PRÁCTICAS DE APLICACIONES DISTRIBUIDAS EN EQUIPO

FECHA DE ENTREGA: Pendiente

Los equipos deberán están formados de la siguiente manera:

- Tres equipos de 4 integrantes cada uno.

Introducción: Cada práctica con lleva una previa investigación, la cual debe realizar cada equipo.

Cada práctica debe contar con un reporte, que se entregará impreso. Las indicaciones están dadas a lo largo del presente documento.

Práctica 1: Utilizar la terminal para probar los comandos básicos de linux y verificar que se cuente con las librerías necesarias para la clase.

ls Listado de archivos
cp Copia de archivos y direct.
rm Borrar archivos y direct.
mv Mover/renombrar arch. y dir.
grep Filtra líneas de un archivo por expresiones regulares
ps Muestra un listado con el estado de los procesos
kill Mata un proceso
fg Pasa un proceso a primer plano
bg Pasa un proceso a segundo plano
ifconfig Configura interfaces de red
ping Comprueba el estado de la Redes
find Busca archivos en el sistema de archivos
locate Busca archivos en una DB de archivos del sistema
su permite abrir una sesión con el ID de un otro usuario
sudo Ejecuta un comando con el ID de otro usuario
who Muestra los usuarios que están conectados
w Muestra los usuarios que estan conectados y que están haciendo
write Comando para comunicarse con otros usuarios
mesg Controla recepción de mensajes en nuestra terminal
netstat Muestra conexiones de red
traceroute Traza el camino que sigue un paquete hasta llegar a su destino, mencionando los routers por los que va pasando.
nmap Escaner de red

Nota. Buscar la librería socket.h en el sistema, la cual es la principal para los programas.

-Realizar múltiples usos de los comandos, como comunicación entre ellos y su forma de terminarlos.

n > archivo	redirecciona la salida desde el descriptor del archivo n a un archivo. Si el archivo no existe, éste es creado. Si ya existe, los contenidos existentes se pierden sin previo aviso.
n >> archivo	redirecciona la salida desde el descriptor del archivo n a un archivo. Si el archivo no existe, éste es creado. Si existe, la salida se agrega al archivo existente.
c1 | c2	Dirige el stdout del primer comando al stdin del segundo. Podrá construir una tubería más larga agregando más comandos y más operadores |.

	Cualquiera de los comandos puede tener opciones o argumentos.
c1 &	Lanza proceso en segundo plano

Práctica 2: Aprender el uso del compilador de C en el servidor “GCC”.

Una vez comprobado que se cuenta con las librerías necesarias deberán realizar un programa en el lenguaje C, el cual deberá manipular el contenido de archivos.

Ejemplo:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

void contenido(void){
FILE *archivo;
char *temp;
int nchar=0;
int nren=0;
clrscr();
archivo = fopen("archivo.txt","r+");
if(archivo)
printf("archivo abierto\n");
printf("\ncontenido:\n\n");
while(fgets(temp,2,archivo)){
printf("%s",temp);
nchar++;
if(temp[0]==10)
nren++;
}
printf("\n\nnumero de caracteres: %d\n",nchar);
printf("numero de renglones : %d",nren+1);
fclose(archivo);
getch();
}
void acaracter(void){
int pos;
char car;
FILE *archivo;
printf("\n\ncaptura caracter\n");
car=getch();
printf("posicion del caracter\n");
scanf("%d",&pos);
archivo=fopen("archivo.txt","r+");
fseek(archivo,pos,SEEK_CUR);
fprintf(archivo,"%c",car);
fclose(archivo);

printf("caracter capturado");
}
```

```
void arenglon(void){
char *cadena;
FILE *archivo;
archivo = fopen("archivo.txt","r+");
fseek(archivo,0,SEEK_END);
printf("\n\ncaptura cadena\n");
gets(cadena);
fprintf(archivo,"\n%s",cadena);
close(archivo);
printf("cadena capturada");
}
void main(){
char op;
contenido();
printf("\n\n1: agregar caracter\n2: agregar renglon");
op=getch();
switch(op){
case '1': acaracter();
break;
case '2': arenglon();
break;
}
getch();
}
```

Nota. Documentar los problemas encontrados al compilar y la forma correcta de hacerlo, así como la forma de ejecución en sistema linux.

Práctica 3: En esta práctica el alumno debe compilar para probar el uso de semáforos, colas.

