Colección de Ejercicios

Autores:

Rubén Gran

Pablo Ibáñez

Carlos J. Pérez

Teresa Monreal

José Luís Briz

Esta colección de ejercicios incluye ejercicios de análisis e implementación relacionados con conceptos básicos sobre sistemas operativos y llamadas al sistema operativo.

Todos los ejercicios aquí incluidos forman parte de propuestas de ejercicios y exámenes que también podéis encontrar disponibles en el directorio de Hendrix `/misc/practicas/sistemas`.

1º. Ejercicio

Tenemos los siguientes programas escritos en C: `padre.c e hijo.c`. ¿En qué orden se harán las escrituras? ¿Cuál será el resultado final en la pantalla de la terminal y en el fichero `salida.dat` al ejecutar `$hendrix> ./padre`? Justifica tu respuesta.

padre.c

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

int main() {
    int id,estado;

    close(1);
    creat("salida.dat",0777);
    write(1,"linea de texto n 1\n",19);
    if((id=fork())==0) {
        execl("hijo","hijo",0);
        exit(1);
    }
    else {
        while(wait(&estado)!=id);
        write(1,"linea de texto n 5\n",19);
        exit(0);
    }
}
```

hijo.c

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

int main() {
    int idf;

    write(1,"linea de texto n 2\n",19);
    idf=open("salida.dat",O_WRONLY);
    lseek(idf,0L,L_SET);
    write(1,"linea de texto n 3\n",19);
    write(idf,"linea de texto n 4\n",19);
    close(idf);
    close(1);
    exit(0);
 }
```

2º. Ejercicio

Supongamos los siguientes programas en lenguaje C: `primero.c` y `segundo.c`. Comenta y justifica cuál es su comportamiento.

```
$hendrix> ./primero
```

primero.c

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

int main() {
    int id, fd[2];

    id = fork();
    pipe(fd);
    switch (id) {
        case 0:
            close(0);
            dup(fd[0]);
            close(fd[0]);
            close(fd[1]);
            execl("segundo","segundo",0);
            exit(1);
        case -1:
            exit(1);
        default:
            close(1);
            dup(fd[1]);
            close(fd[0]);
            close(fd[1]);
            execl("segundo","segundo",0);
            exit(1);
    }
    exit(0);
}
```

segundo.c:

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

int main() {
    char c;

    while(read(0,&c,1)!=0)
        write(1,&c,1);
    exit(0);
}
```

3º. Ejercicio

Dados los siguientes programas escritos en lenguaje C: `primero.c` y `segundo.c`. Comenta y justifica cuál es el comportamiento. ¿Qué valor imprimirá el proceso primero?

```
$hendrix> ./primero
```

primero.c

```
#include <stdio.h><unistd.h><stdlib.h><sys/wait.h><signal.h>
int contador = 0;

void tratar_alarma(int n) {
    signal(SIGALRM,tratar_alarma);
    write(1,"alarma \n",8);
}

int main() {
    int id, i, estado;

    signal(SIGALRM,tratar_alarma);
    switch (id=fork()) {
        case 0: execl("segundo","segundo",0);
                printf("Soy segundo y termino\n");
                exit(1);
        case -1:exit(2);
    }
    for (i=0;i<4;i++) {
        contador=contador+1;
        alarm(3);
        pause();
    }
    while(id!=wait(&estado));
    printf("%d\n",contador);
    exit(0);
}
```

segundo.c

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

int contador = 0;

void tratar_alarma(int n) {
    signal(SIGALRM,tratar_alarma);
    write(1,"alarma \n",8);
}

int main(){
    int id, i, estado;

    for (i=0;i<4;i++) {
        contador=contador+1;
        alarm(3);
        pause();
    }
    printf("%d\n",contador);
    exit(0);
}
```

4º. Ejercicio

El comando "`$hendrix> ps -lu usuario`" muestra lo siguiente por pantalla:

```
F S UID PID  PPID C  PRI NI ADDR     SZ WCHAN   TTY   TIME COMD
1 S 243 7152 7046 0  158 20 4004dec0 58 1d79e12 ttyqa 0:00 ksh
1 R 243 7788 5654 10 180 20 4007b200 16         ttyq6 0:00 ps
```

Escribir un programa en lenguaje C que sea capaz de matar a todos los procesos del usuario vivos en el momento de su ejecución. (Para que funcione correctamente dejará vivos al intérprete de comandos desde el que se lanzó el comando `ps`, y al propio comando `ps -lu usuario`.)

5º. Ejercicio

Escribir un programa en lenguaje C que cree tres procesos y una tubería que los comunique. Los dos primeros tendrán la salida estándar asociada a la tubería y el tercero la entrada estándar. El programa recibe como argumentos los nombres de los ficheros que deben ejecutar los tres procesos.

```
$hendrix> ./mi_prog ejecutable1 ejecutable2 ejecutable3
```

6º. Ejercicio

Implementar el programa "mover" (similar al comando "mv"). El programa recibe dos parámetros, el primero de ellos corresponde a un fichero ya existente que desaparecerá después de la ejecución de "mover", apareciendo un nuevo fichero con los mismos contenidos que el anterior y con nombre igual al segundo parámetro. Para dicha implementación solo se pueden usar llamadas al sistema operativo. Es importante que la implementación sea óptima.

7º. Ejercicio

Escribir una aplicación (`redi.c`) en lenguaje C utilizando llamadas al sistema que redireccione la salida estándar y la salida de error de un programa hacia un fichero cuyo nombre se le pasará como primer argumento. El nombre del programa a ejecutar y sus parámetros se le pasaran en el segundo argumento y sucesivos.

Ejemplo `$hendrix> ./redi salida ls -l pepe`

Debe redireccionar hacia el fichero `salida` (creándolo si no existe o truncándolo) las salidas estándar y de error del comando `ls -l pepe`.

Notas:

- El comando a ejecutar puede producir salida estándar y salida de error mezcladas en el tiempo.
- La ejecución de `redi` solo producirá un proceso.
- El programa `redi.c` tiene como máximo diez líneas.

8º. Ejercicio

Explicar detalladamente (máximo un folio por una cara) cómo se comporta el siguiente programa:

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

int pid;

void trata_alarma(int n){
    kill(pid,SIGKILL);
}

int main(){
    int status;

    pid=fork();
    if (pid!=0) {
        signal(SIGALRM,trata_alarma);
        alarm(10);
        wait(&status);
        alarm(0);
    }
    else {
        execl("otro","otro",0);
        printf("Termina el otro\n");
        exit(1);
    }
}
```

9º. Ejercicio

Sea un sistema controlado por un computador con un Sistema Operativo. El sistema a controlar está compuesto por un sensor y un actuador. El computador debe leer el sensor periódicamente y operar el actuador dependiendo del valor del sensor.

