```
ls -d */
export SLEEP_SECS=20 //acceso a variables del sistema
```

```
#define errnoexit() do{ perror("Error"); exit(EXIT_FAILURE);} while(0);
#define KO -1
#define OK 0
#define MAX_SIZE 100
```

Práctica 2.1: Introducción

Actividad 16: strftime (3)

```
char buff [100];
time_t t;
1. time(&t);
2. struct tm *tm = localtime(&t);
3. strftime (buff, 100, "%A, %d de %B de %y, %H: %M", tm);
printf ("%s\n", buff);
```

1. accedemos al tiempo del sistema con la función time, nos devuelve por referencia el tiempo desde Epoch
2. Asignando el tiempo en la estructura tm, con localtime
3. Formateando la fecha como se le indica por parametro

Practica 2.2: Sistema de ficheros

Actividad 11: extraer una parte del nombre

```
1. char *extractname (char* path) {
   2.  char *file_name = basename (&path[0]);
   3.  char *token = strtok (file_name, ".");
4. return &token [0];
}
```

1. pasamos el puntero por parámetro, path = "<~/nombre. extension>"
2.basename extrae del path el ultimo string después del último "/": file_name = "nombre. extension"
3.strtok hace un Split por "." y nos devuelve un array: token = "nombre, extension"
4. Nos quedamos con la primera posición del array que contiene el nombre.

Actividad 11: hacer una copia de un puntero

```
1. char *name = malloc(sizeof(char) * strlen (argv [1]) + 1);
2. strcat (name, argv [1]);
3. strcat (name, ". extension");
```

1.Creamos un montículo de memoria para la nueva variable, es decir generamos un nuevo puntero de memoria.
2.Copiamos el string en la variable creada, esto nos permite cualquier modificación de la nueva variable sin modificar o dañar la variable original.
3.Añadimos una extensión.
arg [1] = "hola" name = "hola. extension"

Actividad 15: crear un cerrojo

```
int fd;
struct flock flk;

1.
if ((fd = open (argv [1], O_CREAT | O_APPEND | O_RDWR, S_IRWXU)) == KO) {…}

2.
flk.l_type = F_WRLCK;
flk.l_whence = SEEK_CUR;
flk.l_start = 0;
flk.l_len = MAX_SIZE;

3.
if (fcntl(fd, F_SETFL, &flk) == KO) {
   if (errno == EACCESS | errno == EAGAIN) {
        printf ("Ocupado...");
        exit;
   }
   else {
     exit;
   }
}
else {
4.
char buff[MAX_SIZE] = "Estoy escribiendo";
write(fd, buff, MAX_SIZE);
5.
flk.l_type = F_UNLCK;
flk.l_whence = SEEK_CUR;
flk.l_start = 0;
flk.l_len = MAX_SIZE;

6.
if (fcntl(fd, F_SETFL, &flk) == KO) {
    exit;
}
```

1. Abrir el archivo que deseamos bloquear con permisos de escritura, lectura y con el modo de S_IRWXU.
2. creamos una structura con esas caracterisiticas en caso de que vayamos a escribir en una porcion del fichero.
3. Comprobamos que el cerrojo es correcto y podemos escribir, si no informamos al usuario.
4. Creamos el string que vamos a escribir en el caso de que se pueda.
5. se libera el cerrojo con las caracteristicas anteriores.
6. se asigna el cerrojo de nuevo al fichero.

```
exit;
}
```

Práctica 2.3: Procesos

Actividad 6: como crear un demonio

```
pid_t pid, sid;

1. pid = fork();
switch (pid) {
   case -1: exit;
   case 0: //hijo
      2.umask(0);
      3.sid = getsid(pid);
      4.if (chdir("/temp") == KO) {}
       break;
   default: //padre
       wait(pid);
}
```

