Socket II

Leonardo Querzoni

querzoni@dis.uniroma1.it
http://www.dis.uniroma1.it/~querzoni

MIDLAB
http://www.dis.uniroma1.it/~midlab

Sommario

- Richiami sui processi
- Echo server TCP (multi processo)
- Richiami sui thread
- Echo server TCP (multi thread)
- I/O multiplexing
- Esempio di domanda d'esame

Richiami sui processi

Funzioni di base:

```
pid_t fork();
```

Crea un nuovo processo figlio. La funzione copia l'immagine di memoria del processo padre in quella del processo figlio. Conclusa questa operazione, entrambi i processi riprendono l'esecuzione dall'istruzione successiva a *fork()*.
La funzione ritorna il valore 0 al processo figlio, mentre riporta il pID del nuovo processo creato al processo padre.

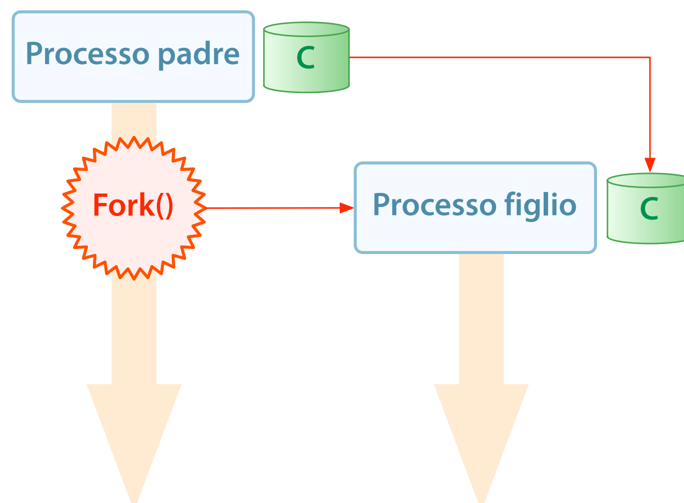
```
pid_t getpid();
pid_t getppid();
```

Restituiscono il pID rispettivamente del processo corrente o di quello padre.

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

Mette il processo corrente in attesa che il processo figlio termini. Il pID del processo figlio viene restituito, mentre il codice di ritorno con cui è terminato viene scritto nella locazione puntata da *stat_loc*.

Richiami sui processi



Richiami sui processi

Struttura di un generico programma multiprocesso:

```
/* inizializzazione comune */

if (!fork()) {

        /* codice eseguito SOLO dal processo figlio */

} else {

        /* codice eseguito SOLO dal processo padre */

}

/* finalizzazione comune */
```


TCP server multiprocesso (1)

```
/*
        TCP Server multiprocesso
*/

/* dichiarazioni ed include */

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

        /* definizione variabili */

        /* Interpretazione linea di comando */
        ParseCmdLine(argc, argv, &endptr);

        /* creazione socket */

        /* bind e listen sul socket */

        /* loop infinito */


        /* Chiusura del socket in ascolto */
        if ( close(list_s) < 0 ) {
            fprintf(stderr, "server: errore durante la close.\n");
            exit(EXIT_FAILURE);
        }
}
```

TCP server multiprocesso (2)

```
/* dichiarazioni ed include */

#include <sys/types.h>        /*  socket types            */
#include <sys/socket.h>       /*  socket definitions      */
#include <arpa/inet.h>        /*  inet (3) funtions       */
#include <unistd.h>           /*  misc. UNIX functions    */
#include <stdlib.h>
#include <stdio.h>

/* Costanti global  */
#define MAX_LINE           (1000)

/* Prototipi funzioni */
int ParseCmdLine(int argc, char *argv[], char **szPort);
ssize_t Readline(int fd, void *vptr, size_t maxlen);
ssize_t Writeline(int fc, const void *vptr, size_t maxlen);
```