```
//sem1.c
#include <iostream.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
// Esta union hay que definirla o no según el valor de los defines aqui indicados.
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
// La union ya está definida en sys/sem.h
#else
// Tenemos que definir la union
union semun
{
int val;
struct semid_ds *buf;
unsigned short int *array;
struct seminfo *__buf;
};
#endif
main()
{
key_t Clave;
```

```
int Id_Semaforo;
struct sembuf Operacion;
union semun arg;
int i=0;
// Igual que en cualquier recurso compartido (memoria compartida, semaforos o colas) se obtiene una clave
a partir de un fichero existente cualquiera

// y de un entero cualquiera. Todos los procesos que quieran compartir este // semaforo, deben usar el
mismo fichero y el mismo entero.
Clave = ftok ("/bin/ls", 33);
if (Clave == (key_t)-1)
{
cout << "No puedo conseguir clave de semaforo" << endl;
exit(0);
}
// Se obtiene un array de semaforos (10 en este caso, aunque solo se usara uno.
// El IPC_CREAT indica que lo cree si no lo está ya el 0600 con permisos de lectura y escritura para el usuario
que lance
// los procesos. Es importante el 0 delante para que se interprete en octal.
Id_Semaforo = semget (Clave, 10, 0600 | IPC_CREAT);
if (Id_Semaforo == -1)
{
cout << "No puedo crear semaforo" << endl;
exit (0);
}
// Se inicializa el semáforo con un valor conocido. Si lo ponemos a 0, es semáforo estará "rojo". Si lo
ponemos a 1, estará "verde".
// El 0 de la función semctl es el índice del semáforo que queremos inicializar dentro del array de 10 que
hemos pedido.
arg.val = 0;
semctl (Id_Semaforo, 0, SETVAL, &arg);
// Para "pasar" por el semáforo parándonos si está "rojo", debemos rellenar esta estructura.
// sem_num es el indice del semáforo en el array por el que queremos "pasar"
// sem_op es -1 para hacer que el proceso espere al semáforo.
// sem_flg son flags de operación. De momento nos vale un 0.
Operacion.sem_num = 0;
Operacion.sem_op = -1;
Operacion.sem_flg = 0;
// Bucle infinito indicando cuando entramos al semáforo y cuándo salimos de él.
// i hace de contador del número de veces que hemos salido del semáforo.
while (1) {
cout << i << " Esperando Semaforo" << endl;

// Se hace la espera en el semáforo. Se le pasa un array de operaciones y el número de elementos
en dicho array.
// En nuestro caso solo 1.
semop (Id_Semaforo, &Operacion, 1);
cout << i << " Salgo de Semaforo " << endl;
cout << endl;
i++;
}
}
```

```
//sem2.c
#include <iostream.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
#include <unistd.h>
// Esta union hay que definirla o no según el valor de los defines aqui indicados.
//
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
// La union ya está definida en sys/sem.h
#else
// Tenemos que definir la union
union semun
{
int val;
struct semid_ds *buf;
unsigned short int *array;
struct seminfo *__buf;
};
#endif
main()
{
key_t Clave;
int Id_Semaforo;
struct sembuf Operacion;
union semun arg;
int i;
// Igual que en cualquier recurso compartido (memoria compartida, semaforos o colas) se obtien
una clave a partir de un fichero existente
// cualquieray de un entero cualquiera. Todos los procesos que quieran compartir este
// semaforo, deben usar el mismo fichero y el mismo entero.
Clave = ftok ("/bin/ls", 33);
if (Clave == (key_t)-1)
{
cout << "No puedo conseguir clave de semáforo" << endl;
exit(0);
}
// Se obtiene un array de semaforos (10 en este caso, aunque solo se usara
// uno.
// El IPC_CREAT indica que lo cree si no lo está ya
// el 0600 con permisos de lectura y escritura para el usuario que lance
// los procesos. Es importante el 0 delante para que se interprete en
// octal.
Id_Semaforo = semget (Clave, 10, 0600 | IPC_CREAT);
if (Id_Semaforo == -1)
{
cout << "No puedo crear semáforo" << endl;
exit (0);
}
// Se levanta el semáforo. Para ello se prepara una estructura en la que
// sem_num indica el indice del semaforo que queremos levantar en el array
// de semaforos obtenido.
```

