```
#include <time.h>
#include "socket_utils.h"

// Limpa uma string
void ClearStr(char* buffer) {
        int i;
        for(i = 0; i < MAXDATASIZE; i++) {
                buffer[i] = '\0';
        }
}

// Envia o resultado do comando para o cliente
void WriteCmd(int connfd, char *client) {
        int backup, p[2], cont = 0;
        char c, cmd[MAXDATASIZE];
        backup = dup(1);
        Close(0);
        Close(1);
        pipe(p);
        system(client);
        dup2(backup, 1);
        while ((c = getchar()) && c != EOF){
                cmd[cont] = c;
                cont++;
        }
        cmd[cont] = '\0';
        write(connfd, cmd, strlen(cmd));
}

/*
   Servidor
   Aplicacao simples de servidor tcp que recebe varias
   conexoes na porta passada por parametro e executa
   o comando e envia o resultado para o cliente
*/
int main (int argc, char **argv) {
   // Declaracao de variaveis
   int listenfd, connfd;
   pid_t pid;
   struct sockaddr_in servaddr;
   struct sockaddr_in clientaddr;
   char   buf[MAXDATASIZE], error[MAXDATASIZE], client[MAXDATASIZE],
        openClient[MAXDATASIZE], closeClient[MAXDATASIZE];
   time_t ticks;
   FILE *file;

   // Checa a presenca do parametro Porta
   // caso ausente, fecha o programa
   if (argc != 3) {
      strcpy(error,"uso: ");
      strcat(error, argv[0]);
      strcat(error," <Port>");
      strcat(error," <Backlog Size>");
      perror(error);
      exit(1);
   }

   // abre um arquivo texto
   file = fopen("log_tcp_server.txt", "w");

   // Tenta criar um socket local TCP IPv4
   listenfd = Socket(AF_INET, SOCK_STREAM, 0);

   // Limpa o que estiver no ponteiro do socket que representa o servidor
   // Seta o socket do servidor como IPv4 e seta a porta de conexao passada por
parametro.
   // Seta uma mascara para aceitar conexoes de qualquer IP
```

```
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family      = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port        = htons(atoi(argv[1]));

   // Tentar fazer o bind do socket de servidor na porta escolhida
   Bind(listenfd, servaddr);

   // Setar socket como passivo (aceita conexoes)
   // Em caso de falha, fechar o programa
   Listen(listenfd, atoi(argv[2]));

   // Loop infinito
   for ( ; ; ) {
      // Se chegou uma conexao
      // Em caso de falha fechar o programa
      connfd = Accept(listenfd, &clientaddr);

      // Pegar o horario de conexao do cliente
      // limpa o buffer
                ClearStr(openClient);
                ClearStr(closeClient);
      ticks = time(NULL);
      snprintf(openClient, MAXDATASIZE, "%.24s\r", ctime(&ticks));

      // cria um processo filho (copia identica do pai)
      if( (pid = fork()) == 0) {
        // fecha a conexÃ£o com o processo pai
        Close(listenfd);

                    // Converter informacao do IP de binario para string
                    // armazenar o resultado no buffer
                    InetNtop(AF_INET, buf, clientaddr);

                    // Escrever IP, porta e string do cliente na saida padrao
                         printf("OPEN -> Client - IP: %s - Port: %d\n", buf, htons(clienta
ddr.sin_port));

                         // enquanto o comando for diferente de exit
                         do {
                                // limpa o buffer
                                ClearStr(client);

                                // Recebe o comando do cliente
                                Read(connfd, client);

                                // Escrever IP, porta e string do cliente na sai
da padrao
                                printf(" CMD -> Client - IP: %s - Port: %d - String: %s", buf
, htons(clientaddr.sin_port), client);

                    // Envia a mensagem de volta para o cliente com o resultado do c
omando executado
                                WriteCmd(connfd, client);

        } while(strcmp(client, "exit\n"));

        // retarda o fechamento da conexÃ£o cliente
         sleep(5);

        // fecha a conexÃ£o do processo filho
        Close(connfd);

        // Pegar o horario de conexao do cliente
                    ticks = time(NULL);
                    snprintf(closeClient, MAXDATASIZE, "%.24s\r\n", ctime(&ticks));
```

```
                // Escrever IP, porta e string do cliente que se desconectou
                        printf("CLOSE-> Client - IP: %s - Port: %d\n", buf, htons(clienta
ddr.sin_port));

