Sistemi di Calcolo 2

May 24, 2023

1 Thread in C-Primitive

1.1 Create

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

Where:

- thread: puntatore a variabile di tipo **pthread_t**
- attr: attributi di creazione (sempre NULL)
- start_routine: funzione da eseguire
- arg: puntatore da passare come argomento alla funzione *start_routine*

Return: 0 se successo, altrimenti la causa dell'errore

1.2 Exit

void **pthread_exit**(void* value_ptr);

- Termina il thread corrente, rendendo disponibile il valore puntato da *value_ptr* ad un eventuale join

1.3 Join

int **pthread_join**(phtread_thread, void** value_ptr);

- Attende esplicitamente la terminazione del thread con ID *thread*
- se *value_ptr != NULL*, vi memorizza il valore restituito dal thread

Return: 0 se successo, altrimenti la causa dell'errore

1.4 Detach

int **pthread_detach**(pthread_t thread);

- Notifica il sistema che non ci sarà join su thread

Return: 0 se successo, altrimenti la causa dell'errore

1.5 EXAMPLE

```
#include<errno.h>
#include<pthread.h>

void* thread_stuff(void *arg){
    return NULL;
}

int ret;
pthread_t thread;
ret = pthread_create(&thread,NULL,thread_stuff,NULL);
if (ret != 0) {
fprintf(stderr, 'ERROR with pthread_create!\n');
exit(EXIT_FAILURE);
}
// codice main indipendente dal thread
ret = pthread_join(thread, NULL);
if (ret != 0) [...]


//MULTIPLE THREADS
#define NUM_THREADS 4

void *hello(void* arg){
    printf('Hello');
}
main(){
    pthread_t tid[NUM_THREADS];
    for(int i=0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],NULL,hello,NULL);

    for(int i=0; i < NUM_THREADS; i++)
        pthread_join(&tid[i],NULL);
}


//PASSAGGIO ARGOMENTI
pthread_t* threads = malloc(N * sizeof(pthread_t));
type_t* objs = malloc(N * sizeof(type_t));
for (i=0; i<N; i++) {
objs[i] = [...] // imposto argomenti thread i-esimo
ret = pthread_create(&threads[i], NULL, foo, &objs[i]);
if (ret != 0) [...]
}
```

2 Fork - Parent&Child

```
// create the N children
    for (i = 0; i < n; i++) {
        pid_t pid = fork(); // PID of child process
        if (pid == -1) {
            printf("Error creating child process #%d: %s\n", i, strerror(errno));
            exit(EXIT_FAILURE); // note that you have to kill manually any other
    process forked so far!

        } else if (pid == 0) {
            // child process, its id is i, exit from cycle
            child_process(i);
            _exit(EXIT_SUCCESS);
        } else {
            // main process, go on creating all required child processes
            continue;
        }
    }
```

```
//Attesa del termine effettivo dei figli
int child_status;
```

```
 for(i = 0; i < n; i++){
    wait(&child_status);
    if(ret == -1) {
        handle_error("wait failed");
        }
        if (WEXITSTATUS(child_status)) {
            fprintf(stderr, "ERROR: child died with code %d\n", WEXITSTATUS(
    child_status));
            exit(EXIT_FAILURE);
        }
}
```

Remember: **fflush(stdout);**
Is used to immediately flush out the contents of an output stream.

3 Concorrenza-Semafori

Semaphores:

- Initialized to a nonnegative integer value
  - 0 nessuna risorsa
  - 1 una risorsa
  - >1 es. array
- semWait decrements the value
- semSignal increments the value

Posix:

- int sem_init(sem_t* sem, int pshared, unsigned value); //initialization
- int sem_wait(sem_t* sem); //wait
- int sem_post(sem_t* sem); //signal
- int sem_destroy(sem_t* sem); //destruction

Parameters:

- sem: the semaphore
- pshared: 0 if sem shared among threads(SEMPRE), 1 if among processes
- value: how many resources we can share

Return: -1 se errore, 0 altrimenti

3.1 EXAMPLE

```
#include<semaphore.h>
...
sem_t sem;
...
sem_init(&sem,phared,value);
...
sem_wait(&sem);
...
sem_post(&sem);
...
sem_destroy(&sem);
```

4 Named Semaphore

Identificato univocamente dal suo nome:
stringa con terminatore che inizia con uno slash (es. */semaforo*)

4.1 Creazione

Due possibili signature:

- sem_t ***sem_open**(const char *name, int oflag);
- sem_t ***sem_open**(const char *name, int oflag, mode_t mode, unsigned int value);

Parametri:

- Nome del semaforo: stringa con terminatore che inizia con uno slash (es. */semaforo*)
- Flag:
  - O_CREAT: il sem viene creato se non esiste già
  - O_CREAT | O_EXCL: se il sem esiste viene lanciato un errore
  - 0 se non ci sono flag da specificare

  Se c'è O_CREAT:
- mode: 0660 o 0666 (permessi)
- value: non negativo, valore a cui è inizializzato

Return:

- Successo: puntatore al named semaphore
- Fallimento: SEM_FAILED

4.2 Chiusura e distruzione

int **sem_close**(sem_t* sem);

- argomento: puntatore sem_t* ottenuto da sem_open

int **sem_unlink**(const char *name);

