```
/* Servidor daemon de echo */

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

#define RUNNING_DIR     "/tmp"
#define LOCK_FILE       "EchoDaemon.lock"
#define LOG_FILE        "EchoDaemon.log"

void log_message( char* message )
{
  FILE* logfile = fopen( LOG_FILE, "a");

  if( !logfile ) return;

  fprintf( logfile, "%s\n", message );

  fclose( logfile );
}

void log_message_ip( char* message, char* ip )
{
  FILE* logfile = fopen( LOG_FILE, "a");

  if( !logfile ) return;

  fprintf( logfile, "[%s]: %s\n", ip, message );

  fclose( logfile );
}

void log_message_stats( int lines, int chars )
{
  FILE* logfile = fopen( LOG_FILE, "a");

  if( !logfile ) return;

  fprintf( logfile, "Stats >> Lines: %d  Chars: %d\n", lines, chars );

  fclose( logfile );
}


void signal_handler( int sig )
{
  switch(sig)
    {

    case SIGHUP:
      log_message( "hangup signal catched" );
      break;

    case SIGTERM:
      log_message( "terminate signal catched" );
      exit(0);
      break;

    }
}
```

```
void daemonize( int method )
{

  int i,lfp;
  char str[10];

  /* already a daemon */
  if( getppid() == 1 ) return;

  i = fork();
  if ( i < 0 ) exit( 1 ); // fork error
  if ( i > 0 ) exit( 0 ); // parent exits

  /* child (daemon) continues */

  /* obtain a new process group and be it’s leader */
  setsid( );

  /* Avoid that deamon open a terminal device automaticly */
  if( method == 1 )
    {
      // refork
      i = fork();
      if ( i < 0 ) exit( 1 ); // fork error
      if ( i > 0 ) exit( 0 ); // parent exits
    }
  else if( method == 2 )
    {

      // Method 2
      /*  close all descriptors */
      for ( i = getdtablesize( ); i >= 0; --i )
        close( i );

      /*  handle standart I/O */
      i = open( "/dev/null", O_RDWR );
      dup( i );
      dup( i );
    }

  /* change running directory to one not mounted by the system */
  chdir( RUNNING_DIR );

  /* set newly created file permissions */
  umask( 027 );


  /* Create a lock file to make sute only one instance is running */
  lfp = open( LOCK_FILE, O_RDWR | O_CREAT, 0640 );
  if ( lfp < 0 ) exit( 1 ); // can not open
  if ( lockf( lfp, F_TLOCK, 0 ) < 0 ) exit( 0 ); // can not lock

  /* first instance continues */

  /* record pid to lockfile */
  sprintf( str, "%d\n", getpid( ) );
  write( lfp, str, strlen( str ) );

  /* ignore child */
  signal( SIGCHLD, SIG_IGN );

  /* ignore tty signals */
  signal( SIGTSTP, SIG_IGN );
  signal( SIGTTOU, SIG_IGN );
  signal( SIGTTIN, SIG_IGN );

  /* catch hangup signal */
  signal( SIGHUP, signal_handler );

  /* catch kill signal */
  signal( SIGTERM, signal_handler );
}
```

```
/*
  UNIX Daemon Server Programming Sample Program
  Levent Karakas <levent at mektup dot at> May 2001
  Taken from: http://www.enderunix.org/documents/eng/daemon.php
*/

/* End Deamonize */


/* server_echo.c - Servidor simples */


#define MYPORT 40001    /* the port users will be connecting to */
#define BACKLOG 10     /* how many pending connections queue will hold */
#define BUFF_SIZE 1000

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

  if( argc != 2 )
    {
      printf("Method 1 or Method 2 ?\n");
      exit(1);
    }

  daemonize( atoi(argv[1]) );

  int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
  struct sockaddr_in my_addr;    /* my address information */
  struct sockaddr_in their_addr; /* connector’s address information */
  int sin_size;
  char buffer[BUFF_SIZE];
  int numInLines = 0, totalInChars = 0;
  FILE *rsock, *wsock;

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  my_addr.sin_family = AF_INET;         /* host byte order */
  my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
  my_addr.sin_addr.s_addr = INADDR_ANY; /* automatically fill with my IP */
  bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

  if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
  }
  if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
  }

  log_message("Deamon ready");

  while(1) {  /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, (socklen_t *)&sin_size)) == -1)
{
      perror("accept");
      continue;
    }
    numInLines = 0; totalInChars = 0;

    log_message_ip( "Client Connect", inet_ntoa( their_addr.sin_addr ) );

    if ((rsock = fdopen(new_fd, "r")) == NULL) {
      perror("fdopen");
      exit(1);
    }
    if ((wsock = fdopen(new_fd, "w")) == NULL) {
      perror("fdopen");
      exit(1);
    }

    char SendText[] = "Conectado, envie uma mensagem que eu devolvo.\n";
    if (fputs(SendText, wsock) == EOF) {
      perror("send");
      exit(1);
    }
    fflush(wsock);
    while ( fgets(buffer, BUFF_SIZE, rsock) != NULL ) { // recebe msg do cliente
      fflush(rsock);
      numInLines++;
      totalInChars += strlen(buffer);
      fputs(buffer, wsock); // devolve a mesma coisa
      fflush(wsock);
    }

    log_message_stats( numInLines, totalInChars );
    log_message_ip( "Client Disconnect", inet_ntoa( their_addr.sin_addr ) );

    fclose(rsock);
    fclose(wsock);
    close(new_fd);
  }
  return 0;
}
```