El software para controlar este sistema consta de dos procesos llamados "sensor" y "actuador".

El proceso "sensor" crea primero el proceso "actuador" estableciendo una pipe entre los dos. Luego el proceso "sensor" llamara con una periodicidad de 1 segundo a una función "`leer_sensor()`" que devolverá el valor del sensor en forma de entero. Una vez conseguido el valor, el proceso "sensor" lo enviara al proceso "actuador" a través de la pipe.

El proceso "actuador" crea inicialmente un fichero "`datos.dat`". Luego empieza a leer de la pipe los datos enviados por el proceso "sensor". Con cada dato leído debe, en primer lugar, llamar a la función "`actua(dato)`" pasándole como parámetro el dato, escribiendo después dicho dato en el fichero "datos.dat". La rutina "`actua(dato)`" activara el actuador dependiendo del valor recibido.

Se pide escribir en lenguaje C y con llamadas al sistema: el código de los procesos "sensor" y "actuador", suponiendo que las funciones "`leer_sensor()`" y "`actua(dato)`" ya están escritas.

10º. Ejercicio

Escribe un programa en C que utilizando solo llamadas al sistema, nos dé como resultado el número de ficheros que hay en un directorio (cuyo nombre se le pasa como argumento) y el tamaño en bytes que ocupan estos. Como ayuda recuerda que el comando `ls -l` ofrece el siguiente formato de salida:

```
Protection links UID      GID      size date   time  name
-rw-r-----     1 teresa   arqcomp  3783 Nov  3 09:42 gauss.c
drwxr-x---     3 teresa   arqcomp  1024 Jul 14 13:53 postgrado
-rw-r-----     1 teresa   arqcomp  3783 Nov 23 11:35 gauss2.c
-rw-r-----     1 teresa   arqcomp  3769 Nov  8 15:51 tpc.c
drwxr-x---     3 teresa   arqcomp  1024 Dec  2 16:06 examenes
drwxr-x---     2 teresa   arqcomp  1024 Nov 21 16:39 practicas
```

Nota: No se contaran los ficheros directorio.

11º. Ejercicio

A continuación se muestran dos programas en lenguaje C. ¿Qué ocurre si ejecutamos `primero.c`? ¿Y si ejecutamos `primero.c` redireccionando la salida estándar sobre un fichero?

Responde suponiendo que, después del `fork()` se ejecuta primero el padre y acaba antes de entrar el hijo. Luego, supón el entrelazado contrario.

primero.c

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

int main() {
    printf("Mensaje1");
    write(1,"Mensaje2\n",9);(con 10 escribe 1 byte de basura)
    if (fork()) {
        printf("Mensaje3\n");
        execlp("segundo","segundo",0);
        printf("Mensaje4");
    }
    printf("Mensaje5");
}
```

segundo.c

```
#include <stdio.h>

int main() {
    printf("Mensaje6");
}
```

12º. Ejercicio

Implementar una función “ejecutar” capaz de poner en ejecución un comando con argumentos. A la función se le pasaran dos parámetros:

- (comando) un string que contiene el nombre del fichero donde está el ejecutable seguido de los argumentos que se le han de pasar. Todo ellos separado por espacios en blanco o tabuladores.
- (espera) un byte de modo de ejecución que puede tomar dos valores: 0) la función devuelve el control en cuanto se lanza la ejecución del comando (ejecución asíncrona). 1) La función devuelve el control cuando el comando a ejecutar ha terminado (ejecución síncrona).

“ejecutar” devolverá como resultado el identificador del proceso encargado de ejecutar el comando.

13º. Ejercicio

Indica qué ficheros se generan y cuál es el contenido de cada uno de ellos después de ejecutar la siguiente línea de comandos:

```
$hendrix> primero >f.out 2>f.err
```

primero.c

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

int fd0=0;
int fd2=2;

int main(){
    int fd1 = 1;
    int fd3 = 3;
    int fd4 = 4;

    fd1 = creat("fich1",0755);
    fd3 = dup(fd1);
    if (!fork()) {
        fd2 = creat("fich2",0750);
        write(fd1,"Mensaje1",8);
        write(fd2,"Mensaje2\n",9);
        execlp("segundo","segundo",0);
    } else {
        wait(NULL);
        fd0=dup(fd2);
        fd4 = open("fich2", 1);
        write(fd0,"M1",2);
        write(fd3,"M2",2);
        write(fd4,"M3",2);
    }
}
```

segundo.c

```
#include <unistd.h>

int main() {
    write(1,"A1",2);
    write(2,"A2",2);
    write(3,"A3",2);
    write(4,"A4",2);
    write(5,"A5",2);
    write(6,"A6",2);
}
```

14º. Ejercicio

Al ejecutar el comando "ls -li" en un sistema aparece la siguiente salida en pantalla:

```
$hendrix> ls -li
total 1
54686 -rw-r--r-- 2 user1 compu 0 Jun 19 10:21 fichero1
54686 -rw-r--r-- 2 user1 compu 0 Jun 19 10:21 fichero2
```

La primera columna aparece como respuesta a la opción "`-i`" del comando, e indica el número de nodo-i de cada uno de los ficheros que contiene el directorio. El resto de columnas ya las conocéis, son las usuales de la opción "`-l`".