TCP server multiprocesso (3)

```
/* definizione variabili */

int       list_s;                /*  listening socket descriptor  */
int       conn_s;                /*  connection socket descriptor */
short int port;                  /*  port number                  */
struct    sockaddr_in servaddr;  /*  socket address structure     */
struct    sockaddr_in their_addr;
char      buffer[MAX_LINE];      /*  character buffer             */
char     *endptr;                /*  for strtol()                 */
int       sin_size;
```

TCP server - ParseCmdLine()

```
/* Interpreta i parametri forniti da linea di comando */

int ParseCmdLine(int argc, char *argv[], char **szPort) {
    int n = 1;
    *szPort = 0;

    while ( n < argc ) {
        if ( !strncmp(argv[n], "-p", 2) || !strncmp(argv[n], "-P", 2) )
            *szPort = argv[++n];
        else if ( !strncmp(argv[n], "-h", 2) || !strncmp(argv[n], "-H", 2) ) {
            printf("Sintassi:\n\n");
            printf("    server -p (porta) [-h]\n\n");
            exit(EXIT_SUCCESS);
        }
        ++n;
    }

    if (*szPort==0) {
        printf("Sintassi:\n\n");
        printf("    server -p (porta) [-h]\n\n");
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```


TCP server multiprocesso (4)

```
/* creazione socket */

port = strtol(endptr, &endptr, 0);
if ( *endptr ) {
        fprintf(stderr, "server: porta non riconosciuta.\n");
        exit(EXIT_FAILURE);
}

printf("Server in ascolto sulla porta %d\n",port);

/*  Create the listening socket  */
if ( (list_s = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        fprintf(stderr, "server: errore nella creazione della socket.\n");
        exit(EXIT_FAILURE);
}
```

TCP server multiprocesso (5)

```
/* bind e listen sul socket */

/* Vuotiamo la struttura servaddr e riempiamo i campi necessari */
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(port);

/* Effettua la bind sull'indirizzo e porta ora specificati */
if ( bind(list_s, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
        fprintf(stderr, "server: errore durante la bind.\n");
        exit(EXIT_FAILURE);
}

/* Mette il socket in modalità di "ascolto" tramite la listen() */
if ( listen(list_s, LISTENQ) < 0 ) {
        fprintf(stderr, "server: errore durante la listen.\n");
        exit(EXIT_FAILURE);
}
```


TCP server multiprocesso (6)

```
/* loop infinito */

while ( 1 ) {

        /*  Attende una nuova connessione tramite la accept() */
        sin_size = sizeof(struct sockaddr_in);
        if ( (conn_s = accept(list_s, (struct sockaddr *)&their_addr, &sin_size) ) < 0 ){
            fprintf(stderr, "server: errore nella accept.\n");
            exit(EXIT_FAILURE);
        }

        printf("server: connessione da %s\n", inet_ntoa(their_addr.sin_addr));

        /*  Crea un nuovo processo a cui affida la comunicazione con il client */
        if (!fork()) {

            /* codice eseguito dal processo figlio */

        }

        /* Chiude la connessione */
        if ( close(conn_s) < 0 ) {
            fprintf(stderr, "server: errore durante la close.\n");
            exit(EXIT_FAILURE);
        }
}
```

TCP server multiprocesso (7)

```
/* codice eseguito dal processo figlio */

/* Chiude il socket su cui è in ascolto il padre */
close(list_s);

/* Legge una riga dal socket e semplicemente la riscrive nel socket stesso */
Readline(conn_s, buffer, MAX_LINE-1);

printf("server: il client ha scritto\n\t%s\n",buffer);

Writeline(conn_s, buffer, strlen(buffer));

