

# PRÁCTICA 2

## Uso de llamadas al sistema

(userland)



## ÍNDICE

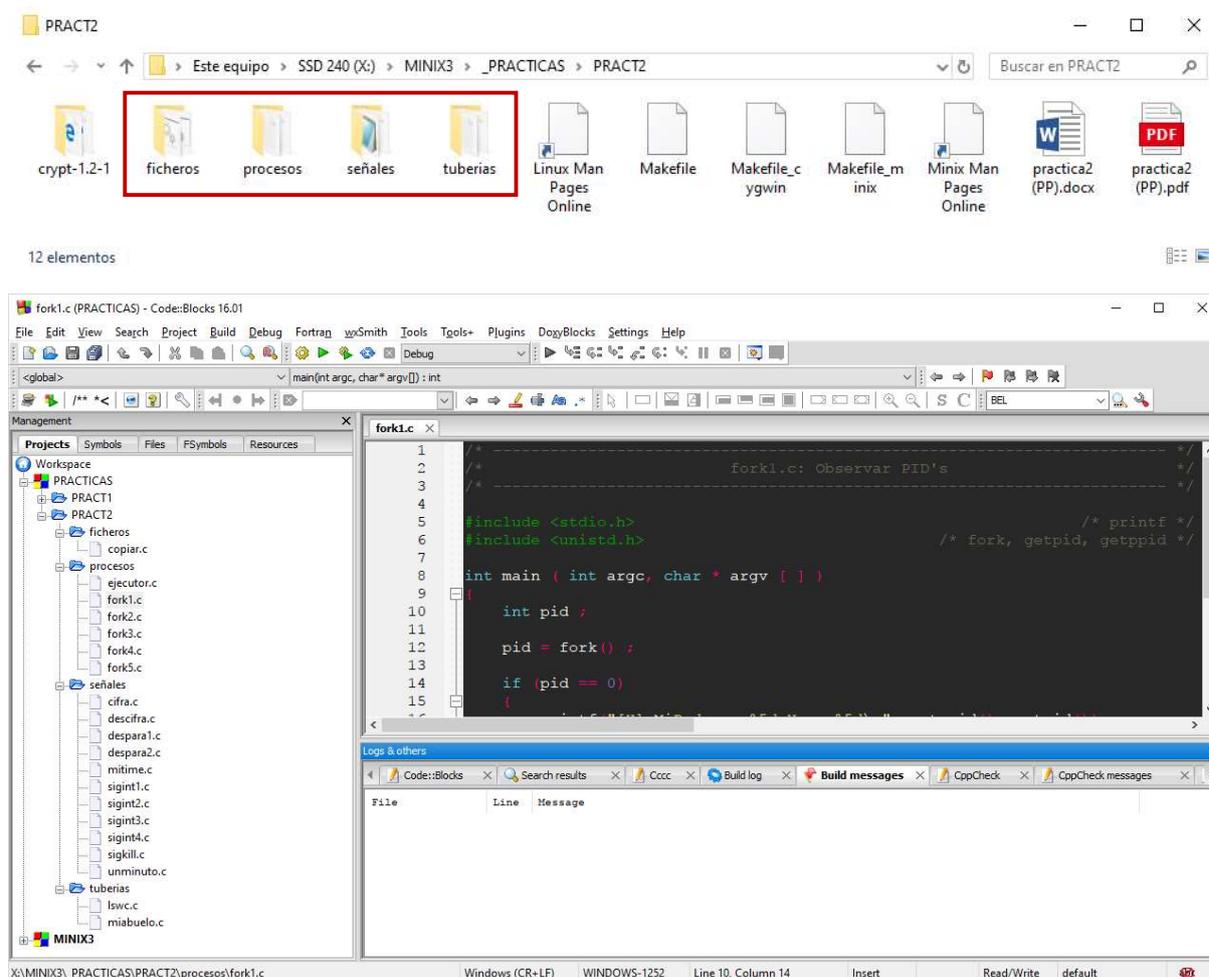
1	OBJETIVOS.....	3
2	LLAMADAS RELACIONADAS CON PROCESOS.....	8
3	LLAMADAS RELACIONADAS CON FICHEROS.....	21
4	LLAMADAS PARA COMUNICAR PROCESOS.....	23
5	LLAMADAS RELACIONADAS CON SEÑALES.....	27
6	PUNTUACIÓN.....	45

## 1 OBJETIVOS

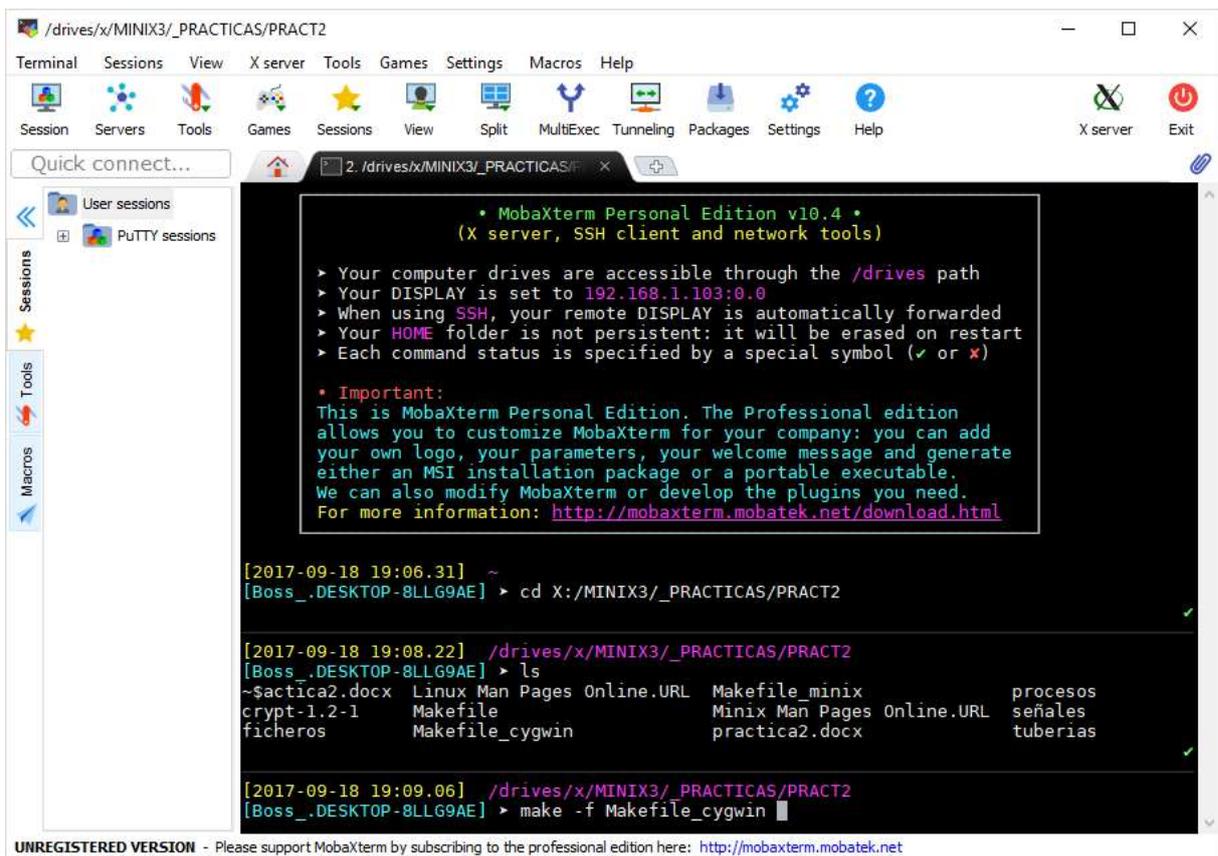
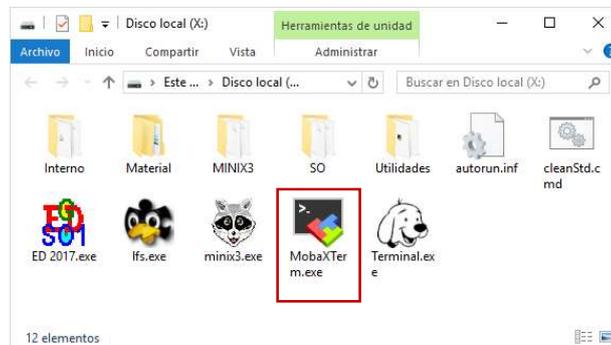
El objetivo de esta práctica es mostrar al alumno cuáles son los **servicios** básicos ofrecidos por el sistema operativo a los **programas de usuario**. Aunque los programas que hacen uso de esos servicios no tienen porqué haber sido escritos en lenguaje de alto nivel, vamos a restringirnos de momento a programas escritos en C que hacen uso de los servicios del sistema llamando a **funciones de interfaz** de las **llamadas al sistema** disponibles a través de **bibliotecas** estándar de C. Para familiarizarnos con esos servicios escribiremos diversos programas ilustrando el uso y aplicación de las llamadas al sistema más representativas, agrupándolas en las siguientes cuatro clases:

- Uso de llamadas relacionadas con procesos: **fork**, **wait**, **exit** y **exec (getpid, getppid)**
- Uso de llamadas relacionadas con ficheros: **open**, **read**, **write** y **close**
- Uso de llamadas para comunicar procesos: **pipe** y **dup2**
- Uso de llamadas relacionadas con señales: **kill**, **sigaction**, **alarm** y **pause**

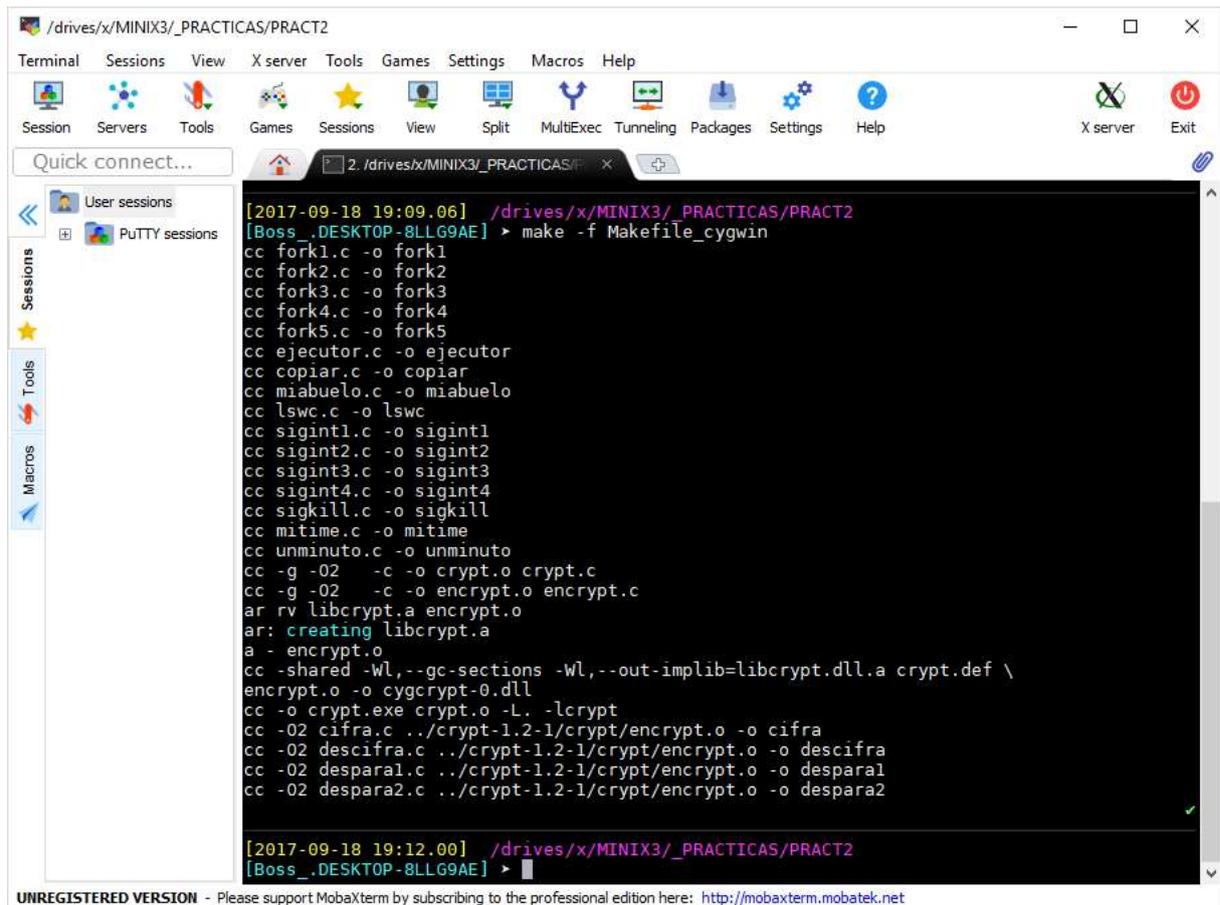
Para cada conjunto de llamadas estudiaremos casos de uso plasmados en varios programas que vamos a tener que entender, compilar, ejecutar y quizás modificar. Estos programas están disponibles en el directorio X:\MINIX3\\_PRACTICAS\PRACT2 del pen drive de prácticas, organizados en cuatro carpetas separadas:



Como se muestra en la figura anterior, podemos ver todos esos programas abriendo con Codeblocks el fichero X:\MINIX3\MINIX3.workspace. Antes de llevar los programas a Minix es conveniente comprobar rápidamente en Windows que tenemos una copia operativa de esos programas. Con ese fin lanzamos X:\MobaXTerm.exe, abrimos un intérprete de comandos bash local, nos movemos con el comando `cd` al directorio X:/MINIX3/\_PRACTICAS/PRACT2 e intentamos compilar todos los programas con el comando `make -f Makefile_cygwin`.



Si la salida del comando `make -f Makefile_cygwin` coincide con la de la figura siguiente, es que partimos de una colección de programas correcta. Podríamos ejecutar los programas compilados desde el intérprete de comandos utilizado, o desde Windows (Cygwin), pero nuestro objetivo es ejecutarlos, a ser posible, bajo Minix, para lo cual tendremos que trasvasar dichos programas a la máquina virtual Minix (soportada por qemu).



```

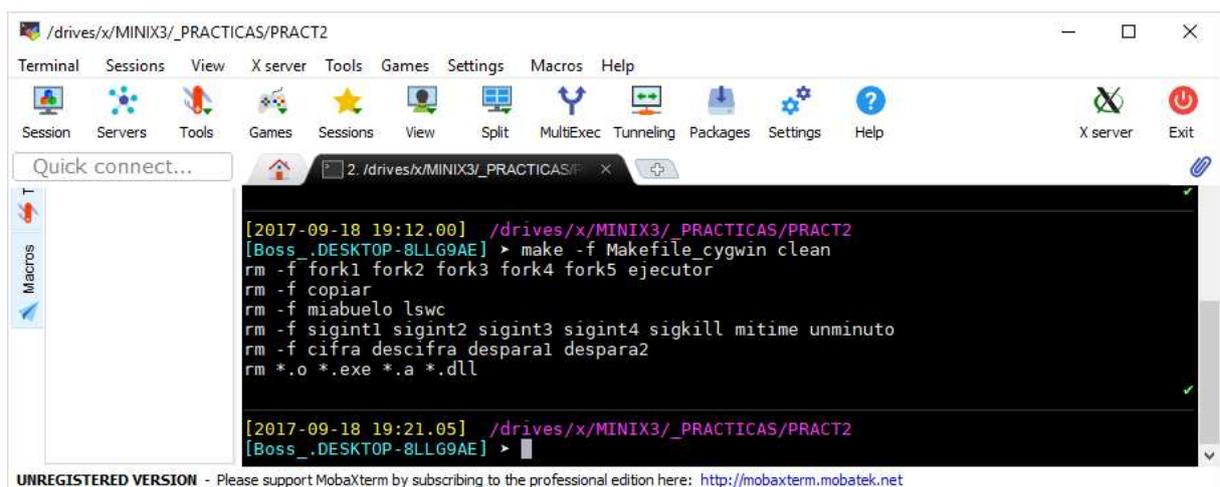
[2017-09-18 19:09.06] /drives/x/MINIX3/_PRACTICAS/PRACT2
[Boss_.DESKTOP-8LLG9AE] > make -f Makefile_cygwin
cc fork1.c -o fork1
cc fork2.c -o fork2
cc fork3.c -o fork3
cc fork4.c -o fork4
cc fork5.c -o fork5
cc ejecutor.c -o ejecutor
cc copiar.c -o copiar
cc miabuelo.c -o miabuelo
cc lswc.c -o lswc
cc sigint1.c -o sigint1
cc sigint2.c -o sigint2
cc sigint3.c -o sigint3
cc sigint4.c -o sigint4
cc sigkill.c -o sigkill
cc mitime.c -o mitime
cc unminuto.c -o unminuto
cc -g -O2 -c -o crypt.o crypt.c
cc -g -O2 -c -o encrypt.o encrypt.c
ar rv libcrypt.a encrypt.o
ar: creating libcrypt.a
a - encrypt.o
cc -shared -Wl,--gc-sections -Wl,--out-implib=libcrypt.dll.a crypt.def \
encrypt.o -o cygcrypt-0.dll
cc -o crypt.exe crypt.o -L. -lcrypt
cc -O2 cifra.c ../crypt-1.2-1/crypt/encrypt.o -o cifra
cc -O2 descifra.c ../crypt-1.2-1/crypt/encrypt.o -o descifra
cc -O2 desparal.c ../crypt-1.2-1/crypt/encrypt.o -o desparal
cc -O2 despara2.c ../crypt-1.2-1/crypt/encrypt.o -o despara2

[2017-09-18 19:12.00] /drives/x/MINIX3/_PRACTICAS/PRACT2
[Boss_.DESKTOP-8LLG9AE] >

```

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: <http://mobaxterm.mobatek.net>

Para llevar a qemu los programas anteriores, conviene eliminar primero los resultados de la compilación anterior sobre Cygwin, para lo cual ejecutamos un comando similar al anterior pero especificando como objetivo, clean: **make -f Makefile\_cygwin clean**.



```

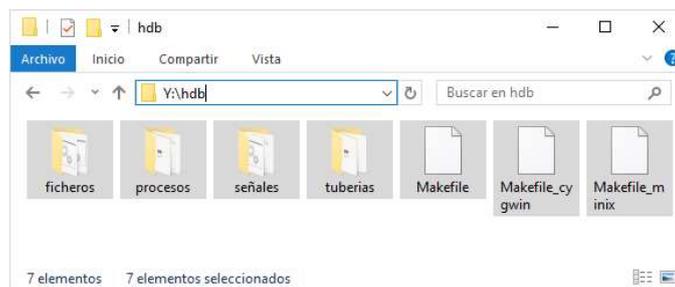
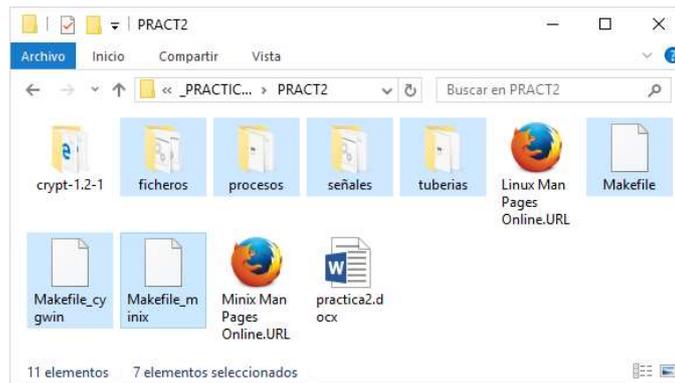
[2017-09-18 19:12.00] /drives/x/MINIX3/_PRACTICAS/PRACT2
[Boss_.DESKTOP-8LLG9AE] > make -f Makefile_cygwin clean
rm -f fork1 fork2 fork3 fork4 fork5 ejecutor
rm -f copiar
rm -f miabuelo lswc
rm -f sigint1 sigint2 sigint3 sigint4 sigkill mitime unminuto
rm -f cifra descifra desparal despara2
rm *.o *.exe *.a *.dll

[2017-09-18 19:21.05] /drives/x/MINIX3/_PRACTICAS/PRACT2
[Boss_.DESKTOP-8LLG9AE] >

```

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: <http://mobaxterm.mobatek.net>

A continuación, utilizando la interfaz gráfica de Windows (el explorador de ficheros) copiamos las cuatro carpetas: ficheros, procesos, señales y tuberías, junto con los tres ficheros Makefile al directorio **Y:\hdb** (%TMP%\-so-\hdb).



