PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA
SISTEMAS OPERATIVOS
LABORATORIO

SOCKETS

1.0 Hilos en Linux

También conocidos como proceso ligeros. Nosotros emplearemos ***POSIX Threads*** definidas por el estándar *POSIX.1.c*, también conocidas como ***pthreads***. En Linux se encuentran implementadas como parte de las librerías del compilador.

Los hilos son ampliamente usados debido a la característica de compartir el segmento de data del proceso. Recuerde que en el segmento de data de un proceso se encuentran las variables estáticas y las variables globales inicializadas, y no inicializadas (*bss*). Usted puede revisar el siguiente *enlace* para una revisión del esquema de memoria de un proceso en ejecución.

1.1 Llamadas básicas a funciones de *pthreads*.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Todos los hilos *pthreads* tienen ciertas propiedades. Cada uno tiene un identificador, un conjunto de registros (incluyendo el contador de programa), y un conjunto de atributos, los que son almacenados en una estructura. Los atributos incluyen el tamaño de la pila, parámetros de planificación, y otros datos necesarios para el uso de los hilos.

Un hilo es creado usando la llamada a *pthread_create*. El identificador de un hilo recién creado es retornado como el valor de una función. La llamada es intencionalmente muy parecida a la llamada al sistema fork (excepto por los parámetros), con el identificador de hilo jugando el rol de PID, principalmente para identificar hilos que se hacen referencia otras llamadas.

Cuando un hilo ha finalizado el trabajo que le ha sido asignado, éste puede terminar invocando *pthread_exit*. Esta llamada detiene el hilo y libera su pila.

A menudo un hilo, antes de continuar, necesita esperar que otro hilo termine su trabajo y salga. El hilo que desea esperar invoca *pthread_join* para esperar por un hilo especifico, antes de terminar. El identificador del hilo a esperar es dado como parámetro.

Algunas veces sucede que un hilo que se ha ejecutado el tiempo suficiente, quiere darle la oportunidad a otro hilo de ejecutarse, entonces lo puede hacer invocando *pthread_yield*.

Las siguientes dos llamadas tratan con los atributos de los hilos. *Pthread_attr_init* crea la estructura de atributo asociada con el hilo y lo inicializa con los valores por defecto. Estos valores (tales como prioridad) pueden ser modificados manipulando los campos de la estructura de atributo.

Finalmente, *pthread_attr_destroy* la estructura de atributos de un hilo, liberando su memoria. Esto no afecta a los hilos que la usan, ellos continúan existiendo.

A continuación un programa que muestra el uso de algunas de las llamadas a funciones *pthreads*.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

/* Tomado del libro Modern Operating System by Andrew Tanenbaum and
                                            Herbert Bos             */

void *print_hello_world(void *tid)
{    /* Esta funcion imprime el orden en que han sido creado cada hilo */
     long id;

     id = (long) tid;
     fprintf(stderr,"Hello World. Greetings from thread %ld\n",id);
     pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* El hilo principal crea 10 hilos y termina */
     pthread_t threads[NUMBER_OF_THREADS];
     int status;
     long i;

     for(i=0; i < NUMBER_OF_THREADS; i++) {
         printf("Main here. Creating thread %d\n", i);
         status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
         if (status != 0) {
            printf("Oops. pthread create returned error code %d\n", status);
            exit(-1);
         }
     }
     exit(0);
}
```

En una terminal, en el mismo directorio donde se encuentra el código fuente, compile el programa tal como se muestra abajo, para obtener la salida correspondiente.

