Enero 2020-Turno 9:30

1) Despliega la topología de red que se muestra en la figura usando `vtopol` y la configuración proporcionada:

- Configura los interfaces de forma manual, eligiendo adecuadamente sus direcciones.
- Configura los encaminadores RouterA y RouterB para que anuncien todas las redes usando RIP.
- Comprueba que todas las máquinas son alcanzables entre sí.

```
//en VM1
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip a add 192.168.0.130/25 dev eth0

[root@localhost ~]# ip route add default via 192.168.0.131


//en el router1
[root@localhost ~]# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip link set eth1 up
[root@localhost ~]# ip a add 192.168.0.131/25 dev eth0
[root@localhost ~]# ip a add 10.0.0.1/24 dev eth1

[root@localhost ~]# sudo gedit /etc/quagga/ripd.conf

router rip
version 2
network eth0
network eth1

[root@localhost ~]# service ripd start
Redirecting to /bin/systemctl start ripd.service


//en el router2
[root@localhost ~]# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[root@localhost ~]# ip link set eth1 up
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip a add 10.0.0.2/24 dev eth1
[root@localhost ~]# ip a add 172.16.0.1/24 dev eth0

[root@localhost ~]# nano /etc/quagga/ripd.conf

router rip
version 2
network eth0
network eth1

[root@localhost ~]# service ripd start
Redirecting to /bin/systemctl start ripd.service
```

//CONECTAMOS ROUTER2 CON ROUTER1

[root@localhost ~]# ping 10.0.0.1

PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.

64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.893 ms

64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.917 ms

64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.920 ms

//CONECTAMOS VM1 con ROUTER2

[root@localhost ~]# ping 172.16.0.1

PING 172.16.0.1 (172.16.0.1) 56(84) bytes of data.

64 bytes from 172.16.0.1: icmp_seq=1 ttl=63 time=0.701 ms

64 bytes from 172.16.0.1: icmp_seq=2 ttl=63 time=1.82 ms

64 bytes from 172.16.0.1: icmp_seq=3 ttl=63 time=1.87 ms

2) Escribe un programa servidor UDP que escuche peticiones realizadas a una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor devolverá la hora (en formato HH:MM:SS) al recibir cualquier mensaje. En cada mensaje, el servidor mostrará la dirección y el puerto del cliente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <arpa/inet.h>

#define MAX_BUFFER_SIZE 1024

void handle_client(int sockfd) {
   struct sockaddr_in client_addr;
   socklen_t addr_len = sizeof(client_addr);
   char buffer[MAX_BUFFER_SIZE];

    while (1) {
      // Recibe el mensaje del cliente
      ssize_t recv_len = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct
sockaddr*)&client_addr, &addr_len);
      if (recv_len < 0) {
         perror("Error al recibir datos del cliente");
         exit(EXIT_FAILURE);
      }

       // Obtiene la hora actual
      time_t rawtime;
```

```
      struct tm* timeinfo;
      time(&rawtime);
      timeinfo = localtime(&rawtime);
      char current_time[9];
      strftime(current_time, sizeof(current_time), "%T", timeinfo);

       // Imprime la dirección y el puerto del cliente
      char client_ip[INET6_ADDRSTRLEN];
      inet_ntop(AF_INET, &(client_addr.sin_addr), client_ip, sizeof(client_ip));
      printf("Mensaje recibido de %s:%d\n", client_ip, ntohs(client_addr.sin_port));

      // Envía la hora actual al cliente
      ssize_t send_len = sendto(sockfd, current_time, strlen(current_time), 0, (struct
sockaddr*)&client_addr, addr_len);
      if (send_len < 0) {
         perror("Error al enviar datos al cliente");
         exit(EXIT_FAILURE);
      }
   }
}

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

   if (argc < 3) {
      printf("Uso: %s <dirección> <puerto>\n", argv[0]);
      return EXIT_FAILURE;
   }

   const char* host = argv[1];
   int port = atoi(argv[2]);

   // Crea el socket UDP
   int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
   if (sockfd < 0) {
      perror("Error al crear el socket");
      return EXIT_FAILURE;
   }

   // Configura la dirección y el puerto del servidor
   struct sockaddr_in server_addr;
   memset(&server_addr, 0, sizeof(server_addr));
   server_addr.sin_family = AF_INET;
   server_addr.sin_addr.s_addr = inet_addr(host);
   server_addr.sin_port = htons(port);

    // Vincula el socket a la dirección y el puerto del servidor
   if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
      perror("Error al vincular el socket");
      return EXIT_FAILURE;
```

```
    }

    printf("Servidor UDP escuchando en %s:%d\n", host, port);

    // Maneja las conexiones entrantes
    handle_client(sockfd);

    // Cierra el socket
    close(sockfd);

    return EXIT_SUCCESS;
}
```

gcc udp_server.c -o udp_server

./udp_server localhost 12345

Desde otra consola: *echo "Hola" | nc -u localhost 12345*

3) Escribe un programa que lea simultáneamente de dos tuberías con nombre (tuberia1 y tuberia2). El programa mostrará el nombre de la tubería desde la que se leyó y los datos leídos. El programa detectará cuándo se cierra el extremo de escritura de una tubería (read() devuelve 0) para cerrarla y volver a abrirla.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/select.h>
#include <sys/stat.h>
#include <sys/types.h>

#define MAX_BUFFER_SIZE 1024

int main() {
    const char *tuberia1 = "./tuberia1";
    const char *tuberia2 = "./tuberia2";

    // Crea las tuberías si no existen
    mkfifo(tuberia1, 0666);
    mkfifo(tuberia2, 0666);

    // Abre las tuberías en modo lectura
    int pipe1 = open(tuberia1, O_RDONLY | O_NONBLOCK);
    int pipe2 = open(tuberia2, O_RDONLY | O_NONBLOCK);

    // Lista de descriptores de archivo de las tuberías
    fd_set read_fds;
    FD_ZERO(&read_fds);
```

```
    FD_SET(pipe1, &read_fds);
    FD_SET(pipe2, &read_fds);

    int max_fd = (pipe1 > pipe2) ? pipe1 : pipe2;

    while (1) {
        // Utiliza select para esperar a que haya datos disponibles en alguna tubería
        fd_set temp_fds = read_fds;
        int ready_fds = select(max_fd + 1, &temp_fds, NULL, NULL, NULL);

        if (ready_fds == -1) {
            perror("Error en select");
            exit(1);
        }

        for (int fd = 0; fd <= max_fd; fd++) {
            if (FD_ISSET(fd, &temp_fds)) {
                char buffer[MAX_BUFFER_SIZE];
                int bytes_read = read(fd, buffer, sizeof(buffer) - 1);

                if (bytes_read == 0) {
                    // Si read() devuelve 0, significa que se cerró el extremo de escritura de la tubería
                    printf("Se cerró el extremo de escritura de %s\n", (fd == pipe1) ? "tuberia1" :
"tuberia2");
                    close(fd);
                    FD_CLR(fd, &read_fds);
                    fd = open((fd == pipe1) ? tuberia1 : tuberia2, O_RDONLY | O_NONBLOCK);
                    FD_SET(fd, &read_fds);
                    max_fd = (pipe1 > pipe2) ? pipe1 : pipe2;

                }else {
                    buffer[bytes_read] = '\0';
                    printf("%s: %s", (fd == pipe1) ? "tuberia1" : "tuberia2", buffer);
                }
            }
        }
    }

    return 0;
}
```

```
gcc -std=c99 Ejercicio3.c -o programa
```

```
./programa
```

En la segunda consola, escribe datos en una de las tuberías usando el siguiente comando:

```
echo "Datos para tuberia1" > tuberia1
```

Vuelve a la primera consola y verifica que el programa haya leído los datos y los haya mostrado en la salida.

Ahora, en la segunda consola, escribe datos en la otra tubería con el siguiente comando:

```
echo "Datos para tuberia2" > tuberia2
```

Vuelve a la primera consola y verifica que el programa haya leído los nuevos datos y los haya mostrado en la salida.

Enero 2020-Turno 16:00

1) Despliega la topología de red que se muestra en la figura usando `vtopol` y la configuración proporcionada:

- Configura los interfaces de forma manual, eligiendo adecuadamente sus direcciones.
- Configura el encaminador Router para que anuncie prefijos en ambas redes.
- Comprueba que todas las máquinas son alcanzables entre sí.

