# PRÁCTICA 2

## Uso de llamadas al sistema

## (userland)

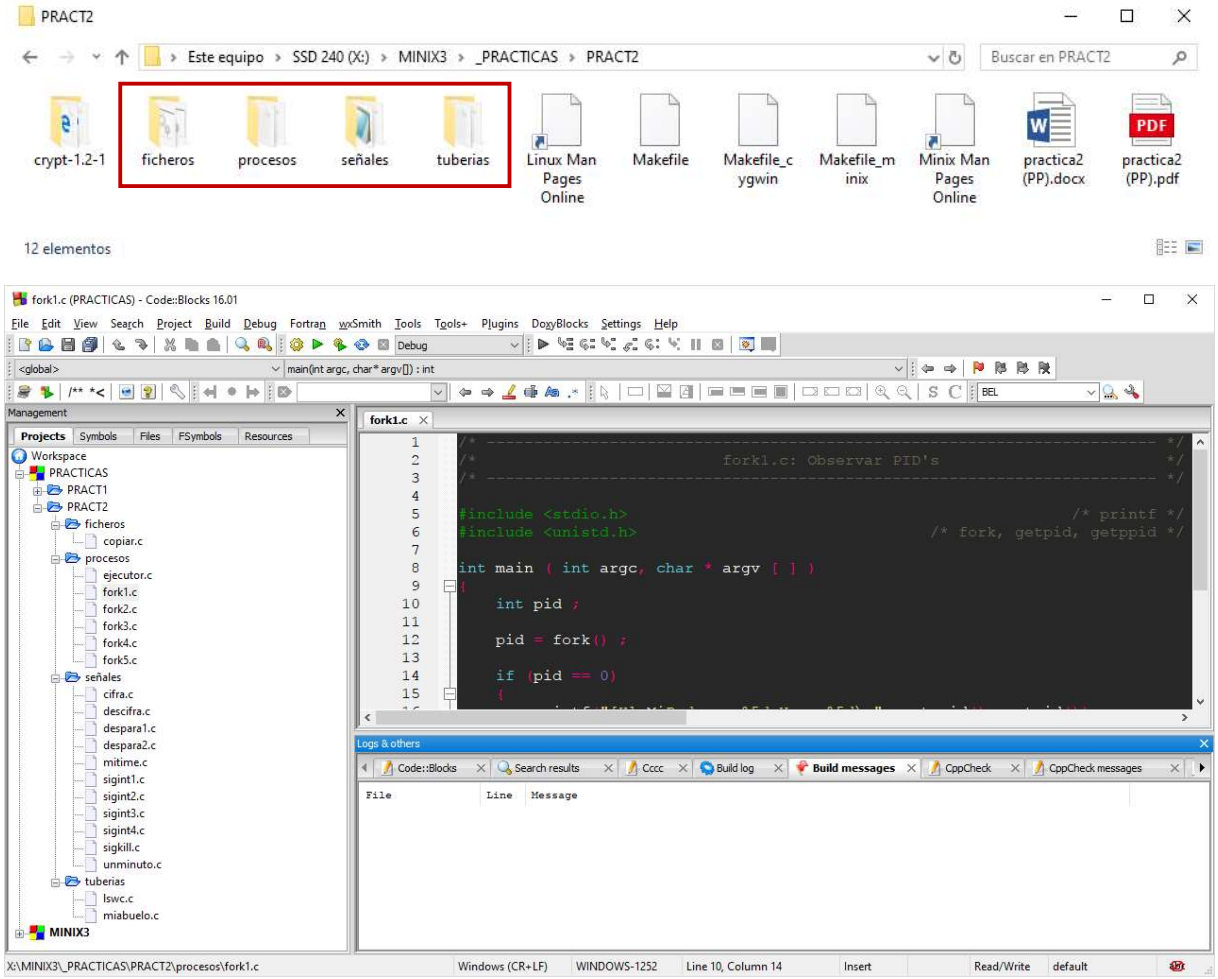
# ÍNDICE

# 1 OBJETIVOS

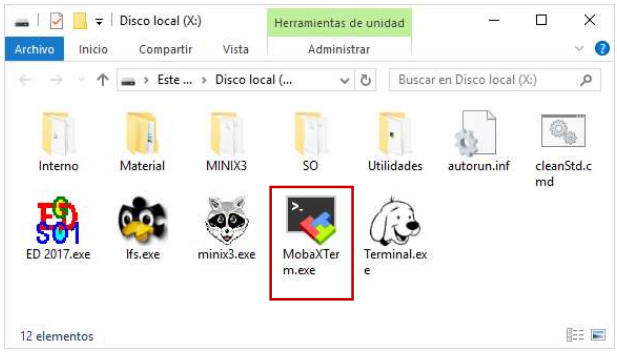
El objetivo de esta práctica es mostrar al alumno cuáles son los **servicios** básicos ofrecidos por el sistema operativo a los **programas de usuario**. Aunque los programas que hacen uso de esos servicios no tienen porqué haber sido escritos en lenguaje de alto nivel, vamos a restringirnos de momento 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:

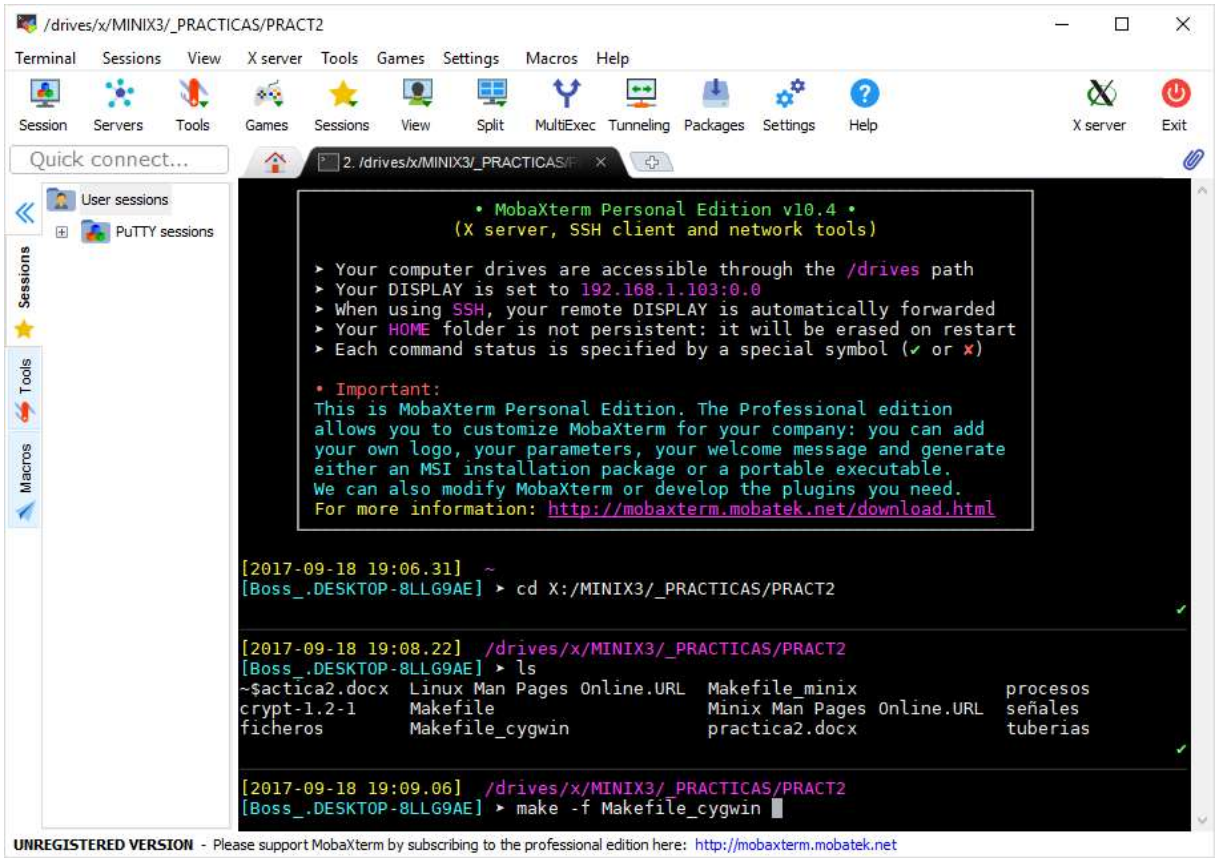
- Uso de llamadas relacionadas con procesos: **fork**, **wait**, **exit** y **exec (getpid**, **getppid**)
- Uso de llamadas relacionadas con ficheros: **open**, **read**, **write** y **close**
- Uso de llamadas para comunicar procesos: **pipe** y **dup2**
- Uso de llamadas relacionadas con señales: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

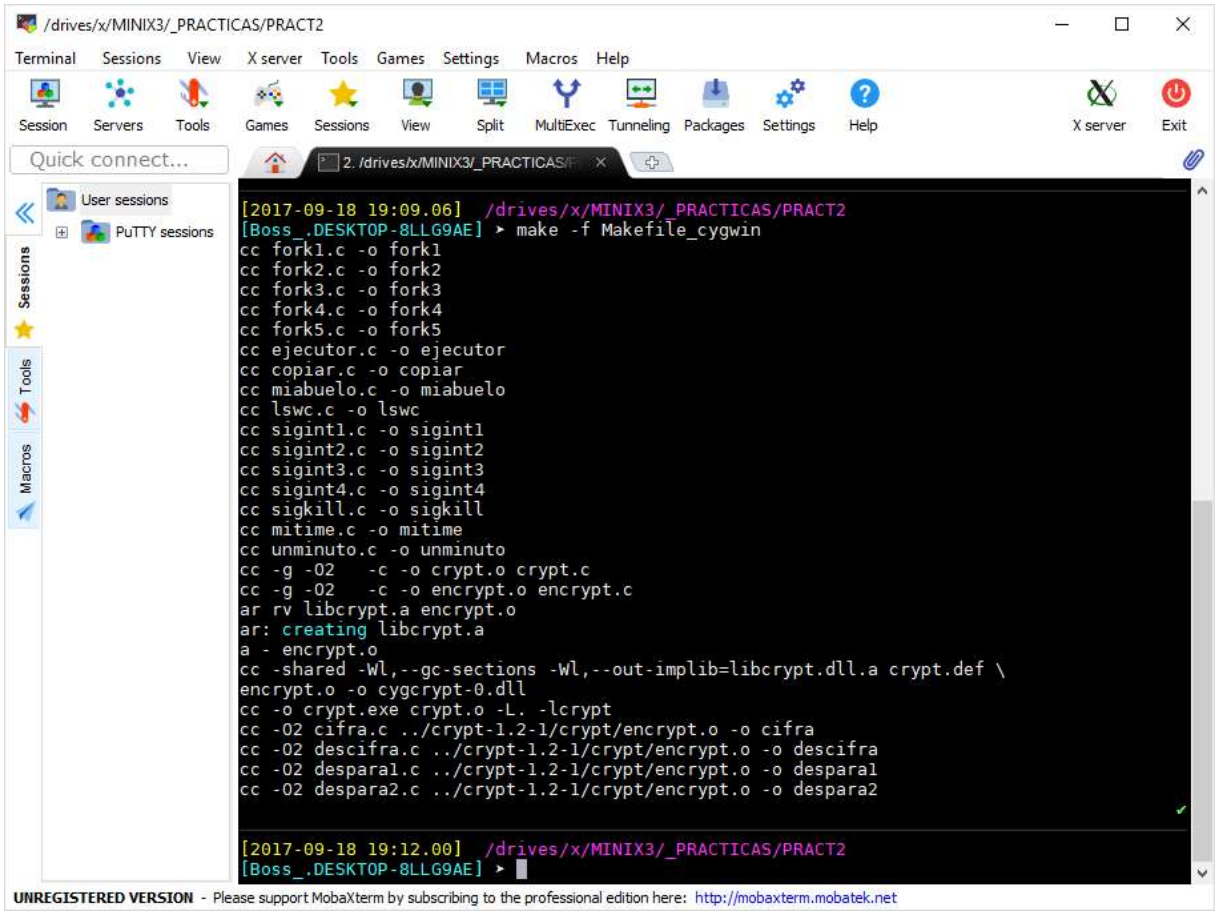


Como se muestra en la figura anterior, podemos ver todos esos programas abriendo con Codeblocks el fichero X:\MINIX3\MINIX3.workspace. Antes de llevar los programas a Minix es conveniente comprobar rápidamente en Windows que tenemos una copia operativa de esos programas. Con ese fin lanzamos **X:\MobaXTerm.exe**, 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**.