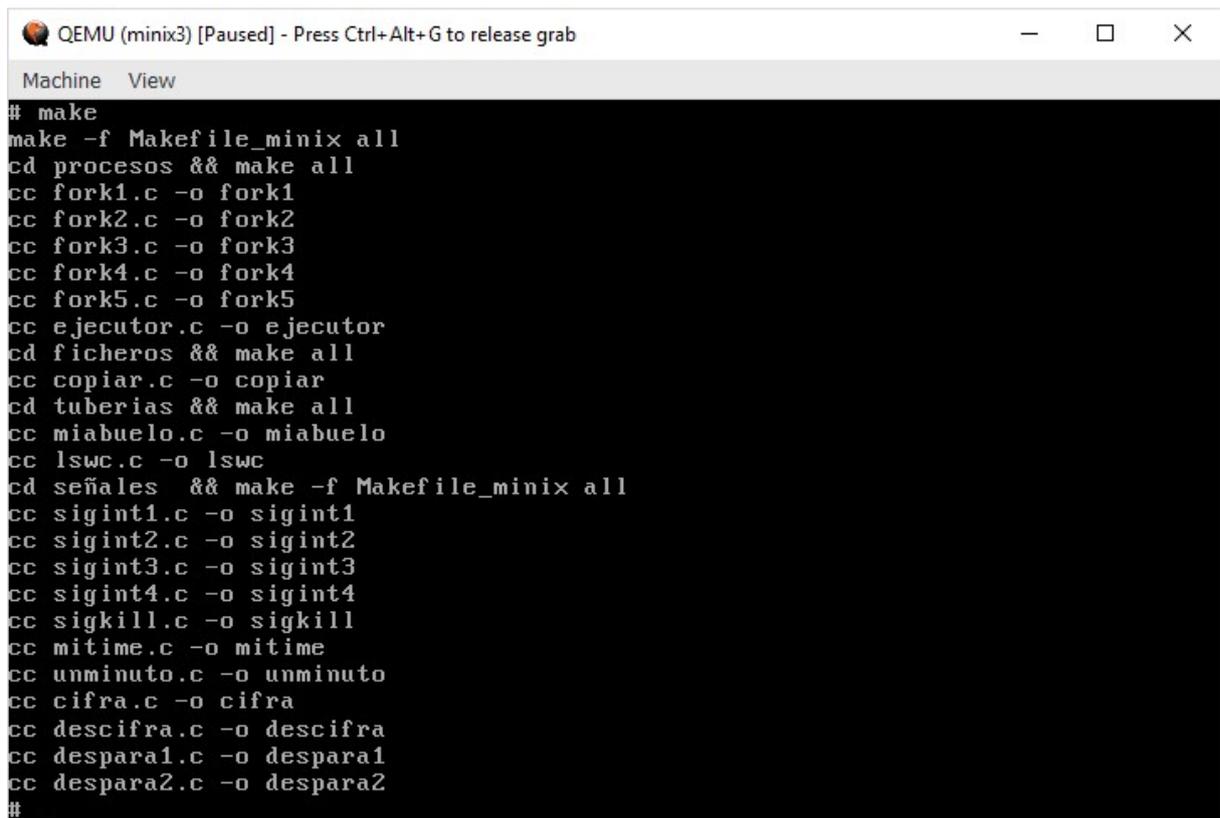
Ahora arrancamos la máquina virtual **X:\minix3.exe** , entramos como root, creamos un directorio `/root/pract2`, nos movemos a ese directorio y copiamos a él todo el contenido de `Y:\hdb`, gracias al comando `mtools copy -s c0d1p0:/* .`

```

QEMU (minix3) - Press Ctrl+Alt+G to release grab
Machine  View
To get rid of this message, edit /etc/motd.
# ls
.lashrc  .ellepro.b1  .ellepro.e  .exerc  .profile  .vimrc
# mkdir pract2
# cd pract2/
# mtools copy -s c0d1p0:/* .
# ls
Makefile          Makefile_minix  ficheros        tuberias
Makefile_cygwin  SES~1           procesos
# mv SES* señales
# ls
Makefile          Makefile_minix  procesos        tuberias
Makefile_cygwin  ficheros        señales
# ls -l
total 8
-rw-r--r-- 1 root  operator   166 Sep 18 18:31 Makefile
-rw-r--r-- 1 root  operator   429 Sep 18 12:44 Makefile_cygwin
-rw-r--r-- 1 root  operator   139 Sep 18 13:22 Makefile_minix
drwxr-xr-x 2 root  operator   384 Sep 18 19:31 ficheros
drwxr-xr-x 2 root  operator   576 Sep 18 19:31 procesos
drwxr-xr-x 2 root  operator  1408 Sep 18 19:31 señales
drwxr-xr-x 2 root  operator   320 Sep 18 19:31 tuberias
# make

```

Como se ve hemos tenido que renombrar el directorio `SES~1` que corresponde al directorio `señales`. Finalmente tenemos ya todos los programas en Minix, por lo que procedemos a su compilación con el comando `make`.



```
QEMU (minix3) [Paused] - Press Ctrl+Alt+G to release grab
Machine View
# make
make -f Makefile_minix all
cd procesos && make all
cc fork1.c -o fork1
cc fork2.c -o fork2
cc fork3.c -o fork3
cc fork4.c -o fork4
cc fork5.c -o fork5
cc ejecutor.c -o ejecutor
cd ficheros && make all
cc copiar.c -o copiar
cd tuberias && make all
cc miabuelo.c -o miabuelo
cc lswc.c -o lswc
cd señales && make -f Makefile_minix all
cc sigint1.c -o sigint1
cc sigint2.c -o sigint2
cc sigint3.c -o sigint3
cc sigint4.c -o sigint4
cc sigkill.c -o sigkill
cc mitime.c -o mitime
cc unminuto.c -o unminuto
cc cifra.c -o cifra
cc descifra.c -o descifra
cc despara1.c -o despara1
cc despara2.c -o despara2
# _
```

Con esto hemos terminado de llevar los programas a Minix y de comprobar que están en perfecto estado sintáctico. Estamos por tanto listos para comenzar a estudiar su comportamiento en tiempo de ejecución.

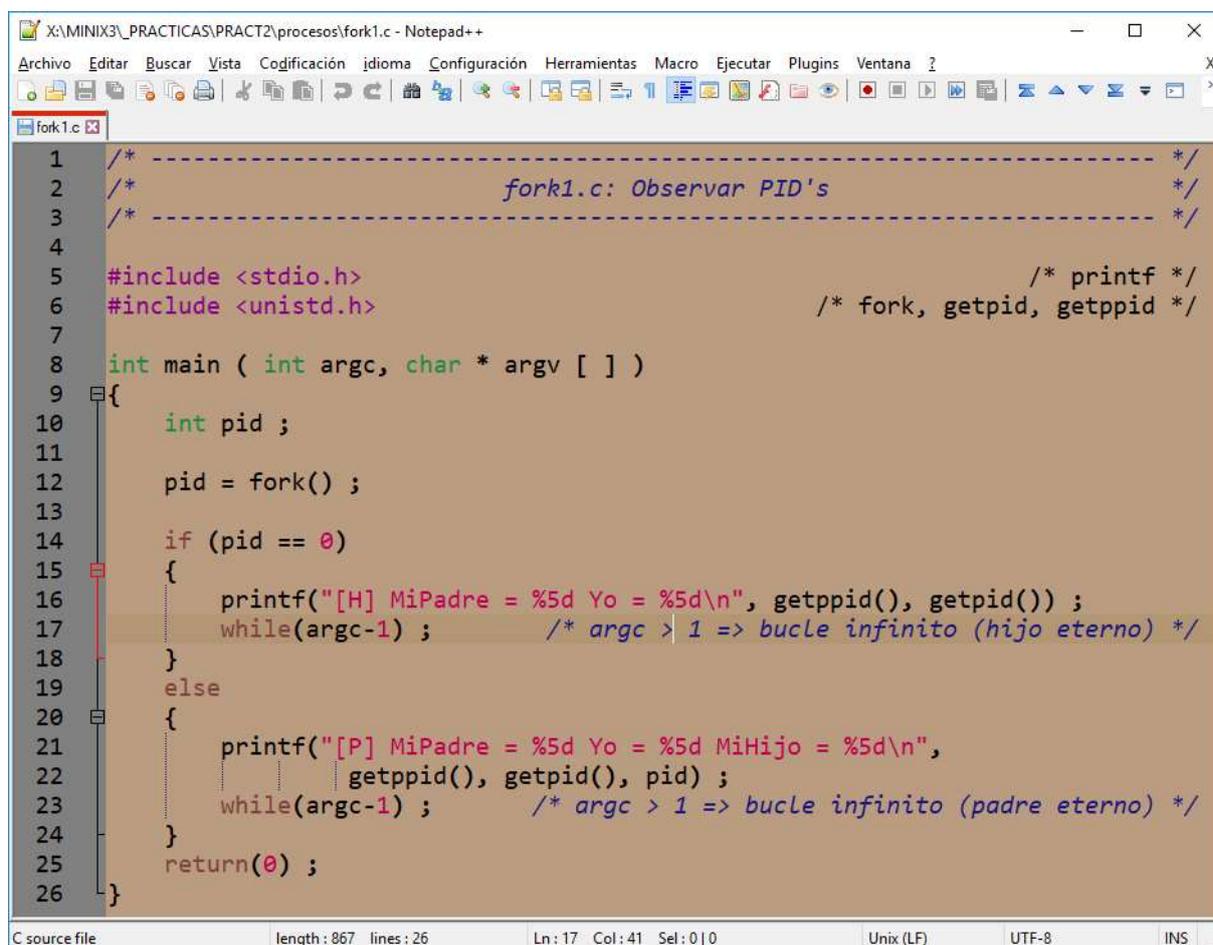
## 2 LLAMADAS RELACIONADAS CON PROCESOS

Para comprender cómo se invocan las **rutinas de interfaz**, así como la funcionalidad de algunas de las **llamadas al sistema** operativo relacionadas con procesos, vamos a compilar y ejecutar los programas que se indican a continuación:

1. fork1.c Observar PID's
2. fork2.c Observar PID's retrasando el mensaje escrito en pantalla por el hijo
3. fork3.c Observar PID's esperando a que termine el hijo
4. fork4.c Observar PID's con terminación explícita del hijo
5. fork5.c Observar los efectos de la multiplexación de la CPU
6. ejecutor.c Una mini shell

Todos estos programas están disponibles originalmente en la carpeta **procesos** del directorio X:\MINIX3\\_PRACTICAS\PRACT2 del pendrive de prácticas, y en el apartado anterior hemos hecho una copia al directorio /root/pract2 de MINIX.

- Arrancar MINIX pinchando sobre el icono X:\minix3.exe.
- Comprobar que en /root/pract2/procesos están los 6 ficheros.
- Echar un vistazo al programa fork1.c



```
1  /* ----- */
2  /*                fork1.c: Observar PID's                */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <unistd.h>                                   /* fork, getpid, getppid */
7
8  int main ( int argc, char * argv [ ] )
9  {
10     int pid ;
11
12     pid = fork() ;
13
14     if (pid == 0)
15     {
16         printf("[H] MiPadre = %5d Yo = %5d\n", getppid(), getpid()) ;
17         while(argc-1) ; /* argc > 1 => bucle infinito (hijo eterno) */
18     }
19     else
20     {
21         printf("[P] MiPadre = %5d Yo = %5d MiHijo = %5d\n",
22                getppid(), getpid(), pid) ;
23         while(argc-1) ; /* argc > 1 => bucle infinito (padre eterno) */
24     }
25     return(0) ;
26 }
```

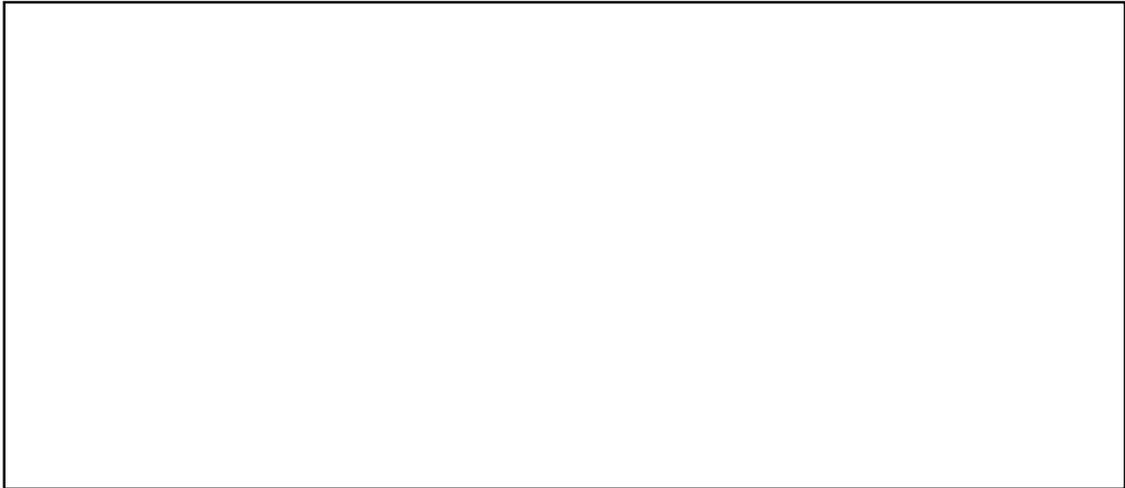
- ¿Qué **rutinas de interfaz** de llamadas al sistema operativo utiliza este programa?

- Si se duda sobre la funcionalidad de alguna de estas llamadas al SO, consultar el manual en línea (comando [man](#)) o comentarlo con el profesor. Escribir a continuación la salida que creéis que aparecerá por pantalla al ejecutar este programa (inventarse los identificadores de proceso) precisando el hecho de si **escribe** antes el padre o el hijo:

¿Tras el fork se **ejecutó** antes el padre o el hijo? [Cuestionario 1]

- Compilar el programa: **cc fork1.c -o fork1** (make fork1, make clean, make)
- Ejecutar este programa con **./fork1 eterno**. ¿Ha salido lo que se esperaba?

- Ejecutar en otra consola (Alt + F2) el comando **ps -l** y escribir (y explicar el significado de) las líneas correspondientes al proceso padre e hijo (consultar el manual: **man ps**).



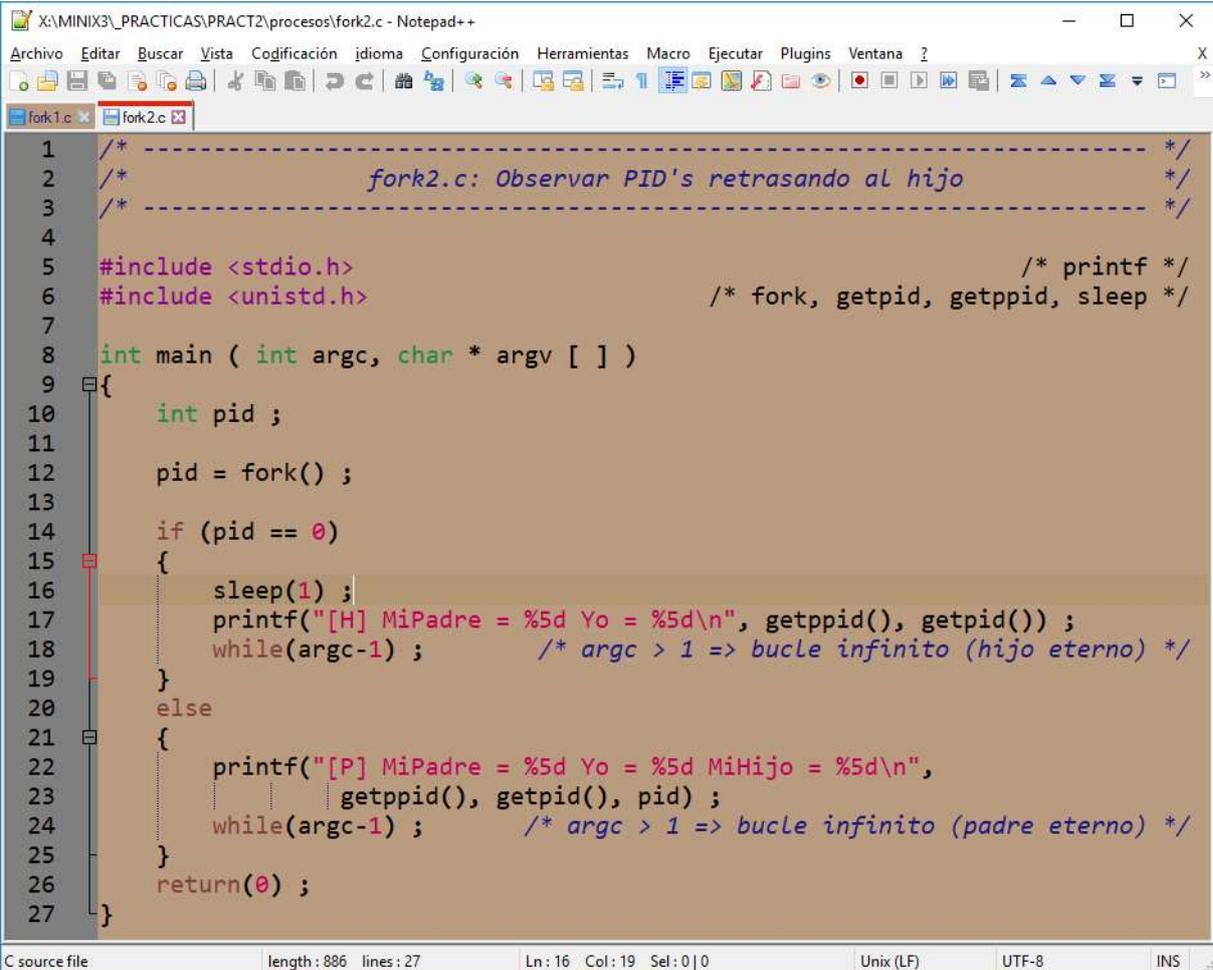
- Matar al proceso padre con el comando **kill <PID>**, ejecutar **ps -l** y escribir y explicar el significado de la línea correspondiente al proceso hijo.