```
// El 1 indica que se levanta el semaforo
// El sem_flg son banderas para operaciones raras. Con un 0 vale.
Operacion.sem_num = 0;
Operacion.sem_op = 1;

Operacion.sem_flg = 0;
// Vamos a levantar el semáforo 10 veces esperando 1 segundo cada vez.
for (i = 0; i<10; i++)
{
cout << "Levanto Semáforo" << endl;
// Se realiza la operación de levantar el semáforo. Se pasa un array
// de Operacion y el número de elementos en el array de Operacion. En
// nuestro caso solo 1.
semop (Id_Semaforo, &Operacion, 1);
sleep (1);
}
}
```

```
// Programa para demostración del uso de colas de mensajes 1
#include <iostream.h>
#include <sys/msg.h>
#include <errno.h>
// Estructura para los mensajes que se quieren enviar y/o recibir. Deben llevar
// obligatoriamente como primer campo un long para indicar un identificador
// del mensaje.
// Los siguientes campos son la información que se quiera transmitir en el
// mensaje. Cuando más adelante, en el código, hagamos un cast a
// (struct msgbuf *), todos los campos de datos los verá el sistema como
// un único (char *)
typedef struct Mi_Tipo_Mensaje
{
long Id_Mensaje;
int Dato_Numerico;
char Mensaje[10];
};
main()
{
key_t Clave1;
int Id_Cola_Mensajes;
Mi_Tipo_Mensaje Un_Mensaje;
// Igual que en cualquier recurso compartido (memoria compartida, semaforos
// o colas) se obtien una clave a partir de un fichero existente cualquiera
// y de un entero cualquiera. Todos los procesos que quieran compartir este
// semaforo, deben usar el mismo fichero y el mismo entero.
Clave1 = ftok ("/bin/ls", 33);
if (Clave1 == (key_t)-1)
{
cout << "Error al obtener clave para cola mensajes" << endl;
exit(-1);
}
```

```
// Se crea la cola de mensajes y se obtiene un identificador para ella.
// El IPC_CREAT indica que cree la cola de mensajes si no lo está ya.
// el 0600 son permisos de lectura y escritura para el usuario que lance
// los procesos. Es importante el 0 delante para que se interprete en
// octal.
Id_Cola_Mensajes = msgget (Clave1, 0600 | IPC_CREAT);
if (Id_Cola_Mensajes == -1)
{
cout << "Error al obtener identificador para cola mensajes" << endl;
exit (-1);
}
// Se rellenan los campos del mensaje que se quiere enviar.
// El Id_Mensaje es un identificador del tipo de mensaje. Luego se podrá
// recoger aquellos mensajes de tipo 1, de tipo 2, etc.
// Dato_Numerico es un dato que se quiera pasar al otro proceso. Se pone,
// por ejemplo 29.
// Mensaje es un texto que se quiera pasar al otro proceso.
Un_Mensaje.Id_Mensaje = 1;
Un_Mensaje.Dato_Numerico = 29;
strcpy (Un_Mensaje.Mensaje, "Hola");
// Se envia el mensaje. Los parámetros son:
// - Id de la cola de mensajes.
// - Dirección al mensaje, convirtiéndola en puntero a (struct msgbuf *)
// - Tamaño total de los campos de datos de nuestro mensaje, es decir
// de Dato_Numerico y de Mensaje
// - Unos flags. IPC_NOWAIT indica que si el mensaje no se puede enviar
// (habitualmente porque la cola de mensajes esta llena), que no espere
// y de un error. Si no se pone este flag, el programa queda bloqueado
// hasta que se pueda enviar el mensaje.
msgsnd (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,
sizeof(Un_Mensaje.Dato_Numerico)+sizeof(Un_Mensaje.Mensaje),
IPC_NOWAIT);
// Se recibe un mensaje del otro proceso. Los parámetros son:
// - Id de la cola de mensajes.
// - Dirección del sitio en el que queremos recibir el mensaje,
// convirtiéndolo en puntero a (struct msgbuf *).
// - Tamaño máximo de nuestros campos de datos.
// - Identificador del tipo de mensaje que queremos recibir. En este caso
// se quiere un mensaje de tipo 2. Si ponemos tipo 1, se extrae el mensaje
// que se acaba de enviar en la llamada anterior a msgsnd().
// - flags. En este caso se quiere que el programa quede bloqueado hasta
// que llegue un mensaje de tipo 2. Si se pone IPC_NOWAIT, se devolvería
// un error en caso de que no haya mensaje de tipo 2 y el programa
// continuaría ejecutándose.
msgrcv (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,
sizeof(Un_Mensaje.Dato_Numerico) + sizeof(Un_Mensaje.Mensaje),
2, 0);
cout << "Recibido mensaje tipo 2" << endl;
cout << "Dato_Numerico = " << Un_Mensaje.Dato_Numerico << endl;
cout << "Mensaje = " << Un_Mensaje.Mensaje << endl;
// Se borra y cierra la cola de mensajes.
```