- argomento: nome del semaforo

4.3 Lettura valore

int **sem_getvalue** (sem_t *sem, int *sval);

- Parametri:
  - Puntatore al semaforo
  - Puntatore ad un int che verrà settato al valore del semaforo
- Return: Successo 0, Errore -1
- Se la coda di attesa non è vuota, *sval sarà 0

5 Shared memory

5.1 shm_open()

int shm_open (const char *name, int oflag, mode_t mode);

- Crea e apre una shared memory, o ne apre una esistente
- Argomenti:
  - *name*:specifica l'oggetto di memoria da creare o aprire, stringa del tipo '/nome'
  - *oflag*:
    - O_CREAT: crea l'oggetto se non esiste
    - O_EXCL: se insieme a O_CREAT e esiset già l'oggetto restituisce errore
    - O_RDONLY: accesso in sola lettura
    - O_WRONLY: accesso in sola scrittura
    - O_RDWR: accesso in lettura/scrittura
  - *mode*: permessi(0666 o 0660)
- Return:
  - Successo: descrittore shared memory
  - Errore: -1

5.2 ftruncate()

int ftruncate(int fd, off_t lenght)

- Dimensiona la memoria condivisa a una dimensione *lenght*
- Argomenti:
  - fd: descrittore ottenuto da shm_open()
  - lenght: dimensione shm
- Return:
  - Successo: 0
  - Errore: -1

5.3 mmap()

void *mmap(void *addr, size_t lenght, int prot, int flags, int fd, off_t offset);

- Mappa la shared memory nella memoria
- Argomenti:
  - addr: 0 (il kernel decide dove posizionare la memoria)
  - lenght: dimensione shm
  - prot: permessi
    - PROT_READ: permesso lettura
    - PROT_WRITE: permesso scrittura
    - PROT_EXEC: permesso di esecuzione
    - PROT_NONE: nessun permesso
  - flags: MAP_SHARED (rende le modifiche visibili)
  - fd: descrittore della shm_open()
  - offset: 0 (permette di mappare la shm in una posizione diversa da quella iniziale)
- Return:
  - Successo: puntatore all'area di memoria dove risiede la shm
  - Errore: MAP_FAILED

5.4 munmap()

int munmap(void *addr, size_t lenght);

- Cancella il mapping tra il processo e la shm
- Argomenti:
  - addr: puntatore alla memoria ottenuto da mmap
  - lenght: dimensione shm
- Return:
  - Successo: 0
  - Errore: -1

5.5 close()

- Chiude il descrittore della shm
- fd: descrittore shm ottenuto da shm_open
- Return:
  - Successo: 0
  - Errore: -1

5.6 shm_unlink()

int shm_unlink(const char *name);

- Rimuove una memoria condivisa
- name: identificatore shm, lo stesso di shm_open
- Return:
  - Successo: 0
  - Errore: -1

5.7 EXAMPLE

```
/* Program to write some data in shared memory */
int main() {
const int SIZE = 4096; /* size of the shared page */
const char * name = 'MY_PAGE';/* name of the shared page */
const char * msg = 'Hello World!';
int shm_fd;
char * ptr;

shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);
ftruncate(shm_fd, SIZE);
ptr = (char *) mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
sprintf(ptr, '%s', msg);
close(shm_fd);
return 0;
}


/* Program to read some data from shared memory */
int main() {
const int SIZE = 4096; /* size of the shared page */
const char * name = 'MY_PAGE';/* name of the shared page */
int shm_fd;
char * ptr;
```

```
24
25 shm_fd = shm_open(name, O_RDONLY, 0666);
26 ptr = (char *) mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
27 printf('%s\n', ptr);
28 shm_unlink(name);
29 return 0;
30 }
```

5.8 Buffer state in shared memory

- Buffer full:
  - in==out;
  - sem_empty.val==0;
  - sem_filled.val== BUFFER_SIZE
  - (in+1)% BUFFER_SIZE == out (when in != out)
- Buffer empty:
  - in==out;
  - sem_empty.val== BUFFER_SIZE;
  - sem_filled.val==0

6 File Descriptor

I file descriptor (FD) sono un'astrazione per accedere a file o altre risorse di input/output come pipe e socket

6.1 Letture con descrittori

ssize_t read(int fd, void *buf, size_t nbyte);

- fd: descrittore risorsa
- buf: puntatore al buffer dove scrivere il messaggio letto
- nbyte: numero massimo di byte da leggere

Ritorna:

- Successo: numero byte letti
- Errore: -1
- Chiamata interrotta prima di riuscire a leggere: -1 && errno==EINTR
- File: 0 se end-of-file
- Socket: 0 se connessione chiusa

```
1 while(<not all bytes have been read>) {
2 // read from fd up to n bytes and store into buf
3 int ret=read(fd, buf, n);
4 // no more bytes to read, quit
5 if (ret==0) break;
6 if (ret==-1 && errno==EINTR) continue;/* interrupted before reading any byte, retry */
7 if (ret==-1) handle_error();  // an error occurred...
8 }
9 /* if interrupted when less than n bytes were read, pay attention to where to write on
      buf on resume! */
10 <do something with read bytes>
11 }
```

6.2 Scritture con descrittori

ssize_t write(int fd, void *buf, size_t nbyte);