- Matar al proceso hijo con el comando **kill <PID>**.

[Repetir: 1) matando al hijo antes que al padre, y 2) ejecutando **./fork1 eterno &**]

- Echar ahora un vistazo al programa **fork2.c**



```
1  /* ----- */
2  /*          fork2.c: Observar PID's retrasando al hijo          */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <unistd.h>                                   /* fork, getpid, getppid, sleep */
7
8  int main ( int argc, char * argv [ ] )
9  {
10     int pid ;
11
12     pid = fork() ;
13
14     if (pid == 0)
15     {
16         sleep(1) ;
17         printf("[H] MiPadre = %5d Yo = %5d\n", getppid(), getpid()) ;
18         while(argc-1) ; /* argc > 1 => bucle infinito (hijo eterno) */
19     }
20     else
21     {
22         printf("[P] MiPadre = %5d Yo = %5d MiHijo = %5d\n",
23                getppid(), getpid(), pid) ;
24         while(argc-1) ; /* argc > 1 => bucle infinito (padre eterno) */
25     }
26     return(0) ;
27 }
```

- Observar que se ha introducido un **retardo pasivo** de un segundo, **sleep(1)**, en el código del proceso hijo (ver [sleep](#)).
- Compilar (make fork2) y ejecutar este programa con el comando **./fork2 eterno &**.
- ¿Qué se observa al ejecutar este programa?

- Matar a los procesos padre e hijo con **kill <PID1> <PID2>**.

- Echar ahora un vistazo al programa **fork3.c** cuyo aspecto es el siguiente:

```

1  /* ----- */
2  /*      fork3.c: Observar PID's esperando a que termine el hijo      */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <unistd.h>                                   /* fork, getpid, getppid */
7  #include <sys/wait.h>                                 /* wait */
8
9  int main ( int argc, char * argv [ ] )
10 {
11     int pid, estado ;
12
13     pid = fork() ;
14
15     if (pid == 0)
16     {
17         printf("[H] MiPadre = %5d Yo = %5d\n", getppid(), getpid()) ;
18         while(argc-1) ;          /* argc > 1 => bucle infinito (hijo eterno) */
19     }
20     else
21     {
22         printf("[P] MiPadre = %5d Yo = %5d MiHijo = %5d\n",
23                getppid(), getpid(), pid) ;
24
25         pid = wait(&estado) ;
26         printf("[P]: Termino el hijo %d con estado %d\n", pid, estado) ;
27         while(argc-1) ;          /* argc > 1 => bucle infinito (padre eterno) */
28     }
29     return(0) ;
30 }

```

- Observar que se ha introducido una espera (*wait*) del padre a que termine un/el hijo.
- Compilar (`make fork3`), y ejecutar este programa con el comando `./fork3 eterno &`.
- Ejecutar el comando `ps -l` y escribir y explicar el significado de la línea correspondiente al proceso padre.

- Matar al proceso hijo con el comando `kill <PID>`.

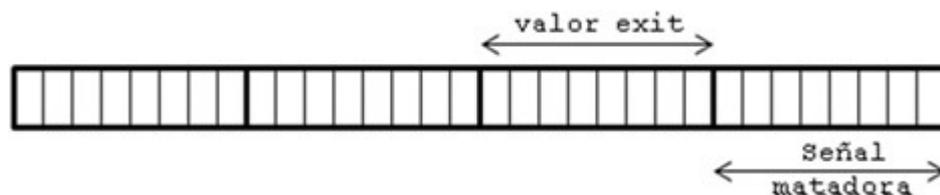
- Ejecutar el comando **ps -l** y escribir y explicar el significado de la línea correspondiente al proceso padre.

- ¿Qué se ha observado en relación con el estado de terminación del proceso hijo?

- Matar al proceso padre con el comando **kill <PID>**.
- Para acabar con esta primera aproximación a la comprensión de las llamadas *fork* y *wait*, vamos a utilizar la llamada **exit** cuyo encabezamiento (ejecutar **man -s 2 exit**) es:

```
void _exit ( int status ) ;
```

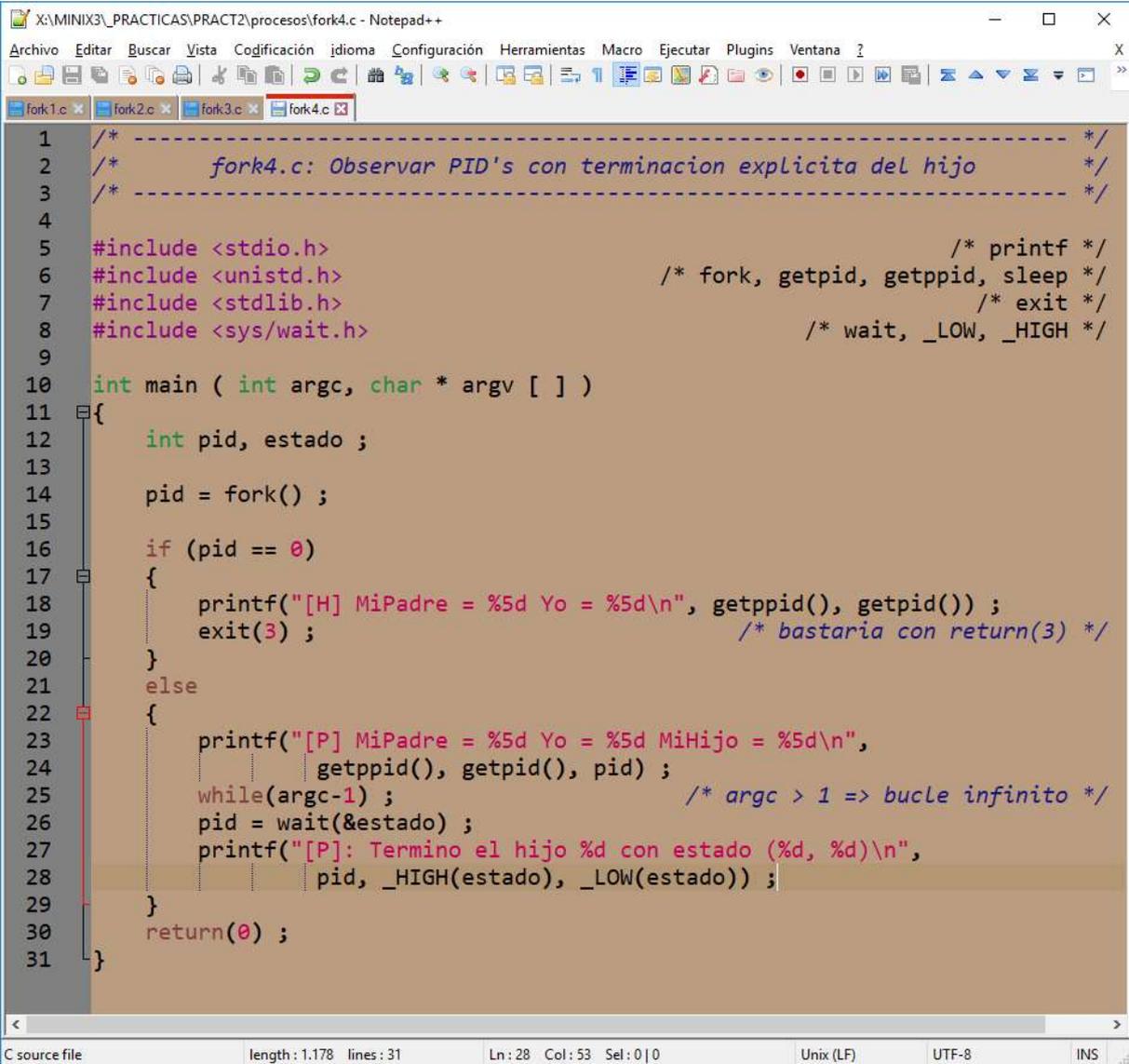
Ahora podemos comprender todavía mejor el comportamiento del programa anterior. Cuando un proceso padre hace *wait*, la variable “estado”, donde se recibe el estado de terminación del hijo, tiene el significado siguiente:



Las macros `_HIGH( )` y `_LOW( )` definidas en `/usr/include/sys/wait.h` nos permiten acceder de forma cómoda a los octetos representativos del estado según que el proceso haya terminado por sí mismo o haya sido matado a consecuencia de una señal.

[**Nota:** recordar de TSO que cuando termina un comando con `exit(status)` podemos conocer el status de terminación con el comando `echo $?`]

- Echar ahora un vistazo al programa **fork4.c** cuyo aspecto es el siguiente:



```
1  /* ----- */
2  /*      fork4.c: Observar PID's con terminacion explicita del hijo      */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <unistd.h>                                   /* fork, getpid, getppid, sleep */
7  #include <stdlib.h>                                  /* exit */
8  #include <sys/wait.h>                                /* wait, _LOW, _HIGH */
9
10 int main ( int argc, char * argv [ ] )
11 {
12     int pid, estado ;
13
14     pid = fork() ;
15
16     if (pid == 0)
17     {
18         printf("[H] MiPadre = %5d Yo = %5d\n", getppid(), getpid()) ;
19         exit(3) ;                                     /* bastaria con return(3) */
20     }
21     else
22     {
23         printf("[P] MiPadre = %5d Yo = %5d MiHijo = %5d\n",
24             getppid(), getpid(), pid) ;
25         while(argc-1) ;                               /* argc > 1 => bucle infinito */
26         pid = wait(&estado) ;
27         printf("[P]: Termino el hijo %d con estado (%d, %d)\n",
28             pid, _HIGH(estado), _LOW(estado)) ;
29     }
30     return(0) ;
31 }
```

- Observar que se ha introducido una terminación explícita del hijo que devuelve el estado “3” (0x00000300) al padre.
- Compilar (make fork4), y ejecutar este programa con el comando **./fork4**.
- ¿Qué ha ocurrido antes, el **exit** del proceso hijo, o el **wait** del proceso padre?

- Para ver qué ocurre cuando el proceso hijo muere sin que el padre esté esperando su terminación, vamos a habilitar el bucle infinito antes del **wait** del padre, cosa que conseguimos ejecutando el comando **./fork4 eterno &**.
- Ejecutar el comando **ps -l** y escribir y explicar el significado de las líneas correspondientes al proceso padre e hijo.



- Matar al proceso hijo y padre con el comando **kill** a ver que pasa.
- Si queda alguna duda en relación con *fork*, *wait* o *exit*, preguntar al profesor.

Ahora vamos a observar el efecto de la multiplexación de la CPU realizada por el sistema operativo para dar la ilusión de que se están ejecutando dos o más procesos de forma concurrente. Para ello, utilizaremos un programa en el que el padre creará un proceso (hijo) y ambos se pondrán a escribir caracteres indefinidamente por la pantalla: el padre escribirá sólo cifras (**0..9**) y el hijo escribirá sólo letras (**a..z**). Tanto las cifras como las letras se escribirán de forma ordenada y circular, de manera de después del **9** se escribirá un **0** y después de la **z** se escribirá una **a**.

- Echar un vistazo al programa **fork5.c** siguiente:

```

1  /* ----- */
2  /*   fork5.c: Observar Los efectos de La multiplexacion de La CPU   */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf, scanf */
6  #include <unistd.h>                                   /* fork, write */
7
8  void retardar ( int vueltas )
9  {
10     int global [100], aux, i ;
11     while (vueltas > 0)
12     {
13         vueltas-- ;
14         i = vueltas % 100 ;
15         aux = global[99 - i] ;
16         global[99 - i] = global[i] ;
17         global[i] = aux ;
18     }
19 }
20
21 void imprimirIntervalo ( char min, char max, int retardo )
22 {
23     char car = min ;
24     while (min <= max)                                /* o no se entra o bucle infinito */
25     {
26         printf("%c", car) ;                            /* write(STDOUT_FILENO, &car, 1) ; */
27         if (++car > max)
28             car = min ;
29         retardar(retardo) ;
30     }
31 }
32
33 int main ( int argc, char * argv [ ] )
34 {
35     int pid, vueltasDeRetardo ;
36
37     if ((argc != 2) || ((scanf(argv[1], "%d", &vueltasDeRetardo) != 1)))
38     {
39         vueltasDeRetardo = 10000 ;                    /* valor por defecto */
40     }
41
42     pid = fork() ;
43
44     if (pid == 0)
45         imprimirIntervalo('a', 'z', vueltasDeRetardo) ;    /* Letras */
46     else
47         imprimirIntervalo('0', '9', vueltasDeRetardo) ;    /* cifras */
48     return(0) ;
49 }

```

C source file      length: 1.415    lines: 49      Ln: 42    Col: 19    Sel: 0 | 0      Unix (LF)      UTF-8      INS

- Compilar el programa con el comando **cc fork5.c -o fork5** (make fork5).

- Ejecutar `X:\Utilidades\hypertrm.exe` y conectar el terminal a MINIX.
- Ejecutar el programa redirigiendo su salida al terminal conectado al primer puerto serie (`/dev/tty00`) con el comando `./fork5 20000 > /dev/tty00 &`. Al cabo de varios segundos observando la salida por el terminal, mate al padre y al hijo con el comando `kill`.
- Inspeccione la salida de `./fork5` enviada al terminal ¿Qué se observa en dicha salida?

- Cuente el número de letras (a..z) que consigue escribir seguidas el proceso hijo.

- ¿Ese número es siempre el mismo en las diferentes secuencias de letras seguidas que escribe el proceso hijo? ¿depende del parámetro **vuelasDeRetardo** de `fork5`?

- Cuente el número de cifras (0 .. 9) que consigue escribir seguidas el proceso padre.

- ¿Ese número es siempre el mismo en las diferentes secuencias de cifras seguidas que escribe el proceso padre? ¿depende del parámetro **vuelasDeRetardo** de `fork5`?

- Dar una explicación razonada de lo observado (sugerencia: ver comentario del [printf](#)).

Para terminar este apartado, vamos a probar el ejemplo de un ejecutor (intérprete) de comandos muy simple como el contado en la teoría y que hemos llamado **ejecutor.c** (ver man [execl](#)).

```

1  /* ----- */
2  /*           ejecutor.c: Una mini shell           */
3  /* ----- */
4
5  #include <stdio.h>                               /* printf, scanf */
6  #include <unistd.h>                               /* fork, execl, access, R_OK, X_OK */
7  #include <stdlib.h>                              /* exit */
8  #include <sys/wait.h>                            /* wait */
9  #include <string.h>                              /* strcmp */
10
11 int main ( void )
12 {
13     int pid, estado ;
14     char programa [ 20 ] ;
15
16     printf("> " ) ;                               /* prompt */
17     scanf("%s", programa ) ;
18
19     while (strcmp(programa, "fin") != 0) /* se sale si el comando es "fin" */
20     {
21         if (strcmp(programa, "ver") == 0) /* ejemplo de comando interno */
22             printf(" version de ejecutor (P.P.L.R.)\n") ;
23         else if (access(programa, R_OK | X_OK) == 0) /* comando externo */
24         {
25             if (fork() == 0)
26             {
27                 estado = execl(programa, programa, NULL) ;
28                 printf("Error en %s => %d\n", programa, estado) ;
29                 exit(1) ;                          /* bastaria con return(1) */
30             }
31             else
32             {
33                 pid = wait(&estado) ;
34             }
35         }
36         else
37             printf(" comando inexistente\n") ;
38         printf("> " ) ;
39         scanf("%s", programa ) ;
40     }
41     return(0) ;
42 }

```

C source file      length : 1.559    lines : 42      Ln : 27    Col : 31    Sel : 0 | 0      Unix (LF)    UTF-8    INS

- Compilar (make ejecutor) y ejecutar este programa. Cuando aparezca el “prompt” – indicador de nuestra *minishell* de que desea atendernos–, teclear **fork1**. Deberá ejecutarse el programa **fork1** tal y como se ejecutó en páginas anteriores.

- Introducir ante el prompt de ejecutor el nombre de un programa que no exista, como por ejemplo **tenedor** o **cuchara**. ¿Qué sucede?

- Introducir ante el prompt de ejecutor, el comando **ps**. ¿Por qué no funciona?