Paso nº1 ROUTER

[root@localhost ~]# ip a add fd00:1:1:a::2/64 dev eth0

[root@localhost ~]# ip a add fd00:1:1:b::2/64 dev eth1

[root@localhost ~]# sudo ip link set dev eth0 up

[root@localhost ~]# sudo ip link set dev eth1 up

[root@localhost ~]# gedit /etc/quagga/zebra.conf

interface eth0

    no ipv6 nd suppress-ra

    ipv6 nd prefix fd00:1:1:a::/64

interface eth1

    no ipv6 nd suppress-ra

    ipv6 nd prefix fd00:1:1:b::/64

sudo sysctl -w net.ipv6.conf.all.forwarding=1

net.ipv6.conf.all.forwarding = 1

service zebra start

VM1

[root@localhost ~]# ip a add fd00:1:1:a::1/64 dev eth0

LA MISMA RED

[root@localhost ~]# ping -6 fd00:1:1:a::2 -I eth0

PING fd00:1:1:a::2(fd00:1:1:a::2) from fd00:1:1:a::1 eth0: 56 data bytes

64 bytes from fd00:1:1:a::2: icmp_seq=1 ttl=64 time=0.717 ms

64 bytes from fd00:1:1:a::2: icmp_seq=2 ttl=64 time=0.785 ms

64 bytes from fd00:1:1:a::2: icmp_seq=3 ttl=64 time=0.593 ms

64 bytes from fd00:1:1:a::2: icmp_seq=4 ttl=64 time=0.687 ms

64 bytes from fd00:1:1:a::2: icmp_seq=5 ttl=64 time=0.717 ms

^C

--- fd00:1:1:a::2 ping statistics ---

5 packets transmitted, 5 received, 0% packet loss, time 4002ms

rtt min/avg/max/mdev = 0.593/0.699/0.785/0.070 ms

LA OTRA PARTE DE LA RED

[root@localhost ~]# ping -6 fd00:1:1:b::2 -I eth0

PING fd00:1:1:b::2(fd00:1:1:b::2) from fd00:1:1:a::1 eth0: 56 data bytes

64 bytes from fd00:1:1:b::2: icmp_seq=1 ttl=64 time=0.329 ms

64 bytes from fd00:1:1:b::2: icmp_seq=2 ttl=64 time=0.645 ms

64 bytes from fd00:1:1:b::2: icmp_seq=3 ttl=64 time=0.609 ms

64 bytes from fd00:1:1:b::2: icmp_seq=4 ttl=64 time=0.954 ms

^C

--- fd00:1:1:b::2 ping statistics ---

4 packets transmitted, 4 received, 0% packet loss, time 3002ms

rtt min/avg/max/mdev = 0.329/0.634/0.954/0.222 ms

VM2(“ip route add default” se debería hacer en el caso de que te dijeran que el eth0 no se anuncian los vecinos)

[cursoredes@localhost ~]$ sudo -i

[root@localhost ~]# ip link set eth0 up

[root@localhost ~]# ip a add fd00:1:1:b::1/64 dev eth0

[root@localhost ~]# ip address

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000

   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

   inet 127.0.0.1/8 scope host lo

      valid_lft forever preferred_lft forever

inet6 ::1/128 scope host

valid_lft forever preferred_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000

link/ether 08:00:27:3f:4d:0e brd ff:ff:ff:ff:ff:ff

inet6 fd00:1:1:b::1/64 scope global

valid_lft forever preferred_lft forever

inet6 fd00::b:a00:27ff:fe3f:4d0e/64 scope global mngtmpaddr dynamic

valid_lft 2591986sec preferred_lft 604786sec

inet6 fe80::a00:27ff:fe3f:4d0e/64 scope link

valid_lft forever preferred_lft forever

HACER PING VM1 A VM2

[root@localhost ~]# ping -6 fd00:1:1:b::1 -I eth0

PING fd00:1:1:b::1(fd00:1:1:b::1) from fd00:1:1:a::1 eth0: 56 data bytes

64 bytes from fd00:1:1:b::1: icmp_seq=1 ttl=63 time=0.862 ms

64 bytes from fd00:1:1:b::1: icmp_seq=2 ttl=63 time=0.942 ms

64 bytes from fd00:1:1:b::1: icmp_seq=3 ttl=63 time=0.971 ms

^C

--- fd00:1:1:b::1 ping statistics ---

3 packets transmitted, 3 received, 0% packet loss, time 2003ms

rtt min/avg/max/mdev = 0.862/0.925/0.971/0.046 ms

2) Escribe un programa servidor TCP que escuche en una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor devolverá lo que el cliente le envíe. En cada conexión, el servidor mostrará la dirección y el puerto del cliente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE];
    ssize_t bytes_received;
```

```
    // Recibir datos del cliente
    bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0);
    if (bytes_received < 0) {
        perror("Error al recibir datos del cliente");
        return;
    }

    // Mostrar la dirección y el puerto del cliente
    struct sockaddr_in6 client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    getpeername(client_socket, (struct sockaddr*)&client_addr, &client_addr_len);
    char client_ip[INET6_ADDRSTRLEN];
    inet_ntop(AF_INET6, &(client_addr.sin6_addr), client_ip, INET6_ADDRSTRLEN);
    int client_port = ntohs(client_addr.sin6_port);
    printf("Cliente conectado desde %s:%d\n", client_ip, client_port);

     // Enviar los datos recibidos de vuelta al cliente
    ssize_t bytes_sent = send(client_socket, buffer, bytes_received, 0);
    if (bytes_sent < 0) {
        perror("Error al enviar datos al cliente");
    }

    // Cerrar el socket del cliente
    close(client_socket);
}

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

    if (argc != 3) {
        printf("Uso: %s <dirección> <puerto>\n", argv[0]);
        return 1;
    }

    const char *address = argv[1];
    int port = atoi(argv[2]);

    // Crear el socket del servidor
    int server_socket = socket(AF_INET6, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Error al crear el socket del servidor");
        return 1;
    }

    // Configurar la dirección y el puerto del servidor
    struct sockaddr_in6 server_addr;
    server_addr.sin6_family = AF_INET6;
    server_addr.sin6_port = htons(port);
    if (inet_pton(AF_INET6, address, &(server_addr.sin6_addr)) <= 0) {
```

```
        perror("Dirección inválida");
        return 1;
    }

    // Enlazar el socket a la dirección y puerto del servidor
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Error al enlazar el socket");
        return 1;
    }

    // Escuchar por conexiones entrantes
    if (listen(server_socket, 5) < 0) {
        perror("Error al escuchar por conexiones entrantes");
        return 1;
    }

    printf("Servidor escuchando en %s:%d\n", address, port);

    while (1) {
        // Aceptar una nueva conexión
        struct sockaddr_in6 client_addr;
        socklen_t client_addr_len = sizeof(client_addr);
        int client_socket = accept(server_socket, (struct sockaddr*)&client_addr,
&client_addr_len);
        if (client_socket < 0) {
            perror("Error al aceptar la conexión entrante");
            return 1;
        }

        // Procesar la conexión en un hilo separado o manejarla aquí mismo
        handle_client(client_socket);
    }

     // Cerrar el socket del servidor
    close(server_socket);

    return 0;
}
```

Terminal1 (Servidor):

Compila el programa del servidor TCP utilizando el siguiente comando:

```
gcc servidor_tcp.c -o servidor_tcp
```

Ejecuta el servidor proporcionando la dirección IP y el puerto como argumentos. Por ejemplo:

```
./servidor_tcp ::1 8080
```

Terminal2 (Cliente):

Conéctate al servidor utilizando el comando telnet. Por ejemplo:

telnet ::1 8080

Escribe un mensaje y presiona Enter para enviarlo al servidor. Por ejemplo:

Hola, servidor!

3) Escribe un programa que ejecute dos comandos de la siguiente forma:
● Los comandos serán el primer y segundo argumento del programa. El resto de argumentos del programa se considerarán argumentos del segundo comando:

$ ./conecta comando1 comando2 arg2_1 arg2_2 ...

● Cada comando se ejecutará en un proceso distinto, que imprimirá su PID por el terminal.
● El programa conectará la salida estándar del primer proceso con la entrada estándar del segundo, y esperará la finalización de ambos para terminar su ejecución.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
   if (argc < 3) {
      printf("Uso: %s comando1 comando2 [arg2_1] [arg2_2] ...\n", argv[0]);
      return 1;
   }

   pid_t pid1, pid2;
   int pipefd[2];

   if (pipe(pipefd) == -1) {
      perror("Error al crear la tubería");
      return 1;
   }

   pid1 = fork();
   if (pid1 == -1) {
      perror("Error al crear el proceso hijo 1");
      return 1;

   }else if (pid1 == 0) {
      // Código del proceso hijo 1
      close(pipefd[0]);  // Cerramos la lectura del extremo del pipe no utilizado

      printf("PID del proceso hijo 1: %d\n", getpid());
      printf("Ejecutando comando 1: %s\n", argv[1]);

      // Conectamos la salida estándar del proceso hijo 1 al extremo de escritura del pipe
```

```
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);  // Cerramos el extremo de escritura del pipe

        // Ejecutamos el primer comando
        execvp(argv[1], &argv[1]);

        perror("Error al ejecutar el comando 1");
        exit(1);

    } else {
        // Código del proceso padre
        pid2 = fork();
        if (pid2 == -1) {
            perror("Error al crear el proceso hijo 2");
            return 1;
        } else if (pid2 == 0) {
            // Código del proceso hijo 2
            close(pipefd[1]);  // Cerramos la escritura del extremo del pipe no utilizado

            printf("PID del proceso hijo 2: %d\n", getpid());
            printf("Ejecutando comando 2: %s\n", argv[2]);

            // Conectamos la entrada estándar del proceso hijo 2 al extremo de lectura del pipe
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);  // Cerramos el extremo de lectura del pipe

            // Ejecutamos el segundo comando
            execvp(argv[2], &argv[2]);

            perror("Error al ejecutar el comando 2");
            exit(1);

        } else {
            // Código del proceso padre
            close(pipefd[0]);  // Cerramos ambos extremos del pipe en el proceso padre
            close(pipefd[1]);

            // Esperamos a que ambos procesos hijos terminen su ejecución
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);

             printf("Ambos procesos han terminado.\n");
        }
    }

    return 0;
}
```

Terminal:

```
gcc -o conecta programa.c
```

Esto ejecutará el comando "cat" para mostrar el contenido de un archivo y el comando "head" con la opción "-n 10" para mostrar las primeras 10 líneas.

```
./conecta cat head -n 10 nombre_archivo.txt
```

```
./conecta ls grep .txt
```