```
alejandro@abdebian:2020-1$ gcc -o hilos1 hilos1.c -lpthread
alejandro@abdebian:2020-1$ ./hilos1
Main here. Creating thread 0
Main here. Creating thread 1
Main here. Creating thread 2
Hello World. Greetings from thread 0
Hello World. Greetings from thread 1
Main here. Creating thread 3
Hello World. Greetings from thread 2
Main here. Creating thread 4
Main here. Creating thread 5
Hello World. Greetings from thread 3
Main here. Creating thread 6
Hello World. Greetings from thread 5
Hello World. Greetings from thread 4
Main here. Creating thread 7
Main here. Creating thread 8
Hello World. Greetings from thread 7
Main here. Creating thread 9
Hello World. Greetings from thread 8
Hello World. Greetings from thread 9
alejandro@abdebian:2020-1$
```

2.0 *Sockets*

Los *sockets* son un método de IPC (*Inter Process Communications*) que permite que la data sea intercambiada entre aplicaciones, ya sea en la misma computadora o en diferentes computadoras conectadas por red.

En un típico escenario cliente-servidor, las aplicaciones se comunican usando *sockets* como sigue:

- Cada aplicación crea un *socket*. Un *socket* es un "aparato" que permiten la comunicación. Y ambos requieren uno.
- El servidor liga o enlaza su *socket* a una dirección conocida de antemano de modo que los clientes puedan ubicarlo.

Un *socket* se crea usando la llamada al sistema *socket()*, la que retorna un descriptor de archivo usado para hacer referencia al *socket* en subsecuentes llamadas al sistema:

```
fd = socket(dominio, tipo, protocolo)
```

A continuación se describe el dominio y el tipo del socket. Para el resto de la guía protocolo siempre será 0.

2.1 Dominios de comunicación

Los *sockets* existen en un dominio de comunicación, los cuales determinan:

- el método de identificar un *socket* (esto es el formato de la dirección de un *socket*); y
- el rango de comunicación (esto es si se lleva a cabo entre aplicaciones en la misma computadora o entre aplicaciones de diferentes computadoras conectadas a través de una red).

Sistemas operativos modernos soportan al menos los siguientes dominios:

a) El dominio *UNIX* (`AF_UNIX`) que permite comunicación en la misma computadora. También puede ser encontrado con el sinónimo de `AF_LOCAL`.

b) El dominio *IPv4* (`AF_INET`) que permite comunicación entre aplicaciones ejecutándose en diferentes computadoras conectadas en red, vía el Protocolo de Internet versión 4.

c) El dominio *IPv6* (`AF_INET6`) que permite comunicación entre aplicaciones ejecutándose en diferentes computadoras conectadas en red, vía el Protocolo de Internet versión 6.

La siguiente tabla resume los dominios:

Domain	Communication performed	Communication between applications	Address format	Address structure
`AF_UNIX`	within kernel	on same host	pathname	*sockaddr_un*
`AF_INET`	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	*sockaddr_in*
`AF_INET6`	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	*sockaddr_in6*

2.2 Tipos de *sockets*

Toda implementación de *sockets* provee al menos dos tipos de *sockets*:

stream sockets (`SOCK_STREAM`)
datagram sockets (`SOCK_DGRAM`)

Estos tipos de *sockets* son soportados tanto por el dominio UNIX como por el dominio de Internet.

2.2.1 *Stream sockets* (`SOCK_STREAM`) provee un canal de comunicación **fiable** (se garantiza que la data transmitida llegará intacta a la aplicación receptora, exactamente como fue transmitida por el que la envía (asumiendo que ni la red ni el receptor fallan), **bidireccional** (la data puede ser transmitida en cualquiera de las dos direcciones entre los *sockets*) y por **flujo de bytes** (al igual como en los *pipes*)

Stream sockets opera entre pares de aplicaciones conectadas. Por esta razón, *stream sockets* son descritas como *orientadas a la conexión*.

El funcionamiento de *stream sockets* se puede explica empleando la analogía del sistema telefónico:

1. La llamada al sistema *socket()*, la misma que crea un *socket*, es el equivalente a instalar un teléfono. Con el fin de comunicar dos aplicaciones, cada una de ellas debe crear un *socket*.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
                                    Returns file descriptor on success, or −1 on error
```

El argumento *domain* especifica el dominio de comunicación para el *socket*. El argumento *type* especifica el tipo de *socket*, ya sea `SOCK_STREAM` o `SOCK_DGRAM`. El argumento *protocol*, siempre se especificará como 0.