                        // Salva um arquivo texto com o historico dos clientes
                        fprintf(file, "IP %s\nPort: %d\nOpen: %s\nClose: %s\n", buf, hton
s(clientaddr.sin_port), openClient, closeClient);

                        // Limpa o que estiver no ponteiro do socket do client
                        bzero(&clientaddr, sizeof(clientaddr));
                exit(0);
                }

      // Finalizar a conexao
      Close(connfd);
   }

   // fecha o arquivo
        fclose(file);

   return(0);
}
```

```
#include <time.h>
#include "socket_utils.h"

#define LISTENQ 10

// Limpa uma string
void ClearStr(char* buffer) {
        int i;
        for(i = 0; i < MAXDATASIZE; i++) {
                buffer[i] = '\0';
        }
}

// Envia o resultado do comando para o cliente
void WriteCmd(int connfd, char *client) {
        int backup, p[2], cont = 0;
        char c, cmd[MAXDATASIZE];
        backup = dup(1);
        Close(0);
        Close(1);
        pipe(p);
        system(client);
        dup2(backup, 1);
        while ((c = getchar()) && c != EOF){
                cmd[cont] = c;
                cont++;
        }
        cmd[cont] = '\0';
        write(connfd, cmd, strlen(cmd));
}

/*
   Servidor
   Aplicacao simples de servidor tcp que recebe varias
   conexoes na porta passada por parametro e executa
   o comando e envia o resultado para o cliente
*/
int main (int argc, char **argv) {
   // Declaracao de variaveis
   int listenfd, connfd;
   pid_t pid;
   struct sockaddr_in servaddr;
   struct sockaddr_in clientaddr;
   char   buf[MAXDATASIZE], error[MAXDATASIZE], client[MAXDATASIZE],
        openClient[MAXDATASIZE], closeClient[MAXDATASIZE];
   time_t ticks;
   FILE *file;

   // Checa a presenca do parametro Porta
   // caso ausente, fecha o programa
   if (argc != 2) {
      strcpy(error,"uso: ");
      strcat(error,argv[0]);
      strcat(error," <Port>");
      perror(error);
      exit(1);
   }

   // abre um arquivo texto
   file = fopen("log_tcp_server.txt", "w");

   // Tenta criar um socket local TCP IPv4
   listenfd = Socket(AF_INET, SOCK_STREAM, 0);

   // Limpa o que estiver no ponteiro do socket que representa o servidor
   // Seta o socket do servidor como IPv4 e seta a porta de conexao passada por
parametro.
```

```
   // Seta uma mascara para aceitar conexoes de qualquer IP
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family      = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port        = htons(atoi(argv[1]));

   // Tentar fazer o bind do socket de servidor na porta escolhida
   Bind(listenfd, servaddr);

   // Setar socket como passivo (aceita conexoes)
   // Em caso de falha, fechar o programa
   Listen(listenfd, LISTENQ);

   // chama waitpid()
   Signal(SIGCHLD, sig_chld);

   // Loop infinito
   for ( ; ; ) {
      // Se chegou uma conexao
      // Em caso de falha fechar o programa
      if ( (connfd = Accept(listenfd, &clientaddr)) < 0) {
                        if (errno != EINTR) {
                                err_sys("accept error");
                        }
                }

                // Pegar o horario de conexao do cliente
                // limpa o buffer
                ClearStr(openClient);
                ClearStr(closeClient);
                ticks = time(NULL);
                snprintf(openClient, MAXDATASIZE, "%.24s\r", ctime(&ticks));

                // cria um processo filho (copia identica do pai)
                if( (pid = fork()) == 0) {
                        // fecha a conexÃ£o com o processo pai
                        Close(listenfd);

                        // Converter informacao do IP de binario para string
                        // armazenar o resultado no buffer
                        InetNtop(AF_INET, buf, clientaddr);

                        // Escrever IP, porta e string do cliente na saida padra
o
                        printf("OPEN -> Client - IP: %s - Port: %d\n", buf, htons(clienta
ddr.sin_port));

                        // enquanto o comando for diferente de exit
                        do {
                                // limpa o buffer
                                ClearStr(client);

                                // Recebe o comando do cliente
                                Read(connfd, client);

                                // Escrever IP, porta e string do cliente na sai
da padrao
                                printf(" CMD -> Client - IP: %s - Port: %d - String: %s", buf