Enero 2023-Turno 10:00

1) RIP

2)Ejercicio 2 (1 punto). Escribe un programa que ejecute dos comandos de la siguiente forma:
● Los comandos serán el primer y segundo argumento del programa.
● El primer comando se ejecutará en un nuevo proceso y el segundo comando se ejecutará en el proceso principal. Ambos procesos imprimirán su PID por el terminal.
● Se conectará la salida estándar del nuevo proceso con la entrada estándar del proceso principal mediante una tubería sin nombre.
Un posible ejemplo de ejecución sería:
$ ./ej2 uname wc
Padre: 22253
Hijo: 22254
1 1 6

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <command1> <command2>\n", argv[0]);
        return 1;
    }

    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // Proceso hijo
        close(pipefd[0]);  // Cerramos la lectura del extremo de la tubería
```

```
        printf("Hijo: %d\n", getpid());

        // Redirigir la salida estándar al extremo de escritura de la tubería
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);

        // Ejecutar el primer comando
        if (execlp(argv[1], argv[1], NULL) == -1) {
            perror("execlp");
            return 1;
        }
    } else {
        // Proceso padre
        close(pipefd[1]);  // Cerramos la escritura del extremo de la tubería

        printf("Padre: %d\n", getpid());

        // Redirigir la entrada estándar al extremo de lectura de la tubería
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);

        // Ejecutar el segundo comando
        if (execlp(argv[2], argv[2], NULL) == -1) {
            perror("execlp");
            return 1;
        }
    }
    return 0;
}
```

[cursoredes@localhost Documents]$ gcc ej2Prueba.c -o prueba
[cursoredes@localhost Documents]$ ./prueba uname wc

3) Ejercicio 3 (1,5 puntos). Escribe un programa servidor TCP que escuche en una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor recibirá el PID de un proceso, imprimirá en el terminal el PID recibido y su valor de nice y enviará al cliente el valor del nice. En caso de que el proceso no exista en el servidor, devolverá un mensaje de error informando.
Un posible ejemplo de ejecución sería:
Servidor:

$ ps -le

F S UID PID PPID C PRI NI ...
...
0 S 1000 1079 1078 0 80 0 ...
0 S 1000 1080 1078 0 80 -20 ...
$ ./ej3 :: 8888
PID[1079]: 0
PID[1080]: -20

Cliente (netcat):

$ nc ::1 8888
1079
0
1080

-20
9999
Proceso 9999 no encontrado

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

#define MAX_BUFFER_SIZE 100

void handle_client(int client_socket) {
   char pid_buffer[MAX_BUFFER_SIZE];
   int pid;
   int nice_value;
   int ret;

   while (1) {
      ret = recv(client_socket, pid_buffer, sizeof(pid_buffer), 0);
      if (ret == -1) {
         perror("Error al recibir el PID");
         break;
      } else if (ret == 0) {
         // El cliente ha cerrado la conexión
         printf("Conexión cerrada por el cliente\n");
         break;
      }

      pid = atoi(pid_buffer);

      errno = 0;  // Reset errno before calling getpriority()

      nice_value = getpriority(0, pid);  // Utilizamos el valor numérico directamente

      char response_buffer[MAX_BUFFER_SIZE];
      memset(response_buffer, 0, sizeof(response_buffer));

      if (nice_value == -1) {
         if (errno == ESRCH) {
            printf("Proceso %d no encontrado\n", pid);
            snprintf(response_buffer, sizeof(response_buffer), "%d\nProceso no encontrado\n",
pid);
         } else {
            perror("Error al obtener el valor de nice");
```

```
            snprintf(response_buffer, sizeof(response_buffer), "%d\nError al obtener el valor de
nice\n", pid);
        }
    } else {
        printf("PID[%d]: %d\n", pid, nice_value);
        snprintf(response_buffer, sizeof(response_buffer), "%d\n%d\n", pid, nice_value);
    }

    send(client_socket, response_buffer, strlen(response_buffer), 0);
  }
  shutdown(client_socket, SHUT_RDWR);  // Cerrar la conexión de forma ordenada
  close(client_socket);
}

int main(int argc, char *argv[]) {
  if (argc != 3) {
    fprintf(stderr, "Uso: %s dirección puerto\n", argv[0]);
    return 1;
  }

  char *address = argv[1];
  char *port = argv[2];

  // Crear el socket
  int server_socket = socket(AF_INET6, SOCK_STREAM, 0);
  if (server_socket == -1) {
    perror("Error al crear el socket");
    return 1;
  }

   // Permitir reutilizar la dirección y el puerto
  int reuse = 1;
  if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse)) == -1) {
    perror("Error al configurar el socket");
    close(server_socket);
    return 1;
  }

  // Configurar la dirección y el puerto
  struct addrinfo hints, *result;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET6;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;

  if (getaddrinfo(address, port, &hints, &result) != 0) {
    perror("Error al obtener la información de la dirección");
    close(server_socket);
    return 1;
```

```
    }

    // Vincular el socket a la dirección y el puerto
    if (bind(server_socket, result->ai_addr, result->ai_addrlen) == -1) {
        perror("Error al vincular el socket");
        freeaddrinfo(result);
        close(server_socket);
        return 1;
    }

    freeaddrinfo(result);

    // Escuchar conexiones entrantes
    if (listen(server_socket, SOMAXCONN) == -1) {
        perror("Error al escuchar conexiones");
        close(server_socket);
        return 1;
    }

    printf("Servidor iniciado. Escuchando en %s:%s\n", address, port);

    while (1) {
        // Aceptar una conexión entrante
        struct sockaddr_storage client_addr;
        socklen_t client_addr_len = sizeof(client_addr);
        int client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_addr_len);
        if (client_socket == -1) {
            perror("Error al aceptar la conexión");
            close(server_socket);
            return 1;
        }

        char client_ip[INET6_ADDRSTRLEN];
        inet_ntop(client_addr.ss_family, &(((struct sockaddr_in6 *)&client_addr)->sin6_addr),
client_ip, sizeof(client_ip));
        printf("Conexión aceptada desde %s\n", client_ip);

        handle_client(client_socket);
    }

    close(server_socket);

    return 0;
}
```

Enero 2023-Turno 14:00

1) Configura las dos interfaces de forma manual, seleccionando las direcciones IP de forma adecuada. Comprueba que ambas máquinas son alcanzables entre sí.
Configura VM1 como servidor DNS para el dominio examenasor.es con la siguiente información:
● El servidor primario es ns.examenasor.es, correspondiente a VM1.
● El e-mail de contacto es contact@examenasor.es.
● Elegir libremente el número de serie y los valores de los temporizadores.
● El servidor de correo electrónico es mail.examenasor.es.
● La dirección IP de www.examenasor.es es 192.168.0.200.
● La dirección IP de mail.examenasor.es es 192.168.0.250.
● El nombre canónico de correo.examenasor.es es mail.examenasor.es.
Configura VM2 para que use VM1 como servidor DNS por defecto.

EN VM1

```
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip a add 192.168.0.1/24 dev eth0
[root@localhost ~]# gedit /etc/named.conf

Comentar allow query y recursion
//allow-query     { localhost; };
//recursion yes;

//y añadir la zone

zone "examenasor.es." {
        type master;
        file "db.examenasor.es";
};

[root@localhost named]# gedit /var/named/db.examenasor.es

$TTL 2d
examenasor.es. IN SOA ns.examenasor.es. contact.examenasor.es. (

                    2003080800 ; serial number
                    3h ; refresh
                    15M ; update retry
                    3W12h ; expiry
                    2h20M ; nx ttl
                    )
        IN      NS      ns.examenasor.es.
        IN      MX 10   mail.examenasor.es
ns.examenasor.es.       IN      A       192.168.0.1
www.examenasor.es       IN      A       192.168.0.200
www.examenasor.es       IN      AAAA    fd00::1
mail.examenasor.es      IN      A       192.168.0.250
correo.examenasor.es    IN      CNAME   mail.examenasor.es


[root@localhost ~]# named-checkzone examenasor.es. /var/named/db.examenasor.es
zone examenasor.es/IN: loaded serial 2003080800
OK

[root@localhost ~]# service named start
Redirecting to /bin/systemctl start named.service
```

EN VM2

```
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip a add 192.168.0.100/24 dev eth0

[root@localhost ~]# gedit /etc/resolv.conf

; generated by /usr/sbin/dhclient-script
search ns.examenasor.es.
nameserver 192.168.0.1

[root@localhost ~]# dig examenasor.es.

; <<>> DiG 9.9.4-RedHat-9.9.4-61.el7_5.1 <<>> examenasor.es.
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 28502
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;examenasor.es.                 IN      A

;; AUTHORITY SECTION:
examenasor.es.          8400    IN      SOA     ns.examenasor.es. contact.examenasor.es.
2003080800 10800 900 1857600 8400

;; Query time: 1 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Wed Jun 14 10:55:16 CEST 2023
;; MSG SIZE  rcvd: 89

[root@localhost ~]# dig A www.examenasor.es

; <<>> DiG 9.9.4-RedHat-9.9.4-61.el7_5.1 <<>> A www.examenasor.es
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 42084
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.examenasor.es.             IN      A

;; AUTHORITY SECTION:
examenasor.es.          8400    IN      SOA     ns.examenasor.es. contact.examenasor.es.
2003080800 10800 900 1857600 8400

;; Query time: 0 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Wed Jun 14 10:58:36 CEST 2023
;; MSG SIZE  rcvd: 93

[root@localhost ~]# dig MX examenasor.es
```

```
; <<>> DiG 9.9.4-RedHat-9.9.4-61.el7_5.1 <<>> MX examenasor.es
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 3011
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 3

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;examenasor.es.                 IN      MX

;; ANSWER SECTION:
examenasor.es.          172800  IN      MX      10 mail.examenasor.es.examenasor.es.

;; AUTHORITY SECTION:
examenasor.es.          172800  IN      NS      ns.examenasor.es.examenasor.es.

;; ADDITIONAL SECTION:
mail.examenasor.es.examenasor.es. 172800 IN A   192.168.0.250
ns.examenasor.es.examenasor.es. 172800 IN A     192.168.0.1

;; Query time: 0 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Wed Jun 14 11:01:04 CEST 2023
;; MSG SIZE  rcvd: 126
```