1. Suponiendo que cada uno de los siguientes comandos se ejecuta partiendo de la situación inicial. Indica qué salida produciría "`ls -li`" después de ejecutar cada uno de ellos?

```
$hendrix> rm fichero1
$hendrix> chmod u+x fichero1
$hendrix> mv fichero1 ..
$hendrix> cp fichero1 fichero3
```

15º. Ejercicio

Diseñar un programa llamado “crea” que:

- cree tantos hijos como se le indique en el primer parámetro. Todos ejecutan en paralelo.
- cada uno de los hijos debe enviar al padre su pid de proceso mediante una pipe y a continuación debe morir
- El padre deberá escribir en un fichero, cuyo nombre se pasa como segundo parámetro, todos los mensajes que lleguen de su hijo, uno en cada línea

Nota: Todos los hijos deberán utilizar el mismo pipe.

Ejemplo: `$hendrix> crea 15 hijos`

Creará 15 hijos cuyos identificadores aparecerán en el fichero "hijos".

16º. Ejercicio

Explica qué hace el siguiente programa, cuál sería su resultado final y por qué. ¿Y si quitamos la linea `pause()`?

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

void una_alarma(int n) {
    write(2,"UNA_ALARMA\n",11);
}

void otra_alarma(int n) {
    write(2,"OTRA_ALARMA\n",12);
}

int main() {
    int estado=2,i;

    signal (SIGALRM, una_alarma);
    if (fork()) {
        alarm(1);
        i=wait(&estado);
        printf("wait devuelve %d, estado=%x\n",i,estado);
    } else {
        signal(SIGALRM, otra_alarma);
        alarm(3);
        pause();
        execl("/bin/ps","ps",0);
        exit(1);
    }
}
```

17º. Ejercicio

Explicar qué hace nuestro programa y cuál sería su resultado final.

Indica qué variación del resultado habría si añadiésemos al final de código del proceso padre la siguiente línea:

```
wait(NULL);
```

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

#define SIZE 13

char *mensaje1 = "El mensaje 1\n";
char *mensaje2 = "El mensaje 2\n";
char *mensaje3 = "El mensaje 3\n";

void rutina(int n) {
    kill(getpid(),SIGKILL);
}

int main (){
    char buf[SIZE];
    int fd[2],j;

    signal(SIGALRM, rutina);
    if (fork()) {
        if (pipe(fd)<0) {
            fprintf(stderr,"Error en la llamada pipe\n");
            exit(-1);
        }
        write(fd[1],mensaje1, SIZE);
        write(fd[1],mensaje2, SIZE);
        for(j = 0; j < 2; j++) {
            read(fd[0], buf, SIZE);
            write(1, buf, SIZE);
        }
    } else {
        alarm(3);
        for (j=0;j<5;j++) {
            write(1,mensaje1,SIZE);
            sleep(1);
        }
        write(1, mensaje2, SIZE);
    }
}
```

18º. Ejercicio

Explica brevemente cómo invocarías al programa ejecutable “mix” y cuál sería el resultado.

Indica cuál sería el resultado si fallase la instrucción `fd1=open(…)`.

mix.c

```
#include <unistd.h><fcntl.h>

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

    int fd1, fd2, fd3, tam, x;
    char c;

    fd1=open(argv[1],0);
    fd2=open(argv[2],0);
    tam=open(argv[3],0);
    fd3=creat(argv[4],0700);
    x = lseek(fd2,tam,0);
    while (read(fd2, &c,1)>0) {
        while(x>0) {
            read(fd1,&c,1);
            write(fd3,&c,1);
            x--;
        }
        write(fd3,&c,1);
    }
}
```

19º. Ejercicio

Explica cuál será el contenido de “salida.dat” y por qué.

Sustituye la línea `lseek(id,-6L,SEEK_CUR)` de `segundo.c` por otra que tenga el mismo efecto que ésta. Escribe al menos dos sustituciones posibles.

primero.c

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

int main() {
    int id, estado;

    close(1);
    creat("salida.dat",0777);
    write(1,"linea\n",6);
    if ((id=fork()) == 0 ) {
        execl("segundo","segundo",0);
        exit(1);
    } else {
        while (wait(&estado) != id);
        write(1,"linea\n",6);
        exit(0);
    }
}
```

segundo.c

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

int main() {
    int idf, id;

    idf=open("salida.dat",O_WRONLY);
    write(idf,"linea\n",6);
    id = dup(1);
    lseek(id, -6L, SEEK_CUR);
    write(id,"linea\n",6);
    write(idf,"linea\n",6);
    close(idf);
    close(id);
    close(1);
    exit(0);
}
```

20º. Ejercicio

Diseñar un programa llamado "thead" cuyo funcionamiento describimos a partir del siguiente ejemplo de utilización:

```
$hendrix> P1 | thead n F1 | P2
```

donde P1, P2 y thead son programas y F1 es el nombre de un fichero (pasado como segundo parámetro a "thead").

Como resultado de la ejecución de la línea anterior:

- P2 recibe en su entrada estándar las n primeras líneas procedentes de la salida estándar de P1 (n es pasado como primer parámetro a "thead" y cuando n=0, P2 recibe en su entrada estándar la salida estándar de P1).
- En F1 queda escrita la salida estándar de P1 (para cualquier valor de n).

Se pide:

Apartado 1. Implementar el programa "thead" en lenguaje C, usando sólo llamadas al sistema operativo para la entrada/salida.

Apartado 2. Utilizando el programa "thead" del **apartado 1**, escribir las líneas de comandos que ejecuten de la forma más eficiente posible lo que se pide a continuación:

- La salida estándar de P1 quede escrita en F1 y F2 y sus n primeras líneas sean usadas como entrada estándar de P2.
- La salida estándar de P1 quede escrita en F1 y sus n primeras líneas aparezcan en pantalla.
- La salida estándar de P1 quede escrita F1 y en F2.

21º. Ejercicio

Diseñar un programa llamado "replicar" en lenguaje C, usando sólo llamadas al sistema operativo y cuyo funcionamiento describimos a partir de los siguientes ejemplos de utilización:

```
$hendrix> ./replicar F1
```

Copiará lo que le introducimos por teclado sobre el fichero F1 (cuyo nombre le ha sido pasado como primer parámetro) y además lo mostrará por pantalla.

```
$hendrix> ./replicar F3 < F1 > F2
```

Creará dos ficheros, F2 y F3, ambos con el mismo contenido que F1.

```
$hendrix> ./replicar
```

No hará nada. Termina devolviendo un código de error de valor 2, ya que le falta el parámetro.