- Buscar alguna manera alternativa de ejecutar el comando **ps** desde nuestro ejecutor de comandos ([whereis ps](#)). Si no se encuentra ninguna consultar con el profesor. Anotar la solución encontrada:

- Ejecutar ahora el ejecutor desde el propio ejecutor. ¿Qué pasa? Comentar el efecto:

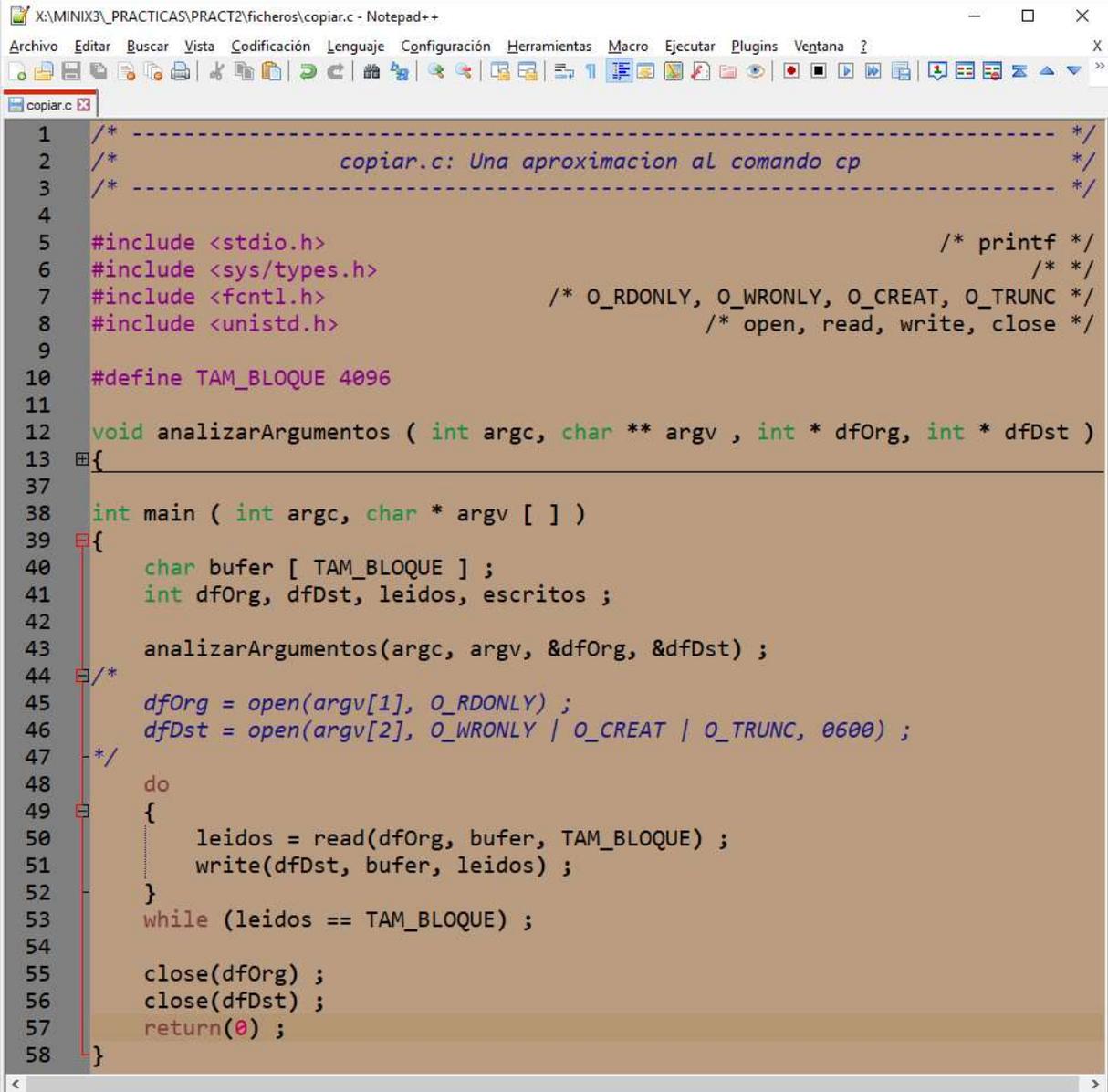
- Modificar ligeramente el programa **ejecutor.c** para que informe justo antes del *prompt*, del identificador del proceso ejecutor y así tener la certeza de qué proceso ejecutor nos está atendiendo. Indicar las modificaciones.



### 3 LLAMADAS RELACIONADAS CON FICHEROS

En esta sección tan solo probaremos el ejemplo contado en clase de teoría que hace una implementación elemental del comando `cp`.

- Entrar en MINIX y echar un vistazo al programa `copiar.c` cuyo aspecto es el siguiente:



```
1  /* ----- */
2  /*           copiar.c: Una aproximacion al comando cp           */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <sys/types.h>                                 /* */
7  #include <fcntl.h>                                     /* O_RDONLY, O_WRONLY, O_CREAT, O_TRUNC */
8  #include <unistd.h>                                   /* open, read, write, close */
9
10 #define TAM_BLOQUE 4096
11
12 void analizarArgumentos ( int argc, char ** argv , int * dfOrg, int * dfDst )
13 {
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38 int main ( int argc, char * argv [ ] )
39 {
40     char bufer [ TAM_BLOQUE ] ;
41     int dfOrg, dfDst, leidos, escritos ;
42
43     analizarArgumentos(argc, argv, &dfOrg, &dfDst) ;
44
45     /*
46     dfOrg = open(argv[1], O_RDONLY) ;
47     dfDst = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0600) ;
48     */
49     do
50     {
51         leidos = read(dfOrg, bufer, TAM_BLOQUE) ;
52         write(dfDst, bufer, leidos) ;
53     }
54     while (leidos == TAM_BLOQUE) ;
55
56     close(dfOrg) ;
57     close(dfDst) ;
58     return(0) ;
59 }
```

C source file      length: 1.615    lines: 58      Ln: 57    Col: 16    Sel: 0 | 0      Windows (CR LF)    UTF-8      INS

- Observar los dos usos que hacemos de la llamada al sistema `open` cuya sinopsis es la siguiente:

```
int open ( const char * path, int flags [, mode_t mode] )
```

Esta función abre el fichero cuyo camino (*path*) absoluto o relativo se indica, devolviendo el correspondiente **descriptor (de fichero)** de la tabla de ficheros abiertos del proceso. En caso de creación el descriptor es el **menor descriptor disponible**.

Los valores que pueden utilizarse como “*flags*” son los siguientes:

O_RDONLY	Sólo lectura
O_WRONLY	Sólo escritura
O_RDWR	Lectura y escritura
O_APPEND	Se sitúa al final del fichero
O_CREAT	Crea el fichero si no existe
O_TRUNC	Trunca el tamaño del fichero a cero
O_EXCL	Error con O_CREAT si existe el fichero

El parámetro “*mode*” indica el modo de protección del fichero si es de nueva creación.

- Compilar (make copiar) y ejecutar el programa para observar su correcto funcionamiento. Probar con el comando “./copiar copiar.c copiarbis.c”.
- Comprobar que se ha copiado correctamente (utilizar comandos: **ls -l** ; **cat** y **diff**).
- Ahora vamos a comprobar la diferencia entre escribir BLOQUE a BLOQUE frente a carácter a carácter. Para ello, vamos a copiar un fichero de mayor tamaño. Teclear el comando “**time ./copiar /boot/image\_big mi\_image**” y anotar el tiempo que tarda:

- Ahora editar el programa **copiar.c** y cambiar el tamaño de BLOQUE de 4096 a 1. Compilarlo, borrar mi\_image, y teclear de nuevo “**time ./copiar /boot/image\_big mi\_image**”. Anotar el tiempo para constatar la diferencia.

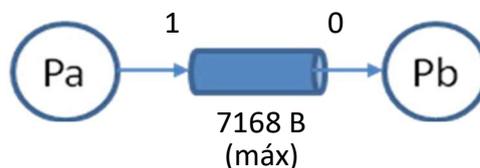
## 4 LLAMADAS PARA COMUNICAR PROCESOS

Una forma de comunicar procesos pesados (que no pueden comunicarse a través de una memoria común compartida) es el uso de ficheros especiales denominados tuberías (*pipes*) que se crean con la llamada al sistema [pipe](#), cuya sinopsis es la siguiente:

```
int pipe ( int p_fd [ 2 ] );
```

Esta llamada crea un pseudofichero accesible a través de dos descriptores de fichero que son inicializados por la propia llamada:

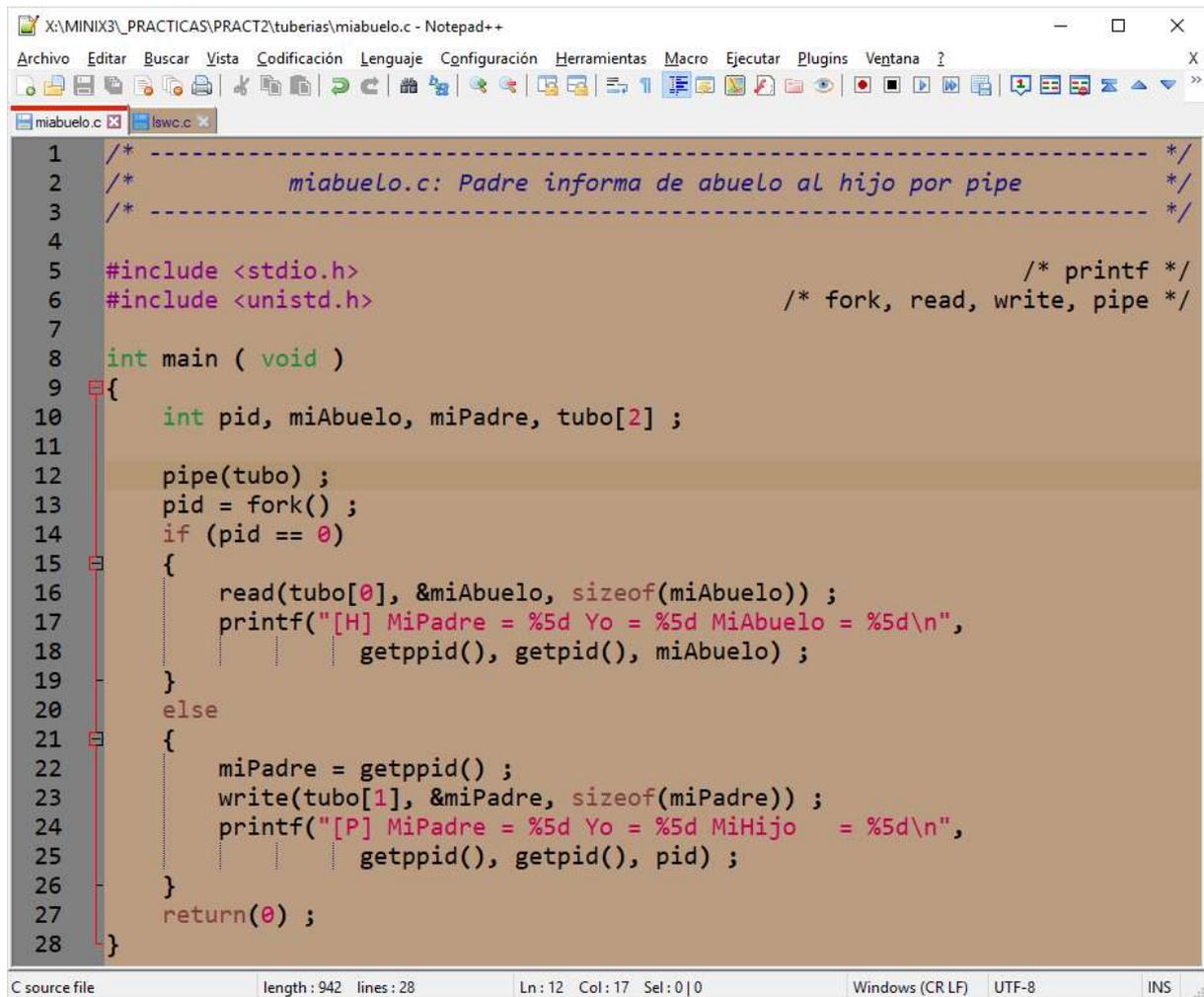
- p\_fd[0] sólo lectura (O\_RDONLY)
- p\_fd[1] sólo escritura (O\_WRONLY)



La escritura en una tubería (a través de p\_fd[1]) permite ir almacenando bytes en la tubería (hasta un máximo de 7.168 bytes en MINIX 3). Si la tubería se llena, la siguiente escritura ocasiona el bloqueo del proceso escritor hasta que puedan almacenarse nuevos bytes en la tubería. Las lecturas de la tubería (a través de p\_fd[0]) leerán en orden los caracteres previamente almacenados por las escrituras, permitiendo el desbloqueo de algún proceso que pudiera estar bloqueado. Una lectura de una tubería vacía ocasiona el bloqueo del proceso lector, hasta que algún otro proceso introduzca nuevos bytes en la tubería.

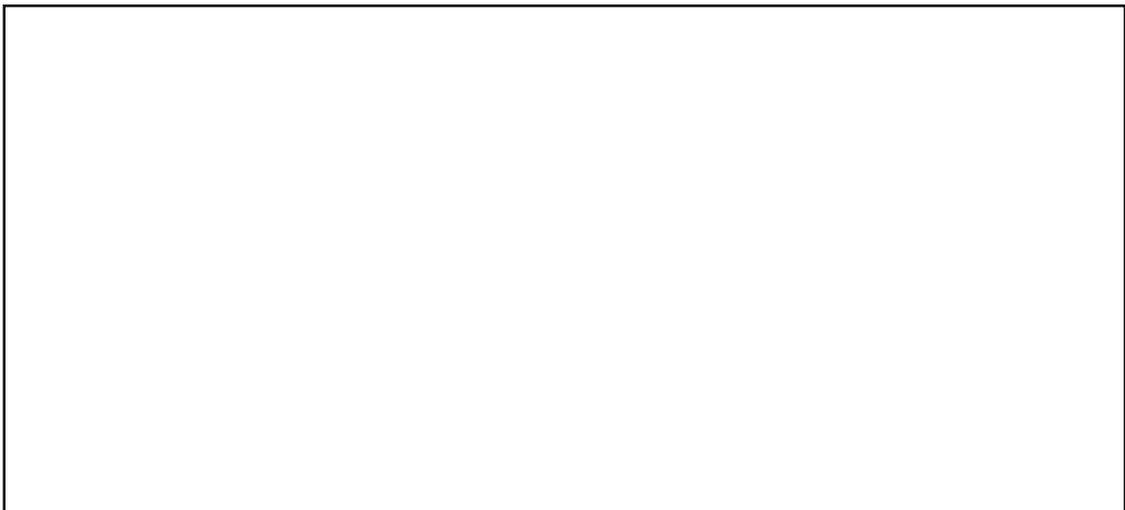
Para que dos procesos se comuniquen a través de una tubería lo único que se necesita añadir es conseguir que un proceso “**Pa**” utilice el extremo de escritura (la entrada a la tubería) y el otro proceso “**Pb**” el extremo de lectura (la salida de la tubería) de una misma tubería. Vamos a utilizar este mecanismo para que un proceso Padre le comunique al Hijo –a través de un pipe común– quien es su abuelo.

- Echar un vistazo al programa **miabuelo.c** cuyo aspecto es el siguiente:



```
1  /* ----- */
2  /*      miabuelo.c: Padre informa de abuelo al hijo por pipe      */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <unistd.h>                                   /* fork, read, write, pipe */
7
8  int main ( void )
9  {
10     int pid, miAbuelo, miPadre, tubo[2] ;
11
12     pipe(tubo) ;
13     pid = fork() ;
14     if (pid == 0)
15     {
16         read(tubo[0], &miAbuelo, sizeof(miAbuelo)) ;
17         printf("[H] MiPadre = %5d Yo = %5d MiAbuelo = %5d\n",
18             getpid(), getpid(), miAbuelo) ;
19     }
20     else
21     {
22         miPadre = getpid() ;
23         write(tubo[1], &miPadre, sizeof(miPadre)) ;
24         printf("[P] MiPadre = %5d Yo = %5d MiHijo = %5d\n",
25             getpid(), getpid(), pid) ;
26     }
27     return(0) ;
28 }
```

- Compilar y ejecutar el programa anterior y anotar lo que sale por pantalla para comprobar que el funcionamiento es el correcto:



Por último, vamos a utilizar la llamada al sistema [dup2](#) cuya sinopsis es la siguiente:

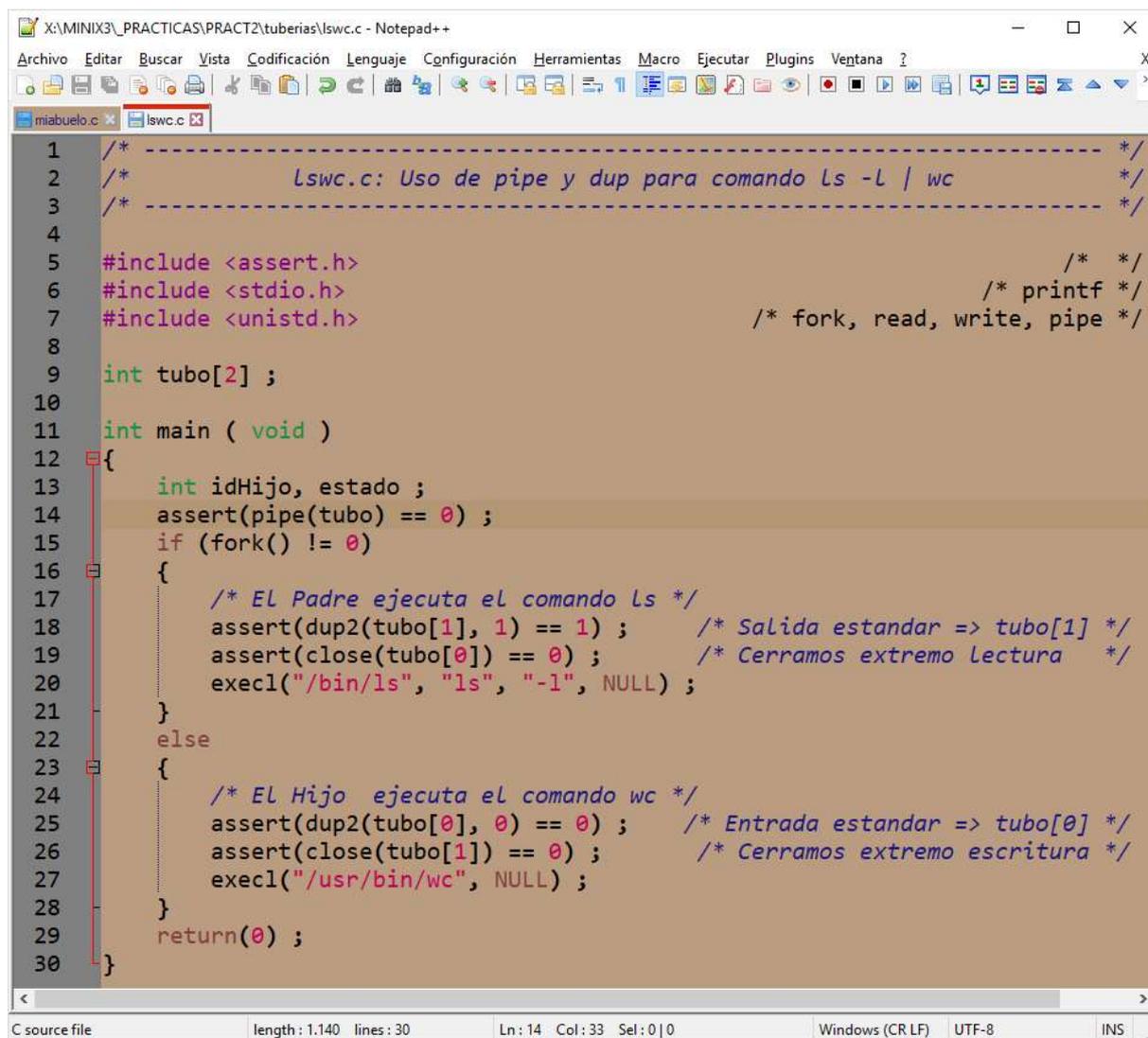
```
int dup2 ( int old_fd, int new_fd )
```

Esta llamada duplica el descriptor de fichero *old\_fd* sobre el descriptor de fichero indicado como *new\_fd*. Si *new\_fd* se corresponde con un fichero previamente abierto, lo cierra antes de hacer la duplicación. Tras ejecutarse **dup2**(*old\_fd*, *new\_fd*), es indiferente utilizar *old\_fd* o *new\_fd* para acceder al fichero que inicialmente sólo manipulábamos a través de *old\_fd*.