```
// IPC_RMID indica que se quiere borrar. El puntero del
final son datos
// que se quieran pasar para otros comandos. IPC_RMID
no necesita datos,
// así que se pasa un puntero a NULL.
msgctl (Id_Cola_Mensajes, IPC_RMID, (struct msqid_ds
*)NULL);
}
// Programa para demostración del uso de colas de
mensajes 2
//
#include <iostream.h>
#include <sys/msg.h>
#include <errno.h>
//
// Estructura para los mensajes que se quieren enviar y/o
recibir. Deben llevar
// obligatoriamente como primer campo un long para
indicar un identificador
// del mensaje.
// Los siguientes campos son la información que se
quiera transmitir en el
// mensaje. Cuando más adelante, en el código, hagamos
un cast a
// (struct msgbuf *), todos los campos de datos los verá
el sistema como
// un único (char *)
//
typedef struct Mi_Tipo_Mensaje
{
long Id_Mensaje;
int Dato_Numerico;
char Mensaje[10];
};
main()
{
key_t Clave1;
int Id_Cola_Mensajes;
Mi_Tipo_Mensaje Un_Mensaje;
//
// Igual que en cualquier recurso compartido (memoria
compartida, semaforos
// o colas) se obtien una clave a partir de un fichero
existente cualquiera
// y de un entero cualquiera. Todos los procesos que
quieran compartir este
// semaforo, deben usar el mismo fichero y el mismo
entero.
//
Clave1 = ftok ("/bin/ls", 33);
if (Clave1 == (key_t)-1)
{
cout << "Error al obtener clave para cola mensajes" <<
```

```
endl;
exit(-1);
}
//
// Se crea la cola de mensajes y se obtiene un
identificador para ella.
// El IPC_CREAT indica que cree la cola de mensajes si no
lo está ya.
// el 0600 son permisos de lectura y escritura para el
usuario que lance
// los procesos. Es importante el 0 delante para que se
interprete en
// octal.
//
Id_Cola_Mensajes = msgget (Clave1, 0600 | IPC_CREAT);
if (Id_Cola_Mensajes == -1)
{
cout << "Error al obtener identificador para cola
mensajes" << endl;
exit (-1);
}
//
// Se recibe un mensaje del otro proceso. Los parámetros
son:
// - Id de la cola de mensajes.
// - Dirección del sitio en el que queremos recibir el
mensaje,
// convirtiéndolo en puntero a (struct msgbuf *).
// - Tamaño máximo de nuestros campos de datos.
// - Identificador del tipo de mensaje que queremos
recibir. En este caso
// se quiere un mensaje de tipo 1, que es el que envia el
proceso cola1.cc
// - flags. En este caso se quiere que el programa quede
bloqueado hasta
// que llegue un mensaje de tipo 1. Si se pone
IPC_NOWAIT, se devolvería
// un error en caso de que no haya mensaje de tipo 1 y el
programa
// continuaría ejecutándose.
//
msgrcv (Id_Cola_Mensajes, (struct msgbuf
*)&Un_Mensaje,
sizeof(Un_Mensaje.Dato_Numerico) +
sizeof(Un_Mensaje.Mensaje),
1, 0);
cout << "Recibido mensaje tipo 1" << endl;
cout << "Dato_Numerico = " <<
Un_Mensaje.Dato_Numerico << endl;
cout << "Mensaje = " << Un_Mensaje.Mensaje << endl;
//
// Se rellenan los campos del mensaje que se quiere
enviar.
```