, htons(clientaddr.sin_port), client);

                                // Envia a mensagem de volta para o cliente com
o resultado do comando executado
                                WriteCmd(connfd, client);

                        } while(strcmp(client, "exit\n"));

                        // fecha a conexÃ£o do processo filho
```

```
                        Close(connfd);

                        // Pegar o horario de conexao do cliente
                        ticks = time(NULL);
                        snprintf(closeClient, MAXDATASIZE, "%.24s\r\n", ctime(&tic
ks));

                        // Escrever IP, porta e string do cliente que se descone
ctou
                        printf("CLOSE-> Client - IP: %s - Port: %d\n", buf, htons(clienta
ddr.sin_port));

                        // Salva um arquivo texto com o historico dos clientes
                        fprintf(file, "IP %s\nPort: %d\nOpen: %s\nClose: %s\n", buf, hton
s(clientaddr.sin_port), openClient, closeClient);

                        // Limpa o que estiver no ponteiro do socket do client
                        bzero(&clientaddr, sizeof(clientaddr));
                        exit(0);
                }

      // Finalizar a conexao
      Close(connfd);
   }

   // fecha o arquivo
        fclose(file);

   return(0);
}
```

```
#include "socket_utils.h"

// Limpa uma string
void ClearStr(char* buffer) {
        int i;
        for(i = 0; i < MAXDATASIZE; i++) {
                buffer[i] = '\0';
        }
}

/*
   Cliente
   Aplicacao simples de cliente tcp que se conecta num
   IP e PORTA passados por parametro, envia um comando ao
   servidor e escreve na saida padrao o retorno do comando
*/
int main(int argc, char **argv) {
   // Declaracao de variaveis
   int sockfd;
   char buf[MAXDATASIZE + 1], error[MAXDATASIZE + 1];
   char server[MAXDATASIZE + 1], server_reply[MAXDATASIZE + 1];
   struct sockaddr_in servaddr;

   // Checa a presenca do parametro de IP e Porta
   // caso ausente, fecha o programa
   if (argc != 3) {
      strcpy(error,"uso: ");
      strcat(error,argv[0]);
      strcat(error," <IPaddress> <Port>");
      perror(error);
      exit(1);
   }

   // Cria um socket
   sockfd = Socket(AF_INET, SOCK_STREAM, 0);

   // Limpa o que estiver no ponteiro do socket que representa o servidor
   // Seta o socket do servidor como IPv4 e seta a porta de conexao para a porta
 da aplicacao.
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port   = htons(atoi(argv[2]));

   // Converte o IP recebido na entrada para a forma binÃ¡ria da struct
   InetPton(AF_INET, argv[1], servaddr);

   // Conecta o socket local com o socket servidor
   Connect(sockfd, servaddr);

        // Escrever IP e porta do servidor na saida padrao
   printf("Server − IP: %s − Port: %d\n", argv[1], atoi(argv[2]));

   // Coletar informacoes sobre o socket com o servidor
   servaddr = Getsockname(sockfd, servaddr);

   // Converter informacao do IP de binario para string
   // armazenar o resultado no buffer
   InetNtop(AF_INET, server, servaddr);

        // Escrever IP e porta do cliente no socket na saida padrao
        printf("Client − IP: %s − Port: %d\n", server, ntohs(servaddr.sin_port));

        printf("Digite os comandos:\n");
        do {
                // limpa o buffer
        ClearStr(buf);
        ClearStr(server_reply);
```

```
                // lÃª uma cadeia de caracteres do teclado
                fgets(buf, MAXDATASIZE, stdin);

                // envia os dados lidos ao servidor
                Write(sockfd , buf);

                // le os dados enviados pelo servidor
                Read(sockfd, server_reply);

                // Imprime a linha de comando devolvida pelo servidor
        printf("%s\n", server_reply);

        } while(strcmp(buf, "exit\n"));

  Close(sockfd);

  exit(0);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <errno.h>

#define MAXDATASIZE 40000

int Accept(int listenfd, struct sockaddr_in *clientaddr);
void Bind(int listenfd, struct sockaddr_in servaddr);
void Close(int connection);
void Connect(int sockfd, struct sockaddr_in sockaddress);
struct sockaddr_in Getsockname(int sockfd, struct sockaddr_in sockaddress);
void InetNtop(int family, char* buffer, struct sockaddr_in sockaddress);
void InetPton(int family, char *ipaddress, struct sockaddr_in sockaddress);
void Listen(int listenfd, int listenq);
void Read(int sockfd, char* buffer);