- fd: descrittore risorsa
- buf: puntatore al buffer contenete il messaggio da scrivere
- nbyte: numero massimo di byte da scrivere

Ritorna:

- Successo: numero byte scritti
- Errore: -1
- Chiamata interrotta prima di riuscire a scrivere: -1 && errno==EINTR

```
while(<not all bytes have been written>) {
// write to fd up to n bytes from buf
int ret=write(fd, buf, n);
if (ret==-1 && errno==EINTR) continue;/* interrupted before writing any byte, retry */
if (ret==-1) exit(EXIT_FAILURE);  // an error occurred...
}
/* if interrupted when less than n bytes were written, pay attention to where you
    start reading from in buf on resume! */
<do something>
}
```

7 Pipe

◇ Meccanismo di comunicazione inter-processo
◇ Canale di comunicazione unidirezionale

int pipe(int fd[2])

- fd[0] descrittore lettura
- fd[1] descrittore scrittura
- Ritorna 0 se successo, -1 altrimenti

◇ Bisogna chiudere i descrittori non necessari
◇ Si scrive nella pipe come in un file
◇ Si legge dalla pipe come da un file

8 Fifo (named pipe)

◇ Consentono comunicazione tra processi non relazionati(nessun legame padre-figlio via fork)
◇ File speciale per comunicazione unidirezionale

8.1 Creazione

int mkfifo(const char *path, mode_t mode)

- path: nome della fifo
- mode: permessi (es. 0666)
- Return: 0 se successo, -1 altrimenti

8.2 Apertura

int open(const char *path, int oflag)

- path: nome fifo
- oflag: modalità apertura (es. O_RDONLY, O_WRONLY, etc)
- Return: descrittore della fifo se successo, -1 altrimenti

8.3 Chiusura e rimozione

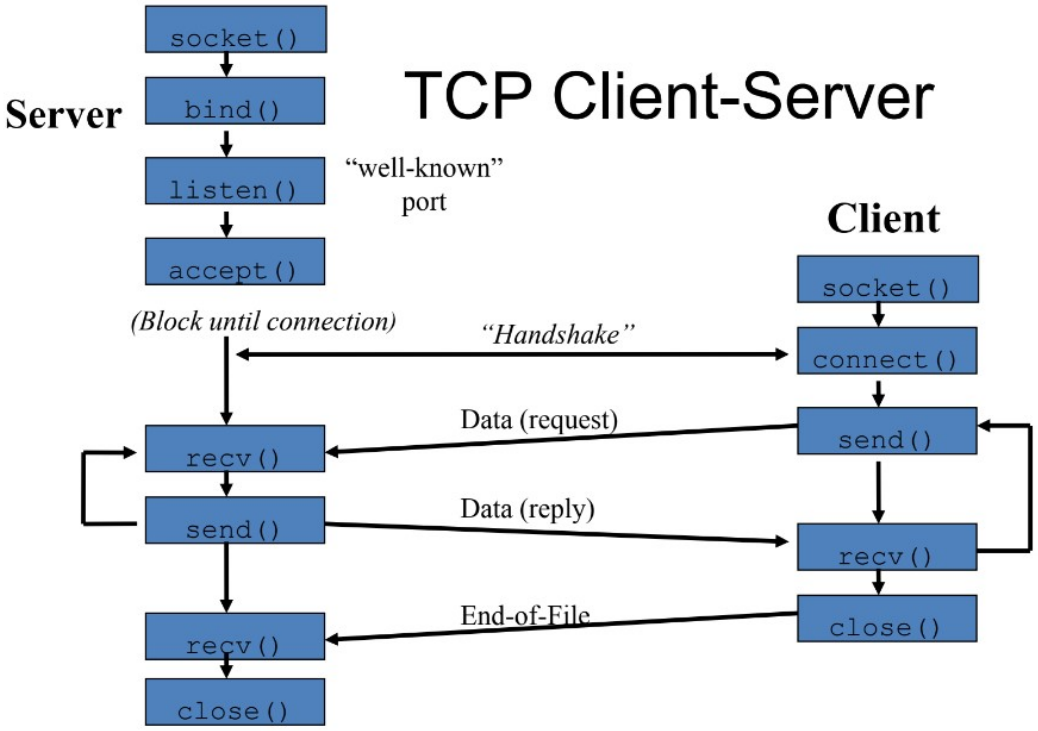
int close(int fd)