```
// El Id_Mensaje es un identificador del tipo de mensaje.
Luego se podrá
// recoger aquellos mensajes de tipo 1, de tipo 2, etc.
// Dato_Numerico es un dato que se quiera pasar al otro
proceso. Se pone,
// por ejemplo 13.
// Mensaje es un texto que se quiera pasar al otro
proceso.
//
Un_Mensaje.Id_Mensaje = 2;
Un_Mensaje.Dato_Numerico = 13;
strcpy (Un_Mensaje.Mensaje, "Adios");
//
// Se envia el mensaje. Los parámetros son:
// - Id de la cola de mensajes.
// - Dirección al mensaje, convirtiéndola en puntero a
(struct msgbuf *)
// - Tamaño total de los campos de datos de nuestro
mensaje, es decir
// de Dato_Numerico y de Mensaje
// - Unos flags. IPC_NOWAIT indica que si el mensaje no
se puede enviar
// (habitualmente porque la cola de mensajes esta llena),
que no espere
// y de un error. Si no se pone este flag, el programa
queda bloqueado
// hasta que se pueda enviar el mensaje.
//
msgsnd (Id_Cola_Mensajes, (struct msgbuf
*)&Un_Mensaje,
sizeof(Un_Mensaje.Dato_Numerico)+sizeof(Un_Mensaje.
Mensaje),
IPC_NOWAIT);
}

//
// Programa para demostración de memoria compartida.
1
//
#include <sys/shm.h>
#include <iostream.h>
#include <unistd.h>
#include <stdio.h>
void main()
{
key_t Clave;
int Id_Memoria;
int *Memoria = NULL;
int i,j;
//
// Conseguimos una clave para la memoria compartida.
Todos los
// procesos que quieran compartir la memoria, deben
```

```
obtener la misma
// clave. Esta se puede conseguir por medio de la funcion
ftok.
// A esta funcion se le pasa un fichero cualquiera que
exista y este
// accesible (todos los procesos deben pasar el mismo
fichero) y un
// entero cualquiera (todos los procesos el mismo
entero).
//
Clave = ftok ("/bin/ls", 33);
if (Clave == -1)
{
printf("No consigo clave para memoria compartida");
exit(0);
}
//
// Creamos la memoria con la clave recien conseguida.
Para ello llamamos
// a la funcion shmget pasandole la clave, el tamano de
memoria que
// queremos reservar (100 enteros en nuestro caso) y
unos flags.
// Los flags son los permisos de
lectura/escritura/ejecucion
// para propietario, grupo y otros (es el 777 en octal) y el
// flag IPC_CREAT para indicar que cree la memoria.
// La funcion nos devuelve un identificador para la
memoria recien
// creada.
//
Id_Memoria = shmget (Clave, sizeof(int)*100, 0777 |
IPC_CREAT);
if (Id_Memoria == -1)
{
printf("No consigo Id para memoria compartida");
exit (0);
}
//
// Una vez creada la memoria, hacemos que uno de
nuestros punteros
// apunte a la zona de memoria recien creada. Para ello
llamamos a
// shmat, pasandole el identificador obtenido
anteriormente y un
```

