.

1.7 Conclusioni.

Nella lezione di oggi, abbiamo imparato che:

- Esistono diverse modalità di gestione dell'I/O. Solitamente, le primitive sono implementate per realizzare la modalità bloccante: se i dati non sono disponibili, il processo viene arrestato in attesa che lo diventino. Abbiamo però visto che esistono anche le modalità non bloccante, la tecnica del multiplexing, l'I/O guidato dai segnali e l'I/O asincrono.
- La funzione *select*() permette di implementare l'I/O multiplexing. Abbiamo visto come può essere usata per scrivere una versione di echo client robusta, e una versione di echo server che non crea un processo per servire ogni nuovo client, ma utilizza il multiplexing.
- La funzione *shutdown*(), che permette di chiudere un socket soltanto in lettura o in scrittura, evita di perdere pacchetti spediti ed ancora in transito sul canale al momento della chiusura della connessione di una delle parti.
- Occorre essere accorti nella realizzazione di un server che gestisce con la tecnica del multiplexing diversi client, al fine di evitare attacchi di tipo denial of service.

1.8 Esercizi

Esercizio 1. Nelle lezioni precedenti abbiamo scritto una piccola applicazione che realizza un mini-smtp client ed un mini-smtp server. Si riscriva mini-smtp client usando la funzione select per la gestione dei due canali stdin e socket.

Esercizio 2. Si modichi echosrv-slct in modo tale che il server stampi a video indirizzo e numero di porta di un client, sia al momento della connessione che al momento della disconnessione.

Esercizio 3. Si scriva un programma di chat. Il server accetta le richieste di connessione da parte dei client e funziona come punto di raccolta dei messaggi. Ogni messaggio spedito da un client al server viene inviato dal server a tutti gli altri client.

Esercizio 4. Si scriva un programma che realizza una crypto-chat. Il server accetta le richieste di connessione da parte dei client e funziona come punto di raccolta dei messaggi. Inoltre,

- Ogni client, nel momento in cui si connette al server, fornisce anche un nickname con cui l'utente compare in chat.
- I messaggi vengono inviati sulla rete in forma cifrata.
- La funzione di cifratura è semplice (Cifrario di Cesare): ogni carattere viene sostituito con quello che segue di *tre* posizioni nell'alfabeto (e.g., "a" viene sostituita con "d", "b" con "e" etc ...). La funzione di decifratura opera la trasformazione inversa.
- Il server stampa a video informazioni (i.e., IP, numero di porta, nickname ...) su ogni nuovo client che si connette e che si disconnette.

Esercizio 5. Si modifichi la crypto-chat precedente in modo tale ogni singolo client possa specificare se invierà e desidera ricevere messaggi in chiaro o in forma cifrata.

Esercizio 6. Si scriva un programma che realizza un server mini-ftp. Il server funziona al modo seguente:

- Ad ogni connessione di un nuovo client, il server invia al client un messaggio di benvenuto ("Mini-ftp server v. 1.0: benvenuto").
- Successivamente, il Client può inviare tre comandi: LS, GET <nomefile> e BYE. Inviando LS riceve la lista dei file presenti nella directory del server. Con GET <nomefile> riceve e memorizza il file remoto <nomefile> nella directory locale in <localnomefile>. Con BYE chiede al server di chiudere la connessione.

Esercizio 7. Si realizzi un'applicazione client-server che simula un BANCOMAT. In particolare, il server memorizza in un file di sistema tuple del tipo (numero scheda, PIN, nome-utente, saldo) per tutti gli utenti del sistema. Per ogni utente, il sistema mantiene un file nominato <nome-utente>, che memorizza tutte le operazioni effettuate con data e ora dell'operazione. Le interazioni vengono gestite come segue:

- Il client chiede all'utente il numero della scheda e il codice PIN e li invia al server. Il server controlla nel file di sistema e, se trova una tupla con il numero di scheda ed il PIN inviati, risponde con un messaggio di benvenuto che riporta il nome dell'utente e con un menù contenente le operazioni attive. Supponiamo siano: *Prelievo, Deposito, Saldo, ListaOperazioniEffettuate.*

- Se l'utente chiede di prelevare, usando il comando *Prelievo: XXX*, il server controlla che la quantità di denaro richiesta sia disponibile, aggiorna il saldo e risponde con il messaggio *Banconote disponibili in pochi secondi.*
- Se l'utente chiede di versare, usando il comando *Deposito: XXX*, il server accredita la quantità di denaro richiesta, aggiorna il saldo e risponde con il messaggio *Deposito effettuato.*
- Se l'utente chiede il saldo, il server legge e invia il saldo al client.
- Se l'utente chiede la lista delle operazioni effettuate, il server apre il file <nome-utente> lo legge e ne invia al client il contenuto riga per riga.