Con esta facilidad y cierta argucia, como veremos más adelante, podemos redirigir la entrada/salida/error estándar del proceso (descriptores de fichero 0, 1 y 2) para que esa redirección sea heredada por un eventual proceso hijo.

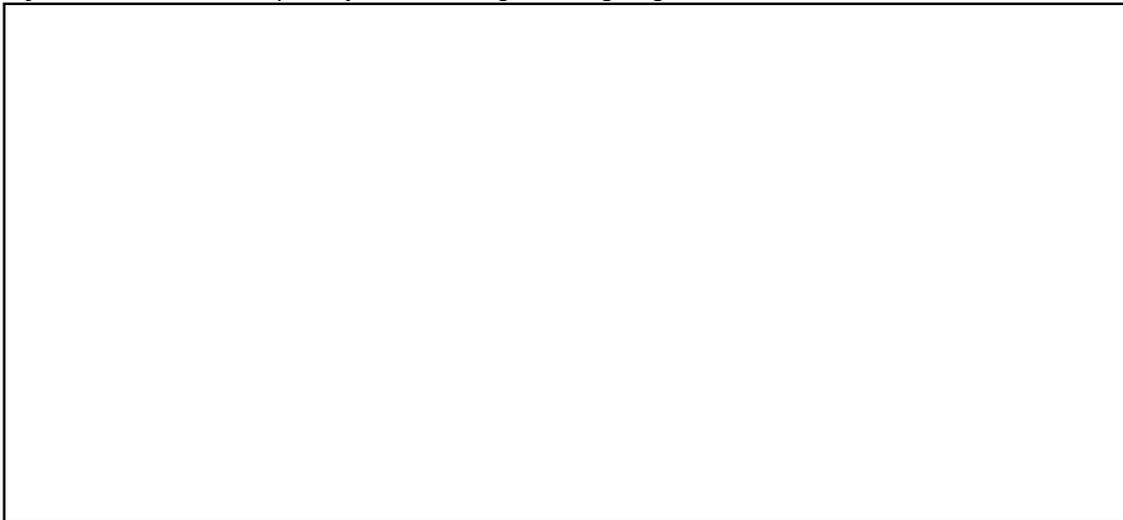
Vamos a escribir un programa que se comporte como el comando: **ls -l | wc**. Para ello, el proceso padre creará un proceso hijo de tal forma que ambos queden conectados mediante una tubería. El padre ejecutará el primer comando “**ls -l**” y el hijo el segundo comando “**wc**”. Lo único que falta es conseguir que la salida estándar (descriptor 1) del padre –por donde escribe el comando **ls**– se redirija a la entrada estándar (descriptor 0) del hijo –por donde lee el comando **wc**–.

- Echar un vistazo al programa **lswc.c** cuyo aspecto es el siguiente:



```
1  /* ----- */
2  /*      lswc.c: Uso de pipe y dup para comando ls -l | wc      */
3  /* ----- */
4
5  #include <assert.h>                                     /* */
6  #include <stdio.h>                                     /* printf */
7  #include <unistd.h>                                    /* fork, read, write, pipe */
8
9  int tubo[2] ;
10
11 int main ( void )
12 {
13     int idHijo, estado ;
14     assert(pipe(tubo) == 0) ;
15     if (fork() != 0)
16     {
17         /* EL Padre ejecuta el comando ls */
18         assert(dup2(tubo[1], 1) == 1) ; /* Salida estandar => tubo[1] */
19         assert(close(tubo[0]) == 0) ; /* Cerramos extremo Lectura */
20         execl("/bin/ls", "ls", "-l", NULL) ;
21     }
22     else
23     {
24         /* EL Hijo ejecuta el comando wc */
25         assert(dup2(tubo[0], 0) == 0) ; /* Entrada estandar => tubo[0] */
26         assert(close(tubo[1]) == 0) ; /* Cerramos extremo escritura */
27         execl("/usr/bin/wc", NULL) ;
28     }
29     return(0) ;
30 }
```

- Intentar comprender el programa anterior y consultar con el profesor en caso de dudas.
- Ejecutar el comando `ls -l` y observar su salida.
- Ejecutar ahora `ls -l | wc` y anotar lo que sale por pantalla:



- Compilar (`make lswc`) y ejecutar el programa `lswc.c` comprobando que funciona correctamente.

Obsérvese que en lo anterior se está utilizando que los comandos del shell trabajan de una manera estándar con los descriptors 0, 1 y 2. El descriptor 0 siempre se utiliza como entrada de los datos (entrada estándar), el descriptor 1 siempre se utiliza para mostrar los resultados (salida estándar) y el descriptor 2 siempre se utiliza para mostrar los mensajes de error (salida de error estándar). La redirección simplemente consiste en hacer que el descriptor 0, 1 o 2 se corresponda con un fichero o dispositivo diferente al utilizado originalmente. Para no tener que recordar los números concretos, la biblioteca `unistd.h` ofrece las siguientes constantes:

```
#define STDIN_FILENO    0           /* file descriptor for stdin */
#define STDOUT_FILENO  1           /* file descriptor for stdout */
#define STDERR_FILENO  2           /* file descriptor for stderr */
```

Para terminar, el alumno debe ser consciente de que **pipe** crea tuberías anónimas que desaparecen con la terminación de los procesos que las usan. Minix/Linux permite también crear tuberías persistentes y con nombre en el sistema de ficheros. Por ejemplo, con los siguientes comandos creamos una tubería con nombre `/root/mitubo`. El comando equivalente a `ls -l | wc` utilizando esa tubería sería:

```
# mkfifo /root/mitubo           (crea la tubería /root/mitubo. prw-r---r-- ...)
```

```
# ls -l /root/mitubo           (muestra los atributos de /root/mitubo.)
```

```
# (wc < /root/mitubo & ls -l > /root/mitubo) (wc procesa lo que ls -l le envía por mitubo)
```

```
                                     (-también se puede hacer en dos terminales-)
```

`mkfifo` es tanto el nombre del comando (`man -s 1 mkfifo`), como el nombre de la llamada al sistema (`man -s 2 mkfifo`) para crear una tubería con nombre. Luego la tubería se puede abrir para lectura o escritura con `open`. [ejercicio: comunicación full-duplex con tuberías (*difícil*)]

## 5 LLAMADAS RELACIONADAS CON SEÑALES

Para comprender cómo se invocan las rutinas de interfaz así como la funcionalidad de algunas de las llamadas al sistema operativo relacionadas con las señales (**kill**, **sigaction**, **alarm** y **pause**), vamos a compilar y ejecutar los programas que se indican a continuación:

1. sigint1.c      Observar el tratamiento por defecto de SIGINT
2. sigint2.c      Capturar la señal SIGINT
3. sigint3.c      Forzar dos SIGINT para matar al proceso
4. sigint4.c      Ignorar la señal SIGINT
5. sigkill.c      Enviar la señal SIGKILL a un proceso
6. mitime.c      Aproximación al comando *time*
7. unminuto.c    Segundero corriendo unos 60 segundos
8. cifra.c        Cifrar con *crypt* una clave de tres letras
9. descifra.c    Descifra por fuerza bruta una clave encriptada
10. despara1.c   Descifra en paralelo con dos procesos
11. despara2.c   Descifra en paralelo usando SIGUSR1

Las señales son “interrupciones” software que pueden llegarle a un proceso comunicando un evento asíncrono (por ejemplo que el usuario ha tecleado <Ctrl-C>) provocando la ejecución de cierta función del proceso. Las señales que gestiona el sistema operativo MINIX están enumeradas en el fichero `/usr/include/signal.h` cuyo contenido es el siguiente, asociando un número a cada tipo de señal:

```
#define SIGHUP            1 /* Hangup */
#define SIGINT            2 /* Interrupt (usually DEL) */
#define SIGQUIT          3 /* Quit (usually CTRL-\) */
#define SIGILL            4 /* Illegal instruction */
#define SIGTRAP          5 /* Trace trap (not reset when caught) */
#define SIGABRT          6 /* Abort program */
#define SIGFPE            8 /* Floating point exception */
#define SIGKILL          9 /* Kill (cannot be caught or ignored) */
#define SIGUSR1         10 /* User defined signal # 1 */
#define SIGSEGV         11 /* Segmentation violation */
#define SIGUSR2         12 /* User defined signal # 2 */
#define SIGPIPE         13 /* Write on a pipe with no reader */
#define SIGALRM         14 /* Alarm clock */
#define SIGTERM         15 /* Software termination signal from kill */
#define SIGCHLD         17 /* Child process terminated or stopped */
#define SIGCONT         18 /* Continue if stopped */
#define SIGSTOP         19 /* Stop signal */
#define SIGWINCH        21 /* Window size has changed */
```

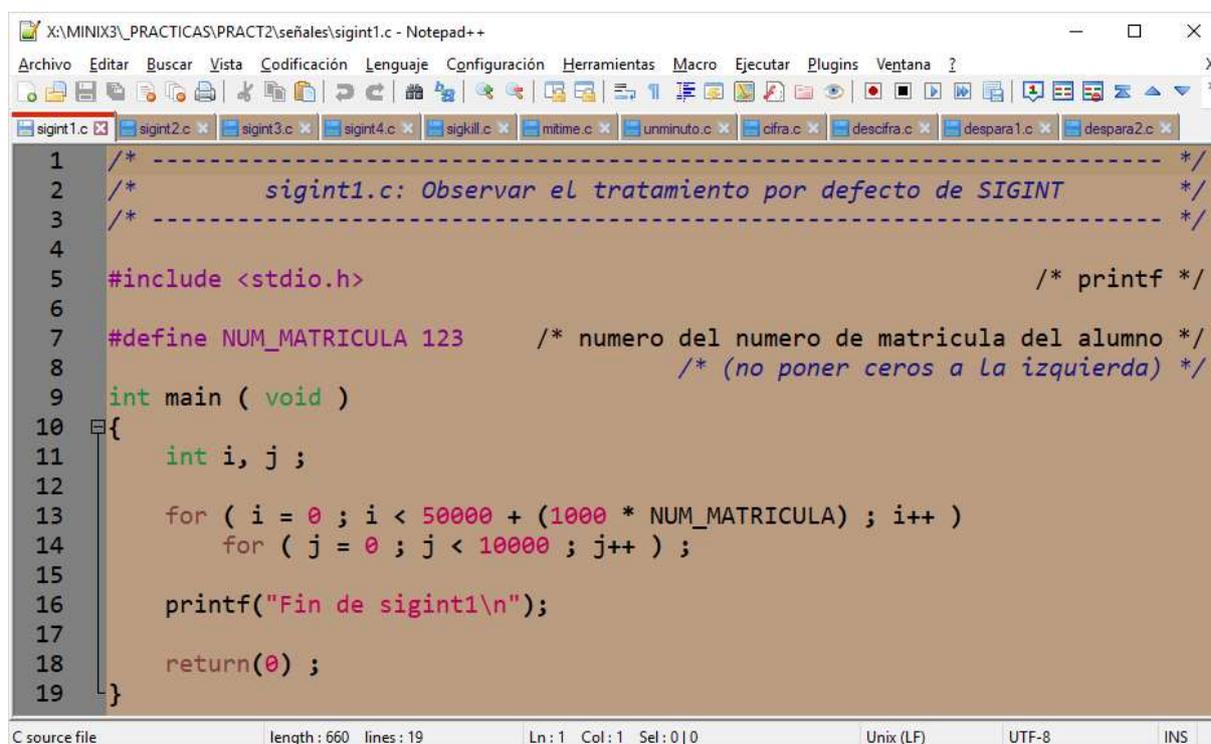
Puede observarse que (además de con **kill**) una señal también puede enviarse por errores de ejecución “excepciones hardware” como SIGILL –intento de ejecutar una instrucción ilegal– o SIGSEGV –intento de acceder a una dirección inválida–. La expiración de una temporización

(puesta por la llamada [alarm](#) que veremos más adelante) también provoca que se envíe una señal (SIGALRM), en este caso al proceso que puso la alarma. Además, hay dos señales que pueden ser utilizadas (enviadas y recibidas) por los procesos de usuario. Un proceso puede elegir qué hacer si le llega una señal concreta, optando entre:

- Dejar el tratamiento por defecto (que es: matar al proceso y generar un volcado del contexto del proceso “core dumped” si procede)
- Ignorar la señal (salvo para SIGKILL)
- Capturar la señal y tratarla de forma específica (indicando la función que la tratará)

Vamos a empezar a ver el comportamiento de las señales con los programas de prueba presentes en el directorio `X:\MINIX3\_PRACTICAS\PRACT2\señales`, que se supone presentes ya en el directorio `/root/pract2/señales`. Comprobar que en `/root/pract2/señales` están los 11 ficheros.

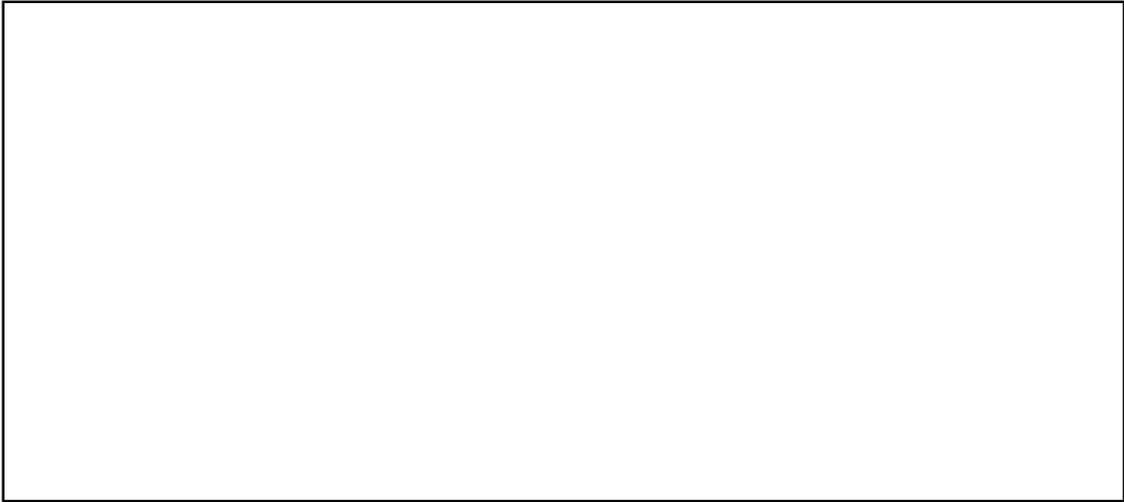
- Echar un vistazo al programa `sigint1.c` cuyo aspecto es el siguiente:



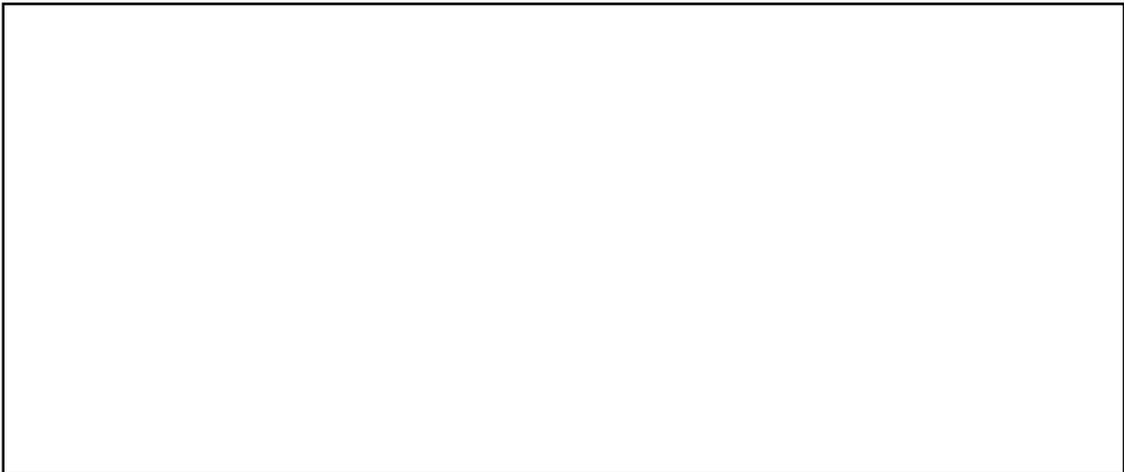
```
1  /* ----- */
2  /*      sigint1.c: Observar el tratamiento por defecto de SIGINT      */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6
7  #define NUM_MATRICULA 123                             /* numero del numero de matricula del alumno */
8  /* (no poner ceros a la izquierda) */
9
10 int main ( void )
11 {
12     int i, j ;
13
14     for ( i = 0 ; i < 50000 + (1000 * NUM_MATRICULA) ; i++ )
15         for ( j = 0 ; j < 10000 ; j++ ) ;
16
17     printf("Fin de sigint1\n");
18
19     return(0) ;
20 }
```

- Modificar la línea 7 de `sigint.c` sustituyendo 123 por el número del número de matrícula del alumno sin ceros a la izquierda.
- Compilar `sigint1.c` y ejecutarlo tecleando: `time ./sigint1` (ver [man time](#)).
- Anotar la salida que aparece finalmente por la pantalla y el valor de  `$?`  (echo  `$?` ):

- Volver a ejecutar el mismo comando, pero ahora pulsar <Ctrl-C> **antes de que termine**. Anotar la salida (debe entenderse qué es lo que ha pasado) y el valor de \$?:



- Volver a ejecutar el programa, ahora sin **time** y pulsando también <Ctrl-C>. Anotar la salida explicando la diferencia que se observa:



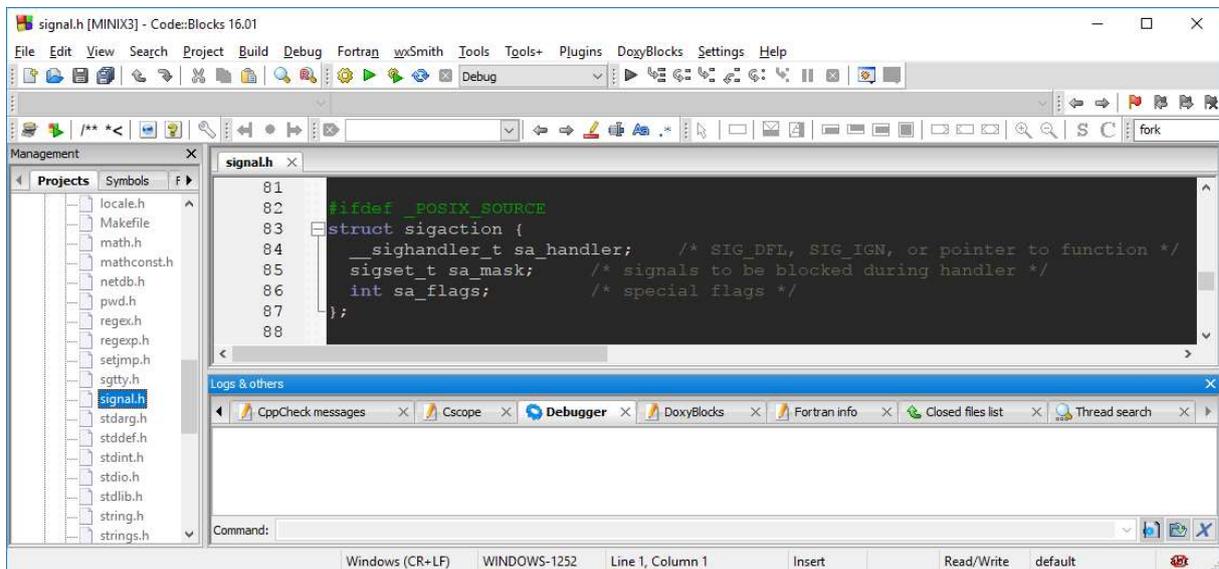
Ahora vamos a capturar la señal SIGINT. Para ello, disponemos de la llamada [sigaction](#) cuyo encabezamiento es el siguiente:

```
int sigaction ( int sig, const struct sigaction * act, struct sigaction * oact );
```

donde:

- sig** Indica la señal que queremos tratar
- act** Indica la dirección de la estructura que hemos inicializado especificando el nuevo comportamiento que queremos para cuando llegue la señal “sig”
- oact** Indica la dirección de la estructura que hemos dispuesto para que el sistema nos deje el comportamiento que tenía la señal “sig” anteriormente, por si mas adelante queremos restaurarlo

La estructura “**sigaction**” que permite definir el nuevo comportamiento ante la llegada de una señal, está declarada en el fichero `/usr/include/signal.h`:



The screenshot shows a code editor window titled "signal.h [MINIX3] - Code:Blocks 16.01". The editor displays the following code snippet:

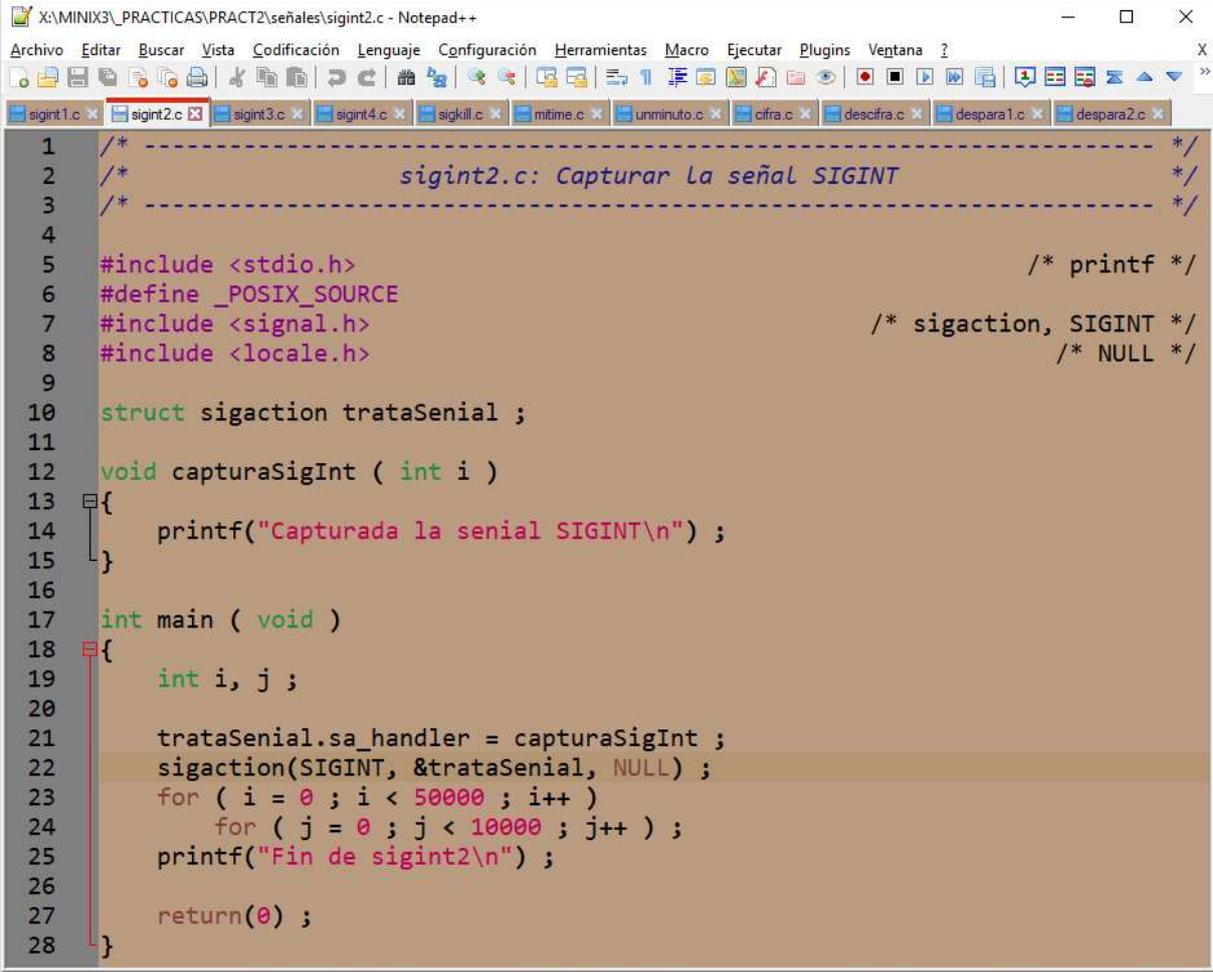
```
81
82
83 #ifdef _POSIX_SOURCE
84 struct sigaction {
85     __sig_handler_t sa_handler; /* SIG_DFL, SIG_IGN, or pointer to function */
86     sigset_t sa_mask; /* signals to be blocked during handler */
87     int sa_flags; /* special flags */
88 };
```

The editor interface includes a "Management" sidebar on the left with a file tree, a "Logs & others" panel at the bottom, and a status bar at the very bottom showing "Windows (CR+LF) WINDOWS-1252 Line 1, Column 1 Insert Read/Write default".

```
struct sigaction {
    __sig_handler_t sa_handler ; /* SIG_DFL, SIG_IGN, or pointer to function */
    sigset_t sa_mask ; /* signals to be blocked during handler */
    int sa_flags ; /* special flags */
};
```

- Indicar en qué línea de qué fichero de MINIX está la declaración con el cuerpo de la rutina de interfaz de la llamada al sistema **sigaction** (Codeblocks).

- Echar un vistazo al programa **sigint2.c** siguiente:



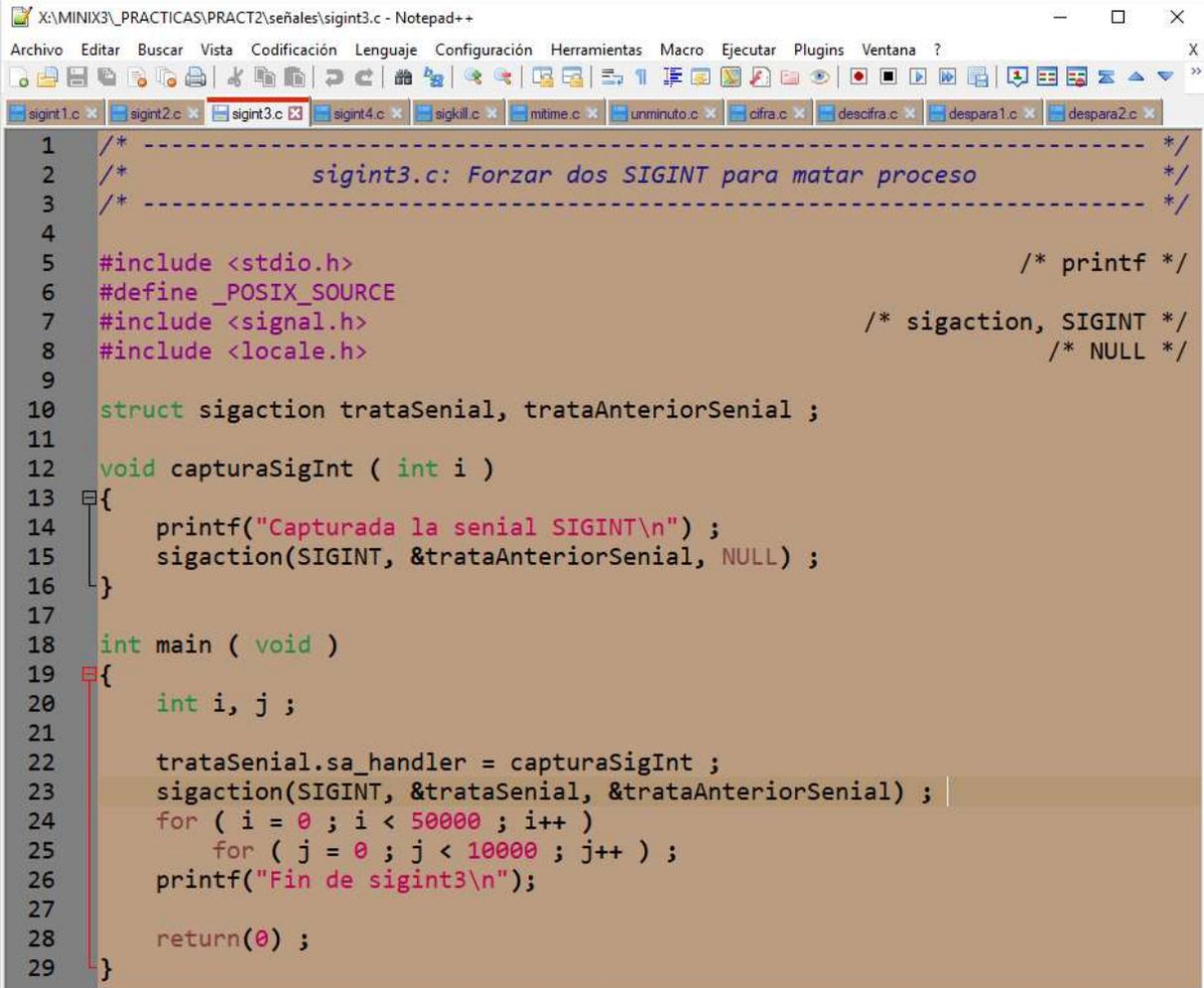
```
1  /* ----- */
2  /*                               sigint2.c: Capturar la señal SIGINT                               */
3  /* ----- */
4
5  #include <stdio.h>                               /* printf */
6  #define _POSIX_SOURCE
7  #include <signal.h>                               /* sigaction, SIGINT */
8  #include <locale.h>                               /* NULL */
9
10 struct sigaction trataSenial ;
11
12 void capturaSigInt ( int i )
13 {
14     printf("Capturada la senial SIGINT\n") ;
15 }
16
17 int main ( void )
18 {
19     int i, j ;
20
21     trataSenial.sa_handler = capturaSigInt ;
22     sigaction(SIGINT, &trataSenial, NULL) ;
23     for ( i = 0 ; i < 50000 ; i++ )
24         for ( j = 0 ; j < 10000 ; j++ ) ;
25     printf("Fin de sigint2\n") ;
26
27     return(0) ;
28 }
```

C source file      length : 853    lines : 28      Ln : 22    Col : 44    Sel : 0 | 0      Unix (LF)      UTF-8      INS

- Compilar y ejecutar **sigint2.c** tecleando <Ctrl-C> tres o cuatro veces antes de que termine el programa. Anotar lo que sale por pantalla y si el resultado es el que se esperaba. ¿Influye esa acción en el valor mostrado con **echo \$??**?

Ahora vamos a probar un programa similar a **sigint2.c**, llamado **sigint3.c**, cuya ejecución va a requerir que se teclee dos veces <Ctrl-C> para matar al proceso. Con el primer <Ctrl-C> se informará que se ha capturado la señal y se restaurará el tratamiento de la señal que tenía por defecto para que con el siguiente <Ctrl-C>, si se teclea antes de que termine el proceso, lo mate.

- Echar un vistazo al programa **sigint3.c**:

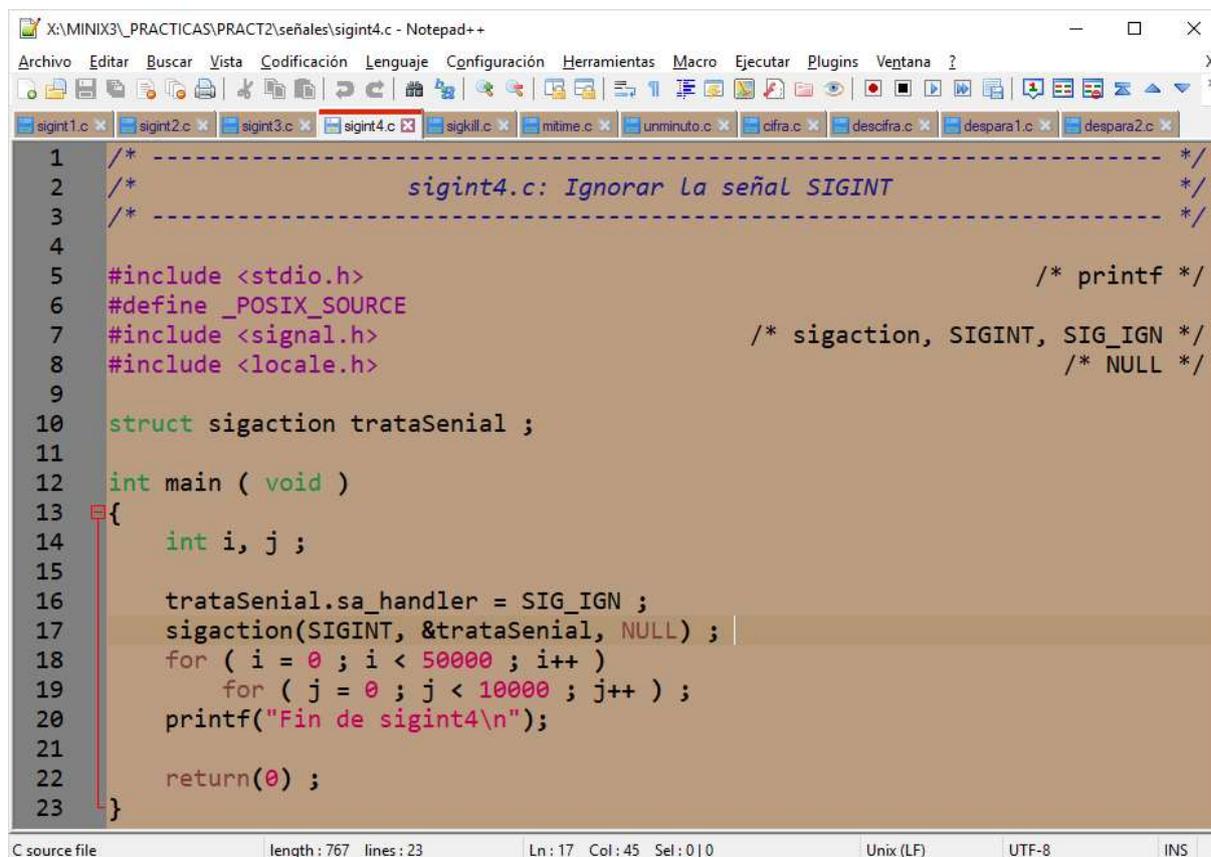


```
1  /* ----- */
2  /*          sigint3.c: Forzar dos SIGINT para matar proceso          */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #define _POSIX_SOURCE
7  #include <signal.h>                                   /* sigaction, SIGINT */
8  #include <locale.h>                                   /* NULL */
9
10 struct sigaction trataSenial, trataAnteriorSenial ;
11
12 void capturaSigInt ( int i )
13 {
14     printf("Capturada la senial SIGINT\n") ;
15     sigaction(SIGINT, &trataAnteriorSenial, NULL) ;
16 }
17
18 int main ( void )
19 {
20     int i, j ;
21
22     trataSenial.sa_handler = capturaSigInt ;
23     sigaction(SIGINT, &trataSenial, &trataAnteriorSenial) ;
24     for ( i = 0 ; i < 50000 ; i++ )
25         for ( j = 0 ; j < 10000 ; j++ ) ;
26     printf("Fin de sigint3\n");
27
28     return(0) ;
29 }
```

C source file      length : 939   lines : 29      Ln : 23   Col : 61   Sel : 0 | 0      Unix (LF)      UTF-8      INS

- Observar cómo se restaura el tratamiento antiguo de la señal tras el primer <Ctrl-C>.
- Compilar y ejecutar el nuevo programa para comprobar su correcto funcionamiento.

- Ahora vamos a ver cómo podemos ignorar el tratamiento de la señal SIGINT de forma que el usuario no pueda matarnos tecleando <Ctrl-C>. Echar un vistazo al programa **sigint4.c** que se muestra a continuación:



```

1  /* ----- */
2  /*          sigint4.c: Ignorar la señal SIGINT          */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #define _POSIX_SOURCE
7  #include <signal.h>                                   /* sigaction, SIGINT, SIG_IGN */
8  #include <locale.h>                                   /* NULL */
9
10 struct sigaction trataSerial ;
11
12 int main ( void )
13 {
14     int i, j ;
15
16     trataSerial.sa_handler = SIG_IGN ;
17     sigaction(SIGINT, &trataSerial, NULL) ;
18     for ( i = 0 ; i < 50000 ; i++ )
19         for ( j = 0 ; j < 10000 ; j++ ) ;
20     printf("Fin de sigint4\n");
21
22     return(0) ;
23 }

```

- Observar cómo se consigue ignorar la señal.
- Compilar y ejecutar el programa. Comprobar que no hay forma de abortar la ejecución con <Ctrl-C>.

Ahora vamos a utilizar la llamada al sistema operativo **kill** cuyo encabezado es el siguiente:

```
int kill ( pid_t pid, int sig ) ;
```

- Indicar en qué línea de qué fichero de MINIX está la declaración con el cuerpo de la rutina de interfaz de la llamada al sistema **kill** (Codeblocks).



Esta llamada envía la señal **sig** al proceso **pid**. Con este mecanismo podríamos enviar señales como SIGUSR1 y SIGUSR2 entre procesos que hayamos creado. Nosotros vamos a utilizarla para matar un proceso enviándole la señal SIGKILL (eso es lo que hace el comando kill).