2. La comunicación vía *stream socket* es análoga a una llamada telefónica. Antes de que la comunicación se lleve a cabo, una aplicación debe conectar su *socket* al *socket* de la aplicación con la que desea contactar. Dos *sockets* se conectan como sigue:

> **a)** Una aplicación invoca *bind()* con el fin de ligar el *socket* a una dirección bien establecida, y luego invoca *listen()* para notificar al *kernel* su disposición de aceptar la conexión entrante. Este paso es análogo a tener un número de teléfono conocido y asegurar que nuestro teléfono está funcionando de forma que la gente nos puede llamar.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                    Returns 0 on success, or −1 on error
```

El argumento *sockfd* es un descriptor de archivo obtenido de llamar a *socket()* previamente. El argumento *addr* es un puntero a una estructura especificando la dirección para el cual este *socket* se puede ligar. El tipo de estructura pasada en este argumento depende del dominio del *socket*. El argumento *addrlen* especifica el tamaño de la estructura de dirección.

Normalmente, nosotros ligamos un *socket* de un servidor a una dirección bien conocida, esto es, una dirección fija que es conocida con anticipación por la aplicación cliente que desea comunicarse con el servidor. También existe la posibilidad de dejar que el *kernel* elija una dirección y después el servidor la debe obtener usando *getsockname()*. Pero este enfoque no lo usaremos.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
                                        Returns 0 on success, or −1 on error
```

Para entender el propósito del argumento *backlog*, primero observemos que el cliente puede invocar *connect()* antes de que el servidor invoque *accept()*. Esto puede suceder, por ejemplo, porque el servidor está ocupado atendiendo otros clientes. Esto resulta en conexiones pendientes. El *kernel* debe registrar alguna información sobre las conexiones pendientes de modo que una subsecuente *accept()* pueda ser procesada. El argumento *backlog* permite limitar el número de las conexiones pendientes.