/* Chiude la connessione */
if ( close(conn_s) < 0 ) {
        fprintf(stderr, "server: errore durante la close.\n");
        exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
```


Richiami sui thread

Funzioni di base:

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

Crea un nuovo thread eseguendo in esso la funzione *start_routine* con i parametri attuali specificati in *arg*. Il tID del nuovo thread creato viene scritto nella variabile puntata da *thread*. La funzione restituisce 0 in tutti i casi in cui non si verifica errore.

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

Rende il thread, il cui tID è stato passato come parametro, un *detached thread* in modo che possa rilasciare le sue risorse alla fine della sua esecuzione.

Richiami sui thread

Funzioni di base:

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

Mette il thread chiamante in attesa che il thread indicato con il primo parametro termini. La locazione di memoria puntata da *value_ptr* può essere usata per raccogliere un eventuale valore restituito dal thread in uscita. La funzione restituisce un valore diverso da 0 in caso di errore.

`int pthread_exit(void *value_ptr);`

Termina il thread chiamante rendendo disponibile il valore puntato da *value_ptr* per un eventuale *pthread_join* eseguito da un altro thread.

`int pthread_cancel(pthread_t thread);`

Termina il thread indicato come parametro.

TCP server multithread (1)

```
/*
        TCP Server multithread
*/

/* dichiarazioni ed include */

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

        /* definizione variabili */

        /* Interpretazione linea di comando */
        ParseCmdLine(argc, argv, &endptr);

        /* creazione socket */

        /* bind e listen sul socket */

        /* loop infinito */

        /* Chiusura del socket in ascolto */
        if ( close(list_s) < 0 ) {
            fprintf(stderr, "server: errore durante la close.\n");
            exit(EXIT_FAILURE);
        }
}
```

TCP server multithread (2)

```
/* dichiarazioni ed include */

#include <sys/types.h>        /*  socket types            */
#include <sys/socket.h>       /*  socket definitions      */
#include <arpa/inet.h>        /*  inet (3) funtions       */
#include <unistd.h>           /*  misc. UNIX functions    */
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>          /*  gestione POSIX threads  */

/* Costanti global  */
#define MAX_LINE           (1000)

/* Prototipi funzioni */
int ParseCmdLine(int argc, char *argv[], char **szPort);
ssize_t Readline(int fd, void *vptr, size_t maxlen);
ssize_t Writeline(int fc, const void *vptr, size_t maxlen);

void * receiveThread(void * param);
int status;
```


TCP server multithread (3)

```
/* definizione variabili */

int       list_s;                /*  listening socket descriptor  */
int       conn_s;                /*  connection socket descriptor */
short int port;                  /*  port number                  */
struct    sockaddr_in servaddr;  /*  socket address structure     */
struct    sockaddr_in their_addr;
char      buffer[MAX_LINE];      /*  character buffer             */
char     *endptr;                /*  for strtol()                 */
int       sin_size;
pthread_t tid;
void     *status;
```

TCP server multithread (4)

```
/* creazione socket */

port = strtol(endptr, &endptr, 0);
if ( *endptr ) {
        fprintf(stderr, "server: porta non riconosciuta.\n");
        exit(EXIT_FAILURE);
}

printf("Server in ascolto sulla porta %d\n",port);

/*  Create the listening socket  */
if ( (list_s = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        fprintf(stderr, "server: errore nella creazione della socket.\n");
        exit(EXIT_FAILURE);
}
```


TCP server multithread (5)

```
/* bind e listen sul socket */

/* Vuotiamo la struttura servaddr e riempiamo i campi necessari */
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(port);

/* Effettua la bind sull'indirizzo e porta ora specificati */
if ( bind(list_s, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
        fprintf(stderr, "server: errore durante la bind.\n");
        exit(EXIT_FAILURE);
}

/* Mette il socket in modalità di "ascolto" tramite la listen() */
if ( listen(list_s, LISTENQ) < 0 ) {
        fprintf(stderr, "server: errore durante la listen.\n");
        exit(EXIT_FAILURE);
}
```

TCP server multithread (6)

```
/* loop infinito */

while ( 1 ) {

        /*  Attende una nuova connessione tramite la accept() */
        sin_size = sizeof(struct sockaddr_in);
        if ( (conn_s = accept(list_s, (struct sockaddr *)&their_addr, &sin_size) ) < 0 ){
            fprintf(stderr, "server: errore nella accept.\n");
            exit(EXIT_FAILURE);
        }

        printf("server: connessione da %s\n", inet_ntoa(their_addr.sin_addr));