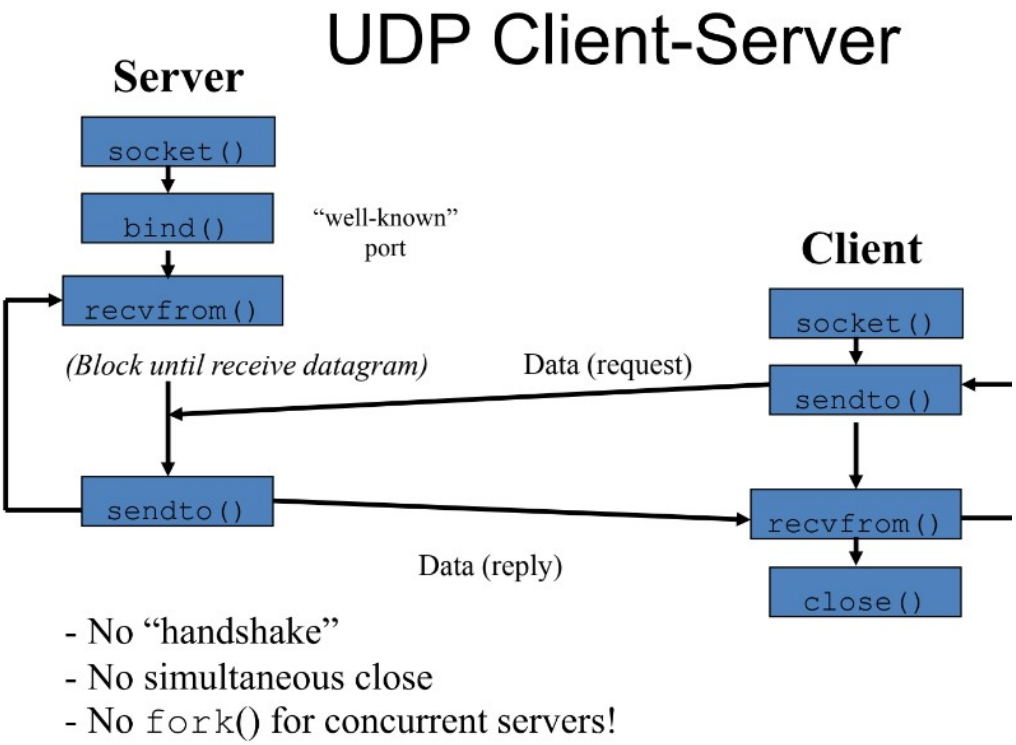
- fd: descrittore restituito dalla open()

int unlink(const char *path)

- path: nome fifo

9 Socket





9.1 Input/Output su socket

9.1.1 Invio messaggi

ssize_t send(int fd,const void *buf, size_t n, int flags);

- fd: descrittore socket
- buf:puntatore al buffer contenente il messaggio da inviare
- n: numero max di byte da scrivere
- flags: fissato a 0 rende send() = write()

Return:

- Successo: il numero di byte letti
- Errore: -1 (con errno==EINTR se prima di trasmettere qualsiasi cosa)

9.1.2 Ricezione messaggi

ssize_t recv(int fd,const void *buf, size_t n, int flags);

- fd: descrittore socket
- buf:puntatore al buffer dove scrivere il messaggio ricevuto
- n: numero max di byte da leggere
- flags: fissato a 0 rende recv() = read()

Return:

- Successo: il numero di byte letti

- Errore: -1 (con errno==EINTR se prima di ricevere qualsiasi cosa)
- 0 in caso di connessione chiusa

9.2 Funzioni di conversione per la porta

- htons()
  uint16_t htons(uint16_t hostshort);
  Converte un ushort da host byte order a network byte order
- ntohs()
  uint16_t ntohs(uint16_t netshort);
  Converte un ushort da network byte order a host byte order

9.3 Strutture dati per le socket

- struct in_addr: indirizzo IP a 32 bit
- struct sockaddr_in: descrizione di una socket, al suo interno:
  - famiglia (sin_family): AF_INET
  - indirizzo ip (sin_addr.s_addr):
    - Server: INADDR_ANY
    - Client: IP del server
  - Numero porta(sin_port): sin_port = htons(port)

9.4 socket()

int socket(int family, int type, int protocol);

- Crea una socket
- Argomenti:
  - family: AF_INET
  - type:
    - SOCK_STREAM (protocollo TCP)
    - SOCK_DGRAM (protocollo UDP)
  - protocol: 0
- Return:
  - Successo: descrittore della socket
  - Errore: -1

9.5 bind()

int bind(int fd, const struct sockaddr *addr, socklen_t len);

- Assegna un indirizzo alla socket
- Argomenti:
  - fd: descrittore socket (restituito da socket())
  - addr: puntatore ad una struttura dati che specifica l'indirizzo (dobbiamo castare *struct sockaddr_in* a *stuct sockaddr*)
  - len: dimensione della struttura dati puntata da addr
- Return:
  - Successo: 0
  - Errore: -1

9.6 listen()

int listen(int socketfd, int backlog);

- Specifica che la socket può essere usata per accettare connessioni tramite *accept()*
- Argomenti:
  - sockfd: descrittore socket(restituito da socket())
  - backlog: lunghezza massima della coda per le connessioni
- Return:
  - Successo: 0
  - Errore: -1

9.7 accept()

int accept(int fd, struct sockaddr *addr, socklen_t *len);

- Accetta una connessione su una socket in ascolto
- Argomenti:
  - fd: descrittore socket(restituito da socket())
  - addr: puntatore ad una struttura dati che verrà riempita con le info della socket del client
  - len: puntatore ad un intero che verrà settato con la dimensione della struttura dati *addr*
- Return:
  - Successo: descrittore per comunicare col client
  - Errore: -1

9.8 connect()

int connect(int fd, const struct sockaddr *addr, socklen_t l);

- Tenta una connessione su una socket in ascolto
- Argomenti:
  - fd: descrittore socket(restituito da socket())
  - addr: puntatore ad una struttura dati che descrive la socket alla quale connettersi
  - len: dimensione della struttura dati puntata da *addr*
- Return:
  - Successo: 0
  - Errore: -1