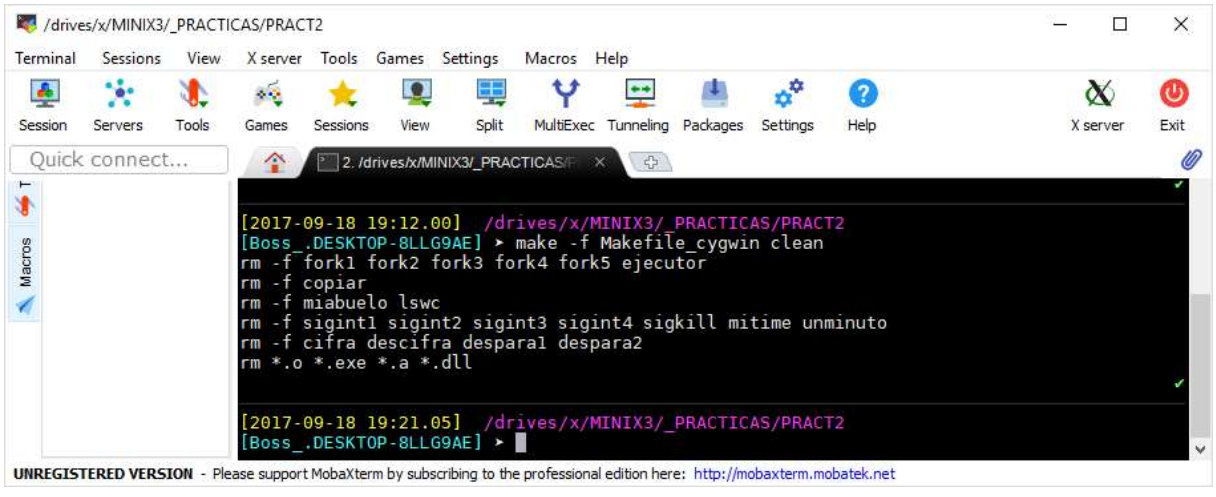


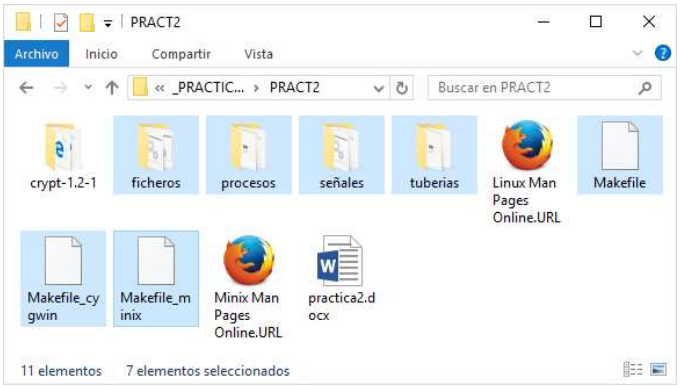

Si la salida del comando **make -f Makefile_cygwin** coincide con la de la figura siguiente, es que partimos de una colección de programas correcta. Podríamos ejecutar los programas compilados desde el intérprete de comandos utilizado, o desde Windows (Cygwin), pero nuestro objetivo es ejecutarlos, a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

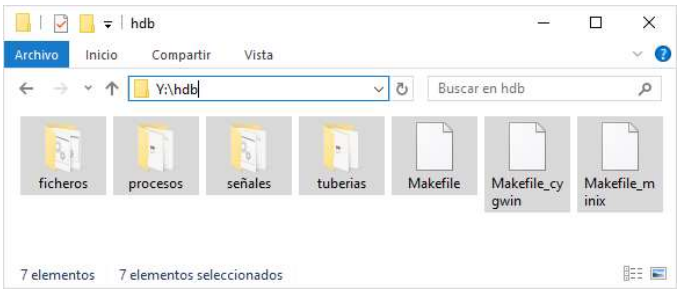



Para llevar a qemu los programas anteriores, conviene eliminar primero los resultados de la compilación anterior sobre Cygwin, para lo cual ejecutamos un comando similar al anterior pero especificando como objetivo, clean: **make -f Makefile_cygwin clean**.

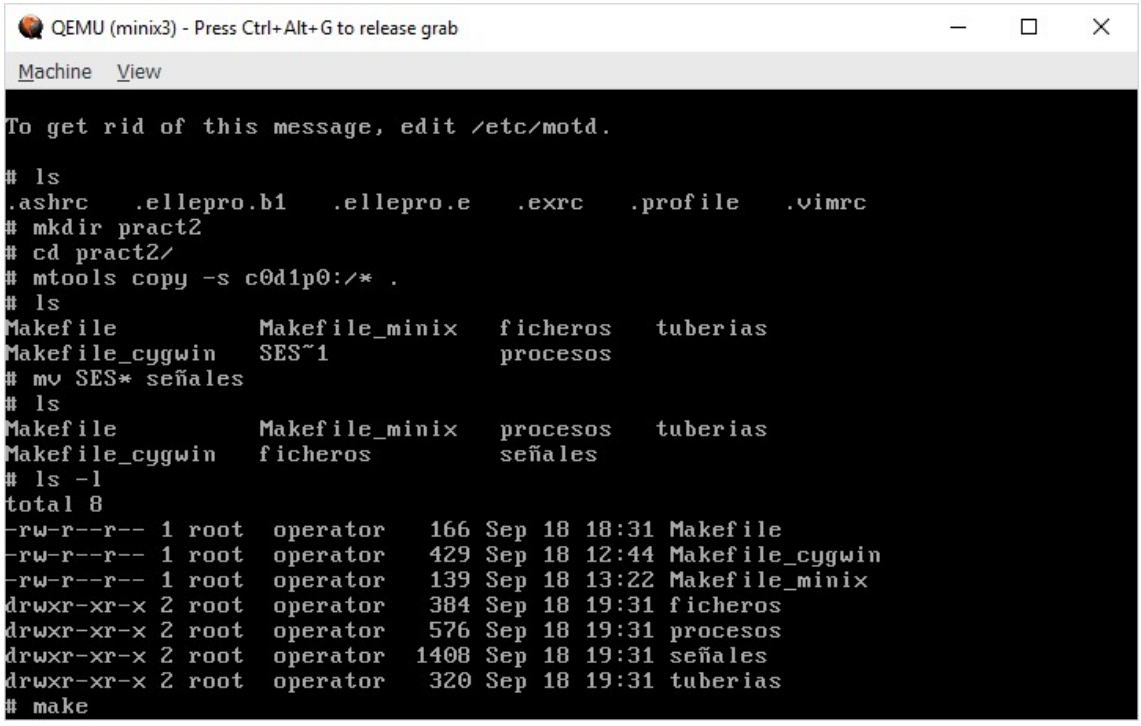



A continuación, utilizando la interfaz gráfica de Windows (el explorador de ficheros) copiamos las cuatro carpetas: ficheros, procesos, señales y tuberías, junto con los tres ficheros Makefile al directorio **Y:\hdb** (%TMP%\-so-\hdb).

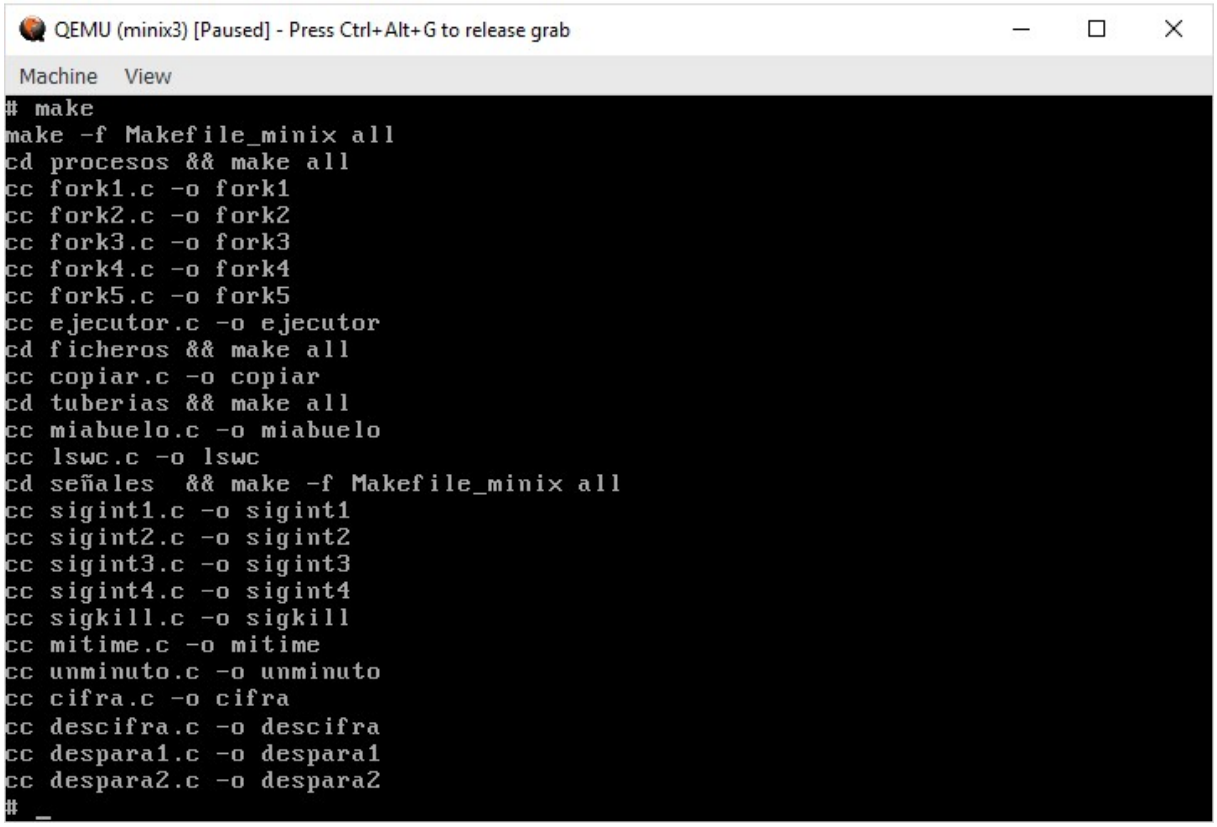



Ahora arrancamos la máquina virtual **X:\minix3.exe**  , entramos como root, creamos un directorio /root/pract2, nos movemos a ese directorio y copiamos a él todo el contenido de Y:\hdb, gracias al comando **mtools copy -s c0d1p0:/* .** .




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**.




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.

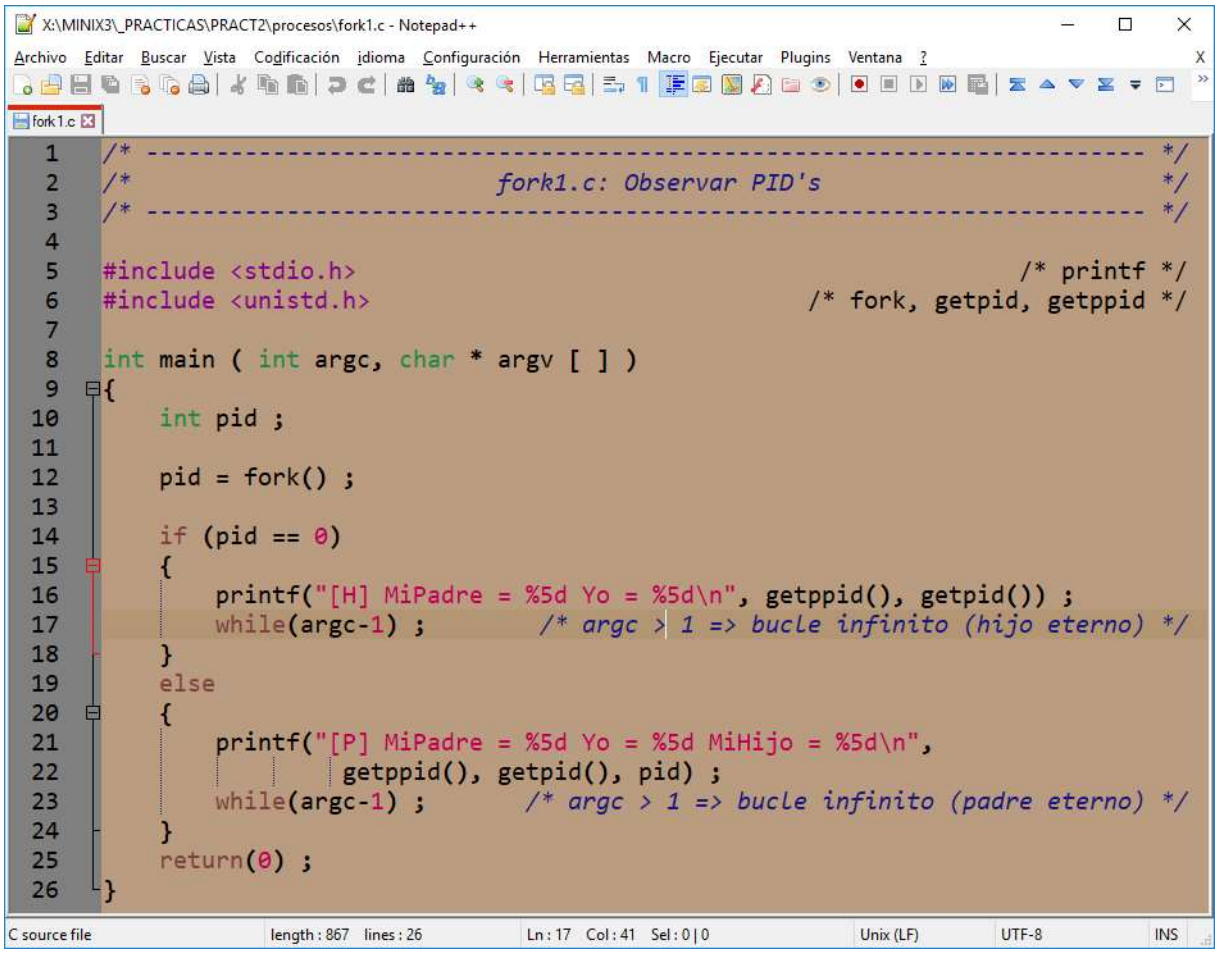
## 2 LLAMADAS RELACIONADAS CON PROCESOS

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**



- ¿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?