        /*  Crea un nuovo thread a cui affida la comunicazione con il client */
        if(pthread_create(&tid, NULL, receiveThread, (void*)&conn_s)){
            printf("server: impossibile creare il thread. Errore: %d\n", proc_id);
            exit(EXIT_FAILURE);
        }
        pthread_detach(tid);

        /* Chiude la connessione */
        if ( close(conn_s) < 0 ) {
            fprintf(stderr, "server: errore durante la close.\n");
            exit(EXIT_FAILURE);
        }
}
```


TCP server multithread (7)

```
/* Funzione gestita dal thread separato */

void * receiveThread(void * param){

        char buf[MAX_LINE];
        /*  Effettuiamo una copia del socket perche' la variabile originaria
            potrebbe essere riutilizzata prima della chiusura di questo thread */
        int my_sock = *((int*)param);

        Readline(my_sock, buf, MAX_LINE-1);
        printf("server: il client ha scritto\n\t%s\n", buf);
        Writeline(my_sock, buf, strlen(buf));

        close(my_sock);

        /*  Termina il thread */
        pthread_exit((void *)&status);
}
```

I/O multiplexing

Spesso si incontra la necessità di dover controllare contemporaneamente l'input e l'output verso differenti canali di comunicazione (standard I/O, files, sockets, etc.).

Questa possibilità è offerta dalla funzione *select()*:

```
int select(int maxfdp1,
           fd_set *readset,
           fd_set *writeset,
           fd_set *exceptset,
           const struct timeval *timeout);
```

La funzione select fa si che il kernel blocchi l'esecuzione del processo e si metta in attesa che uno o più eventi specificati accadano.

Il parametro ***timeout*** specifica il tempo massimo di attesa desiderato. Scaduto questo intervallo senza l'occorrenza di alcuno degli eventi specificati, l'esecuzione del processo riprende dall'istruzione successiva. Specificando NULL al posto di questo parametro l'attesa sarà indefinita; specificando un intervallo di 0 microsecondi si otterrà una funzionalità del tutto simile al *polling* (verrà controllata una volta l'occorrenza degli eventi, dopodiché la *select()* verrà interrotta).

I/O multiplexing

I parametri ***readset***, ***writeset*** *ed* ***exceptset*** specificano i file descriptor che noi desideriamo vengano controllati. In modo più specifico il kernel controllerà l'occorrenza di eventi di lettura sul primo set, scrittura sul secondo ed errori sul terzo. Ciascuno di questi tre argomenti è costituito da un *descriptor set*, ovvero un array di interi in cui ciascun bit di ogni intero rappresenta un *file descriptor*. L'esatta implementazione varia a seconda del SO, ma vengono sempre messe a disposizione delle macro per la gestione dei set:

```
void FD_ZERO(fd_set *fdset);
```

Imposta a 0 tutti i bit di *fdset*.

```
void FD_SET(int fd, fd_set *fdset);
```

Imposta a 1 il bit di *fdset* relativo al file descriptor *fd*.

```
void FD_CLR(int fd, fd_set *fdset);
```

Imposta a 0 il bit di *fdset* relativo al file descriptor *fd*.

```
int FD_ISSET(int fd, fd_set *fdset);
```

Restituisce un valore diverso da 0 se il bit di *fdset* relativo a *fd* è impostato a 1.

I/O multiplexing

Il parametro *maxfdp1* deve essere impostato pari al massimo dei file descriptors da controllare più 1. Dopo la chiamata a *select()* il kernel andrà a testare lo stato dei file descriptors compresi tra 0 e *maxfdp1-1*. Questa variabile deve essere impostata correttamente per ottenere il massimo dell'efficienza.

La funzione ritorna il numero di eventi occorsi; quando il processo viene riavviato negli *fdset* saranno rimasti impostati a 1 solo i bit relativi ai file descriptors sui quali si è verificato un evento.

L'uso della funzione select normalmente prevede i seguenti passi:

1. creazione degli *fdset*
   - 1.a inizializzazione di ogni *fdset* tramite *FD_ZERO*
   - 1.b inserimento dei file descriptors in ogni *fdset* tramite *FD_SET*
2. calcolo di *maxfdp1*
3. chiamata a *select()*
4. check sui bit impostati ad 1 all'interno degli *fdset*

TCP client (1)