1. Hacer un fork y comprobar los 3 estados
2. En el caso del hijo, cambiar la máscara del proceso
3. restablecer el identificador de sesión
4. cambiar el directorio de trabajo

Actividad 11: manejador de señales bloqueadas

```
volatile int variable;
void handler (int signal) {
  if (signal == SIGINT || signal == SIGTSTP) {
     //do something
     variable = 1234;
  }
}
1.
struct sigaction act;
act.sa_handler = handler;
act.sa_flags = SA_SIGINFO;
2.
sigset_t set;
sigemptyset(&set);
sigaddset (&set, SIGINT);
sigaddset (&set, SIGTSTP);
3.
if (sisgprocmask (SIG_BLOCK, &set, NULL) == KO)
{exit;}
4.
if (sigaction (SIGINT, &act, NULL) == KO) {exit}
if (sigaction (SIGTSTP, &act, NULL) == KO) {exit}

//do something
5.
if (sigprocmask (SIG_UNBLOCK, &set, NULL) == KO)
{exit;}
```

1. Rellenar el struct sigaction con el puntero a la función y el flag SA_SIGINFO
2.Inicializar y rellenar la estructura sigset_t con las señales a capturar
3. Enmascarar las señales a capturar con la estructura sigset_t
4. Asociar las señales a la función que va recibir y ejecutar una acción
5.Des enmascarar las señales que se han capturado

Actividad 13: receptor de señal

```
volatile int variable;
void handler (int signal) {
  if (signal == SIGINT || signal == SIGTSTP) {
     //do something
     variable = 1234;
  }
}
1.
struct sigaction act;
sigset_t set;
sigemtotyset(&set);
act.sa_handler = handler;
act.sa_flags = SA_SIGINFO;
2.
if (sigaction (SIGTSTP, &act, NULL) == KO) {exit}
sigsuspend(&set);
```

1. Rellenar el struct sigaction con el puntero a la función y el flag SA_SIGINFO
2. Asociar las señales a la función que va recibir y ejecutar una acción

Práctica 2.4: Tuberías

Actividad 1: tubería unidireccional

```
1.
int tub[2];
if (pipe(tub) == KO) {exit;}
2.
pid = fork();
switch(pid) {
     case -1: exit
```

La tubería sin nombre: tub [1] extremo de escritura, tub [0] extremo de lectura
1. creamos la tubería de tamaño 2 con pipe
2. hacemos fork para trabajar con el padre y el hijo en paralelo
3. en el caso del hijo, cerramos lo que no vamos a utilizar es decir la escritura de la tubería. Se

case 0:
3.
close(tub[1]);
//do something con tub[0]
close(tub[0]);
default:
4.
close(tub[0]);
//do something con tub[1]
close(tub[1]);
 	
}
	ejecuta las acciones sobre la posición 0 de la tubería y lo cerramos una vez ejecutemos la acción.
	4. en el caso del padre, cerramos lo que no vamos a utilizar es decir la lectura de la tubería. Se ejecuta las acciones sobre la posición 1 de la tubería y lo cerramos una vez ejecutemos la acción.

Actividad 2: tubería bidireccional	

1.
int p_h[2];
int h_p[2];
if (pipe(p_h) == KO || pipe(h_p) == KO) {exit;}
2.
pid = fork();
switch(pid) {
case -1: exit
case 0:
3.
close(p_h[1]);
close(h_p[0]);
//do something con h_p[1] (escritura)
//do something con p_h[0] (lectura)
close(p_h[0]);
close(h_p[1]);
default:
4.
close(p_h[0]);
close(h_p[1]);
//do something con h_p[0] (lectura)
//do something con p_h[1] (escritura)
close(p_h[1]);
close(h_p[0]);
 	
}
	La tubería sin nombre: p_h [0] extremo lectura y p_h [1] extremo escritura, h_p [0] extremo escritura y h_p [1] extremo lectura
	1. creamos las tuberías de tamaño 2 con pipe
	2. hacemos fork para trabajar con el padre y el hijo en paralelo
	3. en el caso del hijo, cerramos lo que no vamos a utilizar es decir la escritura de la p_h[1] y el extremo de la lectura de h_p[0]. Se ejecuta las acciones y lo cerramos una vez ejecutemos la acción.
	4. en el caso del padre, cerramos lo que no vamos a utilizar es decir la escritura de la h_p[1] y el extremo de la lectura de p_h[0]. Se ejecuta las acciones y lo cerramos una vez ejecutemos la acción.