22º. Ejercicio

Hemos escrito el siguiente programa en lenguaje C utilizando llamadas al sistema operativo:

primer.c:

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

int main (int argc, char **argv ){
    int fd[2];

    pipe(fd);
    switch (fork()) {
    case -1:
        fprintf(stderr,"Error en fork.");
        exit(1);
    case 0:
        close(fd[0]);
        close(1);
        dup(fd[1]);
        close(fd[1]);
        execlp("cat","cat",argv[1],0);
    default:
        close(fd[1]);
        close(0);
        dup(fd[0]);
        close(fd[0]);
        wait(NULL);
    }
    execlp("sort","sort",0);
}
```

Explica detalladamente cómo se comportará el ejecutable "primer" cuando lo utilicemos desde nuestra cuenta en `hendrix` pasándole como parámetro cualquiera de estos 2 ficheros:

```
$hendrix> ls -l
total 20
-rw-r----- 1 teresa arqcomp 8360 Jul 30 12:28 segun.c
-rw-r----- 1 teresa arqcomp 440  Jul 30 12:23 primer.c
```

¿Tiene el mismo efecto la ejecución de "`$hendrix> ./primer segun.c`" que la de "`$hendrix> ./primer primer.c`"?. Si no es así, explica por qué.

23º. Ejercicio

Explica cuál será el resultado de la ejecución de "programa" y por qué.

programa.c:

```
#include <stdlib.h><unistd.h><fcntl.h>

int fdrd, fdwt;
char c;

void rdwrt() {
    for(;;) {
        if(read(fdrd,&c,1) != 1)
            return;
        write(fdwt,&c,1);
    }
}

int main(int argc, char **argv) {
    if(argc != 3) exit(1);
    if((fdrd = open(argv[1],O_RDONLY)) == -1)
        exit(1);
    if((fdwt = creat(argv[2],0666)) == -1)
        exit(1);
    fork();
    rdwrt();
    exit(0);
}
```

24º. Ejercicio

Explicar detalladamente (máximo un folio por una cara) cómo se comporta el siguiente programa:

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

int pid;

void trata_alarma(int n) {
    kill(pid, SIGKILL);
}

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

    pid=fork();
    if (pid) {
        signal(SIGALRM,trata_alarma);
        alarm(10);
        wait(&estado);
    } else {
        execvp(argv[1],argv+1);
        exit(1);
    }
}
```

25º. Ejercicio

Después de compilar "`programa.c`", responde las siguientes preguntas con respecto a la ejecución del binario `./programa`:

- Explica de forma detallada cuál será el resultado de la ejecución de `./programa` y por qué.
- Explica qué efecto tiene en cualquier programa la línea `alarm(0);`
- Existe alguna diferencia en la ejecución anterior si a "`programa.c`" le quitamos la línea `alarm(0);` de su código?

```
#include <stdlib.h><unistd.h><signal.h>

void sig_alrm(int n) {
    write(2,"RUTINA\n",7);
    return;
}

int main() {
    int fd[2],n;
    char mensaje[8],*s;

    s="MENSAJE\n";
    signal(SIGALRM,sig_alrm);
    pipe(fd);
    if (fork()==0) {
        close(fd[1]);
        alarm(3);
        while ((n=read(fd[0],mensaje,8))>0);
        alarm(0);
        exit(0);
    }
    close(1);
    dup(fd[1]);
    close(fd[0]);
    close(fd[1]);
    while(1)
        write(1,s,8);
}
```

26º. Ejercicio

Hemos escrito el siguiente programa en lenguaje C utilizando llamadas al sistema operativo:

programa.c

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

void funcion(int n) {
    signal(n+1, funcion);
    write(2,"UN MENSAJE\n",11);
}

int main() {
    int pid;

    signal(SIGUSR1,funcion);
    if ((pid=fork()) == 0) {
        pid=getppid();
        kill(pid,SIGUSR1);
    } else
        kill(getpid(),SIGUSR2);
}
```

Después de obtener el ejecutable "programa" y de forma aleatoria, podemos obtener como resultados de la ejecución de "$hendrix> ./programa" lo siguiente:

```
$hendrix> ./programa $hendrix> ./programa $hendrix> ./programa
UN MENSAJE          UN MENSAJE          User signal 2
UN MENSAJE          UN MENSAJE          $hendrix>
UN MENSAJE          $hendrix>
$hendrix>
```

Explica por qué se obtienen estos 3 tipos de resultados.

27º. Ejercicio

Hemos escrito estos 2 programas en lenguaje C utilizando llamadas al sistema operativo. Explica cuál será el contenido del fichero "salida.dat" y por qué al ejecutar programa1 y programa2

programa1.c

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

int main() {
    int id,estado,idf;

    close(1);
    creat("salida.dat",0777);
    write(1,"linea de texto n 1\n",19);
    if ((id=fork())==0) {
        execl("hijo","hijo",0);
        exit(1);
    } else {
        while(wait(&estado)!=id);
        write(1,"linea de texto n 5\n",19);
        exit(0);
    }
}
```

hijo.c

```
#include <stdlib.h><unistd.h><fcntl.h>

int main() {
    int idf;

    write(1,"linea de texto n 2\n",19);
    idf=open("salida.dat",O_WRONLY);
    write(idf,"linea de texto n 3\n",19);
    write(idf,"linea de texto n 4\n",19);
    close(idf);
    close(1);
    exit(0);
}
```

programa2.c

```
#include <stdlib.h><unistd.h><fcntl.h>

int main() {
    int idf;

    close(1);
    creat("salida.dat",0777);
    write(1,"linea de texto n 1\n",19);
    write(1,"linea de texto n 2\n",19);
    idf=dup(1);
    lseek(idf,0L,SEEK_SET);
    write(1,"linea de texto n 3\n",19);
    write(idf,"linea de texto n 4\n",19);
    write(1,"linea de texto n 5\n",19);
    exit(0);
}
```