9.9 close()

int close(int fd);

- Se *fd* è un descrittore di socket, chiude la socket stessa
- Argomenti:
  - fd: descrittore socket(restituito da socket())
- Return:
  - Successo: 0
  - Errore: -1

9.10 UDP Invio messaggi su socket

ssize_t sendto(int fd, const void *buf, size_t n, int flags, const struct sockaddr *dest_addr, socklen_t l);

- Argomenti:
  - fd: descrittore socket
  - buf:puntatore al buffer contenente il messaggio da inviare
  - n: numero max di byte da scrivere
  - flags: fissato a 0 rende sendto() = write()
  - addr: puntatore ad una struttura dati che descrive la socket alla quale connettersi
  - len: dimensione della struttura dati puntata da *addr*
- Return:
  - Successo: byte scritti
  - Errore: -1

9.11 UDP Ricezione messaggi su socket

ssize_t recvfrom(int fd, void *buf, size_t n, int flags,struct sockaddr *src_addr, socklen_t *addrlen);

- Argomenti:
  - fd: descrittore socket
  - buf:puntatore al buffer dove scrivere il messaggio ricevuto
  - n: numero max di byte da leggere
  - flags: fissato a 0 rende recvfrom() = read()
  - se src_addr è una variabile la funzione inserisce le informazione del mittente nella variabile e inserisce in *addrlen* la lunghezza della struttura
    (Utile per rispondere o distinguere tra più possibili mittenti)
  - se src_addr è NULL, non salva il mittente, *addrlen* non viene modificato e può essere NULL
    (Utile quando non ci interessa sapere chi ha inviato il messaggio)
- Return:
  - Successo: byte letti
  - Errore: -1
  - Connessione chiusa: 0

10 SERVER MULTI-PROCESS

```
while (1) {
    int client = accept(server, ....);
    <gestione errori>
    pid_t pid = fork();
    if (pid == -1) {
            <gestione errori>
        } else if (pid == 0) {
            close(server);
            <elaborazione connessione client>
            _exit(0);
        } else {
            close(client);
        }
}
```

11 SERVER MULTI-THREAD

```
while (1) {
    int client = accept(server, ....);
    <gestione errori>


    pthread_t t;
    t_args = ...
    <includere client in t_args>
    pthread_create(&t,NULL,handler,(void*)t_args);
    <gestione errori>
    pthread_detach(t);
}

<gestione socket>
```

12 ES. LETTURA CON DESCRITTORI

```
while (bytes_left > 0) {
 int ret = read(src_fd, buf + read_bytes, bytes_left);

            // no more bytes left to read!
            if (ret == 0) break;

            if (ret == -1){
                if(errno == EINTR) // read() interrupted by a signal
                    continue;
                // handle generic errors
                handle_error("Cannot read from source file");
            }
             bytes_left -= ret;
            read_bytes += ret;
        }
        // no more bytes left to write!
        if (read_bytes == 0) break;//while (1) sopra
```

13 ES. SCRITTURA CON DESCRITTORI

```
int written_bytes = 0; // index for reading from the buffer
        bytes_left = read_bytes; // number of bytes to write

        while (bytes_left > 0) {
            int ret = write(dest_fd, buf + written_bytes, bytes_left);

            if (ret == -1){
                if(errno == EINTR) // write() interrupted by a signal
                    continue;
                // handle generic errors
                handle_error("Cannot write to destination file");
            }

            bytes_left -= ret;
            written_bytes += ret;
        }
    }
```

14 ES. SCRITTURA SU PIPE

```
int written_bytes = 0, ret;

    while (written_bytes < data_len) {
        ret = write(fd, data + written_bytes, data_len - written_bytes);
```

```
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error writing to pipe");
        written_bytes += ret;
    }
    return written_bytes;
}
```

15 ES. LETTURA DA PIPE

```
int read_bytes = 0, ret;
    while (read_bytes < data_len) {
        ret = read(fd, data + read_bytes, data_len - read_bytes);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("error reading from pipe");
        if (ret ==  0) handle_error("unexpected close of the pipe");
        read_bytes += ret;
    }
    return read_bytes;
}
```

16 ES. LETTURA CON FIFO-READ ONE BY ONE

```
int readOneByOne(int fd, char* buf, char separator) {
    int ret;
    int bytes_read = 0;
    do {
        ret = read(fd, buf + bytes_read, 1);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot read from FIFO");
        if (ret ==  0){
            printf("%s\n",buf);
            fflush(stdout);
             handle_error_en(bytes_read,"Process has closed the FIFO unexpectedly!
    Exiting...");
        }
        // we use post-increment on bytes_read so that we first read the
        // byte that has just been written, then we do the increment
    } while(buf[bytes_read++] != separator);
    printf("Read %d bytes\n",bytes_read);
    fflush(stdout);
    return bytes_read;
```

17 ES. SCRITTURA CON FIFO

```
void writeMsg(int fd, char* buf, int size) {
    int ret;
    int bytes_sent = 0;
    while (bytes_sent < size) {
        ret = write(fd, buf + bytes_sent, size - bytes_sent);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot write to FIFO");
        bytes_sent += ret;
    }
    printf("Sent %d bytes\n",bytes_sent);
    fflush(stdout);
}
```