```
/*      TCP Client       */

/* dichiarazioni ed include */

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

        if(argc!=2){
            printf("uso: %s <indirizzo IP>\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(7000);
        inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

        connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

        str_cli(stdin, sockfd,);

        exit(EXIT_SUCCESS);
}
```

TCP client (2)

```
int str_cli(FILE *fp, int sockfd){
        int    maxfdp1;
        fd_set rset;
        char   sendline[MAX_LINE], recvline[MAX_LINE];

        FD_ZERO(&rset); /* Inizializzazione di fdset */

        while( 1 ){
            FD_SET(fileno(fp), &rset); /* Inserimento dei file descriptors */
            FD_SET(sockfd, &rset);     /* in fdset */

            maxfdp1 = max(fileno(fp), sockfd) + 1; /* Calcolo di maxfdp1 */

            select(maxfdp1, &rset, NULL, NULL, NULL); /* Chiamata a select() */

            if(FD_ISSET(sockfd, &rset)){ /* Check sui bit impostati ad 1 */
                if(readline(sockfd, recvline, MAX_LINE))
                    exit(EXIT_FAILURE);
                fputs(recvline, stdout);
            }
            if(FD_ISSET(fileno(fp), &rset)){ /* Check sui bit impostati ad 1 */
                if(fgets(sendline, MAX_LINE, fp) == NULL)
                    return
                writen(sockfd, sendline, strlen(sendline));
            }
        }
}
```


Appendice

Esempio di domanda d'esame

Il seguente frammento di codice appartiene ad un echo server. Completare il codice con le eventuali righe mancanti indicando in quale punto del codice originario ciascuna riga dovrà essere aggiunta. Commentare ogni riga aggiunta con il relativo significato.

```
/* Dichiarazioni ed include */

1: int main(int argc, char **argv){
2:      int                listenfd, connfd;
3:      pid_t              childpid;
4:      socklen_t          clilen;
5:      struct sockaddr_in cliaddr, servaddr;

6:      while(1){
7:             clilen = sizeof(cliaddr);
8:             connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
9:             if ((childpid = fork()) == 0){
10:                    close(listenfd);
11:                    //La seguente funzione esegue il protocollo ECHO
12:                    str_echo(connfd);
13:                    exit(0);
14:            }
15:            close(connfd);
16:     }
}
```


Possibile soluzione

Il seguente frammento di codice va posizionato tra le righe 5 e 6 del codice precedente.

```
/*crea un socket per la comunicazione tramite IPv4/TCP ed assegna ad una variabile
  il relativo file descriptor.*/
listenfd = socket(AF_INET, SOCK_STREAM, 0);

/*svuota il contenuto della struttura sockaddr.*/
bzero(&servaddr, sizeof(servaddr));
/*specifica il tipo di rete a cui fanno riferimento gli indirizzi che poi inseriremo
  nella struttura.*/
servaddr.sin_family = AF_INET;
/*specifica che il socket dovrà mettersi in ascolto su tutti gli indirizzi di
  rete disponibili.*/
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/*specifica la porta su cui il socket si metterà in ascolto*/
servaddr.port = htons(7000);

/*effettua il bind del socket sull'indirizzo/porta specificati nella struttura sockaddr*/
bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

/*Mette in modalità passiva il socket, ovvero lo rende in grado di accettare connessioni
  da altri host della rete*/
listen(listenfd, LISTENQ);
```

ATTENZIONE: le domande all'esame potrebbero essere più articolate e presentare una maggiore difficoltà.

TCP server Win: differenze (1)