28º. Ejercicio

Has escrito y compilado en tu directorio de trabajo el siguiente programa en lenguaje C con llamadas al sistema:

ej1.c:

```
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd, fd1, tam = 8;

    fd = open( "fichero" , O_WRONLY );
    write( fd , "linea A\n", tam - 1 );
    fd1 = creat ( "dir1/fichero" , 0777 );
    write( fd1 , "linea B\n", tam) ;
    write( fd , "linea C\n", tam - 1 );
    close( fd );
    write( fd , "linea D\n" , tam - 1 );
    dup( fd1 );
    write( fd1 , "linea E\n" , tam );
    write( fd , "linea F\n" , tam - 1 );
}
```

El estado de tu directorio de trabajo lo conoces después de ejecutar el siguiente comando:
`$Hendrix> ls -l -R`

Obtienes así un listado (ls) largo (-l) y recursivo (-R) del directorio:

```
$Hendrix> ls -l -R
total 2
drwxr-x--- 2 user1 arqcomp 24 Jun 8 17:56 dir1
-rwxr-x--- 1 user1 arqcomp 24628 Jun 8 18:24 ej1
-rw-r----- 1 user1 arqcomp 337 Jun 8 18:24 ej1.c
-rw-r----- 1 user1 arqcomp 0 Jun 8 17:55 fichero
./dir1:
total 0
```

Llamamos a éste estado del directorio: **situación1**.

situación2: Desde tu directorio de trabajo ejecutas ahora el siguiente comando:

`$hendrix> ln fichero dir1/fichero`

A) Cuál sería el estado de tu directorio de trabajo en la situación2?.

B) Partiendo de situación1 escribe cuál sería el contenido de todos los archivos `fichero` después de ejecutar ej1 (`$hendrix> ./ej1`).

C) Partiendo de situación2 escribe cuál sería el contenido de todos los archivos fichero después de ejecutar ej1 (`$hendrix> ./ej1`).

29º. Ejercicio

Tenemos el siguiente programa en lenguaje C con llamadas al sistema que provoca la ejecución de tres procesos:

ej2.c:

```
1   #include <stdio.h><stdlib.h><unistd.h><sys/wait.h>"error.h"
2
3   int main() {
4      int pid, pid1, estado;
5
6      if ( ( pid = fork( ) ) < 0 )
7         syserr ( "fork" );
8      else
9         if ( pid == 0 ) {
10           if ( ( pid = fork( ) ) < 0 )
11              syserr ( "fork" );
12           else
13              if ( pid ) {
14                 exit ( 0 );
15              }
16           sleep ( 2 );
17           fprintf(stderr,"PPID=%d, pid=%d\n",getppid(),pid);
18           exit ( 0 );
19        }
20     while ( ( pid1 = wait ( &estado ) ) > 0 )
21        fprintf ( stderr , "Wait devuelve %d\n" , pid1 );
22     exit ( 0 );
23  }
```

A) Dibuja un esquema donde aparezcan TODOS los procesos involucrados en la ejecución de ej2, indicando sus relaciones de paternidad.

B) Escribe cuál crees que sería el contenido de la salida estándar de error (stderr) después de ejecutar ej2. Debes indicar qué proceso produce cada salida.

Supón para tu respuesta los siguientes datos:

- El proceso que ejecuta ej2 tiene PID: 25455
- El proceso creado en la línea 6 tiene PID: 25456
- El proceso creado en la línea 10 tiene PID: 25457

C) Modifica ej2.c para que en la salida estándar de error (stderr) aparezcan los PID´s de los dos procesos creados (el de la línea 6 y el de la línea 10).

Nota: En la modificación de **C)** NO se puede utilizar la llamada al sistema getpid ( ) y el ej2.c modificado NO debe superar las 25 líneas de código. No es válido quitar espacios o compactar codigo.

30º. Ejercicio

Se desea realizar un cronómetro de hasta un minuto. Al pulsar Ctrl+C arrancamos el cronómetro, al pulsar por segunda vez Ctrl+C paramos y nos muestra un mensaje por pantalla con el tiempo transcurrido ("Han transcurrido XX.YYYYYY segundos") (XX segundos, YYYYYY microsegundos). Al pulsar Ctrl+C de nuevo el cronómetro iniciará la medición de un nuevo intervalo de tiempo, y así sucesivamente. Con Ctrl+Y sale del programa despidiendo al usuario con otro mensaje por pantalla.
El programa será shareware por lo que si desde que se empieza a ejecutar pasa más de 1 minuto el programa finalizará informando al usuario de que ya no puede seguir ejecutando el programa (programa recibirá la señal SIGALRM).

Ayuda:

```
#include <time.h>
int gettimeofday(struct timeval *ts,NULL);
struct timeval {
     long tv_sec;    // segundos desde (00:00:00 GMT, Jan. 1, 1970)
     long tv_usec;   // microsegundos
}
```

`gettimeofday()` devuelve siempre 0 y rellena la estructura apuntada por `ts` con dos campos: `tv_sec` segundos desde un cierto instante de referencia y `tv_usec` los microsegundos dentro de ese segundo. Los dos campos admiten aritmética entera.

31º. Ejercicio

Analizar el siguiente programa y responder a las siguientes preguntas:

- Explica brevemente qué hace este programa
- ¿Qué salida produce por la terminal?
- ¿Cómo termina? ¿Y si se comenta la linea 9?