Actividad 5: multiplexación	

1.
fd_set readfds;
int fd_select, max_fd;
int fd_1 = open("pipe_1", O_NONBLOCK | O_RDONLY);
int fd_2 = open("pipe_2", O_NONBLOCK | O_RDONLY);
 	
for(;;) {
2.
FD_ZERO(&readfds);
FD_SET(fd_1, &readfds);
FD_SET(fd_2, &readfds);
 	
3.
max_fd = max(fd_1, fd_2);
fd_select = select(max_fd, &readfds, NULL, NULL, NULL);
4.
if(fd_select == KO) { exit }
5.
else if(fd_select) {
6.
if(FD_ISSET(fd_1, &readfds)) {
readfifo(fd_1);
close(fd_1);
int fd_1 = open("pipe_1", O_NONBLOCK | O_RDONLY);
}
else if(FD_ISSET(fd_2, &readfds)) {
//igual que fifo 1
}
}
7.
	1. Abrimos las dos tuberías creadas con anterioridad (mkfifo)
	2. dentro del bucle, se crea un set de descriptores vacíos y se añaden las dos tuberías anteriormente abiertas
	3. se selecciona cual de ellas dos tiene caracteres listos para ser leídos (máximo de ambos)
	4. se comprueba si no ha devuelto un error
	5. si es mayor que 0 entonces significa que hay un descriptor con datos listos para ser leídos.
	6. se comprueba si esta en el conjunto de fd listos (fd_1 o fd_2)
	6. se lee la información se cierra el fd y se vuelve a abrir. Esto se hace porque read devuelve 0 indicando fin de fichero por lo que hay que cerrarla y abrirla de nuevo para iniciar otra vez el puntero, si no select considera que el descriptor esta listo para lectura y no se bloqueará.

else {
printf("No data available");
}
}
8.
close(fd_1);
close(fd_2);	

Práctica 2.5: Sockets	
Actividad 1: Acceso a la información de struct addrinfo	

1.
struct addrinfo *address, hints;
char hbuf[NI_MAXHOST], res[NI_MAXHOST],
name[INET6_ADDRSTRLEN];
 	
memset(&hints, 0, sizeof(hints));
 	
hints.ai_family = AF_UNSPEC;
hints.ai_flags = AI_V4MAPPED | AI_ALL;
 	
2.
getaddrinfo(argv[1], NULL, &hints, &address);
3.
getnameinfo(address ->ai_info, address ->ai_addrlen,
hbuf, sizeof(hbuf), NULL,0, NI_NUMERICHOST |
NI_NUMERICSERV);
4.
const void *addr;
int port;
 	
switch(address->ai_family)
case AF_INET6:
addr = &((struct sockaddr_in6 *)(address-
>ai_addr))->sin6_addr;
port = ((struct sockaddr_in6 *)(address-
>ai_addr))->sin6_port;
case AF_INET:
addr = &((struct sockaddr_in *)(address-
>ai_addr))->sin_addr;
port = ((struct sockaddr_in *)(address-
>ai_addr))->sin_port;
 	