int Socket(int family, int type, int flags);
void Write(int sockfd, char* buffer);
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
void sig_chld(int);
void err_sys(const char* x);
typedef void Sigfunc(int);
Sigfunc * Signal (int signo, Sigfunc *func);
```

```
#include "socket_utils.h"

// Aceita a conexao do cliente
// Em caso de falha fechar o programa
int Accept(int listenfd, struct sockaddr_in *clientaddr) {
  int connfd, clientsize = sizeof(clientaddr);
  if ((connfd = accept(listenfd, (struct sockaddr *)clientaddr, (socklen_t*)&cli
entsize)) == -1 ) {
    perror("accept");
    exit(1);
  }
  return connfd;
}

// Fazer um bind do socket com os parametros escolhidos
// Fechar o programa em caso de erro
void Bind(int listenfd, struct sockaddr_in servaddr) {
  if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
    perror("bind");
    exit(1);
  }
}

// Fecha a conexao
void Close(int connection) {
  close(connection);
}

// Tenta conectar um socket local a um outro socket, que pode ser remoto
// Fecha o programa em caso de erro
void Connect(int sockfd, struct sockaddr_in sockaddress) {
  if (connect(sockfd, (struct sockaddr *)&sockaddress, sizeof(sockaddress)) < 0)
 {
    perror("connect error");
    exit(1);
  }
}

// Coleta informacoes locais sobre um socket, retorna o socket com as informacoe
s preenchidas
// Fecha o programa em caso de erro
struct sockaddr_in Getsockname(int sockfd, struct sockaddr_in sockaddress) {
  socklen_t socksize = sizeof(sockaddress);
  bzero(&sockaddress, sizeof(sockaddress));
  if (getsockname(sockfd, (struct sockaddr *) &sockaddress, &socksize) < 0) {
    perror("getsockname error");
    exit(1);
  }
  return sockaddress;
}

// Converte o IP da forma binaria da struct sockaddr_in para uma string
// e armazena em buffer
// Fecha o programa em caso de erro
void InetNtop(int family, char* buffer, struct sockaddr_in sockaddress) {
  if (inet_ntop(family, &sockaddress.sin_addr, buffer, sizeof(char)*MAXDATASIZE)
 <= 0) {
    perror("inet_ntop error");
    exit(1);
  }
}

// Converte um IP string para a forma binaria da struct sockaddr_in
// Fecha o programa em caso de erro
void InetPton(int family, char *ipaddress, struct sockaddr_in sockaddress) {
  if (inet_pton(family, ipaddress, &sockaddress.sin_addr) <= 0) {
    perror("inet_pton error");
```

```
    exit(1);
  }
}

// Setar socket como passivo (aceita conexoes)
// Fechar o programa em caso de erro
void Listen(int listenfd, int listenq) {
  if (listen(listenfd, listenq) == -1) {
    perror("listen");
    exit(1);
  }
}

// Recebe dados do cliente e escreve em um buffer
// Se retornar algo > 0, ainda ha dados a serem escritos (ultrapassaram o tamanh
o do buffer)
void Read(int sockfd, char* buffer) {
  int read_size = recv(sockfd, buffer, MAXDATASIZE, 0);
  if (read_size < 0) {
    perror("read error");
    exit(1);
  }
}

// Criar um socket com as opcoes especificadas
// Fecha o programa em caso de erro
int Socket(int family, int type, int flags) {
  int sockfd;
  if ( (sockfd = socket(family, type, flags)) < 0) {
    perror("socket error");
    exit(1);
  }
  return sockfd;
}

// Envia dados do cliente e escreve em um buffer
// Se retornar algo > 0, ainda ha dados a serem escritos (ultrapassaram o tamanh
o do buffer)
void Write(int sockfd, char* buffer) {
  int write_size = write(sockfd, buffer, strlen(buffer));
  if (write_size < 0) {
    perror("write error");
    exit(1);
  }
}

// FunÃ§Ã£o que trata o sig_chld
void sig_chld(int signo) {
        pid_t pid;
        int stat;
        while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
                printf("child %d terminated\n", pid);
        return;
}

// funÃ§Ã£o para tratar erro
void err_sys(const char* x)
{
    perror(x);
    exit(1);
}

// funÃ§Ã£o para tratar o sinal
Sigfunc * Signal (int signo, Sigfunc *func)
{
        struct sigaction act, oact;
        act.sa_handler = func;
```

```
        sigemptyset (&act.sa_mask); /* Outros sinais nÃ£o sÃ£o bloqueados*/
        act.sa_flags = 0;
        if (signo == SIGALRM) { /* Para reiniciar chamadas interrompidas */
#ifdef SA_INTERRUPT
                act.sa_flags |= SA_INTERRUPT; /* SunOS 4.x */
#endif
        } else {
#ifdef SA_RESTART
                act.sa_flags |= SA_RESTART; /* SVR4, 4.4BSD */
#endif
        }
        if (sigaction (signo, &act, &oact) < 0)
                return (SIG_ERR);
        return (oact.sa_handler);
}
```