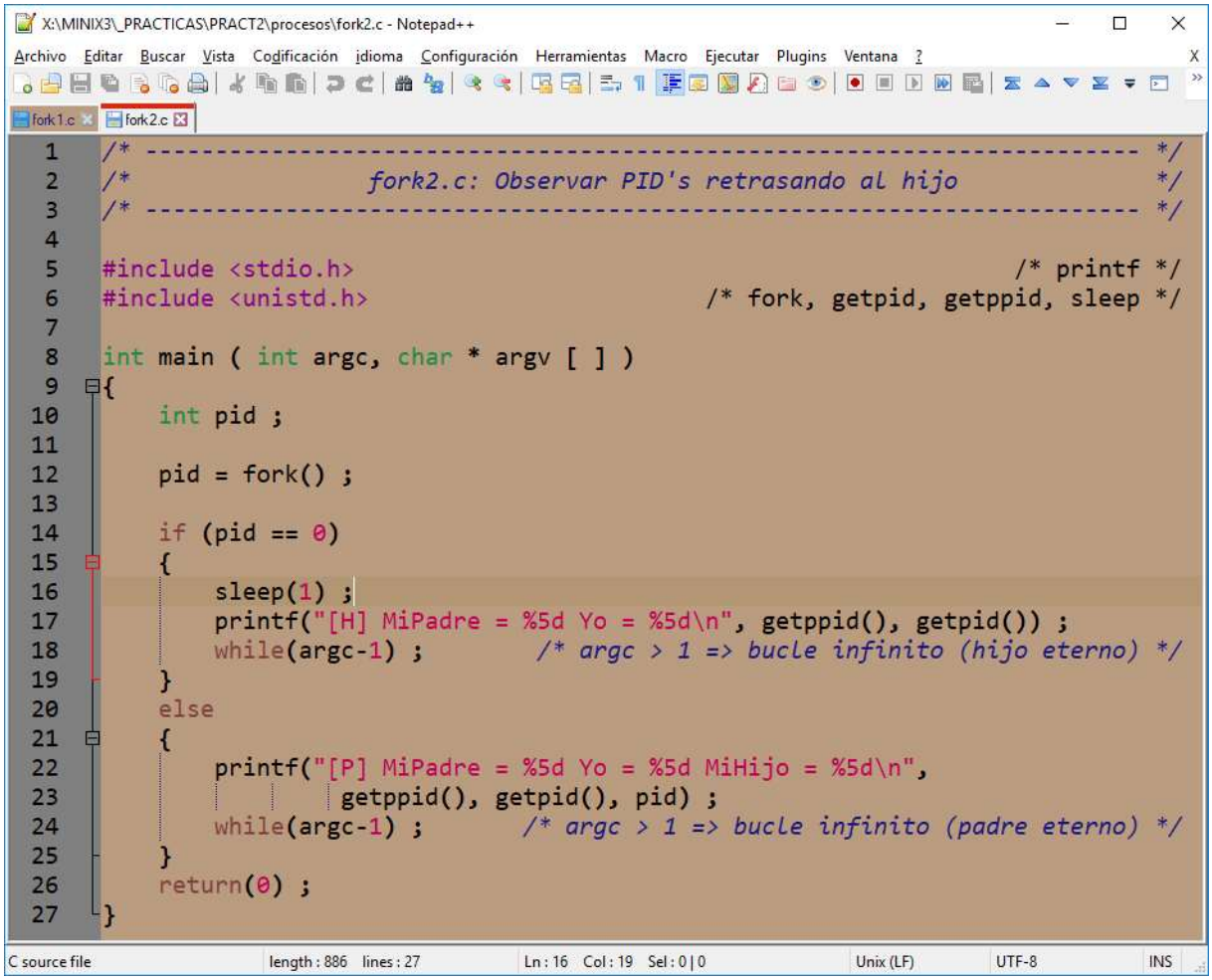
- Ejecutar en otra consola (Alt + F2) el comando **ps -l** y escribir (y explicar el significado de) las líneas correspondientes al proceso padre e hijo (consultar el manual: **man ps**).

- 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**



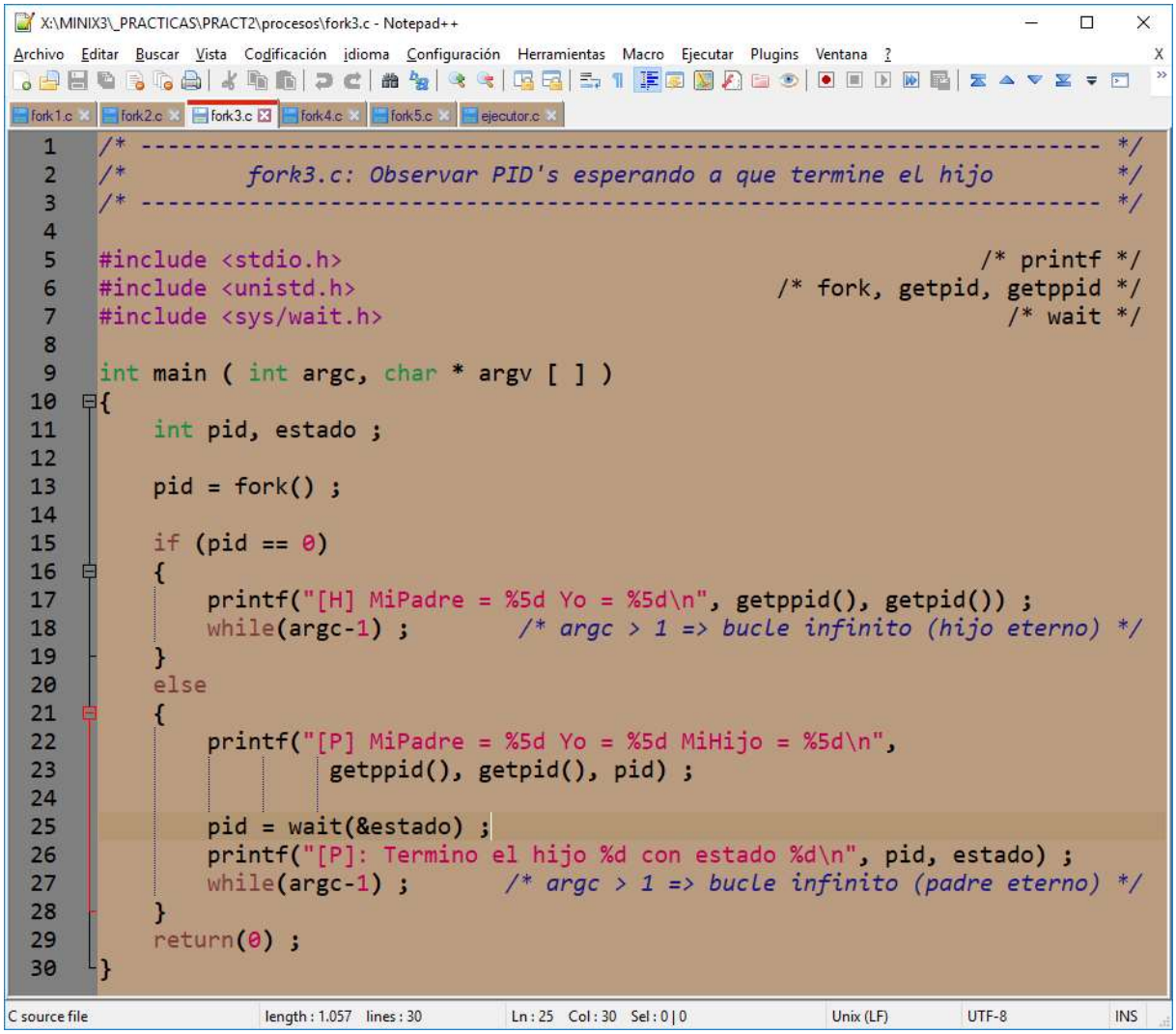

- Observar que se ha introducido un **retardo pasivo** de un segundo, **sleep(1)**, en el código del proceso hijo (ver **sleep**).

- 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:




- 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.

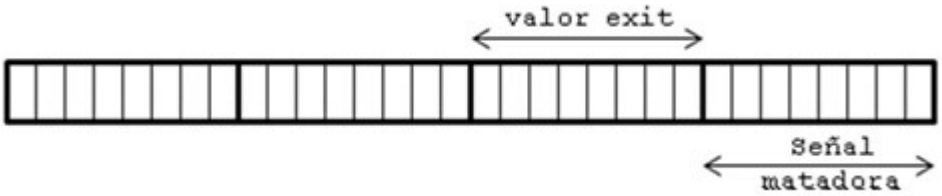
- ¿Qué se ha observado en relación con el estado de terminación del proceso hijo?

- 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 si 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:

X:\MINIX3_PRACTICAS\PRACT2\procesos\fork4.c - Notepad++

Archivo Editar Buscar Vista Codificación idioma Configuración Herramientas Macro Ejecutar Plugins Ventana ?

fork1.c fork2.c fork3.c fork4.c

```c
/* ------------------------------------------------------------------- */
/*       fork4.c: Observar PID's con terminacion explicita del hijo    */
/* ------------------------------------------------------------------- */

#include <stdio.h>                                          /* printf */
#include <unistd.h>                    /* fork, getpid, getppid, sleep */
#include <stdlib.h>                                          /* exit */
#include <sys/wait.h>                          /* wait, _LOW, _HIGH */

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

    pid = fork() ;

    if (pid == 0)
    {
        printf("[H] MiPadre = %5d Yo = %5d\n", getppid(), getpid()) ;
        exit(3) ;                                  /* bastaria con return(3) */
    }
    else
    {
        printf("[P] MiPadre = %5d Yo = %5d MiHijo = %5d\n",
                getppid(), getpid(), pid) ;
        while(argc-1) ;                          /* argc > 1 => bucle infinito */
        pid = wait(&estado) ;
        printf("[P]: Termino el hijo %d con estado (%d, %d)\n",
                pid, _HIGH(estado), _LOW(estado)) ;
    }
    return(0) ;
}
```

C source file | length : 1.178 lines : 31 | Ln : 28 Col : 53 Sel : 0 | 0 | Unix (LF) | UTF-8 | INS

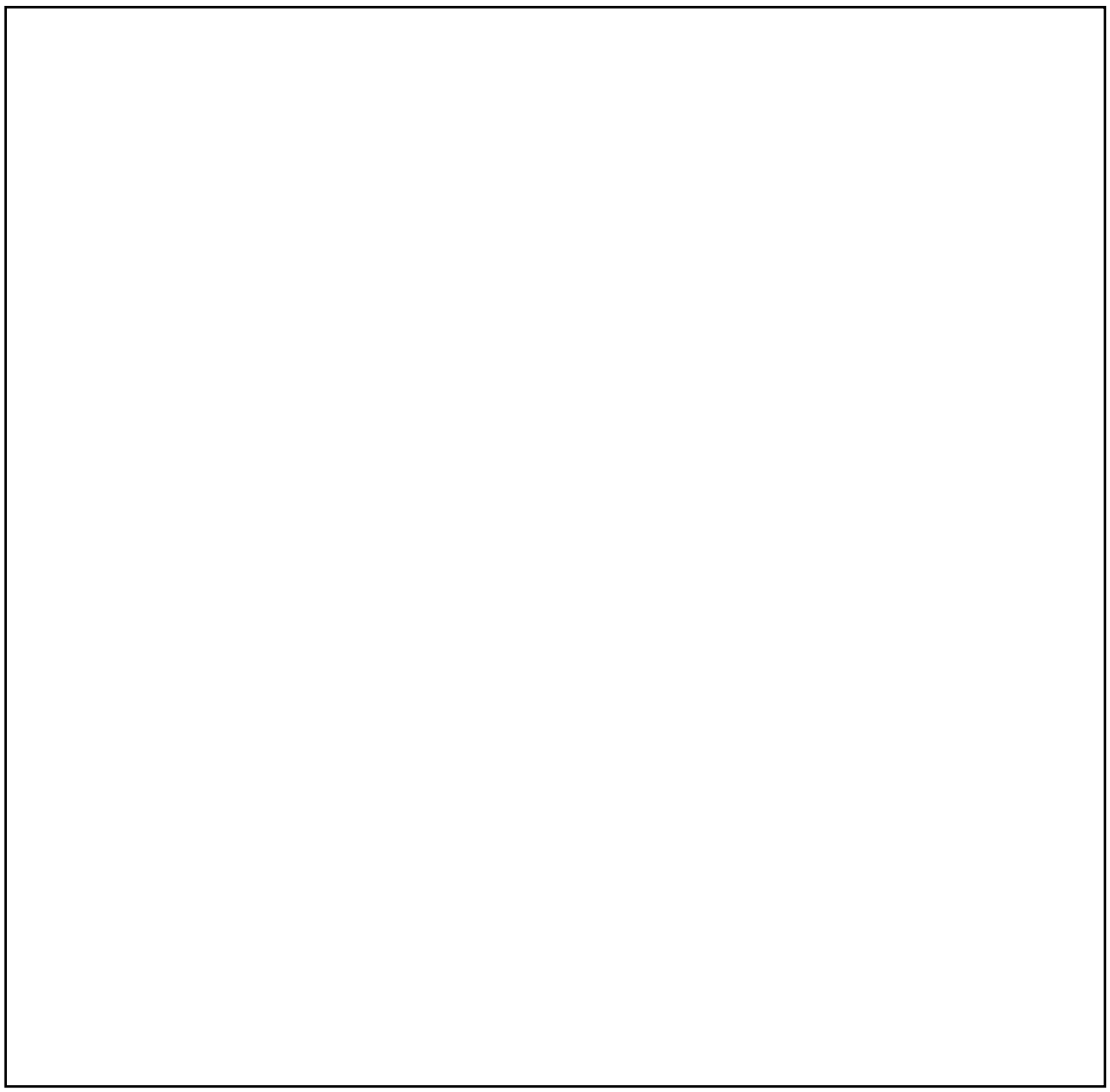
- Observar que se ha introducido una terminación explícita del hijo que devuelve el estado "3" (0x0000**0300**) al padre.