Zona Inversa (Reverse)

VM1

```
[root@localhost ~]# gedit /etc/named.conf

zone "0.168.192.in-addr.arpa." {
        type master;
        file "db.0.168.192";
};

[root@localhost ~]# gedit /var/named/db.0.168.192

$TTL 2d
0.168.192.in-addr.arpa. IN SOA ns.examenasor.es. contact.examenasor.es. (

                        2003080800 ; serial number
                        3h ; refresh
                        15M ; update retry
                        3W12h ; expiry
                        2h20M ; nx ttl
                        )
        IN      NS      ns.examenasor.es.
1       IN      PTR     ns.examenasor.es.
200     IN      PTR     www.examenasor.es
250     IN      PTR     mail.examenasor.es


[root@localhost ~]# named-checkzone 0.168.192.in-addr.arpa. /var/named/db.0.168.192
```

```
zone 0.168.192.in-addr.arpa/IN: loaded serial 2003080800
OK

[root@localhost ~]# service named reload
Redirecting to /bin/systemctl reload named.service
```

VM2

```
[root@localhost ~]# dig PTR 250.0.168.192.in-addr.arpa.

; <<>> DiG 9.9.4-RedHat-9.9.4-61.el7_5.1 <<>> PTR 250.0.168.192.in-addr.arpa.
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 3106
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;250.0.168.192.in-addr.arpa.        IN      PTR
;; ANSWER SECTION:
250.0.168.192.in-addr.arpa. 172800 IN      PTR     mail.examenasor.es.0.168.192.in-addr.arpa.
;; AUTHORITY SECTION:
0.168.192.in-addr.arpa.    172800  IN      NS      ns.examenasor.es.
;; ADDITIONAL SECTION:
ns.examenasor.es.          172800  IN      A       192.168.0.1
;; Query time: 1 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Wed Jun 14 11:34:54 CEST 2023
;; MSG SIZE  rcvd: 134
```

2) Ejercicio 2 (1 punto). Escribe un programa servidor UDP que escuche en una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor recibirá el nombre de un usuario del sistema y devolverá su directorio de inicio (home directory) asociado. En caso de que el usuario no exista, devolverá un mensaje de error informando.
Un posible ejemplo de ejecución sería:
Servidor:

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ grep user /etc/passwd
$ ./ej2 :: 8888
```

Cliente:

```
$ nc -u ::1 8888
root
/root
user
Usuario user no encontrado
^C
$
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <pwd.h>

#define BUFFER_SIZE 1024

// Función para obtener el directorio de inicio de un usuario
char* getHomeDirectory(char* username) {
   struct passwd *pw;
   pw = getpwnam(username);
   if (pw != NULL) {
      return pw->pw_dir;
   }
   return NULL;
}

int main(int argc, char *argv[]) {
   if (argc != 3) {
      fprintf(stderr, "Uso: %s <direccion> <puerto>\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   char *address = argv[1];
   char *port = argv[2];

   int sockfd;
   struct addrinfo hints, *servinfo, *p;
   int status;
   char buffer[BUFFER_SIZE];

   memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_UNSPEC; // Permite tanto IPv4 como IPv6
   hints.ai_socktype = SOCK_DGRAM; // Socket UDP

   // Obtener la información de dirección del servidor
   if ((status = getaddrinfo(address, port, &hints, &servinfo)) != 0) {
      fprintf(stderr, "Error en getaddrinfo: %s\n", gai_strerror(status));
      exit(EXIT_FAILURE);
   }

   // Recorrer todas las direcciones posibles y enlazar el socket al primer resultado válido
   for (p = servinfo; p != NULL; p = p->ai_next) {
      if ((sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
         perror("Error al crear el socket");
         continue;
      }

      if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
```

```
            close(sockfd);
            perror("Error al enlazar el socket");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "No se pudo enlazar el socket\n");
        exit(EXIT_FAILURE);
    }

    freeaddrinfo(servinfo);

    printf("Servidor escuchando en %s:%s\n", address, port);

    while (1) {
        struct sockaddr_storage client_addr;
        socklen_t addr_len = sizeof client_addr;

        ssize_t num_bytes = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0, (struct
sockaddr*)&client_addr, &addr_len);
        if (num_bytes == -1) {
            perror("Error al recibir datos");
            exit(EXIT_FAILURE);
        }

        buffer[num_bytes] = '\0';

        // Eliminar el caracter '\n' final
        if (buffer[num_bytes - 1] == '\n') {
            buffer[num_bytes - 1] = '\0';
        }

        // Obtener el directorio de inicio del usuario
        char *home_directory = getHomeDirectory(buffer);
        if (home_directory != NULL) {
            size_t directory_length = strlen(home_directory);
            strcat(home_directory, "\n");
            sendto(sockfd, home_directory, directory_length + 1, 0, (struct sockaddr*)&client_addr,
addr_len);
        } else {
            sendto(sockfd, "Usuario no encontrado \n", 24, 0, (struct sockaddr*)&client_addr,
addr_len);
        }
    }

    close(sockfd);
```

```
    return 0;
}
```

3) Ejercicio 3 (1,5 puntos). Escribe un programa que lea simultáneamente dos tuberías con nombre usando multiplexación de E/S síncrona. Las tuberías existirán previamente en el directorio actual de trabajo y se llamarán fifo1 y fifo2. El programa mostrará el nombre de la tubería de la que ha leído y los datos leídos. Además, el programa cerrará y volverá a abrir la tubería al detectar el fin de fichero.

Un posible ejemplo de ejecución sería:

Lector:

$ ./ej3

Leído desde fifo1: Hola

Leído desde fifo2: Adiós

Escritor:

$ echo Hola > fifo1

$ echo Adiós > fifo2

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>

#define FIFO1 "fifo1"
#define FIFO2 "fifo2"

int main() {
    int fd1, fd2;
    fd_set readfds;
    char buffer[1024];

    // Abrir las tuberías en modo de solo lectura y bloqueante
    fd1 = open(FIFO1, O_RDONLY);
    fd2 = open(FIFO2, O_RDONLY);

    if (fd1 == -1 || fd2 == -1) {
        perror("Error al abrir las tuberías");
        exit(1);
    }
```

```
    while (1) {
        FD_ZERO(&readfds);
        FD_SET(fd1, &readfds);
        FD_SET(fd2, &readfds);

        // Utilizar select para esperar a que haya datos disponibles en alguna tubería
        int maxfd = (fd1 > fd2) ? fd1 + 1 : fd2 + 1;
        int activity = select(maxfd, &readfds, NULL, NULL, NULL);

        if (activity == -1) {
            perror("Error en select");
            exit(1);
        }

        // Comprobar si hay datos disponibles en la tubería 1
        if (FD_ISSET(fd1, &readfds)) {
            int bytes_read = read(fd1, buffer, sizeof(buffer));
            if (bytes_read <= 0) {
                close(fd1);
                fd1 = open(FIFO1, O_RDONLY);
                printf("Tubería fifo1 cerrada y reabierta\n");
            } else {
                printf("Leído desde fifo1: %.*s", bytes_read, buffer);
            }
        }

        // Comprobar si hay datos disponibles en la tubería 2
        if (FD_ISSET(fd2, &readfds)) {
            int bytes_read = read(fd2, buffer, sizeof(buffer));
            if (bytes_read <= 0) {
                close(fd2);
                fd2 = open(FIFO2, O_RDONLY);
                printf("Tubería fifo2 cerrada y reabierta\n");
            } else {
                printf("Leído desde fifo2: %.*s", bytes_read, buffer);
            }
        }
    }

    // Cerrar las tuberías
    close(fd1);
    close(fd2);

    return 0;
}
```

Terminal 1 :

[cursoredes@localhost 2023-Enero-14-00]$ mkfifo fifo1 fifo2

[cursoredes@localhost 2023-Enero-14-00]$ ./ej3

Terminal2 :

[cursoredes@localhost 2023-Enero-14-00]$ echo Hola > fifo1

[cursoredes@localhost 2023-Enero-14-00]$ echo Adios > fifo2

Enero 2022-Turno 13:00

1) Configura el encaminador (VM3) para que anuncie por DHCP el rango de direcciones 192.168.0.140 - 192.168.0.160 y el encaminador por defecto. Configura el resto de la red para que todas las máquinas sean alcanzables entre sí. Comprueba que VM1 obtiene la dirección del servidor DHCP.