- Echar un vistazo al programa **sigkill.c**:

```
X:\MINIX3_PRACTICAS\PRACT2\señales\sigkill.c - Notepad++
Archivo  Editar  Buscar  Vista  Codificación  Lenguaje  Configuración  Herramientas  Macro  Ejecutar  Plugins  Ventana  ?
signt1.c x  signt2.c x  signt3.c x  signt4.c x  sigkill.c x  mtime.c x  uminuto.c x  cifra.c x  descifra.c x  despara1.c x  despara2.c x

1  /* ----- */
2  /*          sigkill.c: Enviar senial SIGKILL a un proceso          */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf, scanf */
6  #include <unistd.h>                                    /* fork */
7  #include <stdlib.h>                                    /* exit */
8  #define _POSIX_SOURCE
9  #include <signal.h>                                   /* sigaction, kill, SIGKILL */
10 #include <sys/wait.h>                                  /* _HIGH, _LOW */
11
12 void hijo ( void )
13 {
14     int i ;
15     for ( i = 0 ; i < 100000 ; i++ )
16         printf("%c", 'H') ;                               /* write(1, &"H", 1) */
17     printf("\nFin del proceso hijo\n") ;
18     exit(3) ;                                             /* bastaria con return(3) */
19 }
20
21 int main ( void )
22 {
23     int pid, resultado, estado ;
24     char ch ;
25
26     pid = fork() ;
27     if (pid == 0)
28     {
29         hijo() ;
30     }
31     else
32     {
33         scanf("%c", ch) ;
34         resultado = kill(pid, SIGKILL) ;
35         printf("\nEnviado SIGKILL al hijo con resultado %d", resultado) ;
36         resultado = wait(&estado) ;
37         if (estado > 255)
38             printf("\nFin hijo %d con estado %d\n",
39                 resultado, _HIGH(estado)) ;
40         else
41             printf("\nTerminado hijo %d con estado %d\n",
42                 resultado, _LOW(estado)) ;
43     }
44     return(0) ;
45 }
```

C source file: length: 1,414 lines: 45 Ln: 34 Col: 34 Sel: 0|0 Unix (LF) UTF-8 INS

- Observar cómo el proceso padre espera a que tecleemos la tecla RETURN/INTRO, tras lo cual llama a **kill** para matar al proceso hijo, que se limita a ejecutar un bucle relativamente grande escribiendo caracteres 'H'.

- Compilar y ejecutar el programa **sigkill**. Presionar la tecla RETURN en cuanto que comiencen a aparecer los caracteres 'H' escritos por el proceso hijo. Anotar lo que aparece en las dos últimas líneas de la pantalla:

- ¿Podemos asegurar que hemos matado al proceso hijo?

- Ejecutar otra vez el programa pero esperando ahora a que el proceso hijo termine: dejarán de salir 'H' y se indicará "Fin del proceso hijo". En ese momento –no hace falta correr–, pulsar la tecla RETURN y anotar lo que aparece en las dos últimas líneas de la pantalla:

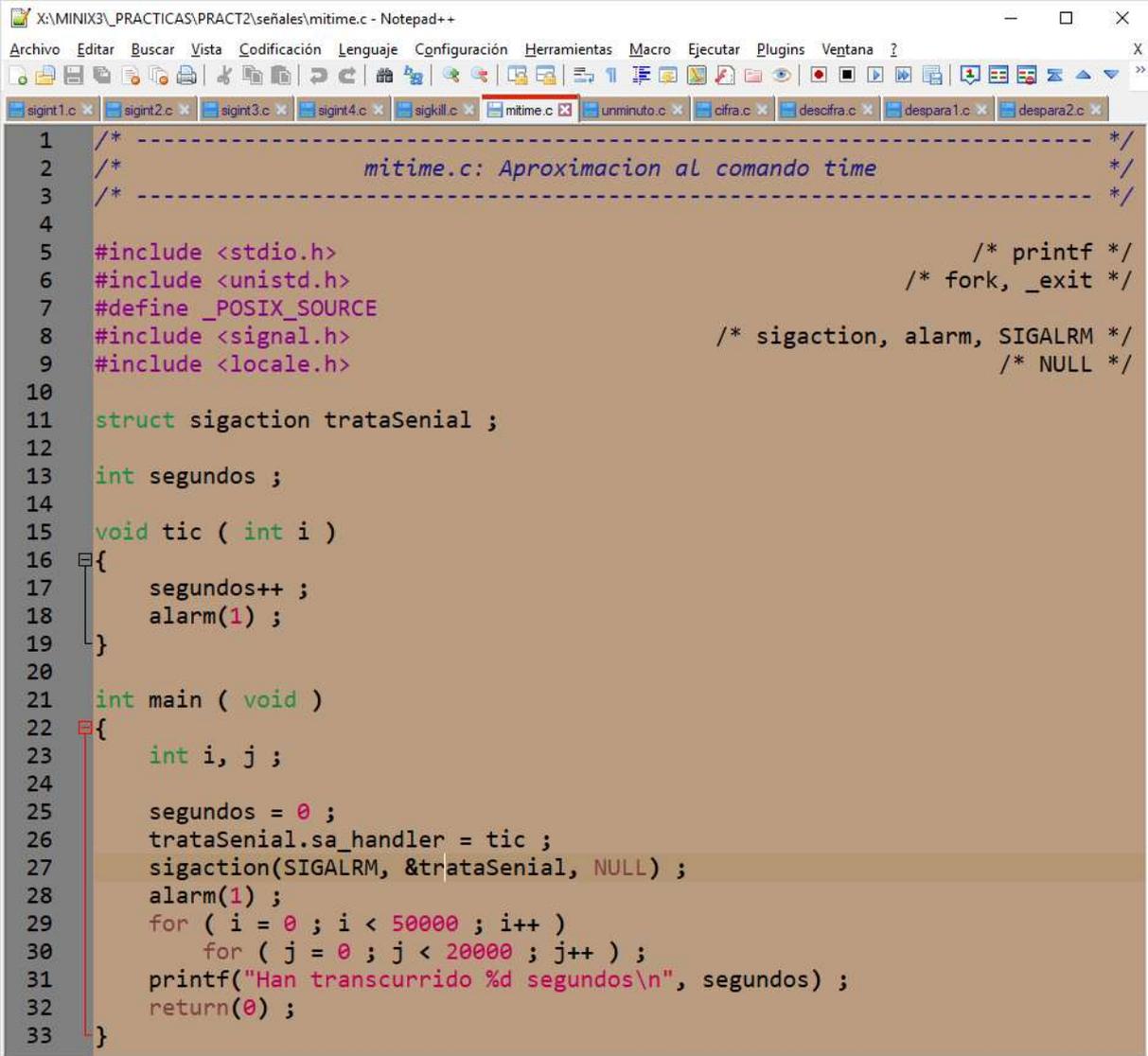
- ¿Cuál es la diferencia entre ambas ejecuciones y a qué es debido?

Ahora vamos a utilizar la llamada al sistema operativo **alarm** cuya sinopsis es la siguiente:

unsigned int **alarm** ( unsigned int **seconds** ) ;

Esta llamada provoca que el sistema operativo le envíe la señal SIGALRM al proceso al cabo de los segundos indicados en el parámetro **seconds**. Vamos a utilizarla para escribir un programa que se parezca al comando **time** que informa del tiempo que tarda en ejecutarse un programa.

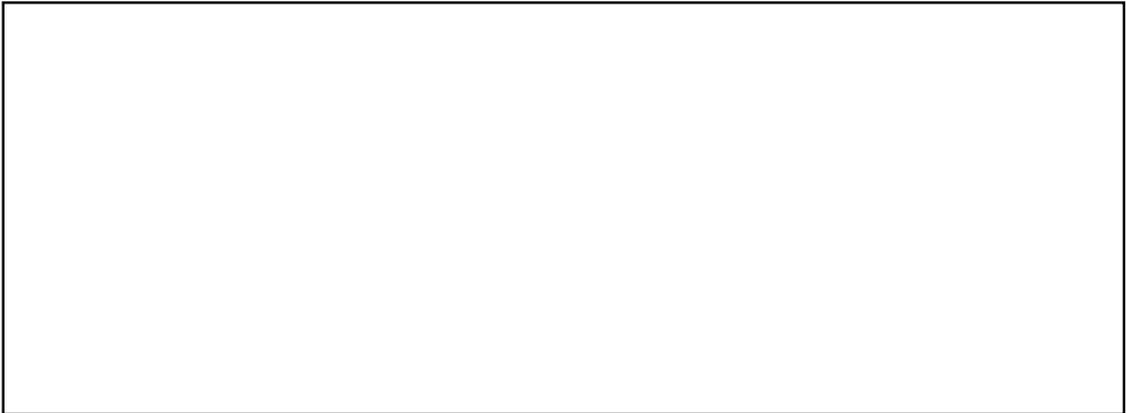
- Echar un vistazo al programa **mitime.c** cuyo aspecto es el siguiente:



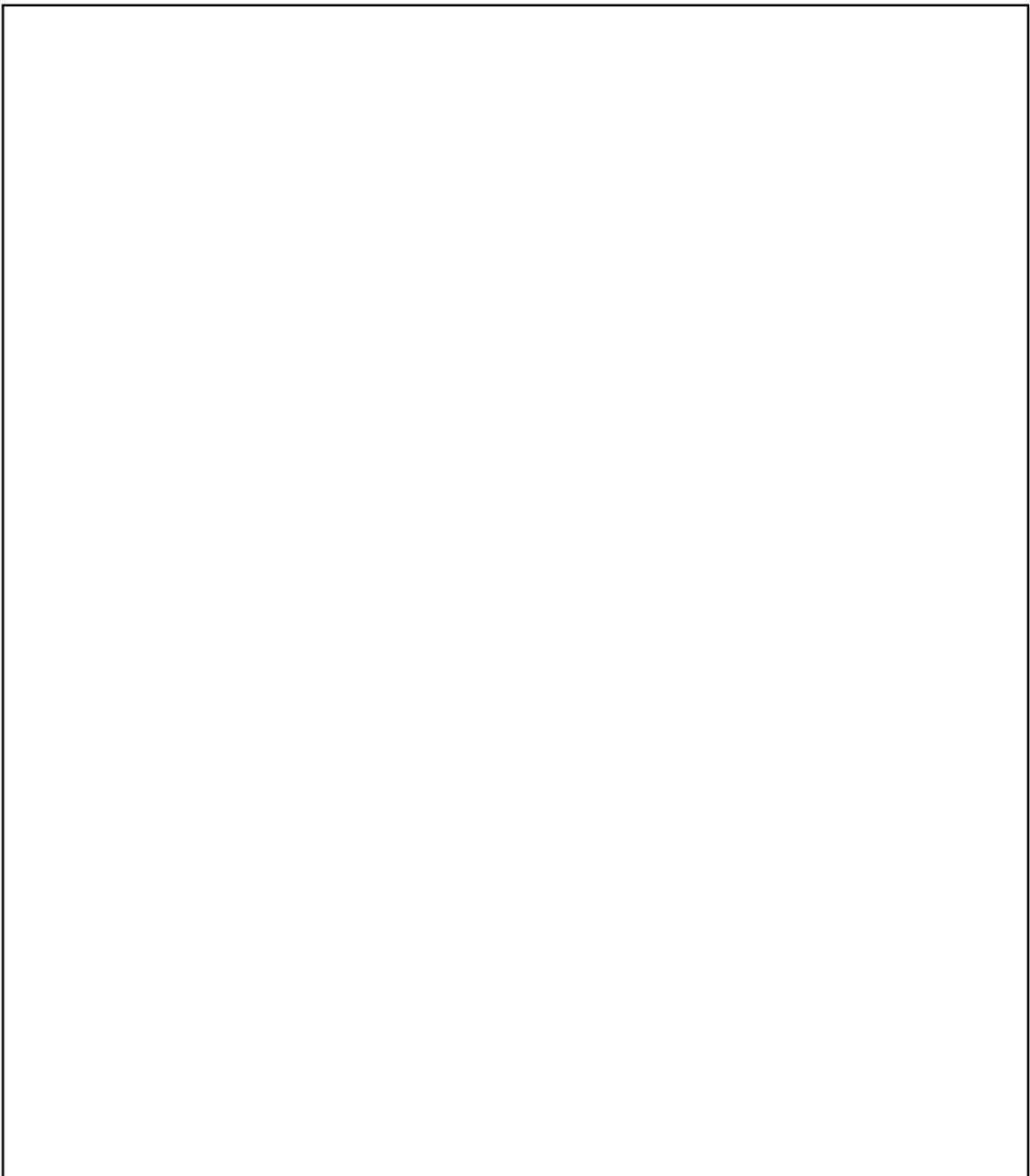
```
1  /* ----- */
2  /*           mitime.c: Aproximacion al comando time           */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #include <unistd.h>                                   /* fork, _exit */
7  #define _POSIX_SOURCE
8  #include <signal.h>                                  /* sigaction, alarm, SIGALRM */
9  #include <locale.h>                                  /* NULL */
10
11 struct sigaction trataSerial ;
12
13 int segundos ;
14
15 void tic ( int i )
16 {
17     segundos++ ;
18     alarm(1) ;
19 }
20
21 int main ( void )
22 {
23     int i, j ;
24
25     segundos = 0 ;
26     trataSerial.sa_handler = tic ;
27     sigaction(SIGALRM, &trataSerial, NULL) ;
28     alarm(1) ;
29     for ( i = 0 ; i < 50000 ; i++ )
30         for ( j = 0 ; j < 20000 ; j++ ) ;
31     printf("Han transcurrido %d segundos\n", segundos) ;
32     return(0) ;
33 }
```

- Observar cómo se utiliza **alarm** en conjunción con **sigaction** para ir anotando (asíncronamente con la ejecución del proceso) el paso del tiempo. Es de destacar que desde la rutina de tratamiento de la alarma se vuelve a llamar a **alarm**.

- Compilar y ejecutar **mitime**. Anotar lo que sale por pantalla:

A large, empty rectangular box with a black border, intended for the student to write down the output of the compilation and execution of the **mitime** program.

- Volver a ejecutar **mitime** pero ahora de la forma siguiente: **time ./mitime**. Anotar lo que sale por pantalla y explicar la diferencia:

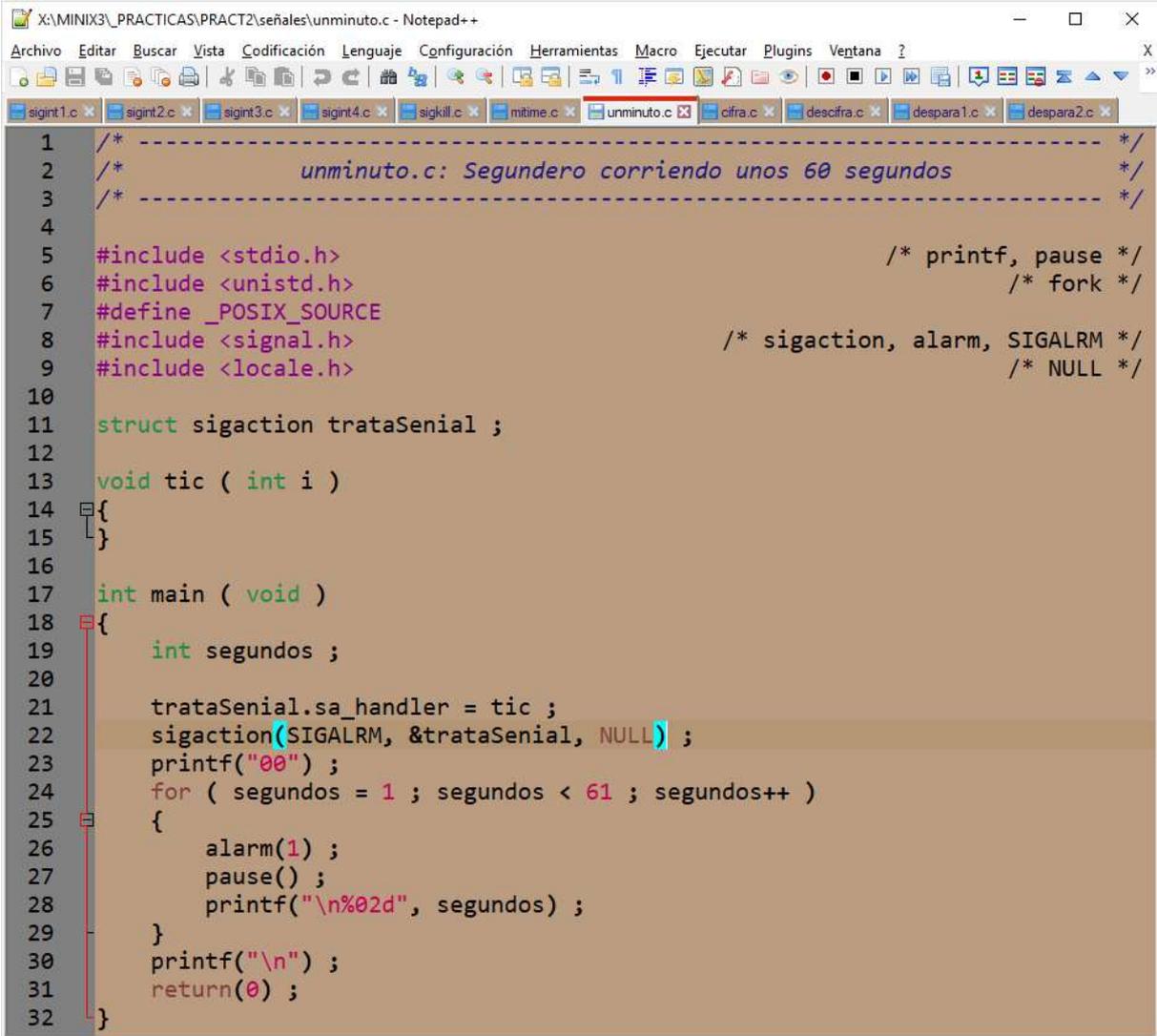
A large, empty rectangular box with a black border, intended for the student to write down the output of the **time ./mitime** command and provide an explanation of the difference compared to the previous step.

Para terminar con esta sección, vamos a utilizar la llamada al sistema operativo [pause](#) cuyo encabezamiento es:

```
int pause ( void );
```

Esta llamada suspende al proceso hasta que llegue una señal a través de **kill** o de un *timeout* programado con **alarm**. Nosotros vamos a utilizarlo para implementar un pseudosegundo.