- 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:

X:\MINIX3_PRACTICAS\PRACT2\procesos\fork5.c - Notepad++

Archivo Editar Buscar Vista Codificación Idioma Configuración Herramientas Macro Ejecutar Plugins Ventana ?

fork1.c fork2.c fork3.c fork4.c fork5.c

```c
/* ------------------------------------------------------------------- */
/*      fork5.c: Observar los efectos de la multiplexacion de la CPU   */
/* ------------------------------------------------------------------- */

#include <stdio.h>                                      /* printf, sscanf */
#include <unistd.h>                                        /* fork, write */

void retardar ( int vueltas )
{
    int global [100], aux, i ;
    while (vueltas > 0)
    {
        vueltas-- ;
        i = vueltas % 100 ;
        aux            = global[99 - i] ;
        global[99 - i] = global[i] ;
        global[i]      = aux ;
    }
}

void imprimirIntervalo ( char min, char max, int retardo )
{
    char car = min ;
    while (min <= max)                    /* o no se entra o bucle infinito */
    {
        printf("%c", car) ;              /* write(STDOUT_FILENO, &car, 1) ; */
        if (++car > max)
            car = min ;
        retardar(retardo) ;
    }
}

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

    if ((argc != 2) || ((sscanf(argv[1], "%d", &vueltasDeRetardo) != 1)))
    {
        vueltasDeRetardo = 10000 ;                    /* valor por defecto */
    }

    pid = fork() ;

    if (pid == 0)
        imprimirIntervalo('a', 'z', vueltasDeRetardo) ;          /* letras */
    else
        imprimirIntervalo('0', '9', vueltasDeRetardo) ;          /* cifras */
    return(0) ;
}
```

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.

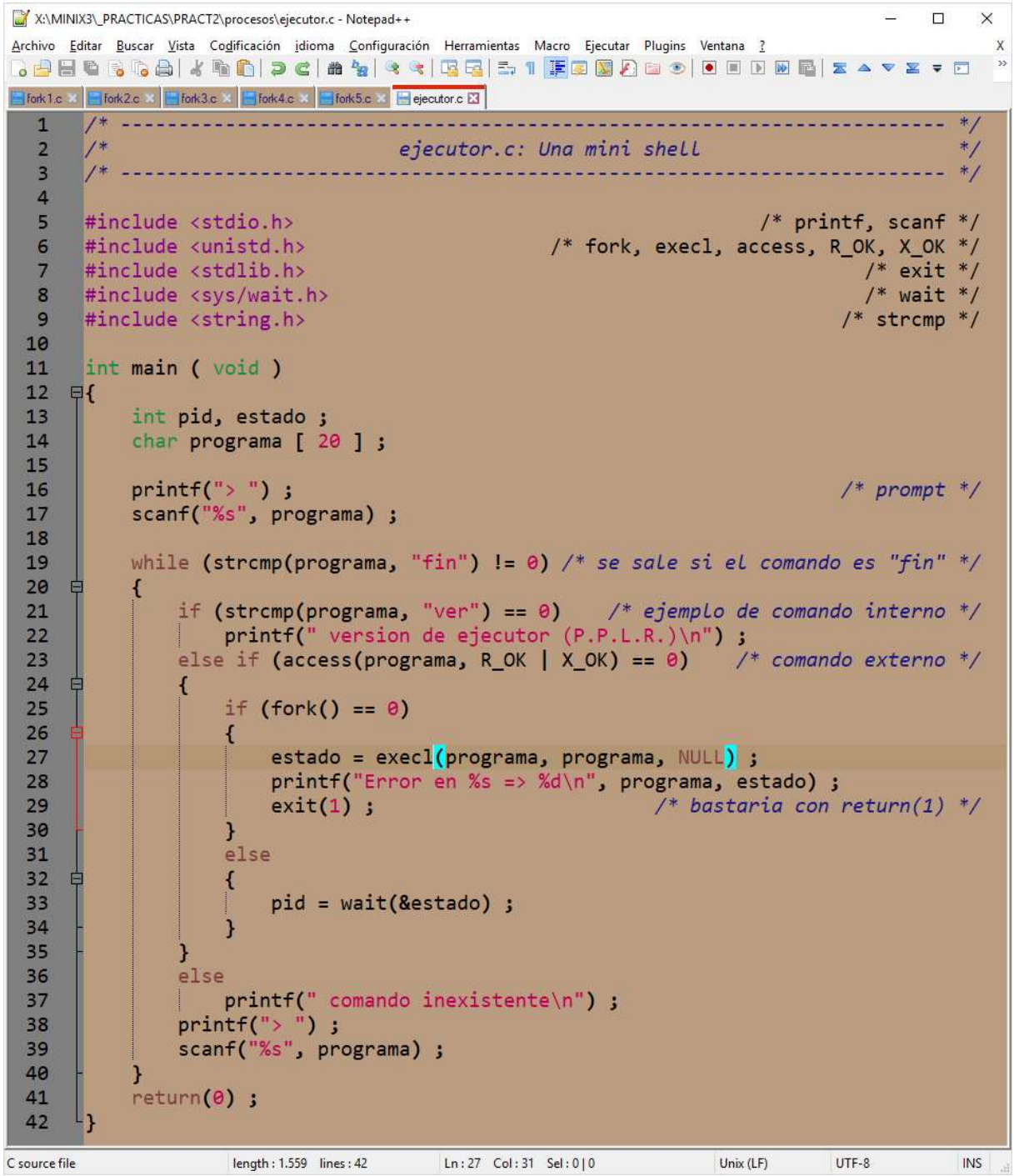
- ¿Ese número es siempre el mismo en las diferentes secuencias de letras seguidas que escribe el proceso hijo? ¿depende del parámetro **vueltasDeRetardo** de fork5?

- 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 **vueltasDeRetardo** 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).



```c
/* ------------------------------------------------------------------- */
/*                          ejecutor.c: Una mini shell                 */
/* ------------------------------------------------------------------- */

#include <stdio.h>                                          /* printf, scanf */
#include <unistd.h>                          /* fork, execl, access, R_OK, X_OK */
#include <stdlib.h>                                                 /* exit */
#include <sys/wait.h>                                               /* wait */
#include <string.h>                                               /* strcmp */

int main ( void )
{
    int pid, estado ;
    char programa [ 20 ] ;

    printf("> ") ;                                                  /* prompt */
    scanf("%s", programa) ;

    while (strcmp(programa, "fin") != 0) /* se sale si el comando es "fin" */
    {
        if (strcmp(programa, "ver") == 0)    /* ejemplo de comando interno */
            printf(" version de ejecutor (P.P.L.R.)\n") ;
        else if (access(programa, R_OK | X_OK) == 0)    /* comando externo */
        {
            if (fork() == 0)
            {
                estado = execl(programa, programa, NULL) ;
                printf("Error en %s => %d\n", programa, estado) ;
                exit(1) ;                          /* bastaria con return(1) */
            }
            else
            {
                pid = wait(&estado) ;
            }
        }
        else
            printf(" comando inexistente\n") ;
        printf("> ") ;
        scanf("%s", programa) ;
    }
    return(0) ;
}
```

- 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.

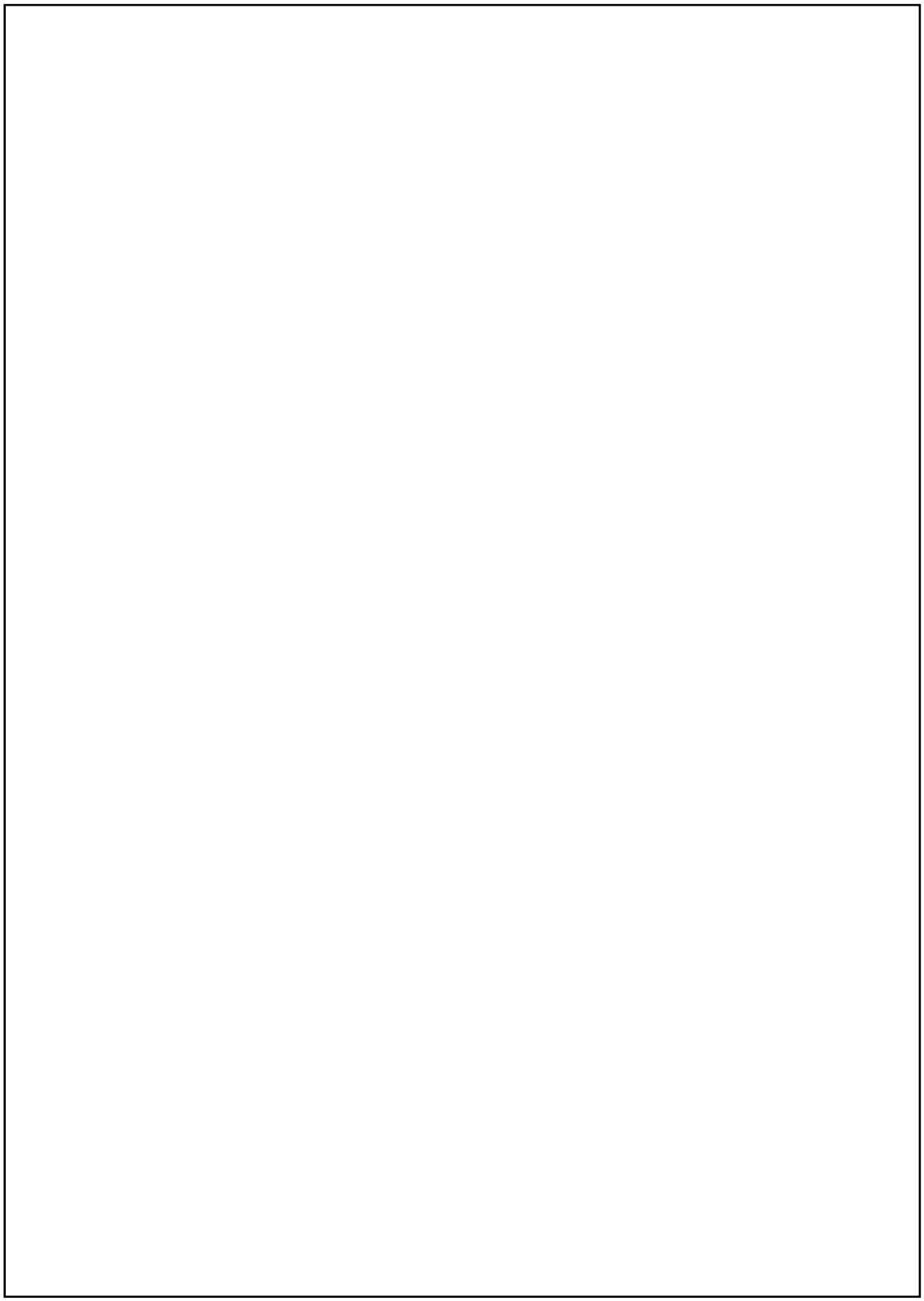
- Introducir ante el prompt de ejecutor el nombre de un programa que no exista, como por ejemplo **tenedor** o **cuchara**. ¿Qué sucede?

- 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:

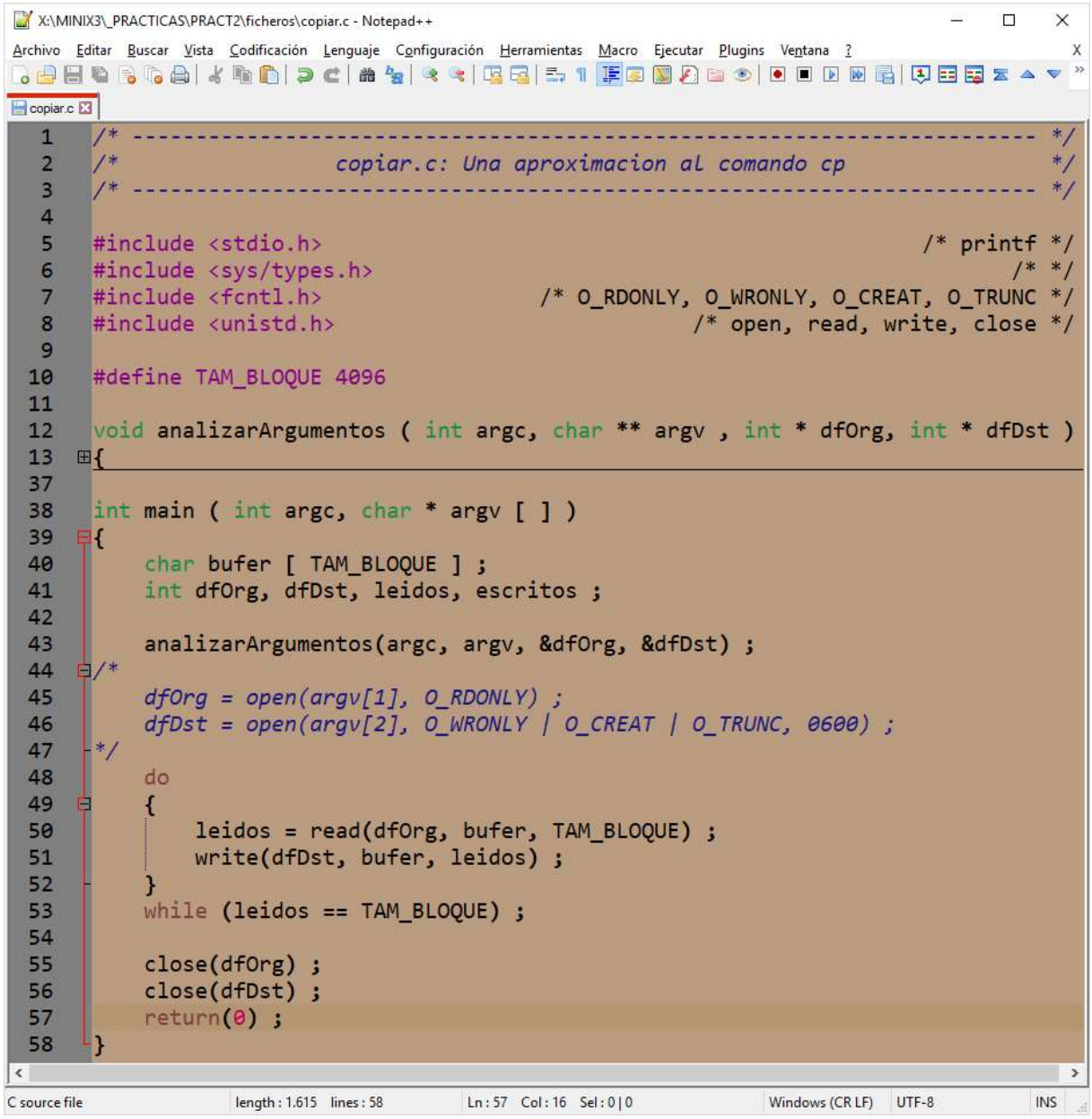
- Modificar ligeramente el programa **ejecutor.c** para que informe justo antes del *prompt,* del identificador del proceso ejecutor y así tener la certeza de qué proceso ejecutor nos está atendiendo. Indicar las modificaciones.