```
    // par de parametros extranos, que con ceros vale.
    //
    Memoria = (int *)shmat (Id_Memoria, (char *)0, 0);
    if (Memoria == NULL)
    {
    printf("No consigo memoria compartida");
    exit (0);
    }
    //
    // Ya podemos utilizar la memoria.
    // Escribimos cosas en la memoria. Los numeros de 1 a 10
    esperando
    // un segundo entre ellos. Estos datos seran los que lea el
    otro
    // proceso.
    //
    for (i=0; i<10; i++)
    {
    for (j=0; j<100; j++)
    {
    Memoria[j] = i;
    }
    printf("Escrito %d",i);
    sleep (1);
    }
    //
    // Terminada de usar la memoria compartida, la liberamos.
    //
    shmdt ((char *)Memoria);
    shmctl (Id_Memoria, IPC_RMID, (struct shmid_ds *)NULL);
    }
    //
// Programa de prueba para la memoria compartida. 2
//
#include <sys/shm.h>
#include <iostream.h>
#include <unistd.h>
void main()
{
key_t Clave;
int Id_Memoria;
int *Memoria = NULL;
int i,j;
//
// Igual que en p1.cc, obtenemos una clave para la memoria compartida
//
Clave = ftok ("/bin/ls", 33);
if (Clave == -1)
{
cout << "No consigo clave para memoria compartida" << endl;
exit(0);
}
//
```

```
// Igual que en p1.cc, obtenemos el id de la memoria. Al no poner
// el flag IPC_CREAT, estamos suponiendo que dicha memoria ya está
// creada.
//
Id_Memoria = shmget (Clave, sizeof(int)*100, 0777 );
if (Id_Memoria == -1)
{
cout << "No consigo Id para memoria compartida" << endl;
exit (0);
}
//
// Igual que en p1.cc, obtenemos un puntero a la memoria compartida
//
Memoria = (int *)shmat (Id_Memoria, (char *)0, 0);
if (Memoria == NULL)
{
cout << "No consigo memoria compartida" << endl;
exit (0);
}
//
// Vamos leyendo el valor de la memoria con esperas de un segundo
// y mostramos en pantalla dicho valor. Debería ir cambiando según
// p1 lo va modificando.
//
for (i=0; i<10; i++)
{
cout << "Leido " << Memoria[0] << endl;
sleep (1);
}
//
// Desasociamos nuestro puntero de la memoria compartida. Suponemos
// que p1 (el proceso que la ha creado), la liberará.
//
if (Id_Memoria != -1)
{
shmdt ((char *)Memoria);
}
}
```

Nota. Documentar y graficar la forma de trabajo de los programas

Práctica 3: Programación cliente-servidor sobre TCP | UDP

Con el código visto en clases sobre servidores TCP – UDP se deberá pasar a su cuenta y compilar para verificar su funcionamiento.

```
//Servidor
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define STDOUT 1
#define SERV_ADDR 1234
void main()
```

```
{
int rval;
int sock,length, msgsock;
struct sockaddr_in server;
char buf[1024];
sock=socket(PF_INET, SOCK_STREAM, 0);
if(sock <0)
{
  perror("NO hay socket de escucha");
  exit(1);
  }
  server.sin_family=AF_INET;
  server.sin_addr.s_addr=htonl(INADDR_ANY);
  server.sin_port=htons(SERV_ADDR);
  if(bind(sock,(struct sockaddr *)&server,sizeof server)<0)
  {
  perror("direccion no asignada");
  exit(1);
  }
  listen(sock,1);
  do{
  msgsock=accept(sock, (struct sockaddr *)0,(int *) 0);
  if (msgsock ==-1){
  perror("Conexion no aceptada!!!!!!!!!!!!!!\n");
  }else do{
  memset(buf,0,sizeof buf);
  rval=read(msgsock, buf,1024);
  if (rval<0)perror("Mensaje no leido..");
  else write(STDOUT,buf,rval);
  }while(rval>0);
  printf("\nCerrando la conexion.....\n");
  close(msgsock);
  }while(1);
  exit(0);
  }