b) La otra aplicación establece la conexión invocando *connect()*, especificando la dirección del *socket* para el cual la conexión se hizo. Esto es análogo a marcar un número de teléfono.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                        Returns 0 on success, or −1 on error
```

Todos los argumentos son especificados de la misma forma como los correspondientes a los argumentos de *bind()*.

c) La aplicación que invocó listen() entonces acepta la conexión usando *accept()*. Esto es análogo a tomar el teléfono cuando timbra. Si el *accept()* se lleva a cabo antes de que la aplicación cliente invoque *connect()*, entonces *accept()* se bloquea (“esperando por el teléfono”).

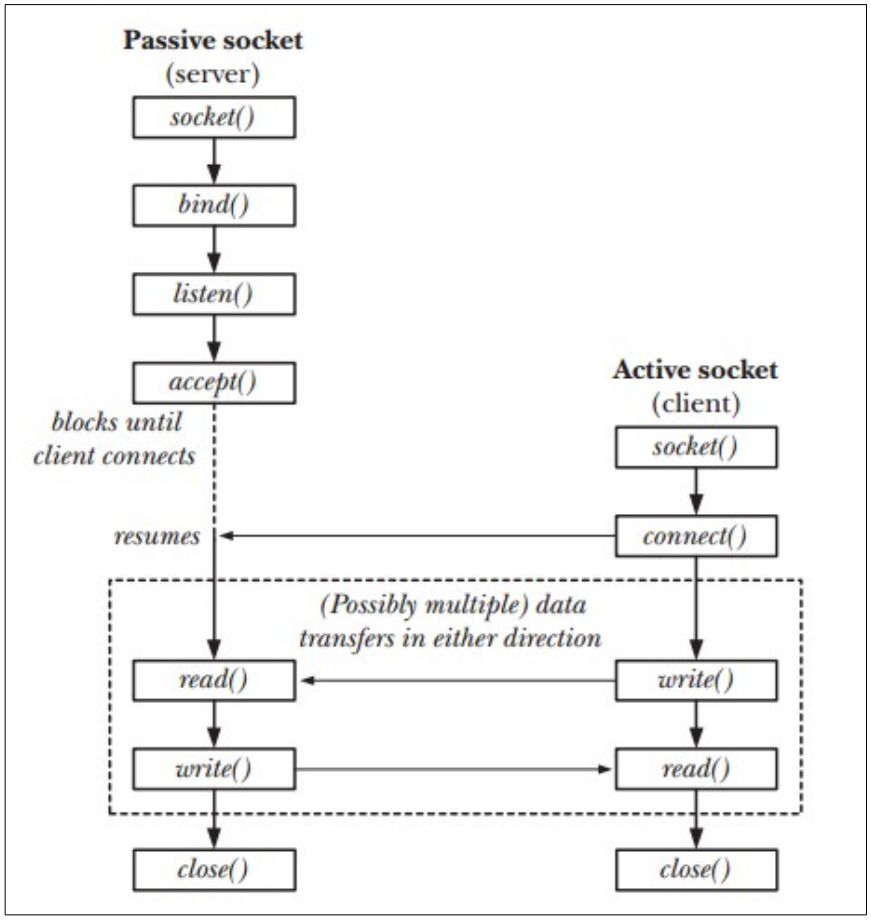
```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
                                        Returns file descriptor on success, or −1 on error
```

El punto clave para entender *accept()* es que éste crea un nuevo *socket*, y es este nuevo *socket* el que es conectado al *socket* de su pareja en el otro extremo, quien invocó a

connect(). Al *socket* que se encuentra en el otro extremo se le denomina ***peer socket***. La función retorna un descriptor para el nuevo *socket* conectado. El *socket* que estuvo escuchando (*sockfd*) permanece abierto, y puede ser usado para seguir aceptando más conexiones.
En *addr* retorna la dirección del peer *socket* en la estructura apuntada por *addr*. Y en *addrlen* retorna el número de bytes copiados a dicha estructura. Si no se está interesado en la dirección del peer socket, entonces *addr* y *addrlen* deberían ser especificados como `NULL` y `0`, respectivamente.

3. Una vez que la conexión se ha establecido, la data puede ser transmitida en ambas direcciones entre las aplicaciones (análoga a la conversación telefónica de doble sentido), hasta que una de ellas cierre la conexión usando *close()*. La comunicación se realiza usando las convencionales llamadas al sistema *read()* y *write()* o a través de un número de llamadas al sistema específica de *sockets*, tales como *send()* y *recv()* que proveen funcionalidad adicional.

A continuación el siguiente esquema ilustra el uso de llamadas al sistema usando *stream sockets*.



2.2.2 Un ejemplo de *stream sockets* en el dominio UNIX

Servidor, el programa servidor listado abajo, lleva a cabo los siguientes pasos:

- Crea un *socket*

- Elimina cualquier archivo existente con el mismo *pathname* que el que queremos ligar al *socket*.

- Construye una estructura de dirección para el socket del servidor, liga el *socket* a esa dirección y marca el *socket* como *socket* de escucha.- Ejecuta un lazo infinito para manejar los pedidos de clientes. En cada iteración se lleva a cabo los siguientes pasos:

° Acepta una conexión, obteniendo un nuevo *socket*, *cfd*, para la conexión.

° Lee toda la data del socket conectado y lo escribe a la salida estándar.

° Cierra el *socket cdf* conectado

El servidor debe ser terminado manualmente (por ejemplo enviándole una señal).

```c
/*************************************************************************\
*                  Copyright (C) Michael Kerrisk, 2010.                   *
*                                                                         *
* This program is free software. You may use, modify, and redistribute *
* it under the terms of the GNU Affero General Public License as       *
* published by the Free Software Foundation, either version 3 or (at   *
* your option) any later version. This program is distributed without  *
* any warranty. See the file COPYING for details.                       *
\*************************************************************************/

/* us_xfr_sv.c

   An example UNIX stream socket server. Accepts incoming connections
   and copies data sent from clients to stdout.