## 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:




```
/* ------------------------------------------------------------------- */
/*                  copiar.c: Una aproximacion al comando cp           */
/* ------------------------------------------------------------------- */

#include <stdio.h>                                          /* printf */
#include <sys/types.h>                                          /* */
#include <fcntl.h>                  /* O_RDONLY, O_WRONLY, O_CREAT, O_TRUNC */
#include <unistd.h>                             /* open, read, write, close */

#define TAM_BLOQUE 4096

void analizarArgumentos ( int argc, char ** argv , int * dfOrg, int * dfDst )
{

int main ( int argc, char * argv [ ] )
{
    char bufer [ TAM_BLOQUE ] ;
    int dfOrg, dfDst, leidos, escritos ;

    analizarArgumentos(argc, argv, &dfOrg, &dfDst) ;
/*
    dfOrg = open(argv[1], O_RDONLY) ;
    dfDst = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0600) ;
*/
    do
    {
        leidos = read(dfOrg, bufer, TAM_BLOQUE) ;
        write(dfDst, bufer, leidos) ;
    }
    while (leidos == TAM_BLOQUE) ;

    close(dfOrg) ;
    close(dfDst) ;
    return(0) ;
}
```

- 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.

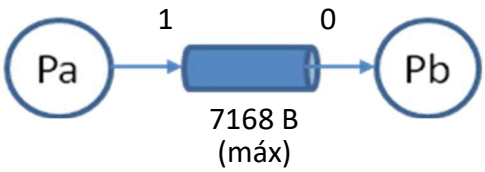
# 4 LLAMADAS PARA COMUNICAR PROCESOS

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:

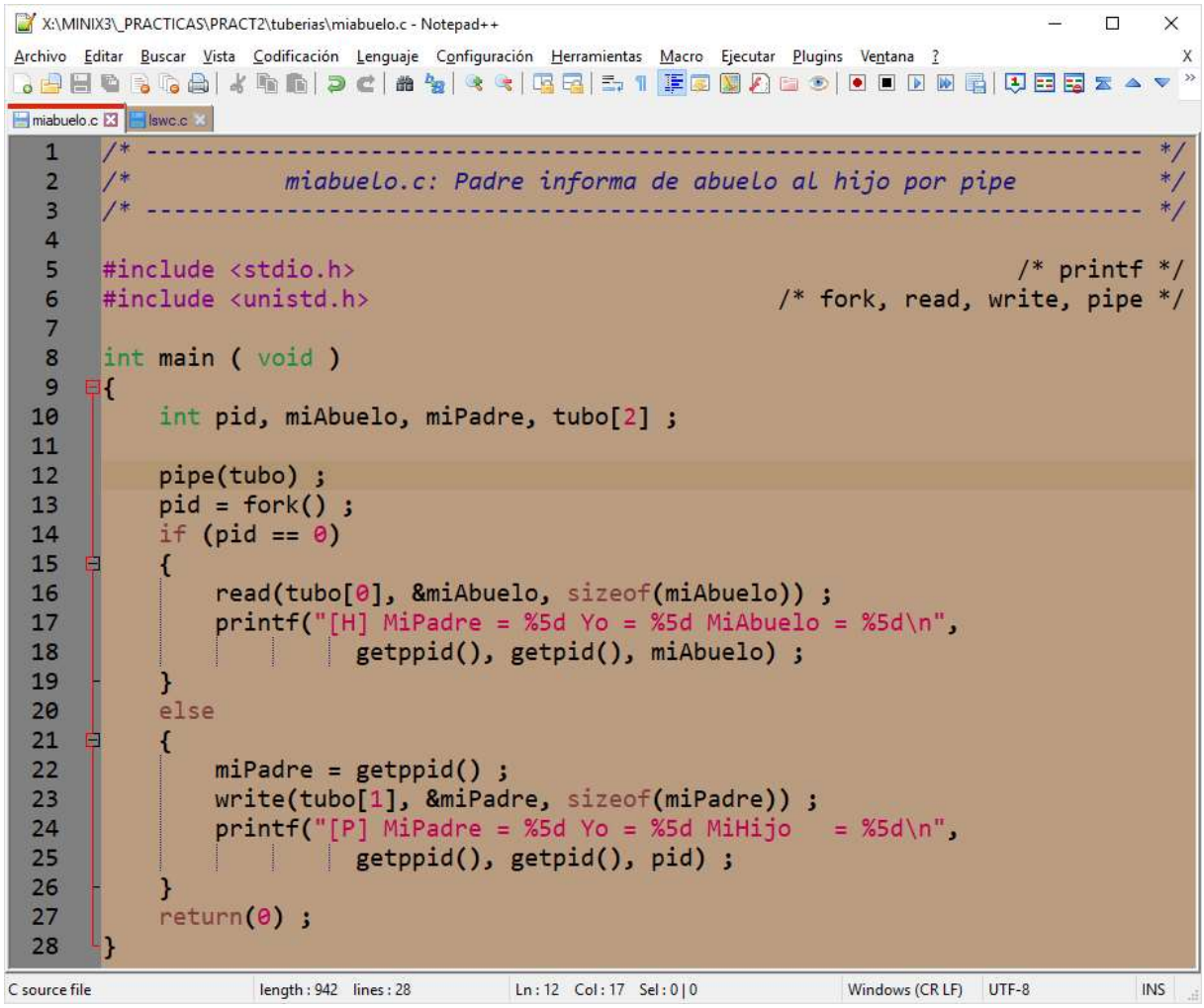
- p_fd[0] sólo lectura (O_RDONLY)
- p_fd[1] sólo escritura (O_WRONLY)




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:




- 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:

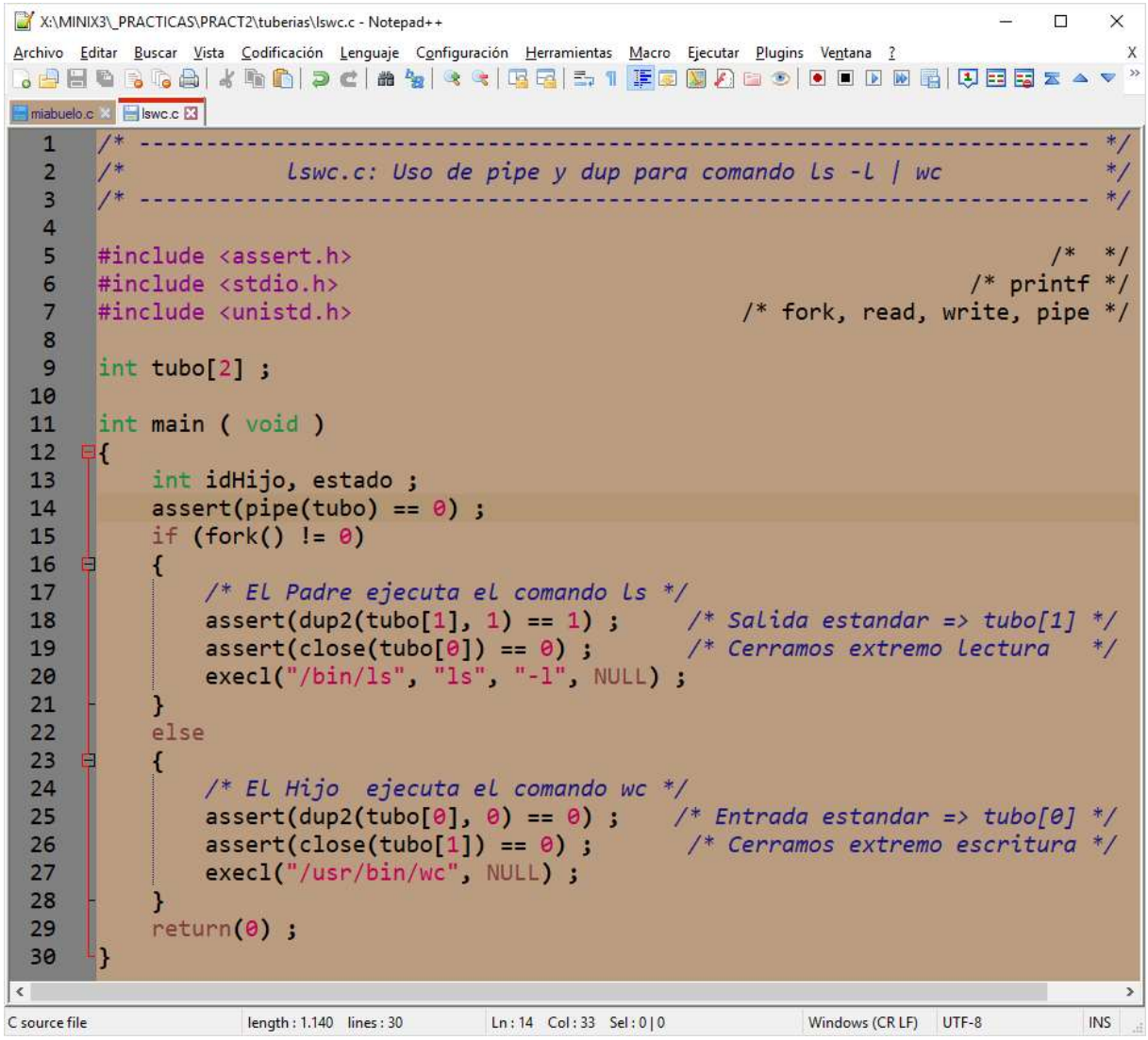
int **dup2** ( int old_fd, int new_fd )

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:



- 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 descriptores 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 */
```

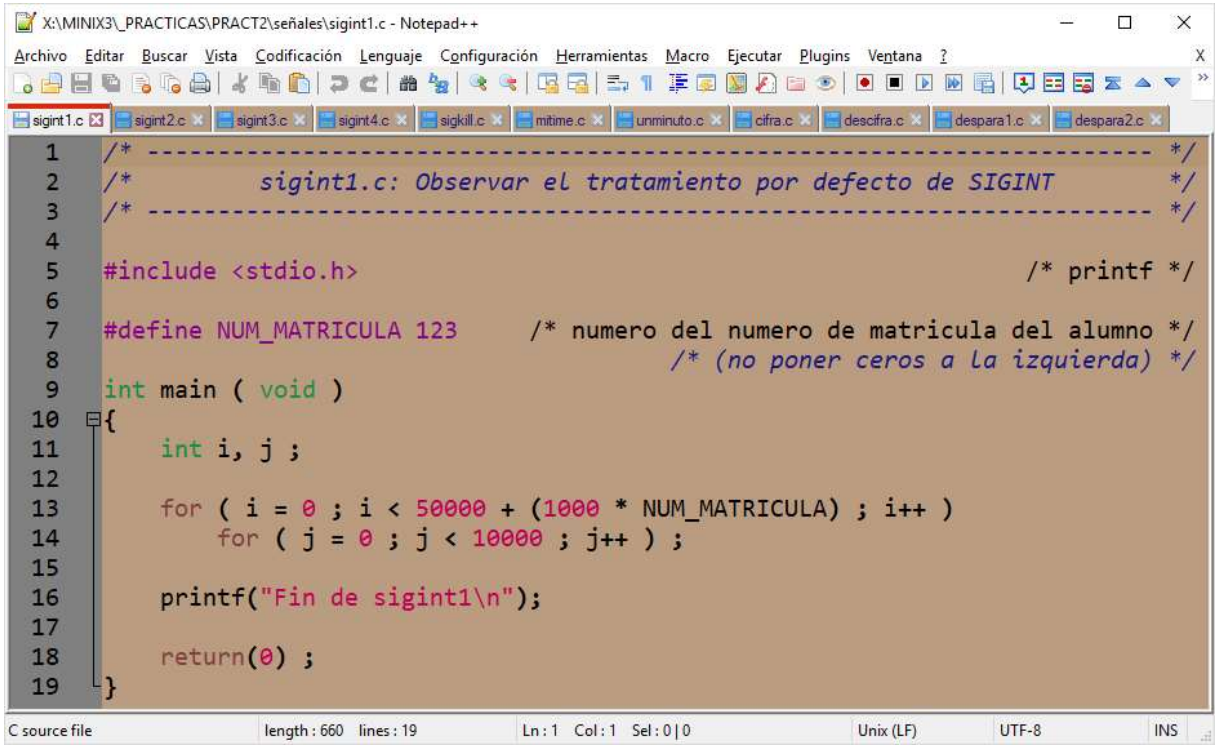
Puede observarse que (además de con **kill**) una señal también puede enviarse por errores de ejecución "excepciones hardware" como SIGILL –intento de ejecutar una instrucción ilegal– o SIGSEGV –intento de acceder a una dirección inválida–. La expiración de una temporización

(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:

a. Dejar el tratamiento por defecto (que es: matar al proceso y generar un volcado del contexto del proceso "*core dumped*" si procede)
b. Ignorar la señal (salvo para SIGKILL)
c. 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:




- 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**) ;

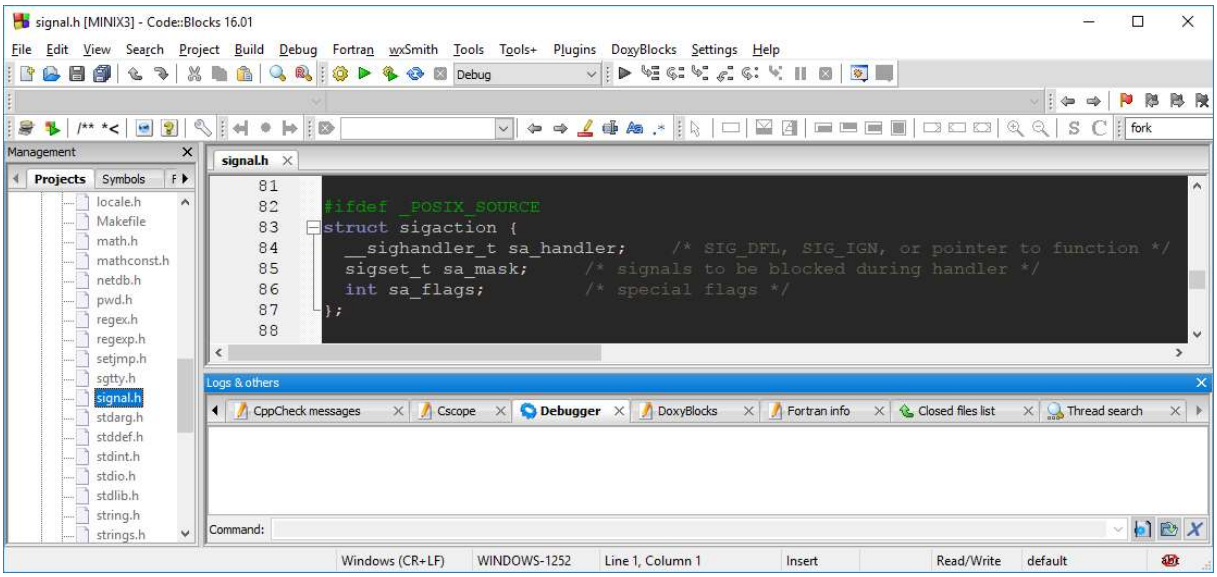
donde:

**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

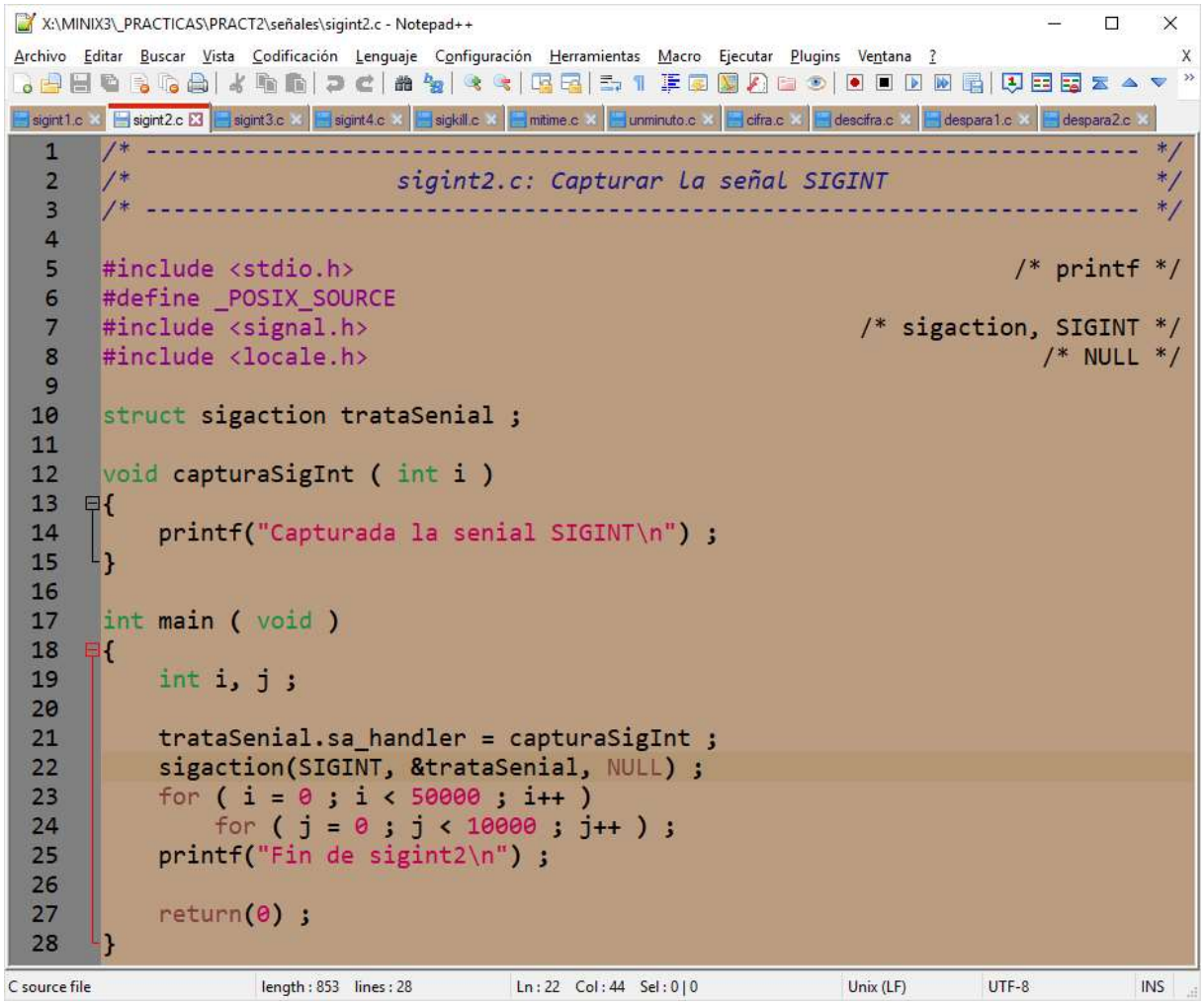
La estructura "**sigaction**" que permite definir el nuevo comportamiento ante la llegada de una señal, está declarada en el fichero **/usr/include/signal.h**:




```
struct sigaction {
  __sighandler_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).

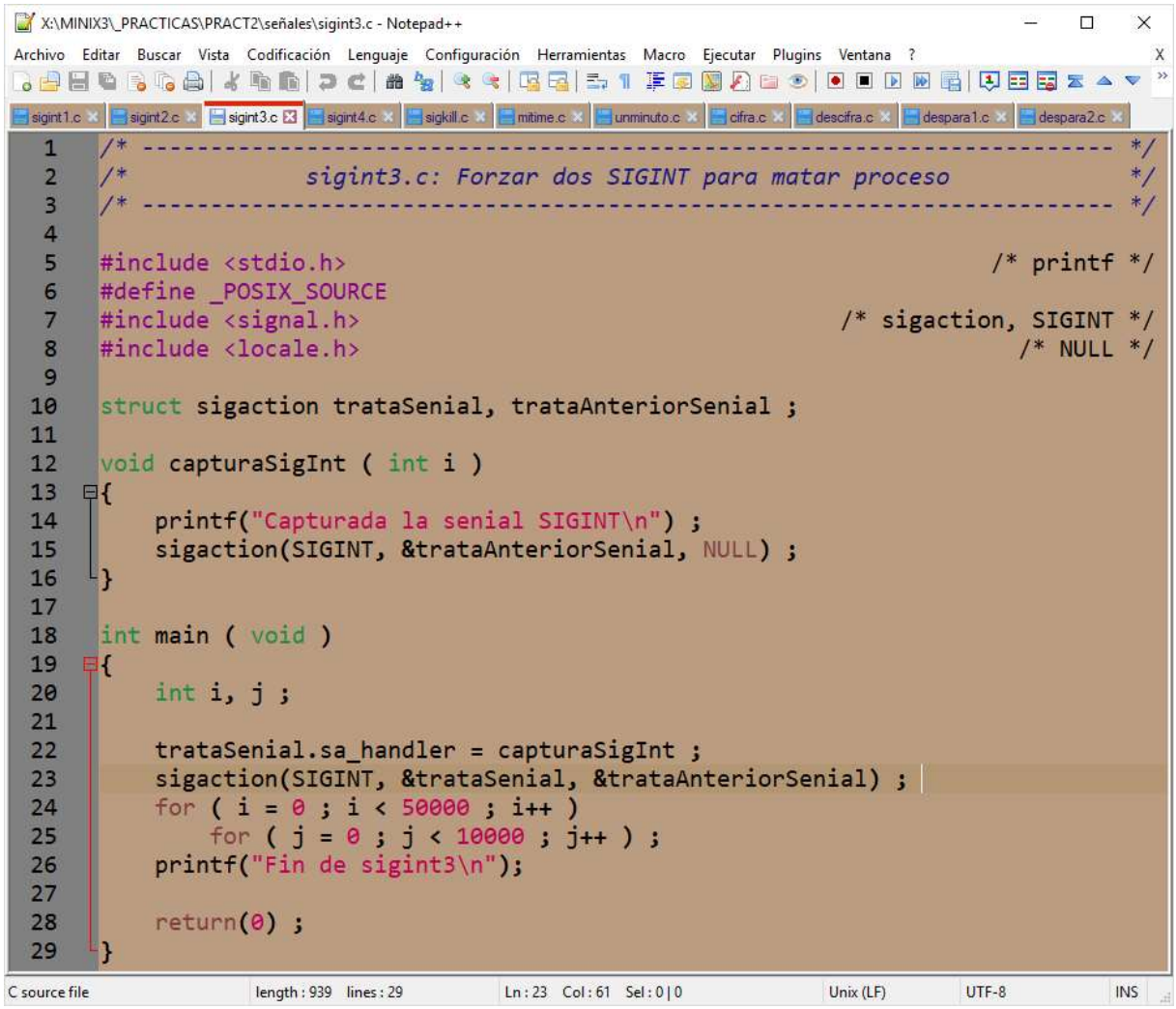
- Echar un vistazo al programa **sigint2.c** siguiente:




- 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.

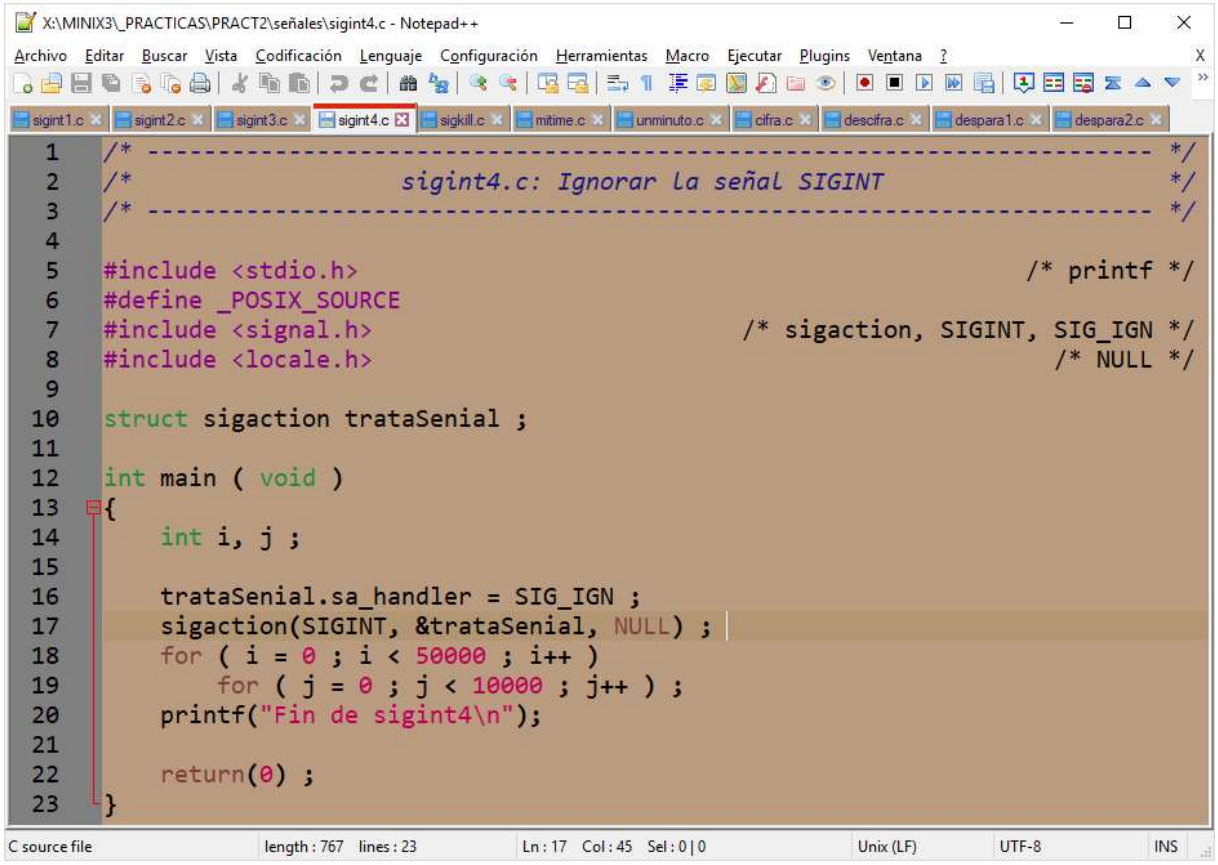
- Echar un vistazo al programa **sigint3.c**:




- Obseervar 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:




- Observar cómo se consigue ignorar la señal.