En VM2:

```
[root@localhost ~]# ip a add 10.0.2.1/24 dev eth0
[root@localhost ~]# ip link set dev eth0 up
[root@localhost ~]# ip route add default via 10.0.2.2
```

En Router:

```
[root@localhost ~]# ip link set dev eth0 up
[root@localhost ~]# ip link set dev eth1 up
[root@localhost ~]# ip a add 192.168.0.130/25 dev eth0
[root@localhost ~]# ip a add 10.0.2.2/24 dev eth1
[root@localhost ~]# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1

[root@localhost ~]# gedit /etc/dhcp/dhcpd.conf
#
# DHCP Server Configuration file.
#   see /usr/share/doc/dhcp*/dhcpd.conf.example
#   see dhcpd.conf(5) man page
#

subnet 192.168.0.128 netmask 255.255.255.128 {
        range 192.168.0.140 192.168.0.160;
        option routers 192.168.0.130;
        option broadcast-address 192.168.0.255;
}
```

hacer ping vm2,router

```
[root@localhost ~]# ping 10.0.2.1

[root@localhost ~]# service dhcpd start
Redirecting to /bin/systemctl start dhcpd.service
```

En VM1:

```
[root@localhost ~]# dhclient -d eth0
[root@localhost ~]# ip a
```

ping a router eth1

```
[root@localhost ~]# ping 10.0.2.2
```

ping a VM2

```
[root@localhost ~]# ping 10.0.2.1
[root@localhost ~]# ping -R -c 3 10.0.2.1
```

2) Ejercicio 2 (1 punto). Escribe un programa servidor TCP que devuelva la dirección del cliente al recibir cualquier mensaje. El programa se ejecutará de la siguiente manera:
$ ./tcp <dir IPv4 o IPv6 en cualquier formato> <puerto>
Por ejemplo:
$ ./tcp :: 7777

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

#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
   if (argc < 3) {
      printf("Uso: %s <dirección IP> <puerto>\n", argv[0]);
      return 1;
   }
   const char *ipAddress = argv[1];
   int port = atoi(argv[2]);

   // Crear el socket
   int serverSocket = socket(AF_INET6, SOCK_STREAM, 0);
   if (serverSocket < 0) {
      perror("Error al crear el socket");
      return 1;
   }
   // Configurar la dirección del servidor
   struct sockaddr_in6 serverAddress;
   memset(&serverAddress, 0, sizeof(serverAddress));
   serverAddress.sin6_family = AF_INET6;
   serverAddress.sin6_port = htons(port);

   if (inet_pton(AF_INET6, ipAddress, &(serverAddress.sin6_addr)) <= 0) {
      perror("Dirección IP inválida");
      return 1;
   }

   // Vincular el socket a la dirección del servidor
   if (bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) {
      perror("Error al vincular el socket");
      return 1;
   }
   // Escuchar en el socket
   if (listen(serverSocket, 1) < 0) {
      perror("Error al escuchar en el socket");
```

```
        return 1;
    }

    printf("Servidor escuchando en %s:%d\n", ipAddress, port);

    // Aceptar conexiones entrantes
    struct sockaddr_in6 clientAddress;
    socklen_t clientAddressLength = sizeof(clientAddress);
    int clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddress,
&clientAddressLength);

    if (clientSocket < 0) {
        perror("Error al aceptar la conexión");
        return 1;
    }

    char clientIP[INET6_ADDRSTRLEN];
    inet_ntop(AF_INET6, &(clientAddress.sin6_addr), clientIP, sizeof(clientIP));
    printf("Conexión aceptada desde %s:%d\n", clientIP, ntohs(clientAddress.sin6_port));

     // Recibir y procesar los mensajes del cliente
    char buffer[BUFFER_SIZE];
    ssize_t bytesRead;
    while ((bytesRead = read(clientSocket, buffer, BUFFER_SIZE)) > 0) {
        printf("Mensaje recibido: %s", buffer);
        printf("Dirección del cliente: %s:%d\n", clientIP, ntohs(clientAddress.sin6_port));

        // Enviar la dirección del cliente de vuelta al cliente
        write(clientSocket, clientIP, strlen(clientIP));
         write(clientSocket, ":", 1);
        char clientPortStr[6];
        sprintf(clientPortStr, "%d", ntohs(clientAddress.sin6_port));
        write(clientSocket, clientPortStr, strlen(clientPortStr));

        memset(buffer, 0, BUFFER_SIZE);
    }

    if (bytesRead == 0) {
        printf("El cliente cerró la conexión\n");
    } else if (bytesRead < 0) {
        perror("Error al leer del socket");
    }

    // Cerrar los sockets
    close(clientSocket);
    close(serverSocket);

    return 0;
}
```

Terminal 1:
[cursoredes@localhost 2022-Enero]$ ./ej1 :: 7777

Terminal 2:
[cursoredes@localhost 2022-Enero]$ nc :: 7777
hola server

Reemplaza <dirección IP del servidor> con la dirección IP del servidor donde está escuchando (en tu caso, :: para todas las interfaces IPv6)

3) Ejercicio 3 (1,5 puntos). Escribe un programa que cree dos procesos y ejecute un comando en cada uno, de forma que la salida del primer comando sea la entrada del segundo. El proceso padre imprimirá el PID y el código de salida de cada proceso. El programa se ejecutará de la siguiente manera:
$ ./conecta comando1 argumento1 comando2 argumento2
Por ejemplo:
$ ./conecta ls -l wc -c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
   if (argc < 5) {
      printf("Uso: %s comando1 argumento1 comando2 argumento2\n", argv[0]);
      return 1;
   }

   // Crear el primer proceso
   pid_t pid1;
   pid1 = fork();

   if (pid1 < 0) {
      perror("Error al crear el primer proceso");
      return 1;
   } else if (pid1 == 0) {
      // Estamos en el primer proceso hijo
      printf("PID del primer proceso hijo: %d\n", getpid());
      printf("Ejecutando comando 1: %s %s\n", argv[1], argv[2]);

      // Redirigir la salida estándar al descriptor de escritura del pipe
      close(STDOUT_FILENO);
      dup2(STDERR_FILENO, STDOUT_FILENO);

       // Ejecutar el primer comando
      execlp(argv[1], argv[1], argv[2], NULL);
      perror("Error al ejecutar el primer comando");
      return 1;
   }
```

```
    // Crear el segundo proceso
    pid_t pid2;
    pid2 = fork();

    if (pid2 < 0) {
        perror("Error al crear el segundo proceso");
         return 1;
    } else if (pid2 == 0) {
        // Estamos en el segundo proceso hijo
        printf("PID del segundo proceso hijo: %d\n", getpid());
        printf("Ejecutando comando 2: %s %s\n", argv[3], argv[4]);

        // Cerrar el descriptor de escritura de la salida estándar
        close(STDOUT_FILENO);

        // Redirigir la entrada estándar al descriptor de lectura del pipe
        close(STDIN_FILENO);
        dup2(STDERR_FILENO, STDIN_FILENO);

        // Ejecutar el segundo comando
        execlp(argv[3], argv[3], argv[4], NULL);
        perror("Error al ejecutar el segundo comando");
        return 1;
    }

    // Estamos en el proceso padre
    printf("PID del proceso padre: %d\n", getpid());

    // Esperar a que ambos procesos hijos terminen
    int status1, status2;
    waitpid(pid1, &status1, 0);
    waitpid(pid2, &status2, 0);

    printf("El primer proceso hijo (PID %d) terminó con código de salida %d\n", pid1,
WEXITSTATUS(status1));
    printf("El segundo proceso hijo (PID %d) terminó con código de salida %d\n", pid2,
WEXITSTATUS(status2));

    return 0;
}
```

Enero 2021-Turno 17:00

1) Ejercicio 1 RIP

2) Ejercicio 2 (1 punto). Escribe un programa servidor UDP que escuche en una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. Recibirá del cliente una ruta de fichero y le devolverá una cadena con el tipo del fichero (regular, directorio, enlace, fifo u otro). Además, al recibir cada mensaje imprimirá en el terminal la dirección y el puerto del cliente.

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

#define BUFFER_SIZE 1024

// Función para determinar el tipo de archivo
const char* get_file_type(const char* path) {
   struct stat st;
   if (stat(path, &st) == 0) {
      if (S_ISREG(st.st_mode))
         return "regular";
      else if (S_ISDIR(st.st_mode))
          return "directorio";
      else if (S_ISLNK(st.st_mode))
         return "enlace";
      else if (S_ISFIFO(st.st_mode))
         return "fifo";
      else
         return "otro";
   }
   return "error";
}

int main(int argc, char *argv[]) {
   if (argc != 3) {
      printf("Uso: %s <dirección> <puerto>\n", argv[0]);
      return 1;
   }

   const char* address = argv[1];
   const int port = atoi(argv[2]);

   int sockfd;
```

```
struct sockaddr_in6 servaddr;
char buffer[BUFFER_SIZE];

// Crear el socket UDP
sockfd = socket(AF_INET6, SOCK_DGRAM, 0);
 if (sockfd == -1) {
    perror("Error al crear el socket");
    return 1;
}

// Configurar la dirección y puerto del servidor
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin6_family = AF_INET6;
servaddr.sin6_port = htons(port);
inet_pton(AF_INET6, address, &(servaddr.sin6_addr));

// Vincular el socket al servidor
if (bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1) {
    perror("Error al vincular el socket");
    close(sockfd);
    return 1;
}

// Esperar y procesar mensajes
while (1) {
    struct sockaddr_in6 cliaddr;
    socklen_t len = sizeof(cliaddr);
    ssize_t n = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0, (struct sockaddr*)&cliaddr, &len);
    if (n == -1) {
        perror("Error al recibir el mensaje");
        close(sockfd);
        return 1;
    }

    buffer[n] = '\0';

    // Imprimir la dirección y el puerto del cliente
    char client_addr_str[INET6_ADDRSTRLEN];
    inet_ntop(AF_INET6, &(cliaddr.sin6_addr), client_addr_str, sizeof(client_addr_str));
    printf("Mensaje recibido de %s:%d\n", client_addr_str, ntohs(cliaddr.sin6_port));