```
/*
        TCP Server Win single thread
*/

/* dichiarazioni ed include */

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

        /* definizione variabili */

        /* Interpretazione linea di comando */
        ParseCmdLine(argc, argv, &endptr);

        /* creazione socket */
        /* bind e listen sul socket */
        /* loop infinito */

        /* Chiusura del socket in ascolto */
        if ( close(list_s) == SOCKET_ERROR ) {
            fprintf(stderr, "server: errore durante la close.\n");
            exit(EXIT_FAILURE);
        }

        WSACleanup();
}
```


TCP server Win: differenze (2)

```
/* dichiarazioni ed include */

#include <stdio.h>
#include <winsock.h>
#include <stdlib.h>

/* Costanti global  */
#define MAX_LINE           (1000)

/* Prototipi funzioni */
int ParseCmdLine(int argc, char *argv[], char **szPort);
SSIZE_T Readline(int fd, void *vptr, SIZE_T maxlen);
SSIZE_T Writeline(int fc, const void *vptr, SIZE_T maxlen);

void receiveThread(void* sd);
int status;
```

TCP server Win: differenze (3)

```
/* definizione variabili */

SOCKET    list_s;              /* listening socket         */
SOCKET    conn_s;              /* connection socket        */
short int port;                /* port number              */
struct    sockaddr_in servaddr; /* socket address structure */
struct    sockaddr_in their_addr;
char      buffer[MAX_LINE];    /* character buffer         */
char     *endptr;              /* for strtol()             */
int       sin_size;
WSADATA   wsaData;
DWORD     nThread;
```


TCP server Win: differenze (4)

```
/* creazione socket */

port = strtol(endptr, &endptr, 0);
if ( *endptr ) {
        fprintf(stderr, "server: porta non riconosciuta.\n");
        exit(EXIT_FAILURE);
}

printf("Server in ascolto sulla porta %d\n",port);

if (WSAStartup(MAKEWORD(1,1), &wsaData) != 0) {
        printf("errore in WSAStartup()\n");
        exit(EXIT_FAILURE);
}

/*  Create the listening socket  */
if ( (list_s = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET ) {
        fprintf(stderr, "server: errore nella creazione della socket.\n");
        exit(EXIT_FAILURE);
}
```

TCP server Win: differenze (5)

```
/* bind e listen sul socket */

/* Vuotiamo la struttura servaddr e riempiamo i campi necessari */
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(port);

/* Effettua la bind sull'indirizzo e porta ora specificati */
if ( bind(list_s, (struct sockaddr *) &servaddr, sizeof(servaddr)) == SOCKET_ERROR ) {
        fprintf(stderr, "server: errore durante la bind.\n");
        exit(EXIT_FAILURE);
}

/* Mette il socket in modalità di "ascolto" tramite la listen() */
if ( listen(list_s, LISTENQ) == SOCKET_ERROR ) {
        fprintf(stderr, "server: errore durante la listen.\n");
        exit(EXIT_FAILURE);
}
```


TCP server Win: differenze (6)

```
/* loop infinito */

while ( 1 ) {

        /*  Attende una nuova connessione tramite la accept() */
        sin_size = sizeof(struct sockaddr_in);
        if ( (conn_s = accept(list_s, (struct sockaddr *)&their_addr, &sin_size)
                                                      ) == INVALID_SOCKET ){
            fprintf(stderr, "server: errore nella accept.\n");
            exit(EXIT_FAILURE);
        }

        printf("server: connessione da %s\n", inet_ntoa(their_addr.sin_addr));

        /*  Crea un nuovo thread a cui affida la comunicazione con il client */
        CreateThread(NULL,
                     0,
                     (LPTHREAD_START_ROUTINE)receiveThread,
                     (LPVOID)conn_s,
                     NORMAL_PRIORITY_CLASS,
                     &nThread);
}
```