18 ES. SERVER TCP

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <arpa/inet.h>  // htons()
#include <netinet/in.h> // struct sockaddr_in
#include <sys/socket.h>

#include "common.h"

// Method for processing incoming requests. The method takes as argument
// the socket descriptor for the incoming connection.
void* connection_handler(int socket_desc) {
    int ret, recv_bytes, bytes_sent;

    char buf[1024];
    size_t buf_len = sizeof(buf);
    int msg_len;
    memset(buf,0,buf_len);

    char* quit_command = SERVER_COMMAND;
    size_t quit_command_len = strlen(quit_command);

    // send welcome message
    sprintf(buf, "Hi! I'm an echo server. I will send you back whatever"
            " you send me. I will stop if you send me %s", quit_command);
    msg_len = strlen(buf);

    bytes_sent=0;
  while ( bytes_sent < msg_len) {
        ret = send(socket_desc, buf + bytes_sent, msg_len - bytes_sent, 0);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot write to the socket");
        bytes_sent += ret;
    }

    if (DEBUG) fprintf(stderr, "Welcome message <<%s>> has been sent\n",buf);

    // echo loop
    while (1) {
        // read message from client
        recv_bytes = 0;
        do {
            ret = recv(socket_desc, buf + recv_bytes, 1, 0);
            if (ret == -1 && errno == EINTR) continue;
            if (ret == -1) handle_error("Cannot read from the socket");
            if (ret == 0) break;
    } while ( buf[recv_bytes++] != '\n' );

        if (DEBUG) fprintf(stderr, "Received command of %d bytes...\n",recv_bytes);

        // check if either I have just been told to quit...

    if (recv_bytes == quit_command_len && !memcmp(buf, quit_command, quit_command_len)
    ){

            if (DEBUG) fprintf(stderr, "Received QUIT command...\n");
             break;
         }

        // ...or I have to send the message back

        bytes_sent=0;
        while ( bytes_sent < recv_bytes) {
            ret = send(socket_desc, buf + bytes_sent, recv_bytes - bytes_sent, 0);
            if (ret == -1 && errno == EINTR) continue;
            if (ret == -1) handle_error("Cannot write to the socket");
            bytes_sent += ret;
        }
```

```
71          if (DEBUG) fprintf(stderr, "Sent message of %d bytes back...\n", bytes_sent);
72      }
73
74      ret = close(socket_desc);
75      if (ret < 0) handle_error("Cannot close socket for incoming connection");
76
77      if (DEBUG) fprintf(stderr, "Socket closed...\n");
78
79      return NULL;
80  }
81
82  int main(int argc, char* argv[]) {
83      int ret;
84
85      int socket_desc, client_desc;
86
87      // some fields are required to be filled with 0
88      struct sockaddr_in server_addr = {0}, client_addr = {0};
89
90      int sockaddr_len = sizeof(struct sockaddr_in); // we will reuse it for accept()
91
92      socket_desc = socket(AF_INET , SOCK_STREAM , 0);
93      if (socket_desc < 0)
94          handle_error("Could not create socket");
95
96      if (DEBUG) fprintf(stderr, "Socket created...\n");
97
98      /* We enable SO_REUSEADDR to quickly restart our server after a crash:
99       * for more details, read about the TIME_WAIT state in the TCP protocol */
100     int reuseaddr_opt = 1;
101     ret = setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_opt, sizeof(
        reuseaddr_opt));
102     if (ret < 0)
103         handle_error("Cannot set SO_REUSEADDR option");
104
105
106     server_addr.sin_addr.s_addr = INADDR_ANY; // we want to accept connections from
        any interface
107     server_addr.sin_family      = AF_INET;
108     server_addr.sin_port        = htons(SERVER_PORT); // don't forget about network
        byte order!
109
110     ret = bind(socket_desc, (struct sockaddr*) &server_addr, sockaddr_len);
111     if (ret < 0)
112         handle_error("Cannot bind address to socket");
113
114     if (DEBUG) fprintf(stderr, "Binded address to socket...\n");
115
116     ret = listen(socket_desc, MAX_CONN_QUEUE);
117     if (ret < 0)
118         handle_error("Cannot listen on socket");
119
120     if (DEBUG) fprintf(stderr, "Socket is listening...\n");
121
122     // loop to handle incoming connections (sequentially)
123     while (1) {
124     // accept an incoming connection
125
126     client_desc = accept(socket_desc, (struct sockaddr*) &client_addr, (socklen_t*) &
        sockaddr_len);
127         if (client_desc < 0)
128         handle_error("Cannot open socket for incoming connection");
129
130         // invoke the connection_handler() method to process the request
131         if (DEBUG) fprintf(stderr, "Incoming connection accepted...\n");
132
133         connection_handler(client_desc);
134
135         if (DEBUG) fprintf(stderr, "Done!\n");
136     }
137
```