   See also us_xfr_cl.c.
*/

#include <sys/un.h>
#include <sys/socket.h>
#include <sys/types.h>  /* Type definitions used by many programs */
#include <stdio.h>      /* Standard I/O functions */
#include <stdlib.h>     /* Prototypes of commonly used library functions,
                           plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>     /* Prototypes for many system calls */
#include <errno.h>      /* Declares errno and defines error constants */
#include <string.h>     /* Commonly used string-handling functions */


#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100

#define BACKLOG 5

int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        perror("socket");

- 1 -
```

```
48      /* Construct server socket address, bind socket to it,
49         and make this a listening socket */
50
51      if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
52          perror("remove");
53
54      memset(&addr, 0, sizeof(struct sockaddr_un));
55      addr.sun_family = AF_UNIX;
56      strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);
57
58      if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
59          perror("bind");
60
61      if (listen(sfd, BACKLOG) == -1)
62          perror("listen");
63
64      for (;;) {          /* Handle client connections iteratively */
65
66          /* Accept a connection. The connection is returned on a new
67             socket, 'cfd'; the listening socket ('sfd') remains open
68             and can be used to accept further connections. */
69
70          cfd = accept(sfd, NULL, NULL);
71          if (cfd == -1)
72              perror("accept");
73
74          /* Transfer data from connected socket to stdout until EOF */
75
76          while ((numRead = read(cfd, buf, BUF_SIZE)) > 0)
77              if (write(STDOUT_FILENO, buf, numRead) != numRead)
78                  perror("partial/failed write");
79
80          if (numRead == -1)
81              perror("read");
82
83          if (close(cfd) == -1)
84              perror("close");
85      }
86  }
87
```


Cliente, el programa cliente listado abajo, lleva a cabo los siguientes pasos:

- Crea un *socket*.

- Construye la estructura de dirección para el *socket* del servidor y conecta al *socket* en esa dirección.

- Ejecuta un lazo que copia su entrada estándar al *socket* conectado. Al encontrar el final de archivo (*end-of-file*) en su entrada estándar, el cliente termina, con el resultado que su *socket* está cerrado y el servidor ve el final de archivo cuando lee el *socket* al otro extremo de la conexión.

```c
/*************************************************************************\
*                  Copyright (C) Michael Kerrisk, 2010.                   *
*                                                                         *
* This program is free software. You may use, modify, and redistribute *
* it under the terms of the GNU Affero General Public License as       *
* published by the Free Software Foundation, either version 3 or (at   *
* your option) any later version. This program is distributed without  *
* any warranty. See the file COPYING for details.                       *
\*************************************************************************/

/* us_xfr_cl.c

   An example UNIX domain stream socket client. This client transmits contents
   of stdin to a server socket.

   See also us_xfr_sv.c.
*/

#include <sys/un.h>
#include <sys/socket.h>
#include <sys/types.h>  /* Type definitions used by many programs */
#include <stdio.h>      /* Standard I/O functions */
#include <stdlib.h>     /* Prototypes of commonly used library functions,
                           plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>     /* Prototypes for many system calls */
#include <errno.h>      /* Declares errno and defines error constants */
#include <string.h>     /* Commonly used string-handling functions */


#define SV_SOCK_PATH "/tmp/us_xfr"

#define BUF_SIZE 100


int
main(int argc, char *argv[])
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);      /* Create client socket */
    if (sfd == -1)
        perror("socket");

    /* Construct server address, and make the connection */

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SV_SOCK_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr,
                sizeof(struct sockaddr_un)) == -1)
        perror("connect");

    /* Copy stdin to socket */

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
        if (write(sfd, buf, numRead) != numRead)
            perror("partial/failed write");

    if (numRead == -1)
        perror("read");

    exit(EXIT_SUCCESS);         /* Closes our socket; server sees EOF */
}
```

Compilando los archivos y luego ejecutando el servidor y el cliente en una terminal.