- Compilar y ejecutar el programa. Comprobar que no hay forma de abortar la ejecución con <Ctrl-C>.

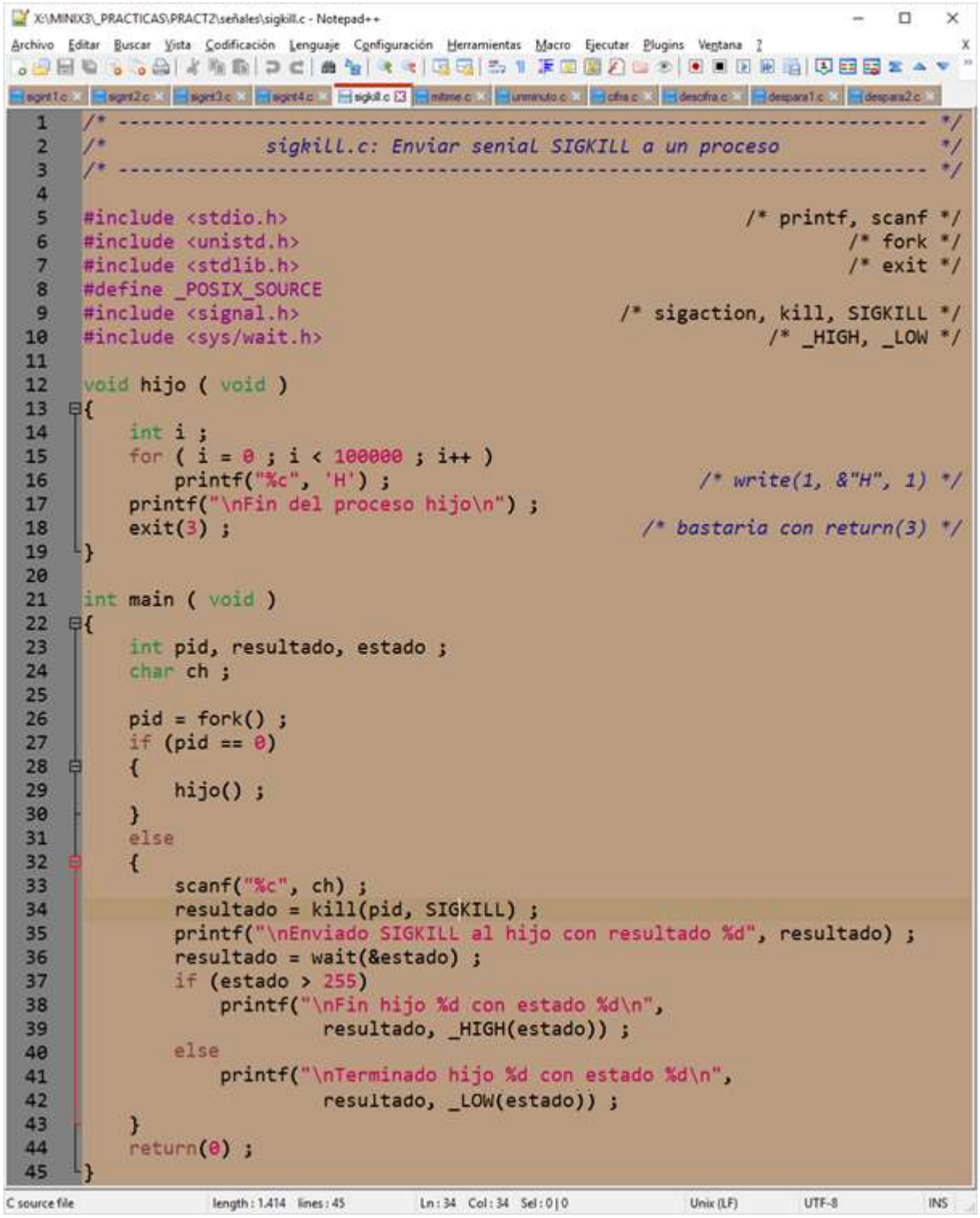
Ahora vamos a utilizar la llamada al sistema operativo **kill** cuyo encabezado es el siguiente:

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**:




- 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:

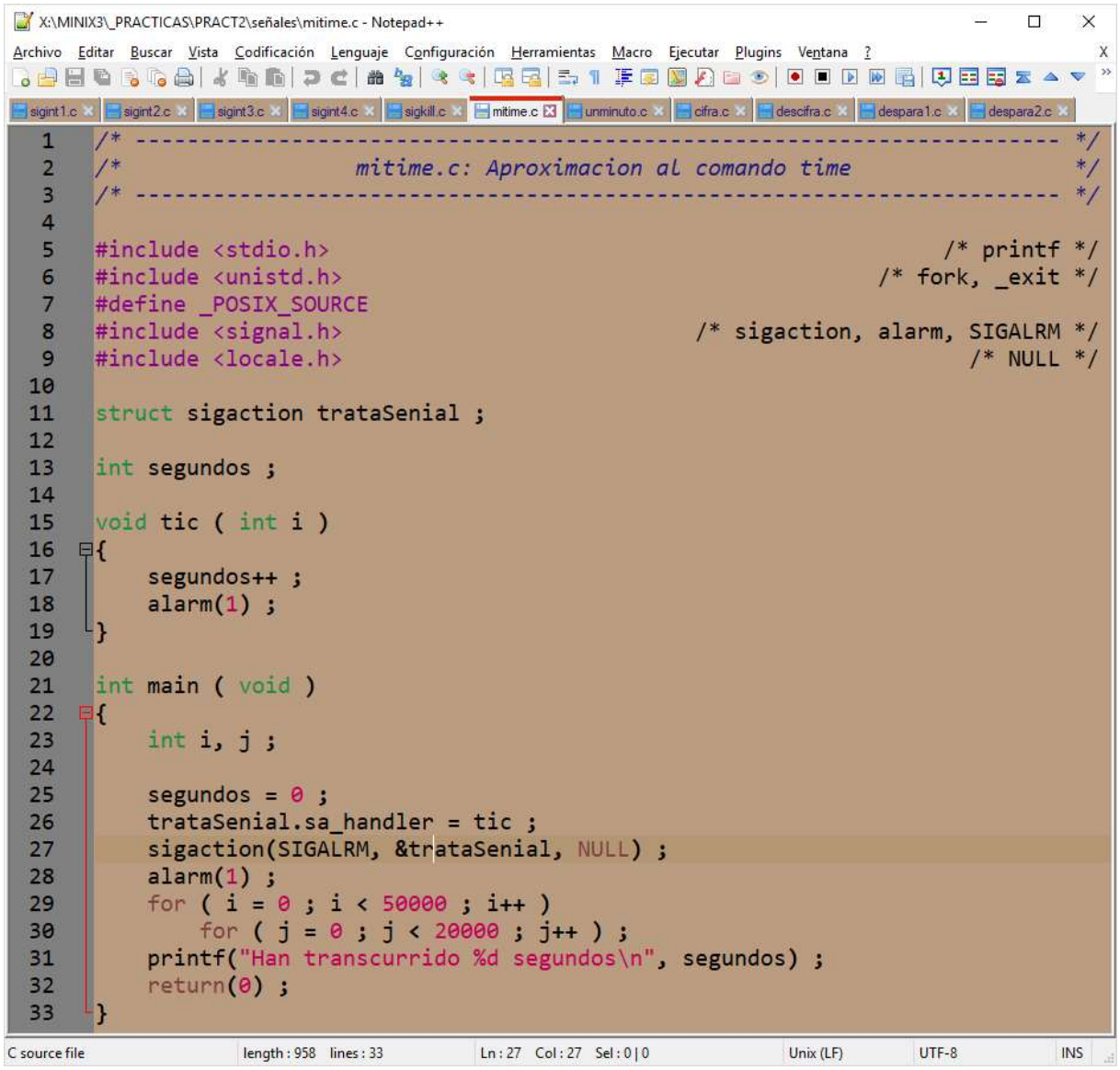
- ¿Cuál es la diferencia entre ambas ejecuciones y a qué es debido?

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:




- 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:

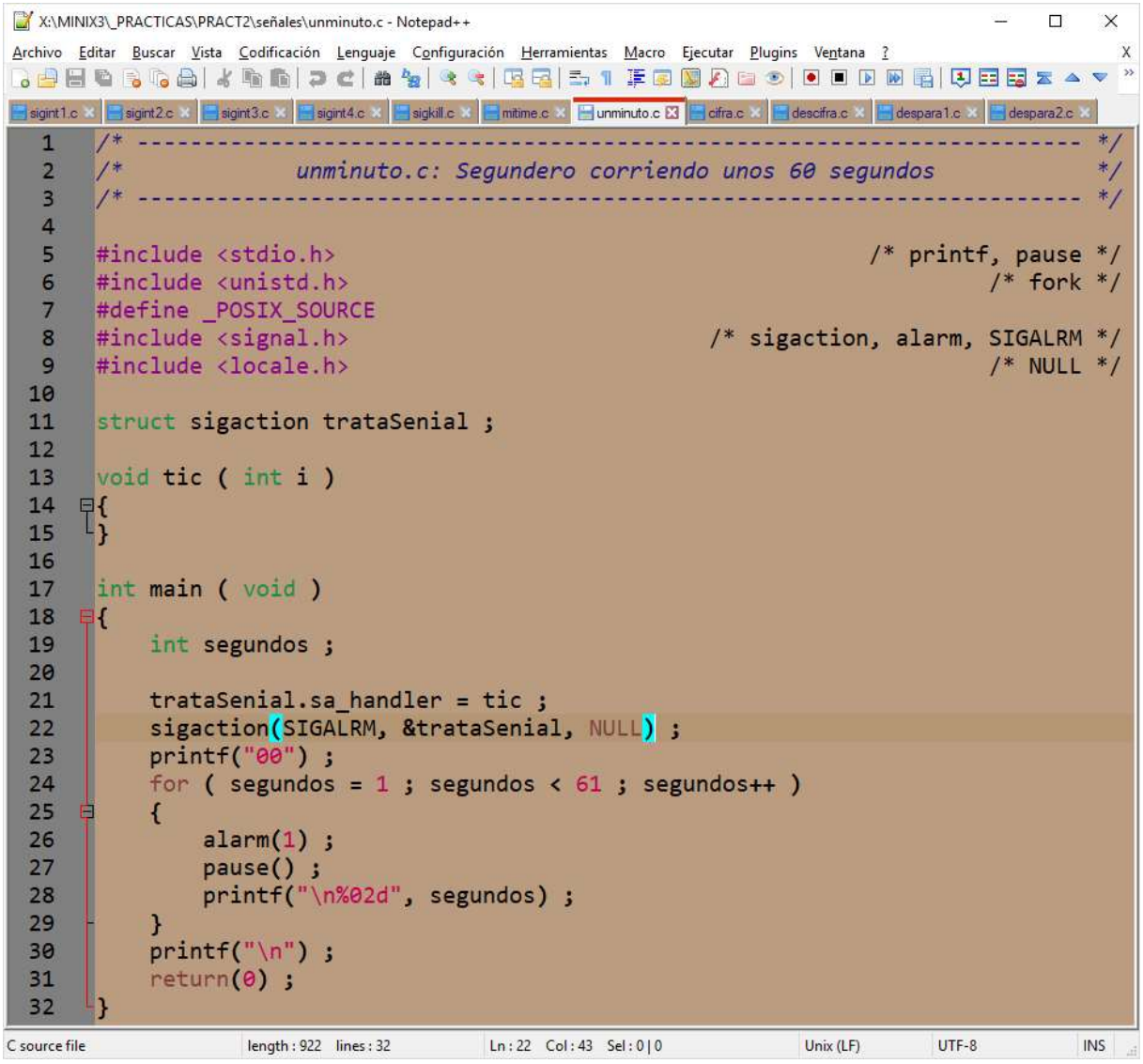
- Volver a ejecutar **mitime** pero ahora de la forma siguiente: **time ./mitime**. Anotar lo que sale por pantalla y explicar la diferencia:

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 pseudosegundero.