    // Obtener el tipo de archivo y enviar la respuesta al cliente
    const char* file_type = get_file_type(buffer);
     if (sendto(sockfd, file_type, strlen(file_type), 0, (struct sockaddr*)&cliaddr, len) == -1) {
        perror("Error al enviar la respuesta");
        close(sockfd);
        return 1;
    }
}
```

```
    // Cerrar el socket
    close(sockfd);

    return 0;
}
```

Terminal 1:

./udp_server ::1 12345

Terminal 2:

echo "/ruta/al/archivo" | nc -u ::1 12345

3) Ejercicio 3 (1,5 puntos). Escribe un programa que lea simultáneamente de una tubería con nombre, que existirá previamente en el directorio actual de trabajo y se llamará tuberia, y del terminal.

● Imprimirá en el terminal los datos leídos y terminará tras 5 segundos sin recibir datos.

● Al detectar el fin de fichero en la tubería, la cerrará y volverá a abrirla .

Nombre Contenido

3read.c Código fuente del programa con lectura simultánea.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <string.h>
#include <signal.h>

#define BUFFER_SIZE 1024

volatile sig_atomic_t timeout_flag = 0;

void handle_timeout(int signal) {
    timeout_flag = 1;
}

int main() {
    int pipe_fd;
    char buffer[BUFFER_SIZE];
    int num_bytes;
    fd_set read_fds;
    struct timeval timeout;

    // Configurar el manejo de señales para la alarma
```

```
signal(SIGALRM, handle_timeout);

// Configurar la alarma para 5 segundos
alarm(5);

// Abrir la tubería con nombre en modo de lectura
pipe_fd = open("tuberia", O_RDONLY);
if (pipe_fd == -1) {
    perror("Error al abrir la tubería");
    exit(EXIT_FAILURE);
}

// Configurar el conjunto de descriptores de archivo para select()
FD_ZERO(&read_fds);
FD_SET(STDIN_FILENO, &read_fds);
FD_SET(pipe_fd, &read_fds);

// Bucle principal
while (!timeout_flag) {
    // Configurar el temporizador para 1 segundo
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    // Configurar el conjunto de descriptores de archivo para select()
    fd_set temp_fds = read_fds;

    // Esperar a que haya datos disponibles o se alcance el tiempo de espera
    if (select(pipe_fd + 1, &temp_fds, NULL, NULL, &timeout) > 0) {
        // Leer desde la tubería
        if (FD_ISSET(pipe_fd, &temp_fds)) {
            num_bytes = read(pipe_fd, buffer, BUFFER_SIZE);
            if (num_bytes <= 0) {
                // Fin de archivo en la tubería, cerrar y volver a abrir
                close(pipe_fd);
                pipe_fd = open("tuberia", O_RDONLY);
                if (pipe_fd == -1) {
                    perror("Error al reabrir la tubería");
                    exit(EXIT_FAILURE);
                }
            } else {
                // Imprimir los datos leídos en el terminal
                 write(STDOUT_FILENO, buffer, num_bytes);
            }
        }

        // Leer desde el terminal
        if (FD_ISSET(STDIN_FILENO, &temp_fds)) {
            num_bytes = read(STDIN_FILENO, buffer, BUFFER_SIZE);
            if (num_bytes > 0) {
```

```
                // Imprimir los datos leídos en el terminal
                write(STDOUT_FILENO, buffer, num_bytes);
            }
        }
    }
}

// Cerrar la tubería
close(pipe_fd);

return 0;
}
```

3read_reopen.c Código fuente del programa con reapertura.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <string.h>

#define BUFFER_SIZE 1024

int main() {
    int pipe_fd;
    char buffer[BUFFER_SIZE];
    int num_bytes;
    fd_set read_fds;
    struct timeval timeout;

    // Abrir la tubería con nombre en modo de lectura
    pipe_fd = open("tuberia", O_RDONLY);
    if (pipe_fd == -1) {
        perror("Error al abrir la tubería");
        exit(EXIT_FAILURE);
    }

    // Configurar el conjunto de descriptores de archivo para select()
    FD_ZERO(&read_fds);
    FD_SET(STDIN_FILENO, &read_fds);
    FD_SET(pipe_fd, &read_fds);

     // Configurar el temporizador para 5 segundos
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
```

```
    // Bucle principal
    while (select(pipe_fd + 1, &read_fds, NULL, NULL, &timeout) > 0) {
        // Leer desde la tubería
        if (FD_ISSET(pipe_fd, &read_fds)) {
            num_bytes = read(pipe_fd, buffer, BUFFER_SIZE);
            if (num_bytes <= 0) {
                // Fin de archivo en la tubería, cerrar y volver a abrir
                 close(pipe_fd);
                pipe_fd = open("tuberia", O_RDONLY);
                if (pipe_fd == -1) {
                    perror("Error al reabrir la tubería");
                    exit(EXIT_FAILURE);
                }
            } else {
                // Imprimir los datos leídos en el terminal
                write(STDOUT_FILENO, buffer, num_bytes);
            }
        }

        // Leer desde el terminal
        if (FD_ISSET(STDIN_FILENO, &read_fds)) {
            num_bytes = read(STDIN_FILENO, buffer, BUFFER_SIZE);
            if (num_bytes > 0) {
                // Imprimir los datos leídos en el terminal
                write(STDOUT_FILENO, buffer, num_bytes);
            }
        }

        // Volver a configurar el conjunto de descriptores de archivo y el temporizador
        FD_ZERO(&read_fds);
        FD_SET(STDIN_FILENO, &read_fds);
        FD_SET(pipe_fd, &read_fds);
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
    }

    // Cerrar la tubería
    close(pipe_fd);

    return 0;
}
```

Abre una terminal y crea la tubería con nombre usando el siguiente comando:

mkfifo tubería

En la misma terminal, compila el programa "3read.c" con el siguiente comando:

gcc 3read.c -o 3read

En la misma terminal, ejecuta el programa "3read" con el siguiente comando:

./3read

Abre otra terminal en el mismo directorio y compila el programa "3read_reopen.c" con el siguiente comando:

gcc 3read_reopen.c -o 3read_reopen

En la segunda terminal, ejecuta el programa "3read_reopen" con el siguiente comando:

./3read_reopen

Ahora, en la segunda terminal, puedes escribir algunos datos en la tubería con nombre usando el siguiente comando:

echo "Hola desde la tubería" > tuberia

Enero 2019-Turno 9:30

1) Ejercicio 1 (1,5 puntos). Configura la topología de red que se muestra en la figura usando vtopol y la configuración proporcionada:
● Configura los interfaces de forma manual, eligiendo adecuadamente sus direcciones IP.
● Configura el encaminamiento y comprueba que todas las máquinas son alcanzables entre sí.
● Configura la traducción de direcciones de red origen considerando que la red de la izquierda es privada y que la red de la derecha es pública.

VM3(Router)

```
[cursoredes@localhost ~]$ sudo -i
[root@localhost ~]# ip lin set eth0 up
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip link set eth1 up
[root@localhost ~]# ip a add 192.168.0.2/24 dev eth0
[root@localhost ~]# ip a add 172.16.0.2/16 dev eth1
[root@localhost ~]# sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

VM1

```
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip a add 192.168.0.1/24 dev eth0
[root@localhost ~]# ip route add default via 192.168.0.2
```

VM2

```
[root@localhost ~]# ip link set eth0 up
[root@localhost ~]# ip a add 172.16.0.1/16 dev eth0
[root@localhost ~]# ip route add default via 172.16.0.2
```

Ping VM2 a VM1(conectan todas las maquinas)

```
[root@localhost ~]# ping 192.168.0.1
```

VM3(Router)

```
[root@localhost ~]# iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

[root@localhost ~]# iptables -t nat -A PREROUTING -d 172.16.0.2 -p tcp --dport 80 -j DNAT --to 192.168.0.2:7777

Ejercicio 2 (1 punto). Escriba un programa que cuente las veces que ha recibido las señales SIGINT y SIGTSTP. El programa principal permanecerá en un bucle que se detendrá cuando se hayan recibido 10 señales en total. El número de señales de cada tipo se mostrará al finalizar el programa.

```
#include <stdio.h>
#include <signal.h>

volatile int sigintCount = 0;
volatile int sigtstpCount = 0;

void handleSigint(int sig) {
   sigintCount++;
}

void handleSigtstp(int sig) {
   sigtstpCount++;
}

int main() {
   struct sigaction sigintAction, sigtstpAction;

   // Configurar el manejador de señal para SIGINT
   sigintAction.sa_handler = handleSigint;
   sigemptyset(&sigintAction.sa_mask);
   sigintAction.sa_flags = 0;
   sigaction(SIGINT, &sigintAction, NULL);