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <signal.h>
4
5   char quien;
6   int testigo=0;
7
8   void trata(int k) {
9        signal(SIGUSR1,trata);
10       testigo=1;
11       write(1,&quien,1);
12  }
13
14  int main() {
15       int pid,destino;
16
17       destino=getpid();
18       signal(SIGUSR1,trata);
19       if (pid=fork()) {
20            quien='A';
21            sleep(1);
22       }
23       else{
24            if (pid=fork()) quien='B';
25            else quien='C';
26       }
27       if (quien=='A') {
28            destino=pid;
29            testigo=1;
30       }
31       if (quien=='B') destino=pid;
32       while (1) {
33            if (testigo==1) {
34                 testigo=0;
35                 kill(destino,SIGUSR1);
36            }
37            pause();
38       }
39  }
```

32º. Ejercicio

Analiza el siguiente código en C para UNIX, explícalo brevemente y escribe de forma clara el resultado que produce tanto en pantalla como en el fichero *salida*.

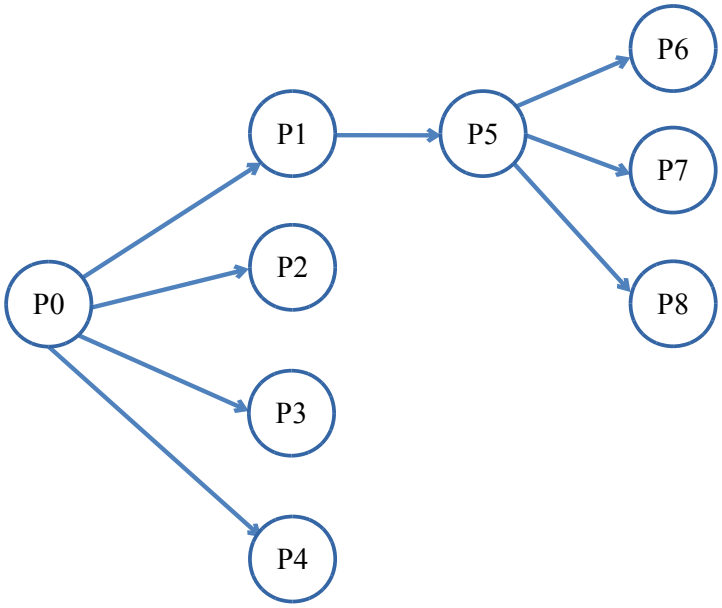
```
#include <stdio.h><unistd.h><fcntl.h><sys/wait.h>

int incrementar(int x) {
     static int i=0;
     i=i+1;
     printf("Función: valor de i: %d\n",i);
     return x+1;
}

int main() {
     int i,k=0;
     char c=5;
     int fd0=0,fd1=2;
     i=1;
     k=incrementar(k);
     printf("Inicio: valor de k: %d",k);
     write(1,"Antes de fork\n",14);
     if (fork()==0){
          close(2);
          fd0=creat("salida",0600);
          i+=5;
          k=incrementar(k);
          k=incrementar(k);
          printf("Hijo: valor de k: %d\n",k);
          printf("Hijo: valor de c: %c,%d\n",c,c);
          write(fd1,"Adios hijo",10);
     }
     else {
          c = c+'0';
          wait(NULL);
          k=incrementar(k);
          printf("Padre: valor de k: %d\n",k);
          printf("Padre: valor de i: %d\n",i);
          printf("Padre: valor de c: %c,%d\n",c,c);
          printf("Padre: Termino");
          write(fd1,"Adios padre",11);
          close(1);
     }
}
```

33º. Ejercicio

Dado el siguiente código:

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

int i=4, j=5, id=10;

void f(int x) {
     printf("%d %d %d %d %d\n", getpid(), id, x, i, j);
}

void main() {
     struct sigaction sin, sout;
     sin.sa_handler = f;
     sigemptyset(&sin.sa_mask);
     sin.sa_flags = 0;
     sigaction(SIGALRM, &sin, &sout);
     sigaction(SIGUSR1, &sin, &sout);
     printf("Empieza programa\n");
     alarm(10);

     for (i=0; i<3; i++){
          if (fork() == 0){
               alarm(i+1);
               switch (id = fork()) {
                    case 0:  pause();
                             j=j+1;
                             printf("%d %d\n", i, j);
                             break;
                    default: pause();
                             j=j+1;
               }
               exit(0);
          }
     }
     pause();
     kill(-1, SIGUSR1);
}
```

a) Dibuja el árbol de procesos que se crea. Asigna nombres a cada uno de los procesos (P0, P1, P2, …). Como ejemplo, el árbol siguiente indica que:
   - P1, P2, P3 y P4 son hijos de P0
   - P5 es hijo de P1
   - P6, P7 y P8 son hijos de P5



b) Indica claramente el resultado obtenido al ejecutar el programa. Utiliza los nombres que has asignado a los procesos en el árbol como sus PIDs cuando tengas que escribirlos en pantalla. Es imprescindible que escribas el resultado que aparece en pantalla en el siguiente cuadrado.

Tiempo	**Texto**
0	Empieza programa

34º. Ejercicio

Crear un programa **monitor** que nos permita redirigir dinámicamente la salida estándar de otro programa **comando** hacia diferentes destinos. El programa **monitor** recibirá como parámetros el nombre de un fichero (**filename**) y el **comando**. El comando estará formado por el nombre de un programa y sus parámetros (entre 0 y 100 parámetros).

Monitor podrá trabajar en tres modos, permitiendo redirigir la salida estándar producida por el comando hacia la pantalla (modo 1), hacia el fichero **filename** (modo 2) o hacia ambos (modo 3). **Monitor** empezará trabajando en modo 1 y cada vez que reciba la señal SIGUSR1 cambiará al siguiente modo (1->2->3->1->2->3-> …).

Monitor escribirá en pantalla, nada más empezar, su identificador de proceso, para que el usuario lo pueda usar para el envío de señales.

Ejemplo de llamada a monitor:

```
hendrix$ monitor fich_salida comando param1 param2
```

1) Escribir el programa **monitor** en C y con llamadas al sistema UNIX.

2) Describe brevemente **(no hace falta implementar)** cómo diseñarías otro programa monitor capaz de redireccionar dinámicamente tanto la salida estándar como la salida de error. Este nuevo monitor recibiría como parámetros los nombres de dos ficheros (el primero para la stdout y el segundo para la stderr) y un comando con sus parámetros. Con SIGUSR1 cambiaremos el modo de la salida estándar y con SIGUSR2 cambiaremos el modo de la salida de error.