5.
struct protoent *protocol = getprotobynumber(address-
>ai_protocol);
protocol ->p_name;
6.
if(addr && inet_ntop(address->ai_family, addr, name,
sizeof(name)))
7.freeaddrinfo(address)
	1.creamos las estructuras, en hints creamos una estructura vacia a la que le añadimos el tipo de servidor que queremos crear: AF_UNSPEC (IPv4, IPv6), AI_V4MAPPED | AI_aLL (resuelve 0.0.0.0 y ::)
	2.obtenemos la información de dirección
	3.obtenemos el nombre de la dirección, para ello pasamos lo que hemos obtenido en el paso anterior y añdimos que el host y el servidor va a ser pasado como número (host=198.162.17.1 port=80)
	4.accedemos a la familia y obtenemos la dirección y el puerto dependiendo de esta
	5.obtenemos el protocolo en la estructura protoent
	6.comprobamos que addr no es vacio y transformarmos la direccion según la familia que sea
	7.liberamos la variable

Actividad 2: crear un servidor UDP	

1.
struct addrinfo *address, *result, hints;
int sfd;
memset(&hints, 0, sizeof(hints));
 	
hints.ai_family = AF_UNSPEC;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_protocol = 0;
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;
 	
2.
getaddrinfo(argv[1], NULL, &hints, &result);
	1. para crear un servidor UDP se debe seleccionar AI_PASSIVE en flags, SOCK_DGRAM en socktype y lo demás a NULL
	2.obtenemos la información de dirección

Actividad 2: conectar a un servidor UDP	
iteramos: address = result until address == NULL int sfd; 1. sfd = socket(address->ai_family, address- >ai_socktype, address->ai_protocol); 2.bind(sfd, address->ai_address, address->ai_addrlen)	iteramos la lista enlazada que se nos devuelve en result 1. asociamos la dirección a un socket, si nos devuelve un descriptor de fichero socket podemos avanzar. Si no continuamos

3.close(sfd); 4.si address == NULL es error y se hace freeaddrinfo(result)	2. establecemos el enlace con servidor creado 3. cerramos el descriptor de fichero 4. liberamos la dirección
Actividad 2: recibir datos de un cliente al servidor UDP	
1. struct sockaddr_storage peer_addr; struct socklen_t peer_addr_len; char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV], name[INET6_ADDRSTRLEN], buf[MAX_SIZE]; 2.iteramos infinitamente (while(0) | for(;;)) peer_addr_len = sizeof(struct sockaddr_storage); 3.nread = recvfrom(sfd, buf, MAX_SIZE, 0, (struct sockaddr *) &peer_addr, &peer_addr_len); 4.getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV);	1. creamos las nuevas estructuras a utilizar 2. iteramos infonitamente, declaramos la longitud del struct de la dirección de la que recibimos datos 3. recibimos al servidor, con el struct asociado 4. obtenemos información de ese cliente
Actividad 2: enviar un buffer al cliente	
1. char *strtime[MAX_SIZE]; size_t tread = MAX_SIZE; strtime = “12:00 PM 0/0/0” 2. sendto(sfd, strtime, tread, 0, (struct sockaddr *) &peer_addr, peer_addr_len) != tread //error	1.creamos un buffer nuevo con la información a enviar 2.enviamos con el struct creado en pasos anteriores
Actividad 3: crear un cliente UDP	
ver crear un servidor udp	en este caso lo único que en ai_flags es 0.
Actividad 3: conectar un cliente UDP a un servidor UDP	
iteramos: address = result until address == NULL int sfd; 1. sfd = socket(address->ai_family, address->ai_socktype, address->ai_protocol); 2.connect(sfd, address->ai_address, address->ai_addrlen) 3.close(sfd); 4.si address == NULL es error y se hace freeaddrinfo(result)	iteramos la lista enlazada que se nos devuelve en result 1. asociamos la dirección a un socket, si nos devuelve un descriptor de fichero socket podemos avanzar. Si no continuamos 2. establecemos la conexión con servidor creado 3. cerramos el descriptor de fichero 4. liberamos la dirección
Actividad 3: el cliente envia al servidor UDP los argumentos del programa	
1. for 3…argc cmd_len = strlen(arv[i]) + 1 1. write(sfd, argv[i], cmd_len) 2.nread = read(sfd, buf, MAX_SIZE)	1. recorremos los argumentos desde la posición 3 2. escribimos el argumento en el servidor 3. leemos del servidor
Actividad 4: servidor UDP multiplexado con la entrada estandar	