   // Configurar el manejador de señal para SIGTSTP
   sigtstpAction.sa_handler = handleSigtstp;
   sigemptyset(&sigtstpAction.sa_mask);
   sigtstpAction.sa_flags = 0;
   sigaction(SIGTSTP, &sigtstpAction, NULL);

   int totalSignals = 0;
   while (totalSignals < 10) {
      // Esperar por una señal
      pause();

      totalSignals = sigintCount + sigtstpCount;
   }

   printf("Número de señales SIGINT: %d\n", sigintCount);
   printf("Número de señales SIGTSTP: %d\n", sigtstpCount);

   return 0;
}
```

Ejercicio 3 (1,5 puntos). Escriba un programa servidor TCP que escuche en una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor devolverá lo que el cliente le envíe y será capaz de atender a varios clientes a la vez. En cada conexión, el servidor imprimirá la dirección y el puerto del cliente, así como el PID del proceso que la atiende. Finalmente, el servidor gestionará adecuadamente la finalización de los procesos hijo que cree.

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

#define MAX_CLIENTS 10
#define BUFFER_SIZE 1024

void handle_client(int client_socket) {
   char buffer[BUFFER_SIZE];
   ssize_t bytes_received;

   // Obtener información del cliente
   struct sockaddr_in6 client_address;
   socklen_t client_address_len = sizeof(client_address);
   getpeername(client_socket, (struct sockaddr*)&client_address, &client_address_len);

   // Obtener PID del proceso actual
   pid_t pid = getpid();

   // Imprimir información del cliente y PID
   printf("Cliente conectado desde %s:%d (PID: %d)\n",
         inet_ntoa(client_address.sin6_addr), ntohs(client_address.sin6_port), pid);

   while (1) {
      // Recibir datos del cliente
      bytes_received = recv(client_socket, buffer, BUFFER_SIZE - 1, 0);
      if (bytes_received <= 0) {
         break;
      }

      buffer[bytes_received] = '\0';
      printf("Cliente (PID: %d): %s", pid, buffer);

      // Enviar los datos de vuelta al cliente
      send(client_socket, buffer, strlen(buffer), 0);
   }

   // Cerrar el socket del cliente
   close(client_socket);
```

```
    // Imprimir mensaje al finalizar el cliente
    printf("Cliente desde %s:%d (PID: %d) desconectado.\n",
           inet_ntoa(client_address.sin6_addr), ntohs(client_address.sin6_port), pid);

    exit(0);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <dirección> <puerto>\n", argv[0]);
        return 1;
    }

    char *address = argv[1];
    int port = atoi(argv[2]);

    // Crear el socket
    int server_socket = socket(AF_INET6, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("Error al crear el socket");
        return 1;
    }

    // Configurar la dirección y el puerto del servidor
    struct sockaddr_in6 server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin6_family = AF_INET6;
    server_address.sin6_port = htons(port);
    inet_pton(AF_INET6, address, &(server_address.sin6_addr));

    // Vincular el socket a la dirección y puerto del servidor
    if (bind(server_socket, (struct sockaddr*)&server_address, sizeof(server_address)) == -1) {
        perror("Error al vincular el socket");
        return 1;
    }

    // Escuchar en el socket
    if (listen(server_socket, MAX_CLIENTS) == -1) {
        perror("Error al escuchar en el socket");
        return 1;
    }

    printf("Servidor escuchando en %s:%d\n", address, port);

    while (1) {
        // Aceptar la conexión entrante del cliente
        struct sockaddr_in6 client_address;
        socklen_t client_address_len = sizeof(client_address);
```

```
        int client_socket = accept(server_socket, (struct sockaddr*)&client_address,
&client_address_len);
        if (client_socket == -1) {
            perror("Error al aceptar la conexión del cliente");
            continue;
        }

        // Crear un proceso hijo para manejar al cliente
        pid_t pid = fork();
        if (pid == -1) {
            perror("Error al crear el proceso hijo");
            close(client_socket);
            continue;
        } else if (pid == 0) {
            // Proceso hijo
            close(server_socket);
            handle_client(client_socket);
        } else {
            // Proceso padre
            close(client_socket);
        }
    }

    // Cerrar el socket del servidor
    close(server_socket);

    return 0;
}
```

./servidor_tcp <dirección> <puerto>

Enero 2019-Turno 12

Ejercicio 2 (1 punto). Escribe un programa servidor TCP que escuche peticiones realizadas a una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor devolverá la dirección del cliente al recibir cualquier mensaje. En cada conexión, el servidor mostrará en el terminal la dirección y el puerto del cliente.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define MAX_BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <dirección> <puerto>\n", argv[0]);
        return 1;
```

```
    }

    // Obtener la dirección y el puerto de los argumentos de línea de comandos
    const char *address = argv[1];
    int port = atoi(argv[2]);

    // Crear el socket
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket == -1) {
        perror("Error al crear el socket");
        return 1;
    }

    // Configurar la dirección del servidor
    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr(address);
    serverAddress.sin_port = htons(port);

    // Vincular el socket a la dirección y puerto
    if (bind(serverSocket, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) == -1) {
        perror("Error al vincular el socket");
        close(serverSocket);
        return 1;
    }

    // Escuchar nuevas conexiones
    if (listen(serverSocket, 5) == -1) {
        perror("Error al escuchar");
        close(serverSocket);
        return 1;
    }

    printf("Servidor escuchando en %s:%d\n", address, port);

    while (1) {
        // Aceptar una nueva conexión
        struct sockaddr_in clientAddress;
        socklen_t clientAddressLength = sizeof(clientAddress);
        int clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddress,
&clientAddressLength);
        if (clientSocket == -1) {
            perror("Error al aceptar la conexión");
            close(serverSocket);
            return 1;
        }

        // Convertir la dirección del cliente a formato legible
        char clientIP[INET6_ADDRSTRLEN];
```

```
        inet_ntop(AF_INET, &(clientAddress.sin_addr), clientIP, INET6_ADDRSTRLEN);
        int clientPort = ntohs(clientAddress.sin_port);
        printf("Cliente conectado desde %s:%d\n", clientIP, clientPort);

        // Leer y procesar los mensajes del cliente
        char buffer[MAX_BUFFER_SIZE];
        ssize_t bytesRead;
        while ((bytesRead = read(clientSocket, buffer, sizeof(buffer))) > 0) {
            // Enviar la dirección del cliente como respuesta
            if (write(clientSocket, clientIP, strlen(clientIP)) == -1) {
                perror("Error al enviar respuesta");
                close(clientSocket);
                close(serverSocket);
                return 1;
            }
        }

        if (bytesRead == -1) {
            perror("Error al leer del socket");
            close(clientSocket);
            close(serverSocket);
            return 1;
        }

        // Cerrar la conexión con el cliente
        close(clientSocket);
        printf("Conexión cerrada con %s:%d\n", clientIP, clientPort);
    }

    // Cerrar el socket del servidor
    close(serverSocket);

    return 0;
}
```

Ejercicio 3 (1,5 puntos). Escriba un programa que ejecute dos comandos de la siguiente forma:
● Los comandos serán el primer y segundo argumento del programa. El resto de argumentos del programa se considerarán argumentos del segundo comando:
$ ./conecta comando1 comando2 arg2_1 arg2_2 ...
● Cada comando se ejecutará en un proceso distinto, que imprimirá su PID por el terminal.
● El programa conectará la salida estándar del primer proceso con la entrada estándar del segundo, y esperará la finalización de ambos para terminar su ejecución.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Uso: %s comando1 comando2 [argumentos...]\n", argv[0]);
        return 1;
    }

    pid_t pid1, pid2;
    int pipefd[2];

    if (pipe(pipefd) == -1) {
        perror("Error en pipe");
        return 1;
    }

    pid1 = fork();
    if (pid1 < 0) {
        perror("Error en fork");
        return 1;
    } else if (pid1 == 0) {
        // Proceso hijo 1 (comando1)
        close(pipefd[0]);  // Cerramos el extremo de lectura del pipe

        printf("PID del proceso hijo 1 (comando1): %d\n", getpid());

        // Conectamos la salida estándar del proceso hijo 1 al extremo de escritura del pipe
        dup2(pipefd[1], STDOUT_FILENO);

        // Ejecutamos el primer comando
        execvp(argv[1], &argv[1]);

        // Si execvp tiene éxito, el código siguiente no se ejecutará
        perror("Error en execvp del comando1");
        return 1;
    } else {
        // Proceso padre
        pid2 = fork();
        if (pid2 < 0) {
            perror("Error en fork");
            return 1;
        } else if (pid2 == 0) {
            // Proceso hijo 2 (comando2)
            close(pipefd[1]);  // Cerramos el extremo de escritura del pipe

            printf("PID del proceso hijo 2 (comando2): %d\n", getpid());

            // Conectamos la entrada estándar del proceso hijo 2 al extremo de lectura del pipe
            dup2(pipefd[0], STDIN_FILENO);

            // Ejecutamos el segundo comando
```

```
        execvp(argv[2], &argv[2]);

        // Si execvp tiene éxito, el código siguiente no se ejecutará
        perror("Error en execvp del comando2");
        return 1;
    } else {
        // Proceso padre
        close(pipefd[0]);  // Cerramos ambos extremos del pipe en el proceso padre
        close(pipefd[1]);

        // Esperamos la finalización de ambos procesos hijos
        waitpid(pid1, NULL, 0);
        waitpid(pid2, NULL, 0);

        printf("Ambos comandos han finalizado\n");
    }
  }

  return 0;
}
```

Examen 2019

Ejercicio 2 (1 punto). Escribe un programa que lance un demonio del sistema. Recibirá el programa a lanzar como primer argumento y el resto de argumentos se interpretarán como argumentos del programa:

$ ./daemonize comando arg1 arg2 ...