```
alejandro@abdebian:2020-1$ gcc -o us_xfr_sv us_xfr_sv.c
alejandro@abdebian:2020-1$ gcc -o us_xfr_cl us_xfr_cl.c
alejandro@abdebian:2020-1$ ./us_xfr_sv > b &
[1] 6272
alejandro@abdebian:2020-1$ ls -lF /tmp/us_xfr
srwxr-xr-x 1 alejandro alejandro 0 may  2 00:14 /tmp/us_xfr=
alejandro@abdebian:2020-1$ cat *.c > a
alejandro@abdebian:2020-1$ ./us_xfr_cl < a
alejandro@abdebian:2020-1$ kill %1
alejandro@abdebian:2020-1$
[1]+  Terminado               ./us_xfr_sv > b
alejandro@abdebian:2020-1$ diff a b
alejandro@abdebian:2020-1$
```

2.2.3 Un ejemplo de *stream sockets* en el dominio de internet

A continuación se muestra el código de un servidor que acepta comunicación de cualquiera de las interfaces que la computadora pueda tener, esto es posible indicando `INADDR_ANY` en la estructura de dirección. En esta misma estructura se indica el puerto 9734 para la comunicación.

```c
/*  Make the necessary includes and set up the variables.  */
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

/*  Remove any old socket and create an unnamed socket for the server.  */
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

/*  Name the socket.  */
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/*  Create a connection queue and wait for clients.  */
    listen(server_sockfd, 5);
    while(1) {
        char ch;

        printf("server waiting\n");

/*  Accept a connection.  */
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);

/*  We can now read/write to client on client_sockfd.  */
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}

```

Cuando se emplean números enteros en la estructura de direcciones, estas deben ser convertidas en números de red (*network order*). Por ejemplo el número del puerto 9734, que es de 16 bits, se emplea `htons` (*host to network short*) para obtener su *network order*.

La dinámica es muy sencilla, el cliente le envía el carácter 'A', y el servidor toma dicho carácter y devuelve el siguiente.

A continuación se lista el cliente.

```c
/*  Make the necessary includes and set up the variables.  */
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{   int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char ch = 'A';

/*  Create a socket for the client.  */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

/*  Name the socket, as agreed with the server.  */
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = htons(9734);
    len = sizeof(address);

/*  Now connect our socket to the server's socket.  */
    result = connect(sockfd, (struct sockaddr *)&address, len);

    if(result == -1) {
        perror("oops: client3");
        exit(1);
    }

/*  We can now read/write via sockfd.  */
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

En este caso indicamos que el server se encuentra en la misma computadora (*localhost* o 127.0.0.1) y la función *inet_addr* ya devuelve el número a *network order*.

En la siguiente ventana se compila los fuentes y se ejecuta. Al final recuerde que se debe de terminar el proceso servidor.