  //cliente
  #include <sys/types.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SERV_ADDR 1981
//(IPPORT_RESERVED+1)
#define DATA "##--##--##----***----##--##--##"
void main(int argc, char** argv)
{
int sock;
struct sockaddr_in server;
struct hostent *hp, *gethostbyname();
if(argc < 2) {
```

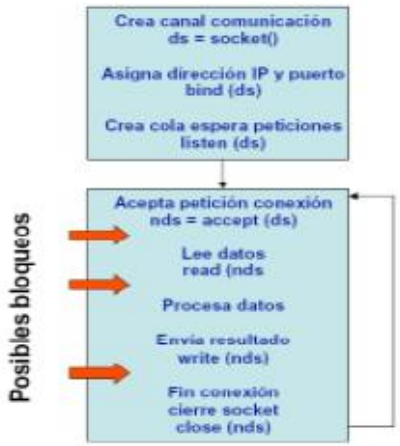
```
printf("Uso: %s nombre_host\n",argv[0]);
exit(1);
}
sock= socket (PF_INET, SOCK_STREAM,0);
if(sock < 0){
perror("No se ha podido conseguir un socket :(");
exit(1);
}
server.sin_family=AF_INET;
hp=gethostbyname(argv[1]);
if(hp==0){
fprintf(stderr, "%s: No conozco a esa compu\n",argv[1]);
exit(2);
}
memcpy((char *)&server.sin_addr, (char *)hp->h_addr,hp->h_length);
server.sin_port=htons(SERV_ADDR);
if (connect(sock, (struct sockaddr *)&server,sizeof (server))<0){
perror("Conexion no aceptada!!!!");
exit(1);
}
if (write(sock, DATA,strlen(DATA)+1)<0)
perror("No he podido escribir el mensaje");
close(sock);
exit(0);
}
```

Utilizar el comando “nc” en apoyo para comprobar el funcionamiento de su programa servidor.

Nc 127.0.0.1 1234 \\ se conectara al servidor que está corriendo en la maquina del usuario local por el puerto especificado “1234” en el programa compilado por el alumno.

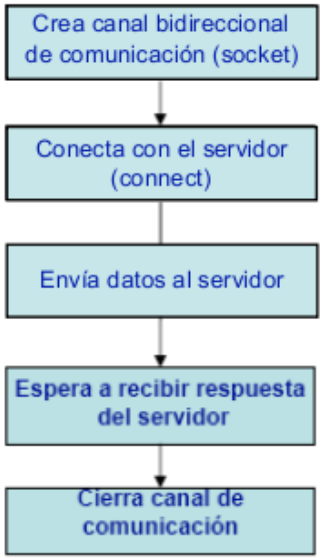
Documentar la utilidad del comando nc y documentación del código de su servidor así como un diagrama de flujo.

Diagrama del servidor:



- `socket()` : crea el canal
- `bind()` : asocia al canal una dirección IP y un puerto
- `listen()` : crea la cola
- `accept()` : acepta una petición y completa la conexión
- `read()` : lee datos del socket
- `write()` : envía datos al cliente a través del socket
- `close()` : cierra el socket
- Dos descriptores de socket:
  - ds: socket de escucha (cola)
  - nds: socket de conexión

Diagrama del cliente:



- `socket()` : crea el canal
- `connect()` : solicita la conexión al servidor
- `write()`: envía datos al servidor a través del socket
- `read()`: lee datos del socket
- `close()`: cierra el socket

Práctica 4: Cliente-Servidor Fork

El alumno deberá modificar su cliente-servidor y agregarle el uso de procesos hijos para ver las ventajas de ellos.

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <stdio.h>
/*
* Programa principal.
* Crea un proceso hijo.
* El proceso hijo escribe su id en pantalla, espera 5 segundos y sale con un
* exit (33).
* El proceso padre espera un segundo, escribe su id, el de su hijo y espera
* que el hijo termine. Escribe en pantalla el valor de exit del hijo.
*/
main()
{
/* Identificador del proceso creado */
pid_t idProceso;
/* Variable para comprobar que se copia inicialmente en cada proceso y que
* luego puede cambiarse independientemente en cada uno de ellos. */
int variable = 1;
/* Estado devuelto por el hijo */
int estadoHijo;
/* Se crea el proceso hijo. En algún sitio dentro del fork(), nuestro
* programa se duplica en dos procesos. Cada proceso obtendrá una salida
* distinta. */
idProceso = fork();
/* Si fork() devuelve -1, es que hay un error y no se ha podido crear el
```

```
* proceso hijo. */
if (idProceso == -1)

{
perror ("No se puede crear proceso");
exit (-1);
}
/* fork() devuelve 0 al proceso hijo.*/
if (idProceso == 0)
{
/* El hijo escribe su pid en pantalla y el valor de variable */
printf ("Hijo : Mi pid es %d. El pid de mi padre es %d\n", getpid(), getppid());
/* Escribe valor de variable y la cambia */
printf ("Hijo : variable = %d. La cambio por variable = 2\n", variable);
variable = 2;
/* Espera 5 segundos, saca en pantalla el valor de variable y sale */
sleep (5);
printf ("Hijo : variable = %d y salgo\n", variable);
exit (33);
}
/* fork() devuelve un número positivo al padre. Este número es el id del
* hijo. */
if (idProceso > 0)
{ /* Espera un segundo (para dar tiempo al hijo a hacer sus cosas y no entremezclar salida en la pantalla) y
escribe su pid y el de su hijo */
sleep (1);
printf ("Padre : Mi pid es %d. El pid de mi hijo es %d\n", getpid(), idProceso);
/* Espera que el hijo muera */
wait (&estadoHijo);
/* Comprueba la salida del hijo */

if (WIFEXITED(estadoHijo) != 0)
{
printf ("Padre : Mi hijo ha salido. Devuelve %d\n", WEXITSTATUS(estadoHijo));
}
/* Escribe el valor de variable, que mantiene su valor original */
printf ("Padre : variable = %d\n", variable);
}
}
```