```
    exit(EXIT_SUCCESS); // this will never be executed
}
```

19 ES. CLIENT TCP

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>  // htons() and inet_addr()
#include <netinet/in.h> // struct sockaddr_in
#include <sys/socket.h>

#include "common.h"

int main(int argc, char* argv[]) {
    int ret,bytes_sent,recv_bytes;

    // variables for handling a socket
    int socket_desc;
    struct sockaddr_in server_addr = {0}; // some fields are required to be filled
    with 0

    socket_desc = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_desc < 0)
        handle_error("Could not create socket");

    if (DEBUG) fprintf(stderr, "Socket created...\n");

    //
    server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(SERVER_PORT); // don't forget about network
    byte order!

    // initiate a connection on the socket
    ret = connect(socket_desc, (struct sockaddr*) &server_addr, sizeof(struct
    sockaddr_in));
    if (ret < 0)
        handle_error("Could not create connection");

    if (DEBUG) fprintf(stderr, "Connection established!\n");

    char buf[1024];
    size_t buf_len = sizeof(buf);
    int msg_len;
    memset(buf,0,buf_len);

    recv_bytes = 0;
    do {
        ret = recv(socket_desc, buf + recv_bytes, buf_len - recv_bytes, 0);
        if (ret == -1 && errno == EINTR) continue;
        if (ret == -1) handle_error("Cannot read from the socket");
        if (ret == 0) break;
  recv_bytes += ret;

    } while ( buf[recv_bytes-1] != '\n' );
    printf("%s", buf);

    if (DEBUG) fprintf(stderr, "Received message of %d bytes...\n",recv_bytes);


    // main loop
    while (1) {
        char* quit_command = SERVER_COMMAND;
        size_t quit_command_len = strlen(quit_command);

        printf("Insert your message: ");
```

```
62
63          /* Read a line from stdin
64           *
65           * fgets() reads up to sizeof(buf)-1 bytes and on success
66           * returns the first argument passed to it. */
67          if (fgets(buf, sizeof(buf), stdin) != (char*)buf) {
68              fprintf(stderr, "Error while reading from stdin, exiting...\n");
69              exit(EXIT_FAILURE);
70          }
71
72          msg_len = strlen(buf);
73  //        buf[--msg_len] = '\0'; // remove '\n' from the end of the message
74
75
76          bytes_sent=0;
77          while ( bytes_sent < msg_len) {
78              ret = send(socket_desc, buf + bytes_sent, msg_len - bytes_sent, 0);
79              if (ret == -1 && errno == EINTR) continue;
80              if (ret == -1) handle_error("Cannot write to the socket");
81              bytes_sent += ret;
82          }
83
84          if (DEBUG) fprintf(stderr, "Sent message of %d bytes...\n", bytes_sent);
85
86
87      if (msg_len == quit_command_len && !memcmp(buf, quit_command, quit_command_len)){
88
89              if (DEBUG) fprintf(stderr, "Sent QUIT command ...\n");
90               break;
91          }
92
93    recv_bytes = 0;
94        do {
95              ret = recv(socket_desc, buf + recv_bytes, buf_len - recv_bytes, 0);
96              if (ret == -1 && errno == EINTR) continue;
97              if (ret == -1) handle_error("Cannot read from the socket");
98            if (ret == 0) break;
99            recv_bytes += ret;
100
101       } while ( buf[recv_bytes-1] != '\n' );
102
103         if (DEBUG) fprintf(stderr, "Received answer of %d bytes...\n",recv_bytes);
104
105         printf("Server response: %s\n", buf); // no need to insert '\0'
106     }
107
108     ret = close(socket_desc);
109     if (ret < 0) handle_error("Cannot close the socket");
110
111     if (DEBUG) fprintf(stderr, "Socket closed...\n");
112
113     if (DEBUG) fprintf(stderr, "Exiting...\n");
114
115     exit(EXIT_SUCCESS);
116 }
```

20 ES. SERVER UDP

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <time.h>
6  #include <unistd.h>
7  #include <arpa/inet.h>  // htons()
8  #include <netinet/in.h> // struct sockaddr_in
9  #include <sys/socket.h>
10
11 #include "common.h"
```

```
12
13 // Method for processing incoming requests. The method takes as argument
14 // the socket descriptor for the incoming connection.
15 void* connection_handler(int socket_desc) {
16     int ret, recv_bytes, bytes_sent;
17
18     char buf[1024];
19     size_t buf_len = sizeof(buf);
20     int msg_len;
21     memset(buf,0,buf_len);
22
23     char* quit_command = SERVER_COMMAND;
24     size_t quit_command_len = strlen(quit_command);
25
26     struct sockaddr_in client_addr;
27
28     int sockaddr_len = sizeof(client_addr); // we will reuse it for accept()
29
30     // echo loop
31     while (1) {
32         // read message from client
33     // - in UDP we don't deal with partially sent messages
34
35         recv_bytes = 0;
36         do {
37             recv_bytes = recvfrom(socket_desc, buf, buf_len, 0, &client_addr, &
    sockaddr_len);
38             if (recv_bytes == -1 && errno == EINTR) continue;
39             if (recv_bytes == -1) handle_error("Cannot read from the socket");
40             if (recv_bytes == 0) break;
41     } while ( recv_bytes <= 0 );
42
43
44         if (DEBUG) fprintf(stderr, "Received command of %d bytes...\n",recv_bytes);
45
46         // check if either I have just been told to quit...
47
48     if (recv_bytes == quit_command_len && !memcmp(buf, quit_command, quit_command_len)
    ){
49
50             if (DEBUG) fprintf(stderr, "Received QUIT command...\n");
51                 continue;
52          }
53
54         // ...or I have to send the message back
55
56         bytes_sent=0;
57         while ( bytes_sent < recv_bytes) {
58             ret = sendto(socket_desc, buf, recv_bytes, 0, &client_addr, sockaddr_len);
59             if (ret == -1 && errno == EINTR) continue;
60             if (ret == -1) handle_error("Cannot write to the socket");
61             bytes_sent = ret;
62         }
63     }
64
65     ret = close(socket_desc);
66     if (ret < 0) handle_error("Cannot close socket for incoming connection");
67
68     if (DEBUG) fprintf(stderr, "Socket closed...\n");
69
70     return NULL;
71 }
72
73 int main(int argc, char* argv[]) {
74     int ret;
75
76     int socket_desc, client_desc;
77
78     // some fields are required to be filled with 0
79     struct sockaddr_in server_addr = {0}, client_addr = {0};
80
```