- Echar un vistazo al programa **unminuto.c** siguiente:



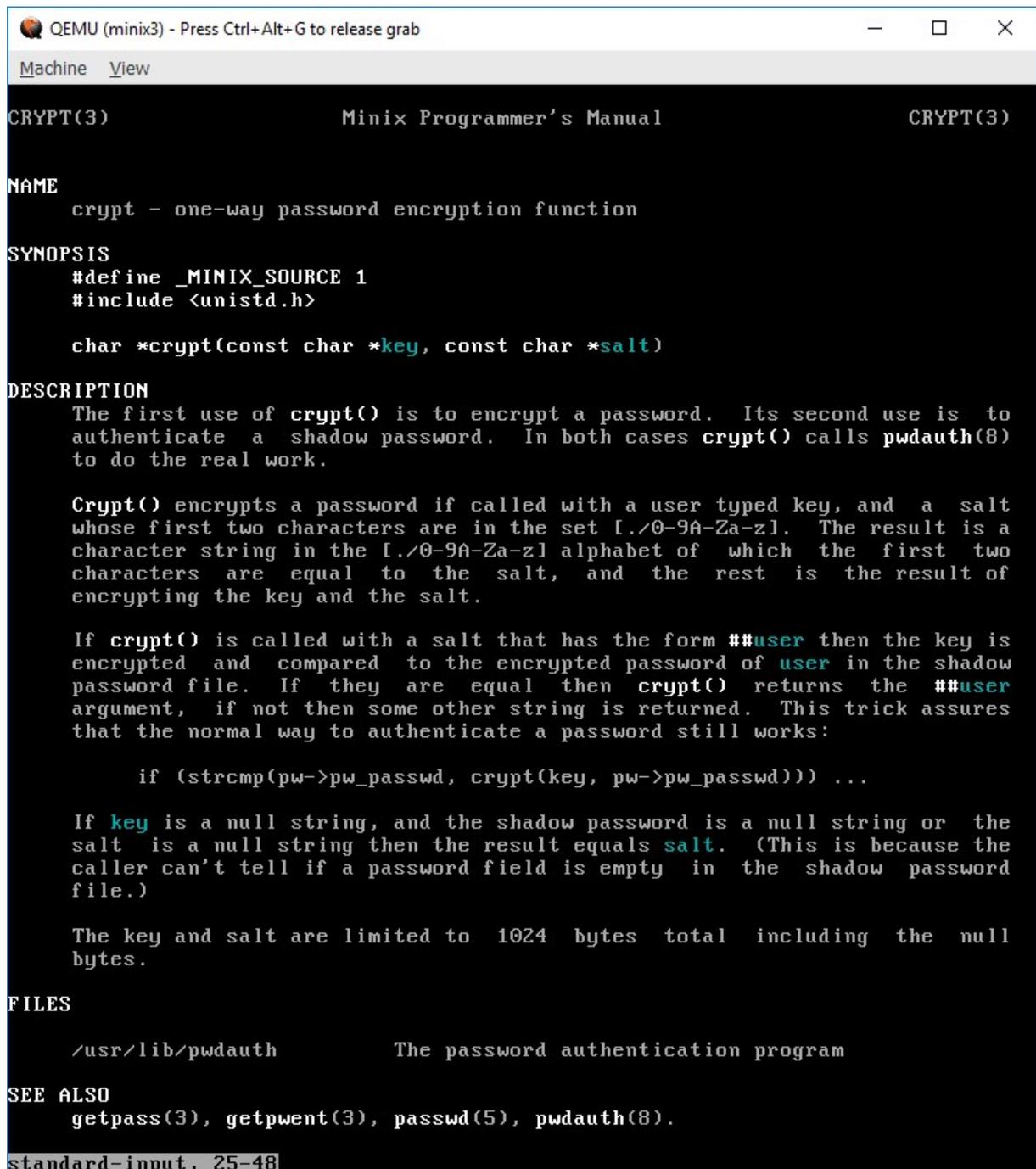
```
1  /* ----- */
2  /*           unminuto.c: Segundero corriendo unos 60 segundos           */
3  /* ----- */
4
5  #include <stdio.h>                /* printf, pause */
6  #include <unistd.h>              /* fork */
7  #define _POSIX_SOURCE
8  #include <signal.h>              /* sigaction, alarm, SIGALRM */
9  #include <locale.h>              /* NULL */
10
11 struct sigaction trataSenial ;
12
13 void tic ( int i )
14 {
15 }
16
17 int main ( void )
18 {
19     int segundos ;
20
21     trataSenial.sa_handler = tic ;
22     sigaction(SIGALRM, &trataSenial, NULL) ;
23     printf("00") ;
24     for ( segundos = 1 ; segundos < 61 ; segundos++ )
25     {
26         alarm(1) ;
27         pause() ;
28         printf("\n%02d", segundos) ;
29     }
30     printf("\n") ;
31     return(0) ;
32 }
```

C source file      length : 922    lines : 32      Ln : 22    Col : 43    Sel : 0 | 0      Unix (LF)      UTF-8      INS

- Compilar y ejecutar el programa para observar su correcto funcionamiento.

Ahora vamos a intentar utilizar la señal SIGUSR1 para intercomunicar procesos. Para ello desarrollaremos de forma muy simplificada un programa para reventar (*crackear*) una clave (*password*) que ejecutaremos tanto en versión secuencial como en versión paralela utilizando dos procesos.

Las claves que utilizaremos serán a lo más de cinco letras ya que reventar una clave de más letras mediante fuerza bruta (probar todas las combinaciones posibles) nos llevaría mucho tiempo. Las claves se cifran con el servicio [crypt](#) que puede consultarse con el manual en línea de MINIX [[man crypt](#)].



```
QEMU (minix3) - Press Ctrl+Alt+G to release grab
Machine View
CRYPT(3)                               Minix Programmer's Manual                               CRYPT(3)

NAME
  crypt - one-way password encryption function

SYNOPSIS
  #define _MINIX_SOURCE 1
  #include <unistd.h>

  char *crypt(const char *key, const char *salt)

DESCRIPTION
  The first use of crypt() is to encrypt a password. Its second use is to
  authenticate a shadow password. In both cases crypt() calls pwdauth(8)
  to do the real work.

  Crypt() encrypts a password if called with a user typed key, and a salt
  whose first two characters are in the set [./0-9A-Za-z]. The result is a
  character string in the [./0-9A-Za-z] alphabet of which the first two
  characters are equal to the salt, and the rest is the result of
  encrypting the key and the salt.

  If crypt() is called with a salt that has the form ##user then the key is
  encrypted and compared to the encrypted password of user in the shadow
  password file. If they are equal then crypt() returns the ##user
  argument, if not then some other string is returned. This trick assures
  that the normal way to authenticate a password still works:

      if (strcmp(pw->pw_passwd, crypt(key, pw->pw_passwd))) ...

  If key is a null string, and the shadow password is a null string or the
  salt is a null string then the result equals salt. (This is because the
  caller can't tell if a password field is empty in the shadow password
  file.)

  The key and salt are limited to 1024 bytes total including the null
  bytes.

FILES
  /usr/lib/pwdauth           The password authentication program

SEE ALSO
  getpass(3), getpwent(3), passwd(5), pwdauth(8).

standard-input, 25-48
```

- Echar un vistazo al programa **cifra.c** cuyo aspecto es el siguiente:

```

1  /* ----- */
2  /*      cifra.c: Cifrar con crypt una clave de tres letras      */
3  /* ----- */
4
5  #include <stdio.h>                                     /* printf */
6  #define _MINIX                                         /* (crypt) */
7  #include <unistd.h>                                    /* crypt */
8
9  #ifdef _MINGW64                                        /* compilador TDM-GCC-64 (gcc) */
15
16 int main ( int argc, char * argv [ ] )
17 {
18     if (argc < 2)
19     {
20         printf(
21             "\n"
22             " formato: cifra <palabra_a_cifrar> \n"      /* # cifra xxxxx */
23         );
24         return(1) ;
25     }
26     printf(
27         "\n"
28         " cifra(\"%s\") = \"%s\"\n"UL, argv[1], crypt(argv[1], "aa")
29     );
30     return(0) ;
31 }

```

- Compilar este programa con **cc cifra.c -o cifra** y ejecutarlo para cifrar las claves “**esesi**” y “**ojosi**” anotando los valores de las claves cifradas en la segunda columna de la tabla que aparece a continuación:

Clave cruda	Clave cifrada	T. secuencial	T. Paralelo1	T. Paralelo2
esesi				
ojosi				

Tabla 1: Tiempos para reventar claves cifradas con **crypt**

- Echar un vistazo al programa **descifra.c** que se muestra en las páginas siguientes.

## descifra.c: Estructura del programa

```

1  /* ----- */
2  /*      descifra.c: Descifra por fuerza bruta una clave encriptada      */
3  /* ----- */
4
5  #include <stdio.h>                                /* printf */
6  #include <string.h>                               /* strcmp, strcpy, strlen */
7  #include <ctype.h>                                /* islower, isalpha */
8  #include <stdlib.h>                               /* malloc, free */
9  #define _MINIX                                    /* (crypt) */
10 #include <unistd.h>                               /* crypt */
11
12 #ifdef _MINGW64_                                  /* compilador TDM-GCC-64 (gcc) */
13
14
15
16
17
18
19 void siguientePassword ( char * pwd )              /* orden Lexicografico */
20 {                                                 /* circular */
21
22
23
24
25
26
27
28
29
30 int valor ( const char * pwd )                    /* valor de La clave (base 'z'-'a' + 1) */
31 {
32
33
34
35
36
37
38
39
40
41 int obtenPwd ( int valor, char * pwd )
42 {
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57 int analizarArgumentos ( int argc, char * argv [ ], int * verbose )
58 {
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123 #define macro_verbose \
124
125
126
127
128
129
130
131
132
133
134
135 int buscaEImprimeClave ( char * claveCifrada,
136                          char * pwd_0,
137                          char * pwd_1,
138                          int verbose )
139 {
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170 int main ( int argc, char * argv [ ] )
171 {
172
173     char * claveCifrada = argv[1] ;              /* resultado de cifrar el password */
174     char * pwd_0 = argv[2] ;                    /* password inicial de busqueda */
175     char * pwd_1 = argv[3] ;                    /* password final de busqueda */
176
177     int verbose ; /* 0, 1 o 2 (grado de escritura de mensajes con printf) */
178
179     int resultado = analizarArgumentos(argc, argv, &verbose) ;
180
181     if (resultado != 0) return(resultado) ;
182
183     return(buscaEImprimeClave(claveCifrada, pwd_0, pwd_1, verbose)) ;
184
185 }

```

C source file      length : 5.523    lines : 185      Ln : 183    Col : 66    Sel : 0 | 0      Unix (LF)    UTF-8      INS

**descifra.c:** Función **buscaEImprimeClave** que busca la clave y la imprime:

```

135 int buscaEImprimeClave ( char * claveCifrada,
136                          char * pwd_0,
137                          char * pwd_1,
138                          int verbose )
139 {
140     char * pwd ;
141     int length_pwd ;
142     int i_0 = valor(pwd_0) ;
143     int i_1 = valor(pwd_1) ;
144     int i, j ;
145
146     length_pwd = strlen(pwd_0) ;
147     pwd = malloc(length_pwd+1) ;
148     strcpy(pwd, pwd_0) ;
149
150     printf("\n") ;
151     for ( i = i_0 ; i <= i_1 ; i++ )
152     {
153         macro_verbose ;      /* macro para mostrar el progreso con -v o -p */
154
155         if (strcmp(claveCifrada, crypt(pwd, "aa")) == 0)
156         {
157             if (verbose > 0) printf("\n\n") ;
158             printf (" la password es: %s (iteraciones = %d = %2d %c)\n"UL,
159                    pwd, i-i_0+1, (100*(i-i_0+1))/(i_1-i_0+1), '%') ;
160             return(0) ;      /* free(pwd) */
161         }
162
163         siguientePassword(pwd) ;
164     }
165     if (verbose > 0) printf("\n\n") ;
166     printf(" la clave \"%s\" no es una clave cifrada \n"UL, claveCifrada) ;
167     return(6) ;      /* free(pwd) */
168 }

```

C source file      length : 5.523    lines : 185      Ln : 183    Col : 66    Sel : 0 | 0      Unix (LF)    UTF-8    INS

Este programa empieza a buscar desde una clave inicial (por ejemplo “aaaaa”) hasta una clave final (por ejemplo la “zzzzz”) cifrando con **crypt** cada clave y finalizando cuando encuentra coincidencia con la clave cifrada pasada como argumento.

[Minix tarda demasiado, mejor **MobaXTerm**]

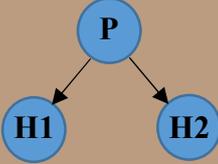
- Compilar el programa **descifra.c** con el comando **cc descifra.c -o descifra**. Ejecutar el comando: **time ./descifra aaxxxxxxxxxxxx aaaaa zzzzz -v** para descifrar las claves cifradas correspondientes a “**esesi**” y “**ojosi**”. Comprobar que localiza las claves y anotar en la tercera columna de la tabla el tiempo (**real**) invertido en la búsqueda de cada clave.
- Ahora vamos a realizar la búsqueda en paralelo con dos procesos, uno buscará en la primera parte del espacio de búsqueda (desde “aaaaa” hasta “mzzzz”) y el otro proceso buscará en la segunda mitad (desde “naaaa” hasta “zzzzz”). Echar un vistazo al programa **despara1.c**.

**despara1.c: programa principal.**

```

186
187 int main ( int argc, char * argv [ ] )
188 {
189     char * claveCifrada = argv[1] ;      /* resultado de cifrar el password */
190     char * pwd_0 = argv[2] ;            /* password inicial de busqueda */
191     char * pwd_1 = argv[3] ;           /* password final de busqueda */
192
193     int verbose ; /* 0, 1 o 2 (grado de escritura de mensajes con printf) */
194
195     int pid, pid_1, pid_2 ;
196     char pwdMedio [20] ;
197     int estado ;
198
199     int resultado = analizarArgumentos(argc, argv, &verbose) ;
200
201     if (resultado != 0) return(resultado) ;
202
203     strcpy(pwdMedio, pwd_0) ;
204
205     obtenPwd((valor(pwd_0)+valor(pwd_1))/2, (char *)&pwdMedio) ;
206
207     printf("\n pwdMedio = \"%s\"\n", pwdMedio) ;
208
209     pid_1 = fork() ;
210     if (pid_1 == 0) {
211         resultado = buscaEImprimeClave(claveCifrada, pwd_0, pwdMedio, 0) ;
212         return(resultado) ;
213     }
214     pid_2 = fork() ;
215     if (pid_2 == 0) {
216         siguientePassword(pwdMedio) ;
217         resultado = buscaEImprimeClave(claveCifrada, pwdMedio, pwd_1, 0) ;
218         return(resultado) ;
219     }
220     pid = wait((int *)&estado) ;
221     pid = wait((int *)&estado) ;
222     printf("\n fin del proceso padre\n\n") ;
223     return(0) ;
224 }

```



C source file      length : 6.369    lines : 224      Ln : 1    Col : 1    Sel : 0 | 0      Unix (LF)    UTF-8    INS

- Compilar **despara1.c** con el comando **cc despara1.c -o despara1** y ejecutar este programa para descifrar en paralelo las claves cifradas correspondientes a “**esesi**” y “**ojosi**”. Comprobar que localiza las claves y anotar en la cuarta columna de la tabla 1 el tiempo (real) invertido en la búsqueda paralela de cada clave. ¿Qué problema se hace evidente en esta forma de búsqueda? ¿Es más rápido que descifra?

- Para intentar resolver el problema del programa **despara1.c** vamos a utilizar la señal SIGUSR1 para que el primer proceso que localice la clave le avise al otro proceso para que termine la búsqueda (ya estéril). A continuación se muestra un **esqueleto** del programa que, **con las modificaciones adecuada por parte del alumno**, deberá mejorar los resultados obtenidos anteriormente. Echar un vistazo al programa para entender dónde insertar código para enviar y tratar la señal SIGUSR1:

### despara2.c: Programa principal

```
X:\MINIX3\PRACTICAS\PRACT2\señales\despara2.c - Notepad++
Archivo  Editar  Buscar  Vista  Codificación  Lenguaje  Configuración  Herramientas  Macro  Ejecutar  Plugins  Ventana  ?
sigint1.c x  sigint2.c x  sigint3.c x  sigint4.c x  sigkill.c x  mtime.c x  unminuto.c x  cifra.c x  descifra.c x  despara1.c x  despara2.c x

182
183 int main ( int argc, char * argv [ ] )
184 {
185
186     char * claveCifrada = argv[1] ;      /* resultado de cifrar el password */
187     char * pwd_0 = argv[2] ;           /* password inicial de busqueda */
188     char * pwd_1 = argv[3] ;           /* password final de busqueda */
189
190     int verbose ; /* 0, 1 o 2 (grado de escritura de mensajes con printf) */
191
192     int pid, pid_1, pid_2 ;
193     char pwdMedio [6] ;
194     int estado ;
195
196     int resultado = analizarArgumentos(argc, argv, &verbose) ;
197
198     if (resultado != 0) return(resultado) ;
199
200     strcpy(pwdMedio, pwd_0) ;
201
202     obtenPwd((valor(pwd_0)+valor(pwd_1))/2, (char *)&pwdMedio) ;
203
204     printf("\n pwdMedio = \"%s\"\n", pwdMedio) ;
205                                     /* ver: setsid, kill(0, _) */
206     pid_1 = fork() ;
207     if (pid_1 == 0) {
208         resultado =
209             buscaEImprimeClave(claveCifrada, pwd_0, pwdMedio, 0, getpid()) ;
210         return(resultado) ;
211     }
212     pid_2 = fork() ;
213     if (pid_2 == 0) {
214         siguientePassword(pwdMedio) ;
215         resultado =
216             buscaEImprimeClave(claveCifrada, pwdMedio, pwd_1, 0, pid_1) ;
217         return(resultado) ;
218     }
219     pid = wait((int *)&estado) ;
220     pid = wait((int *)&estado) ;
221
222     printf("\n fin del proceso padre\n\n") ;
223     return(0) ;
224 }
```

C source file      length : 6.606    lines : 224      Ln : 180    Col : 39    Sel : 0 | 0      Unix (LF)      UTF-8      INS

- Compilar y ejecutar el programa **despara2.c** (con las modificaciones que has realizado) para descifrar en paralelo las claves cifradas correspondientes a “**esesi**” y “**ojosi**”. Comprobar que localiza las claves y anotar en la quinta columna de la tabla 1 los tiempos empleados en la búsqueda paralela de las claves. ¿Qué podemos concluir viendo ahora todos los tiempos de la Tabla 1? [gráficas de tiempo de búsqueda de una clave x]



**Importante:** Para que el programa **despara2** se considere correctamente realizado, todos los procesos que se creen durante la ejecución del programa deben acabar escribiendo un mensaje en la pantalla, concretamente el que aparece antes del último **return** que ejecutan. Dicho de otra manera, todos los procesos deben acabar de forma natural (con un código de terminación mayor que 255) y no siendo matados por una señal (código de terminación menor o igual que 255). El alumno debe darse cuenta de la importancia de esta exigencia imaginándose la mejora de **despara2** permitiendo la búsqueda simultánea de varias claves a partir de sus claves cifradas:

**despara\_n claveCifrada\_1 claveCifrada\_2 ... claveCifrada\_n**

## 6 PUNTUACIÓN

El apartado 2 “Llamadas relacionadas con procesos” supone 0,3 puntos, el apartado 3 “Llamadas relacionadas con ficheros” aporta 0,1 puntos, el apartado 4 “Llamadas para comunicar procesos” aporta 0,2 puntos y el último apartado “Llamadas relacionadas con señales” supone los restantes 0,4 puntos. Los apartados deben ir completándose de forma secuencial. La puntuación total de esta práctica es por lo tanto de un punto (1,0).