2.iteramos infinitamente (while(0) | for(;;))
peer_addr_len = sizeof(struct sockaddr_storage);
3.
FD_ZERO(&readfds);
FD_SET(sfd, &readfds);
FD_SET(sfd, &readfds);
 	
4.
max_fd = max (sfd, 0);
fd_select = select(max_fd, &readfds, NULL, NULL, NULL);
5.
if(fd_select == KO) { exit }
6.
else if(fd_select) {
7.
if(FD_ISSET(0, &readfds)) {
read(0, buf, MAX_SIZE)
strtime_function(buf, &strtime &tread);
}
else if(FD_ISSET(sfd, &readfds)) {
nread = recvfrom(sfd, buf, MAX_SIZE, 0, (struct sockaddr *) &peer_addr, &peer_addr_len);
	1. repetimos crear y conectar a un servidor UDP, de la actividad 2
	2. iteramos infonitamente, declaramos la longitud del struct de la dirección de la que recibimos datos
	3. dentro del bucle, se crea un set de descriptores vacíos y se añaden las dos tuberías anteriormente abiertas
	4. se selecciona cual de ellas dos tiene caracteres listos para ser leídos (máximo de ambos)
	5. se comprueba si no ha devuelto un error
	6. si es mayor que 0 entonces significa que hay un descriptor con datos listos para ser leídos.
	7. se comprueba si esta en el conjunto de fd listos (0 o sfd)
	8. se lee de entrada estándar si hay valores o sino por sfd, luego se sigue los mecanismos ya vistos en la actividad 2

getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV);
 	
strtime_function(buf, &strtime &tread);
sendto(sfd, strtime, tread, 0, (struct sockaddr *) &peer_addr, peer_addr_len) != tread //error
}
}
8.
else {
printf("No data available");
}	

Actividad 4: creamos una función que cree el buffer a imprimir	

int strtime_function (char buf[], char str[], ssize_t *tread) {
if(buf[0] == “d”)
*tread = (ssize_t) strftime(str, MAX_SIZE, “%D”, tm);
else if(buf[0] == “t”)
*tread = (ssize_t) strftime(str, MAX_SIZE, “%T%p”, tm);
else
return -1;
return 0;
}
	Esta función recibe en buf lo que se va querer realizar, en str se devuelve el string de tiempo creado y en tread el tiempo

Actividad 5: servidor UDP pre-fork	

2.
for 0..MAX_CLIENTE
pid = fork()
3. if(pid == 0)
for(;;)
--actividad 2: recibir datos de un cliente al servidor UDP
--actividad 2: enviar un buffer al cliente
4. else if(pid > 0)
wait(&status);
	1. repetimos crear y conectar a un servidor UDP, de la actividad 2
	2.iteramos al máximo de clientes que se puede conectar al servidor
	3.por cada iteración se crea una conexión con un cliente, se ramifica y se empieza a recibir/enviar datos con el cliente UDP
	4.el padre se encarga de esperar a que el hijo termine

Actividad 6: crear un servidor TCP eco	

1.
struct addrinfo *address, *result, hints;
int sfd;
memset(&hints, 0, sizeof(hints));
 	
hints.ai_family = AF_UNSPEC;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = 0;
hints.ai_canonname = NULL;
hints.ai_addr = NULL;
hints.ai_next = NULL;
 	
2.
getaddrinfo(argv[1], NULL, &hints, &result);
	1. para crear un servidor UDP se debe seleccionar 0 en flags, SOCK_STREAM en socktype y lo demás a NULL
	2.obtenemos la información de dirección