```
    int sockaddr_len = sizeof(struct sockaddr_in); // we will reuse it for accept()

    socket_desc = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_desc < 0)
        handle_error("Could not create socket");

    if (DEBUG) fprintf(stderr, "Socket created...\n");

    /* We enable SO_REUSEADDR to quickly restart our server after a crash:
     * for more details, read about the TIME_WAIT state in the TCP protocol */
    int reuseaddr_opt = 1;
    ret = setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_opt, sizeof(
    reuseaddr_opt));
    if (ret < 0)
        handle_error("Cannot set SO_REUSEADDR option");
    server_addr.sin_addr.s_addr = INADDR_ANY; // we want to accept connections from
    any interface
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(SERVER_PORT); // don't forget about network
    byte order!

    ret = bind(socket_desc, (struct sockaddr*) &server_addr, sockaddr_len);
    if (ret < 0)
        handle_error("Cannot bind address to socket");

    if (DEBUG) fprintf(stderr, "Binded address to socket...\n");

    // loop to handle incoming connections (sequentially)
    while (1) {
    // accept an incoming connection
        if (DEBUG) fprintf(stderr, "opening connection handler...\n");

        connection_handler(socket_desc);

    }

    exit(EXIT_SUCCESS); // this will never be executed
}
```

21 ES. CLIENT UDP

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>  // htons() and inet_addr()
#include <netinet/in.h> // struct sockaddr_in
#include <sys/socket.h>

#include "common.h"

int main(int argc, char* argv[]) {
    int ret,bytes_sent,recv_bytes;

    // variables for handling a socket
    int socket_desc;
    struct sockaddr_in server_addr = {0}; // some fields are required to be filled
    with 0

    socket_desc = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_desc < 0)
        handle_error("Could not create socket");

    if (DEBUG) fprintf(stderr, "Socket created...\n");
    server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(SERVER_PORT); // don't forget about network
    byte order!
```

```
27
28
29      char buf[1024];
30      size_t buf_len = sizeof(buf);
31      int msg_len;
32      memset(buf,0,buf_len);
33
34      // main loop
35      while (1) {
36          char* quit_command = SERVER_COMMAND;
37          size_t quit_command_len = strlen(quit_command);
38
39          printf("Insert your message: ");
40
41          /* Read a line from stdin
42           *
43           * fgets() reads up to sizeof(buf)-1 bytes and on success
44           * returns the first argument passed to it. */
45          if (fgets(buf, sizeof(buf), stdin) != (char*)buf) {
46              fprintf(stderr, "Error while reading from stdin, exiting...\n");
47              exit(EXIT_FAILURE);
48          }
49
50          msg_len = strlen(buf);
51  //          buf[--msg_len] = '\0'; // remove '\n' from the end of the message
52
53          bytes_sent=0;
54          while ( bytes_sent < msg_len) {
55              ret = sendto(socket_desc, buf, msg_len, 0, (struct sockaddr*) &server_addr
        , sizeof(struct sockaddr_in));
56              if (ret == -1 && errno == EINTR) continue;
57              if (ret == -1) handle_error("Cannot write to the socket");
58              bytes_sent = ret;
59          }
60
61          if (DEBUG) fprintf(stderr, "Sent message of %d bytes...\n", bytes_sent);
62
63      if (msg_len == quit_command_len && !memcmp(buf, quit_command, quit_command_len)){
64
65              if (DEBUG) fprintf(stderr, "Sent QUIT command ...\n");
66               break;
67          }
68
69    recv_bytes = 0;
70        do {
71              ret = recvfrom(socket_desc, buf, buf_len, 0, NULL, NULL);
72              if (ret == -1 && errno == EINTR) continue;
73              if (ret == -1) handle_error("Cannot read from the socket");
74            if (ret == 0) break;
75            recv_bytes = ret;
76
77        } while ( recv_bytes<=0 );
78
79          if (DEBUG) fprintf(stderr, "Received answer of %d bytes...\n",recv_bytes);
80
81          printf("Server response: %s\n", buf); // no need to insert '\0'
82      }
83
84      ret = close(socket_desc);
85      if (ret < 0) handle_error("Cannot close the socket");
86
87      if (DEBUG) fprintf(stderr, "Socket closed...\n");
88
89      if (DEBUG) fprintf(stderr, "Exiting...\n");
90
91      exit(EXIT_SUCCESS);
92  }
```