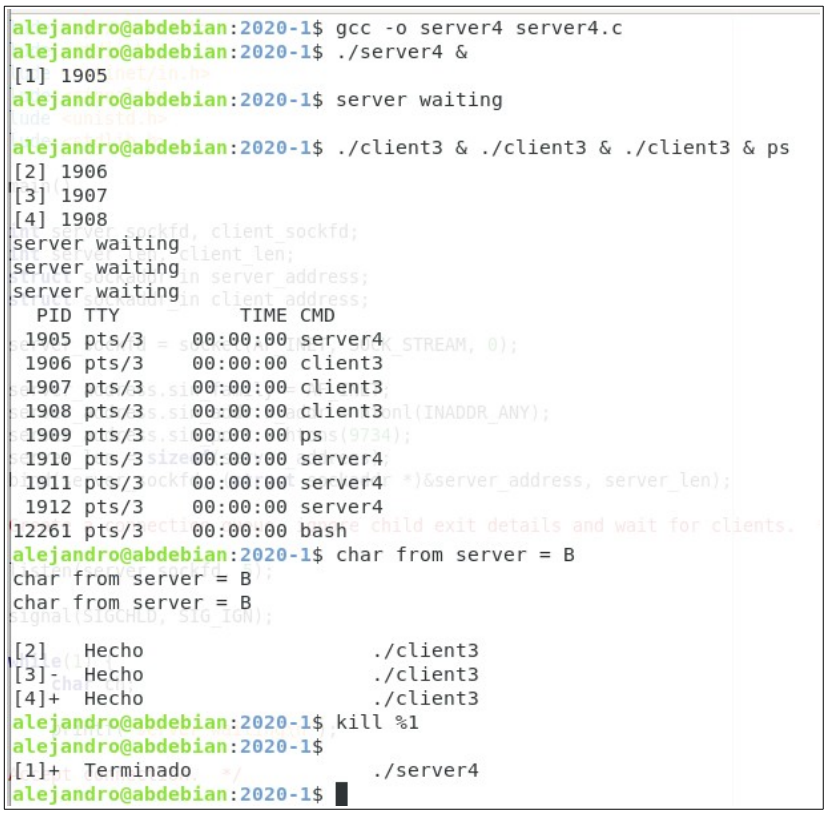
```
alejandro@abdebian:2020-1$ gcc -o server3 server3.c
alejandro@abdebian:2020-1$ gcc -o client3 client3.c
alejandro@abdebian:2020-1$ ./server3 &
[1] 30282
alejandro@abdebian:2020-1$ server waiting
alejandro@abdebian:2020-1$ ./client3
server waiting
char from server = B
alejandro@abdebian:2020-1$ kill %1
alejandro@abdebian:2020-1$
[1]+  Terminado              ./server3
alejandro@abdebian:2020-1$
```

Modificando el servidor para que acepte múltiples clientes

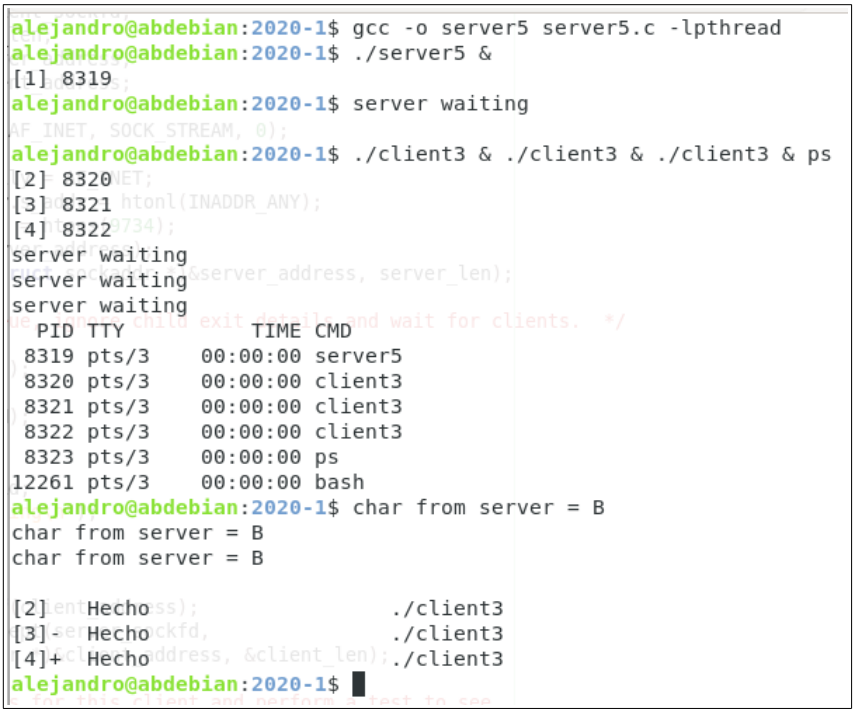
```c
/*  This program, server4.c, begins in similar vein to our last server,
    with the notable addition of an include for the signal.h header file.
    The variables and the procedure of creating and naming a socket are the same.  */

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/*  Create a connection queue, ignore child exit details and wait for clients.  */

    listen(server_sockfd, 5);

    signal(SIGCHLD, SIG_IGN);

    while(1) {
        char ch;

        printf("server waiting\n");

/*  Accept connection.  */

        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);

/*  Fork to create a process for this client and perform a test to see
    whether we're the parent or the child.  */

        if(fork() == 0) {

/*  If we're the child, we can now read/write to the client on client_sockfd.
    The five second delay is just for this demonstration.  */

            read(client_sockfd, &ch, 1);
            sleep(5);
            ch++;
            write(client_sockfd, &ch, 1);
            close(client_sockfd);
            exit(0);
        }

/*  Otherwise, we must be the parent and our work for this client is finished.  */

        else {
            close(client_sockfd);
        }
    }
}
```