35º. Ejercicio

Dado el siguiente código:

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

void f(int x) {
     static char c[3]="0\n";
     write(2,c,2);
     c[0]=c[0]+1;
}

void main(int argc, char *argv[]) {
     int i,j,n,m,pid,*fd;
     char s[3]="F0";
     struct sigaction sin, sout;
     n=atoi(argv[1]);
     fd=calloc(2*n,sizeof(int));
     for (i=0; i<n; i++){
          pipe(fd+2*i);
          if (fork()==0) {
               for(j=0;j<=i;j++) close(fd[2*j+1]);
               s[1]=i+'1';
               close(1);
               creat(s,0600);
               while(read(fd[2*i],&m,sizeof(int))!=0)
                    fprintf(stdout,"%d\n",m);
               sleep(i+1);
               kill(getppid(),SIGUSR1);
               exit(0);
          }
     }
     sin.sa_handler=f;
     sigemptyset(&sin.sa_mask);
     sin.sa_flags=SA_RESTART;
     sigaction(SIGUSR1,&sin,&sout);
     m=0;
     for (i=0;i<n;i++)
          for (j=0;j<n;j++) {
               write(fd[2*j+1],&m,sizeof(int));
               m=m+1;
          }
     for(i=0;i<n;i++) close(fd[2*i+1]);
     close(1);
     creat(s,0600);
     for (i=0;i<n;i++){
          pid=wait(NULL);
          fprintf(stdout,"%d\n",pid);
     }
}
```

Asigna nombres a cada uno de los procesos involucrados (P0, P1, P2, …).

a) Indica claramente la salida (en la terminal y en ficheros) que produce el programa al ejecutarlo con parámetro 3. Utiliza los nombres que has asignado a los procesos como sus PIDs cuando tengas que escribirlos. Además, puedes explicar brevemente el comportamiento del programa en hoja aparte.

b) Indica claramente la salida (en la terminal y en ficheros) que produce el programa al ejecutarlo con parámetro 5.

36º. Ejercicio

Escribir un programa en C con llamadas al sistema UNIX que ejecute un comando descrito en un fichero de control cuyo nombre recibe como parámetro. El fichero tiene 5 líneas con el siguiente formato:

- Comando1 parámetros
- Comando2 parámetros
- Fichero_entrada
- Fichero_salida
- Fichero_errores

Comando1 y Comando2 pueden tener un número de parámetros entre 0 y 100. Cada línea del fichero puede tener un máximo de 1000 caracteres. Nuestro programa pondrá en ejecución Comando1 y Comando2 enlazados por una tubería. La entrada estandar de Comando1 leerá de Fichero_entrada. La salida estandar de Comando2 escribirá en Fichero_salida. El texto escrito por ambos comandos por su salida de error se guardará en Fichero_errores.

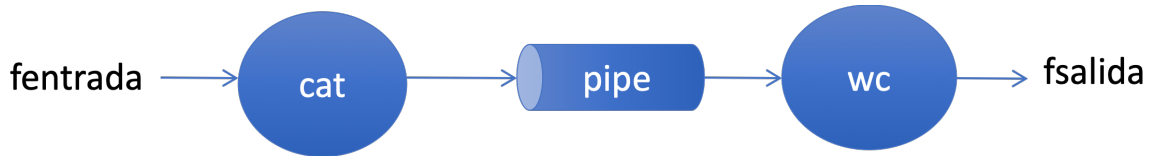
Ejemplo de fichero de control (`fcontrol`):

```
cat -s -v
wc -l
fentrada
fsalida
ferrores
```

Ejemplo de llamada a nuestro programa (`mi_programa`):

```
mi_programa fcontrol
```

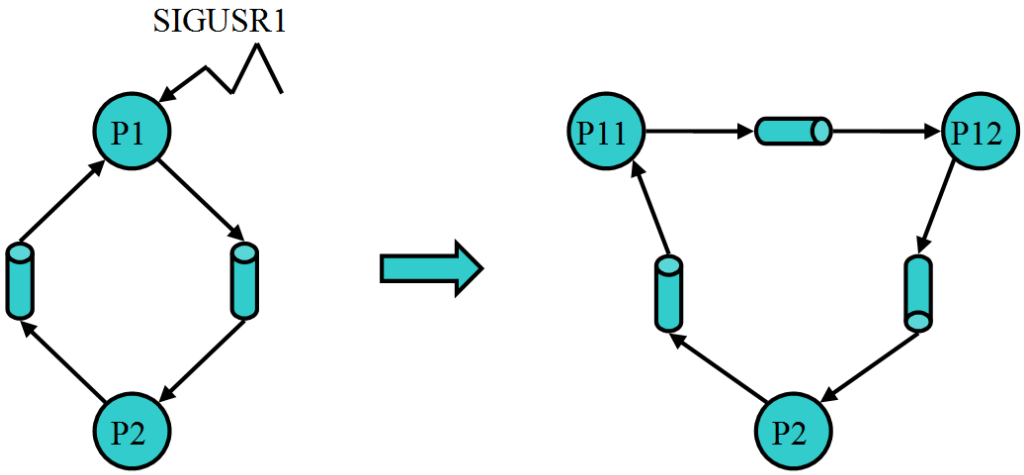
Resultado de la ejecución:



37º. Ejercicio

Escribir un programa en C con llamadas al sistema UNIX que lance 2 procesos (padre, hijo) que se irán pasando cíclicamente (de padre a hijo y de hijo a padre) una pelota mediante tuberías. La pelota consiste en un número que cada proceso incrementará en 1 antes de volverla a enviar. Cada proceso retiene la pelota durante 1 segundo.

Durante la ejecución del programa, cada vez que a un proceso le llegue la señal SIGUSR1 el proceso debe esperar a que le vuelva a llegar la pelota para duplicarse y añadir el nuevo proceso al ciclo de comunicación (si había dos procesos en el ciclo, se pasa a tener 3, si había 3 se pasa a tener 4, etc.)



Ayuda: Se recomienda redirigir los extremos de las tuberías de forma que todos los procesos lean de su entrada estándar y escriban en su salida estándar.

38º. Ejercicio

Diseñar un programa que sume los enteros almacenados en un fichero utilizando varios procesadores. Los enteros se han almacenado en el fichero en binario. Cada entero ocupa 32 bits, lo que coincide con el tamaño del entero en nuestra máquina. El programa recibirá como parámetros el nombre del fichero y el número de procesadores que se quieren utilizar (N). Al finalizar, el programa imprimirá en la salida estándar un mensaje indicando el valor decimal de la suma. Implementarás dos versiones, una basada en procesos independientes y otra basada en hilos del mismo proceso.

Apartado A. La primera versión lanzará N procesos hijo, y utilizará las llamadas al sistema de ficheros para leer la información. En caso de que el número de enteros almacenados en el fichero no sea múltiplo de N, el proceso padre se encargará de procesar el resto. La forma en que los procesos hijos reciben la información del trozo que deben procesar y devuelven la suma parcial calculada, es decisión de diseño. Debes usar la que consideres más apropiada.

Apartado B. La segunda versión lanzará N hilos del mismo proceso además del hilo principal, y usará la llamada mmap() para mapear el fichero en memoria y acceder así a su información. De forma similar al caso anterior, en caso de que el número de enteros almacenados en el fichero no sea múltiplo de N, el hilo principal se encargará de procesar el resto. Igualmente deberás usar la forma que consideres mas apropiada para comunicar a los hilos el trozo de fichero a procesar y para transmitir las sumas parciales de cada hilo.

39º. Ejercicio

Dado el siguiente código:

```
1   #include <stdio.h> <stdlib.h> <unistd.h> <fcntl.h> <signal.h>
2
3   void f(int x){}
4
5   void g(int x){
6        int tmp;
7
8        tmp=dup(4);
9        close(4);dup(6);
10       close(6);dup(tmp);
11       close(tmp);
12  }
13
14  void main(int argc, char *argv[]){
15       int i,j,n,fd[4];
16       char s[6]="F0\0F1";
17       struct sigaction sin;
18       sigset_t seti;
19
20       sin.sa_flags=0;sin.sa_handler = f;
21       sigemptyset(&sin.sa_mask);
22       sigaction(SIGALRM, &sin, NULL);
23       sin.sa_handler = g;
24       sigaction(SIGUSR1, &sin, NULL);
25       for (i=0; i<2; i++){
26            pipe(fd+2*i);
27            if (fork()==0) {
28                 close(fd[1]);
29                 if (i==1) close(fd[3]);
30                 close(1);
31                 creat(s+3*i,0600);
32                 while(read(fd[2*i],&j,sizeof(int))!=0)
33                      fprintf(stdout,"%d\n",j);
34                 exit(0);
35            }
36       }
37       sigemptyset(&seti);sigaddset(&seti,SIGUSR1);
38       n=atoi(argv[1]);
39       j=4;
40       for(i=0;i<n;i++) {
41            write(j,&i,sizeof(int));
42            j=10-j;
43            sigprocmask(SIG_BLOCK,&seti,NULL);
44            alarm(1);
45            pause();
46            sigprocmask(SIG_UNBLOCK,&seti,NULL);
47       }
48  }
```

Asigna nombres a cada uno de los procesos involucrados (P0, P1, P2, …).

a) Explica brevemente el comportamiento del programa.

b) Indica claramente la salida (en la terminal y en ficheros) que produce el programa al ejecutarlo con argumento 10. Para cada mensaje, indica el proceso que lo produce y el instante de tiempo aproximado en el que se produce.

c) Si en el segundo 3,5 se le manda una señal SIGUSR1 al proceso inicial P0, indica claramente la salida (terminal y ficheros) que produce el programa al ejecutarlo con argumento 10. ¿Y si se le envía la señal SIGUSR1 en el segundo 2,5?

d) Indica en qué se modifica el comportamiento del programa si se eliminan las líneas sigprocmask (líneas 43 y 46).

40º. Ejercicio

Indica qué aparecerá en pantalla y cuál será el contenido del fichero *datos.dat* tras ejecutar el comando que se indica en cada apartado, suponiendo que *prog* es el resultado de compilar el código en C que se ofrece, y que *datos.dat* es un fichero de datos cuyo contenido antes de iniciar cada apartado es el siguiente:

```
01234567890123456789012345678901234567890123456789
```

a)

comando: `prog datos.dat`
siendo prog.c:

```
main(int argc, char *argv[]){
   int fd;
   char buf[6]="mnopq";

   fd=open(argv[1], 2);
   read(fd, buf, 5);
   write(fd,"xx", 2);
   printf("%d\n", argc);
   printf("%s", &buf[2]);
}
```

b)

comando: `prog <datos.dat f`
siendo prog.c:

```
main(int argc, char *argv[]){
   int fd;
   char buf[6]="mnopq";

   fd=open("datos.dat", 2);
   read(0, buf, 2);
   write(fd, "abcdefg", 7);
   read(0, buf, 3);
   printf("%d\n", argc);
   write(1, buf+2, 3);
}
```

c)

comando: `prog >datos.dat`

siendo prog.c:

```
main(int argc, char *argv[]) {
   int fd;
   char buf[6] ="mnopq";
   fd=open("datos.dat", 2);
   read(fd, buf, 2);
   write(1, "abcdefg", 7);
   read(fd, buf, 3);
   printf("%d\n", argc);
   write(1, &buf[0], 5);
}
```

d)

comando: `prog datos.dat`
siendo prog.c:

```
main(int argc, char *argv[]) {
   int fd;
   char buf[6] ="mnopq";

   fd=open(argv[1],2);
   if (!fork())
       write(fd, "abcdefg", 7);
   else{
       wait(NULL);
       read(fd, buf, 2);
       write(fd, "abcdefg", 7);
   }
   printf("%d\n", argc);
   write(1, buf+1, 3);
}
```

e)

comando: `prog datos.dat` siendo prog.c:

```
#include <sys/mman.h>
main(int argc, char *argv[]){
    int fd;
    char *v;

    fd=open(argv[1],2);
    v=mmap(0,20, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
    v[7]='x';
    printf("%d\n", argc);
    write(1, v+2, 10);
}
```