Actividad 6: conectar a un servidor TCP eco	

iteramos: address = result until address == NULL
int sfd;
1.
sfd = socket(address->ai_family, address->ai_socktype, address->ai_protocol);
2.
bind(sfd, address->ai_address, address->ai_addrlen)
3.
close(sfd);
4.si address == NULL es error y se hace freeaddrinfo(result)
5.listen (sfd, 10);
	1. asociamos la dirección a un socket, si nos devuelve un descriptor de fichero socket podemos avanzar. Si no continuamos
	2. establecemos la conexión con servidor creado
	3. cerramos el descriptor de fichero
	4. liberamos result
	5. establecer un límite de escucha

Activdad 6: conectar a un cliente TCP	

1.
iteramos infinitamente (while(0) | for(;;))
peer_addr_len = sizeof(struct sockaddr_storage);
 	
2.
cfd = accept(sfd, (struct sockaddr *) &peer_addr, &peer_addr_len);
 	
3.
getnameinfo((struct sockaddr *) &peer_addr, peer_addr_len, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV);
	1. iteramos infonitamente, declaramos la longitud del struct de la dirección de la que recibimos datos
	2. establecemos la conexión con el cliente, para reconocer con que cliente estamos trabajamos guardamos el resultado de accept
	3. obtenemos información del struct

Actividad 6: recibir y enviar datos tipo eco a un cliente TCP	

1.
while(nread = recv(cfd, buf, MAX_SIZE, 0)) {
buf[nread + 1] = '0';
2. send(cfd, buf, nread, 0);
}
3.
close(cfd);
	1. leemos del descriptor del cliente
	2. enviamos tipo eco al propio cliente
	3. cerramos el descriptor del cliente

Actividad 7: el cliente envia al servidor TCP datos por consola	

--actividad 6: crear un servidor TCP eco
--actividad 6: conectar a un servidor TCP eco (sin el listen)
1.
iteramos infinitamente (while(0) | for(;;))
2.
nread = read(0, buf, MAX_SIZE)
3.
write(sfd, buf, nread);
4.
nread = read(sfd, buf, nread);
	2. leemos de consola
	3. enviamos al servidor
	4. nos reenvía la info el servidor, porque estamos en un servidor eco

Actividad 8: servidor TCP accept-and-fork	

--actividad 6: crear un servidor TCP eco
--actividad 6: conectar a un servidor TCP eco
1. iteramos infinitamente (while(0) | for(;;))
peer_addr_len = sizeof(struct sockaddr_storage);
 	
2.cfd = accept(sfd, (struct sockaddr *) &peer_addr, &peer_addr_len);
3.
pid = fork
4. if(pid == 0)
for(;;)
while(nread = recv(cfd, buf, MAX_SIZE, 0)) {
buf[nread + 1] = '0';
send(cfd, buf, nread, 0);
}
5. else if(pid > 0)
close(cfd);
	2. aceptamos la conexión del cliente
	3. hacemos un fork y dividimos las tareas para el padre y el hijo
	4. si es el hijo, hacemos un bucle infinito que reciba la información del cliente, se encarga de leer/enviar
	5. el padre se encarga de cerrar el descriptor de fichero del hijo

Actividad 9: capturar SIGCHLD en un servidor TCP	
--actividad 6: crear un servidor TCP eco --actividad 6: conectar a un servidor TCP eco 1. sigaction(SIGCHLD, &act, NULL); --actividad 6: recibir y enviar datos tipo eco a un cliente TCP ó --actividad 8: servidor TCP accept-and-fork	1. añadimos la señal que queremos capturar
Actividad 9: handler	

void handler (int signal) {
1.
if(signal == SIGCHLD)
pid_t pid;
pid = wait(NULL);
 	
}
	1. el handler se encarga de cerrar la conexión con el hijo cuando este le envia la señal de que ha terminado