El hecho que el *socket* original aún está disponible y que los *sockets* se comportan como descriptores de archivos nos dan un método para atender múltiples clientes al mismo tiempo. Si el servidor invoca a *fork()* para crear una segunda copia de si mismo, el *socket* abierto será heredado por el nuevo proceso hijo. Este puede comunicarse con el cliente conectado mientras que el servidor continúa aceptando más conexiones de clientes.

Debido a que se está creando procesos hijos que no son esperados a que terminen, se debe instruir al servidor que ignore la señal `SIGCHLD` para evitar procesos *zombies*.



Modificaremos el programa para que el servidor emplee hilos en lugar de procesos hijos.



A continuación el programa fuente

```c
/*  This program, server5.c, begins in similar vein to our last server,
    with the notable addition of create threads.
    The variables and the procedure of creating and naming a socket are the same.  */
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>

void *thread_func(void * tsocket) {
    long  c_sockfd = (long)tsocket;
    char ch;

    read(c_sockfd, &ch, 1);
    sleep(5);
    ch++;
    write(c_sockfd, &ch, 1);
    close(c_sockfd);
    pthread_exit(NULL);
}

int main()
{
    long server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;

    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/*  Create a connection queue, ignore child exit details and wait for clients.  */

    listen(server_sockfd, 5);

    while(1) {
         pthread_t thread_id;
        printf("server waiting\n");

/*  Accept connection.  */

        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);

/*  Fork to create a process for this client and perform a test to see
    whether we're the parent or the child.  */

        int ret = pthread_create(&thread_id,NULL, thread_func, (void
        *)client_sockfd);
        if (ret != 0) { printf("Error from pthread: %d\n", ret); exit(1); }
    }
}
```

- 1 -

Nota: El tema del laboratorio estará basado en el material expuesto.

Ejercicios.

1.- Elabore un servidor que devuelve “yes” si el número que un cliente le ha enviado es un número primo y “no” en caso contrario. El servidor debe atender múltiples clientes. El tipo de *socket* debe ser *stream socket* y escriba una versión para el dominio `AF_UNIX` y otra para el dominio `AF_INET`.

2.- Elabore un servidor que devuelva los *n* números de Fibonacci donde el número *n* ha sido proporcionado por el cliente. El servidor debe atender múltiples clientes. El tipo de *socket* debe ser *stream socket* y escriba una versión para el dominio `AF_UNIX` y otra para el dominio `AF_INET`.

3.- Elabore una versión mínima de un servidor web.

Bibliografía

1.- ANDREW S. TANENBAUM, HERBERT BOS. *Modern Operating System. Fourth Edition*

2..- MICHAEL KERRISK. *The Linux Programming Interface.*

3.- NEIL MATTHEW, RICHARD STONES. *Beginning Linux Programming. 4Th Edition*

4.- KURT WALL, MARK WATSON, MARK WHITIS. *Linux Programming Unleashed.*