Nota. Documentar y realizar diagrama de como se realizaron las modificaciones.

Práctica 5:Cliente-Servidor Hilos

El alumno deberá modificar su cliente-servidor y agregarle el uso de hilos para ver las ventajas de ellos.

```
/*
*
* Programa de ejemplo de threads.
*
* Un único contador y dos threads para modificarlo. Uno lo incrementa y pone
* su valor en pantalla precedido de la palabra "Padre". El otro lo
* decrementa y pone su valor en pantalla precedido de la palabra "Hilo".
* Vemos en pantalla como el contador se incrementa y se decrementa
* rápidamente.
*/
#include <pthread.h>
/* Prototipo de la función que va a ejecutar el thread hijo */
void *funcionThread (void *parametro);
/* Contador, global para que sea visible desde el main y desde funcionThread */

int contador = 0;
/*
* Principal
* Lanza un thread para la función funcionThread.
* Después de comprobar el posible error, se mete en un bucle infinito
* incrementando y mostrando el contador.
*/
main()
{
/* Identificador del thread hijo */
pthread_t idHilo;
/* error devuelto por la función de creación del thread */
int error;
/* Creamos el thread.
* En idHilo nos devuelve un identificador para el nuevo thread,
* Pasamos atributos del nuevo thread NULL para que los coja por defecto
* Pasamos la función que se ejecutará en el nuevo hilo
* Pasamos NULL como parámetro para esa función. */
error = pthread_create (&idHilo, NULL, funcionThread, NULL);
/* Comprobamos el error al arrancar el thread */
if (error != 0)
{
perror ("No puedo crear thread");
exit (-1);
}
/* Bucle infinito para incrementar el contador y mostrarlo en pantalla */

while (1)
{
contador++;
printf ("Padre : %d\n", contador);
}
}
```

```
/* Funcion que se ejecuta en el thread hijo.*/
void *funcionThread (void *parametro)
{
/* Bucle infinito para decrementar contador y mostrarlo en pantalla. */
while (1)
{
contador--;
printf ("Hilo : %d\n", contador);
}
}
```

Nota. Documentar y realizar diagrama de como se realizaron las modificaciones.

Práctica 6: Comunicarse con otros servidores

El alumno deberá probar su sistema cliente-servidor con los de sus compañeros, realizando conexiones y recibiendo en su propio servidor.

Al aceptar una conexión deberá mostrar el ip de quien realizo la conexión.