El proceso principal debe crear un proceso y terminar. El nuevo proceso debe crear una nueva sesión, fijar el directorio de trabajo a /tmp, mostrar los identificadores de proceso, de proceso padre, de grupo de procesos y de sesión, y ejecutar el programa con sus argumentos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
  pid_t pid, sid;

  // Verificar que se proporcione al menos un argumento (el programa a lanzar)
  if (argc < 2) {
    fprintf(stderr, "Uso: %s programa [argumentos...]\n", argv[0]);
    exit(EXIT_FAILURE);
  }
```

```
    // Crear un nuevo proceso
    pid = fork();

    // Manejar errores al crear el proceso hijo
    if (pid < 0) {
        fprintf(stderr, "Error al crear el proceso hijo.\n");
        exit(EXIT_FAILURE);
    }

    // Terminar el proceso padre
    if (pid > 0) {
        exit(EXIT_SUCCESS);
    }

    // Crear una nueva sesión
    sid = setsid();
    if (sid < 0) {
        fprintf(stderr, "Error al crear una nueva sesión.\n");
        exit(EXIT_FAILURE);
    }

    // Cambiar el directorio de trabajo a /tmp
    if (chdir("/tmp") < 0) {
        fprintf(stderr, "Error al cambiar el directorio de trabajo.\n");
        exit(EXIT_FAILURE);
    }

    // Cerrar los descriptores de archivo estándar
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    // Mostrar los identificadores de proceso, padre, grupo y sesión
    printf("PID: %d\n", getpid());
    printf("PPID: %d\n", getppid());
    printf("PGID: %d\n", getpgrp());
    printf("SID: %d\n", getsid(0));

    // Ejecutar el programa con sus argumentos
    execvp(argv[1], &argv[1]);

    // Si se ejecuta esta línea, hubo un error al ejecutar el programa
    fprintf(stderr, "Error al ejecutar el programa '%s'.\n", argv[1]);
    exit(EXIT_FAILURE);
}
```

Ejercicio 3 (1,5 puntos). Escribe un programa servidor UDP que escuche peticiones realizadas a una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos y desde el terminal, multiplexando ambos canales. El servidor devolverá la hora al recibir cualquier mensaje. En cada mensaje, el servidor mostrará en el terminal la dirección y el puerto del cliente.

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

#define BUFFER_SIZE 1024

void printClientAddress(struct sockaddr_in clientAddr) {
   char clientIP[INET_ADDRSTRLEN];
   inet_ntop(AF_INET, &(clientAddr.sin_addr), clientIP, INET_ADDRSTRLEN);
   int clientPort = ntohs(clientAddr.sin_port);
   printf("Received message from %s:%d\n", clientIP, clientPort);
}

int main(int argc, char *argv[]) {
   if (argc != 3) {
      fprintf(stderr, "Usage: %s <IP> <port>\n", argv[0]);
      exit(1);
   }

   const char *ipAddress = argv[1];
   int port = atoi(argv[2]);

   // Create socket
   int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
   if (sockfd < 0) {
      perror("Failed to create socket");
      exit(1);
   }

   // Bind to the specified IP and port
   struct sockaddr_in serverAddr;
   memset(&serverAddr, 0, sizeof(serverAddr));
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_addr.s_addr = inet_addr(ipAddress);
   serverAddr.sin_port = htons(port);

   if (bind(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
```

```
        perror("Failed to bind");
        exit(1);
    }

    printf("Server listening on %s:%d\n", ipAddress, port);

    // Receive and respond to messages
    struct sockaddr_in clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);
    char buffer[BUFFER_SIZE];

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);

        // Receive message
        ssize_t numBytes = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
                                    (struct sockaddr *)&clientAddr, &clientAddrLen);
        if (numBytes < 0) {
            perror("Failed to receive message");
            exit(1);
        }

        // Print client address
        printClientAddress(clientAddr);

        // Get current time
        time_t currentTime = time(NULL);
        const char *timeStr = ctime(&currentTime);

        // Send response with current time
        if (sendto(sockfd, timeStr, strlen(timeStr), 0,
                   (struct sockaddr *)&clientAddr, clientAddrLen) < 0) {
            perror("Failed to send response");
            exit(1);
        }
    }

    // Close socket
    close(sockfd);

    return 0;
}
```

2018

Ejercicio 2 (1 punto). Escriba un programa que recorra un directorio, cuya ruta recibe como argumento, y muestre el nombre de los ficheros que contiene, el uid del usuario propietario, número de i-nodo y tamaño.

Nota: Se recomienda hacer este ejercicio en dos partes: primero un programa que muestre el nombre y luego otro que muestre el resto de información.

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <pwd.h>

void mostrarNombresArchivos(const char *ruta);
void mostrarInformacionArchivos(const char *ruta);

int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Uso: %s <ruta>\n", argv[0]);
      return 1;
   }

   const char *ruta = argv[1];

   mostrarNombresArchivos(ruta);
   mostrarInformacionArchivos(ruta);

   return 0;
}

void mostrarNombresArchivos(const char *ruta) {
   DIR *directorio;
   struct dirent *entrada;

   directorio = opendir(ruta);
   if (directorio == NULL) {
      printf("No se pudo abrir el directorio %s\n", ruta);
      exit(1);
   }

   printf("Archivos en %s:\n", ruta);

   while ((entrada = readdir(directorio)) != NULL) {
      if (entrada->d_type == DT_REG) {
         printf("%s\n", entrada->d_name);
      }
   }
```

```
    closedir(directorio);
}

void mostrarInformacionArchivos(const char *ruta) {
    DIR *directorio;
    struct dirent *entrada;
    struct stat informacion;

    directorio = opendir(ruta);
    if (directorio == NULL) {
        printf("No se pudo abrir el directorio %s\n", ruta);
        exit(1);
    }

    printf("\nInformación de archivos en %s:\n", ruta);

    while ((entrada = readdir(directorio)) != NULL) {
        if (entrada->d_type == DT_REG) {
            char archivo_ruta[256];
            sprintf(archivo_ruta, "%s/%s", ruta, entrada->d_name);

            if (stat(archivo_ruta, &informacion) == -1) {
                printf("No se pudo obtener información de %s\n", entrada->d_name);
                continue;
            }

            struct passwd *usuario = getpwuid(informacion.st_uid);

            printf("Archivo: %s\n", entrada->d_name);
            printf("Propietario: %s\n", usuario->pw_name);
            printf("Número de i-nodo: %ld\n", informacion.st_ino);
            printf("Tamaño: %ld bytes\n\n", informacion.st_size);
        }
    }

    closedir(directorio);
}
```

Ejercicio 3 (1,5 puntos). Escriba un programa servidor TCP que escuche en una dirección (IPv4 o IPv6 en cualquier formato) y puerto dados como argumentos. El servidor devolverá lo que el cliente le envíe y será capaz de atender a varios clientes a la vez. En cada conexión, el servidor imprimirá la dirección y el puerto del cliente, así como el PID del proceso que la atiende. Finalmente, el servidor gestionará adecuadamente la finalización de los procesos hijo que cree.

Nota: Se recomienda hacer este ejercicio en tres partes: primero un programa servidor monoproceso, luego modificarlo para que sea multiproceso y, finalmente, gestionar la finalización de los hijos.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFFER_SIZE 1024

// Función para manejar la comunicación con un cliente
void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    // Leer datos del cliente y enviarlos de vuelta
    while ((bytes_read = recv(client_socket, buffer, BUFFER_SIZE, 0)) > 0) {
        if (send(client_socket, buffer, bytes_read, 0) != bytes_read) {
            perror("Error al enviar datos al cliente");
            break;
        }
    }

    // Cerrar el socket del cliente
    close(client_socket);
}

// Función principal del servidor
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Uso: %s <dirección> <puerto>\n", argv[0]);
        return 1;
    }

    const char *address = argv[1];
    int port = atoi(argv[2]);

    // Crear el socket del servidor
    int server_socket = socket(AF_INET6, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Error al crear el socket del servidor");
        return 1;
    }

    // Configurar la dirección del servidor
    struct sockaddr_in6 server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin6_family = AF_INET6;
    server_address.sin6_port = htons(port);
    if (inet_pton(AF_INET6, address, &server_address.sin6_addr) <= 0) {
```

```
        perror("Dirección inválida");
        return 1;
    }

    // Vincular el socket a la dirección del servidor
    if (bind(server_socket, (struct sockaddr *)&server_address, sizeof(server_address)) < 0) {
        perror("Error al vincular el socket a la dirección del servidor");
        return 1;
    }

    // Escuchar en el socket del servidor
    if (listen(server_socket, 5) < 0) {
        perror("Error al escuchar en el socket del servidor");
        return 1;
    }

    printf("Servidor escuchando en %s:%d\n", address, port);

    // Ciclo principal del servidor
    while (1) {
        // Aceptar una nueva conexión de un cliente
        struct sockaddr_in6 client_address;
        socklen_t client_address_length = sizeof(client_address);
        int client_socket = accept(server_socket, (struct sockaddr *)&client_address,
&client_address_length);
        if (client_socket < 0) {
            perror("Error al aceptar la conexión del cliente");
            continue;
        }

        // Crear un proceso hijo para manejar la conexión del cliente
        pid_t child_pid = fork();
        if (child_pid < 0) {
            perror("Error al crear un proceso hijo");
            close(client_socket);
            continue;
        } else if (child_pid == 0) {
            // Proceso hijo
            close(server_socket);

            // Obtener información del cliente
            char client_address_str[INET6_ADDRSTRLEN];
            if (inet_ntop(AF_INET6, &client_address.sin6_addr, client_address_str,
sizeof(client_address_str)) == NULL) {
                perror("Error al obtener la dirección del cliente");
            } else {
                // Imprimir información del cliente y del proceso hijo
                printf("Conexión establecida con %s:%d, PID del proceso hijo: %d\n",
                    client_address_str, ntohs(client_address.sin6_port), getpid());
```

```
        }

        // Manejar la comunicación con el cliente
        handle_client(client_socket);

        // Salir del proceso hijo
        exit(0);
    } else {
        // Proceso padre
        close(client_socket);
    }
  }

  // Cerrar el socket del servidor
  close(server_socket);

  return 0;
}
```