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

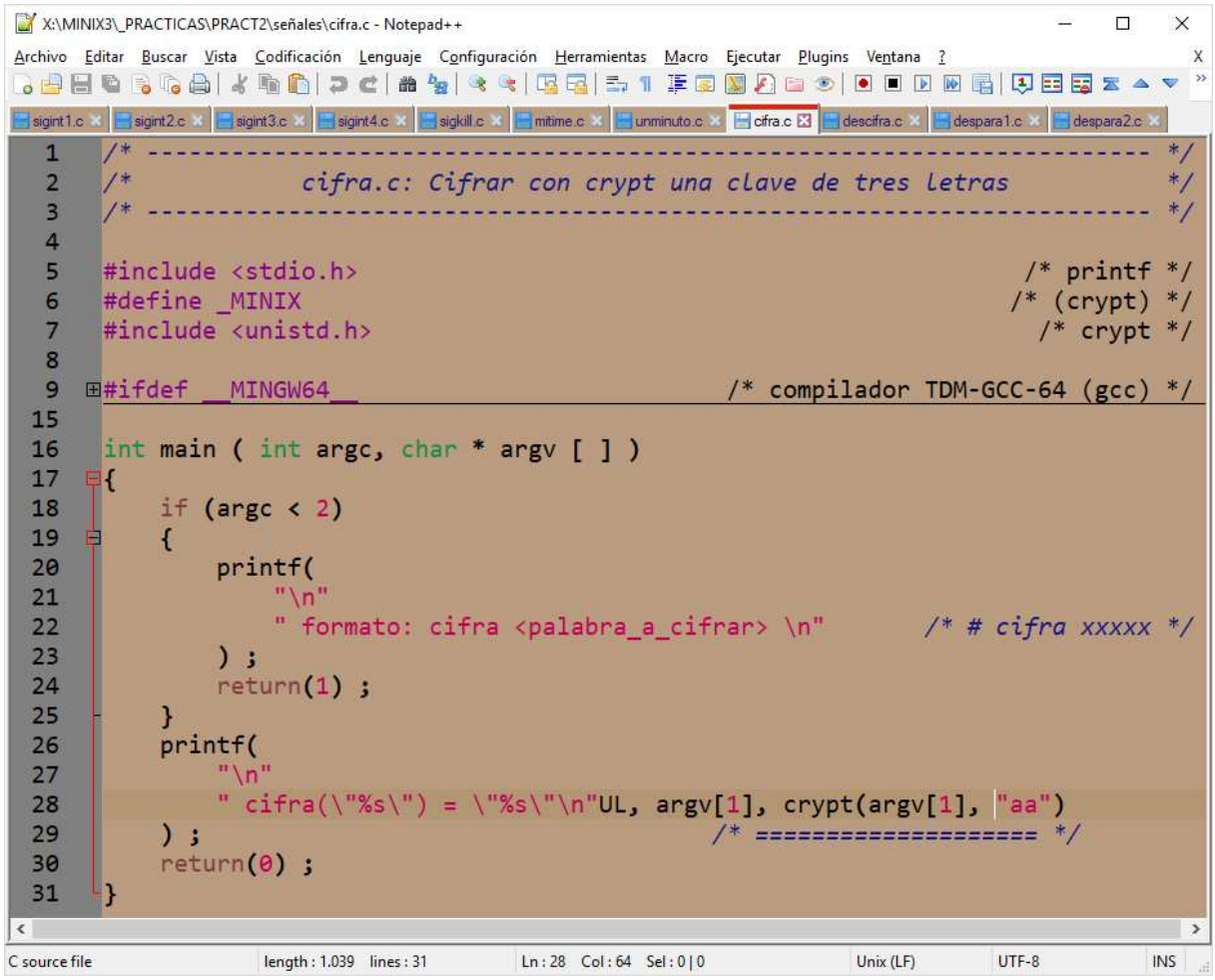



- Compilar y ejecutar el programa para observar su correcto funcionamiento.

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**].


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




- 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

X:\MINIX3_PRACTICAS\PRACT2\señales\descifra.c - Notepad++

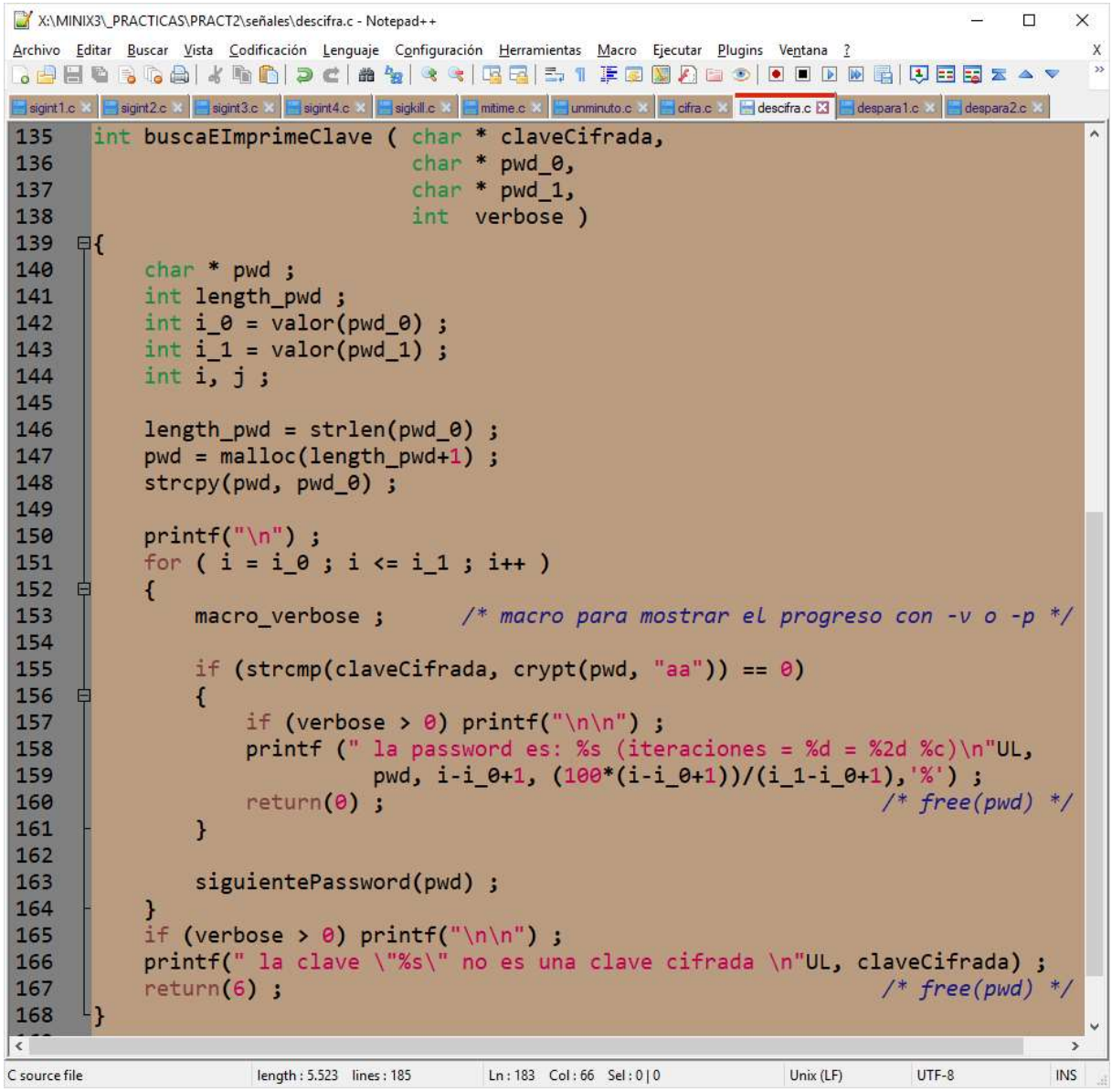
Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?

sigint1.c | sigint2.c | sigint3.c | sigint4.c | sigkill.c | mitime.c | unminuto.c | cifra.c | descifra.c | despara1.c | despara2.c

```
  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) */
 18
 19  void siguientePassword ( char * pwd )              /* orden lexicografico */
 20 {                                                          /* circular */
 29
 30  int valor ( const char * pwd )     /* valor de la clave (base 'z'-'a' + 1) */
 31 {
 40
 41  int obtenPwd ( int valor, char * pwd )
 42 {
 56
 57  int analizarArgumentos ( int argc, char * argv [ ], int * verbose )
 58 {
122
123  #define macro_verbose \
134
135  int buscaEImprimeClave ( char * claveCifrada,
136                           char * pwd_0,
137                           char * pwd_1,
138                           int  verbose )
139 {
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:




```c
int buscaEImprimeClave ( char * claveCifrada,
                         char * pwd_0,
                         char * pwd_1,
                         int  verbose )
{
    char * pwd ;
    int length_pwd ;
    int i_0 = valor(pwd_0) ;
    int i_1 = valor(pwd_1) ;
    int i, j ;

    length_pwd = strlen(pwd_0) ;
    pwd = malloc(length_pwd+1) ;
    strcpy(pwd, pwd_0) ;

    printf("\n") ;
    for ( i = i_0 ; i <= i_1 ; i++ )
    {
        macro_verbose ;      /* macro para mostrar el progreso con -v o -p */

        if (strcmp(claveCifrada, crypt(pwd, "aa")) == 0)
        {
            if (verbose > 0) printf("\n\n") ;
            printf (" la password es: %s (iteraciones = %d = %2d %c)\n"UL,
                     pwd, i-i_0+1, (100*(i-i_0+1))/(i_1-i_0+1),'%') ;
            return(0) ;                                          /* free(pwd) */
        }

        siguientePassword(pwd) ;
    }
    if (verbose > 0) printf("\n\n") ;
    printf(" la clave \"%s\" no es una clave cifrada \n"UL, claveCifrada) ;
    return(6) ;                                                  /* free(pwd) */
}
```

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 aaxxxxxxxxxx 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**.

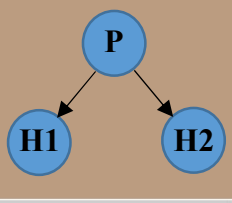
**despara1.c**: programa principal.

X:\MINIX3_PRACTICAS\PRACT2\señales\despara1.c - Notepad++

Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?

sigint1.c | sigint2.c | sigint3.c | sigint4.c | sigkill.c | mitime.c | unminuto.c | cifra.c | descifra.c | despara1.c | despara2.c

```
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  }
```

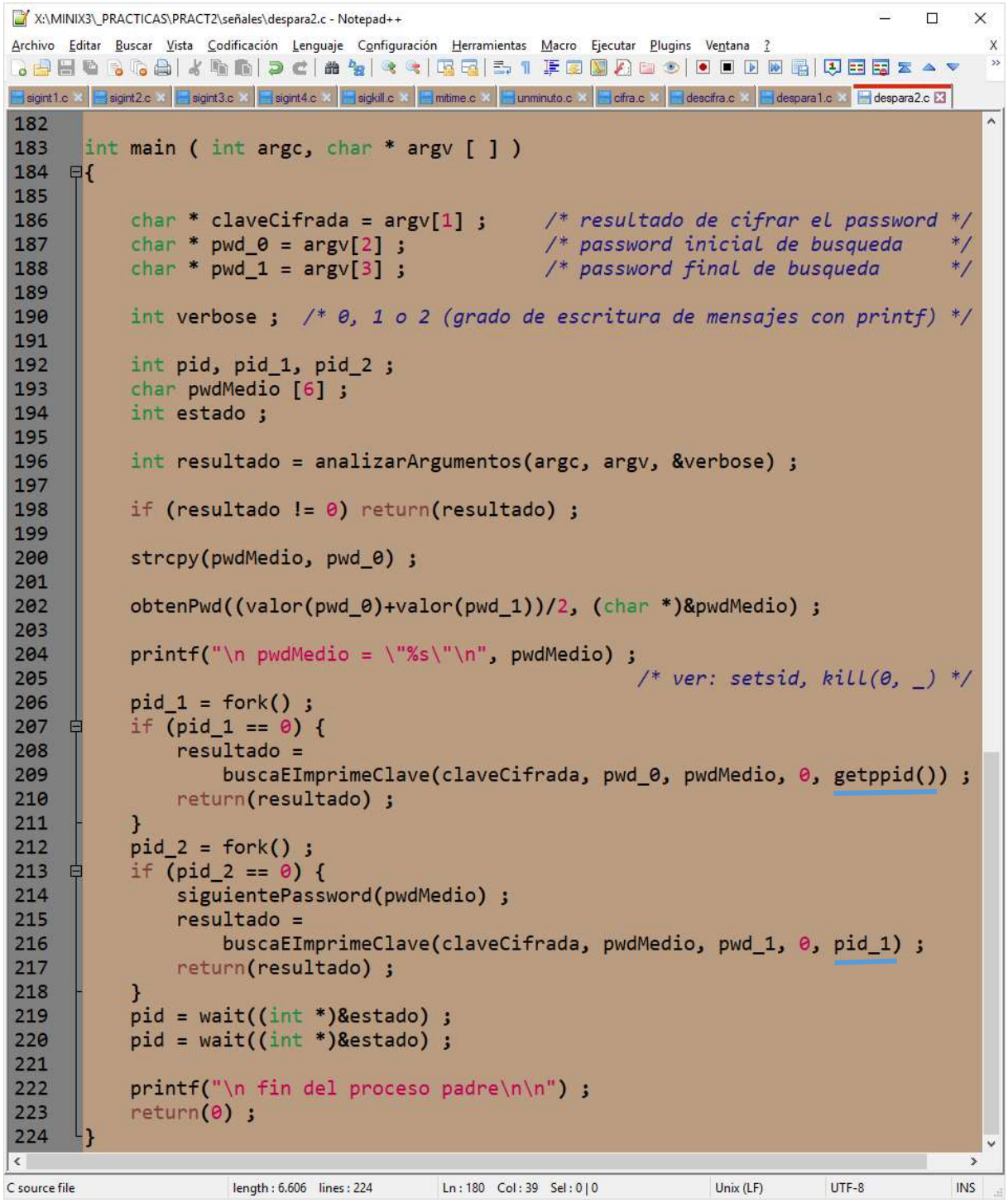



C source file | length : 6.369 lines : 224 | Ln : 1 Col : 1 Sel : 0 | 0 | Unix (LF) | UTF-8 | INS

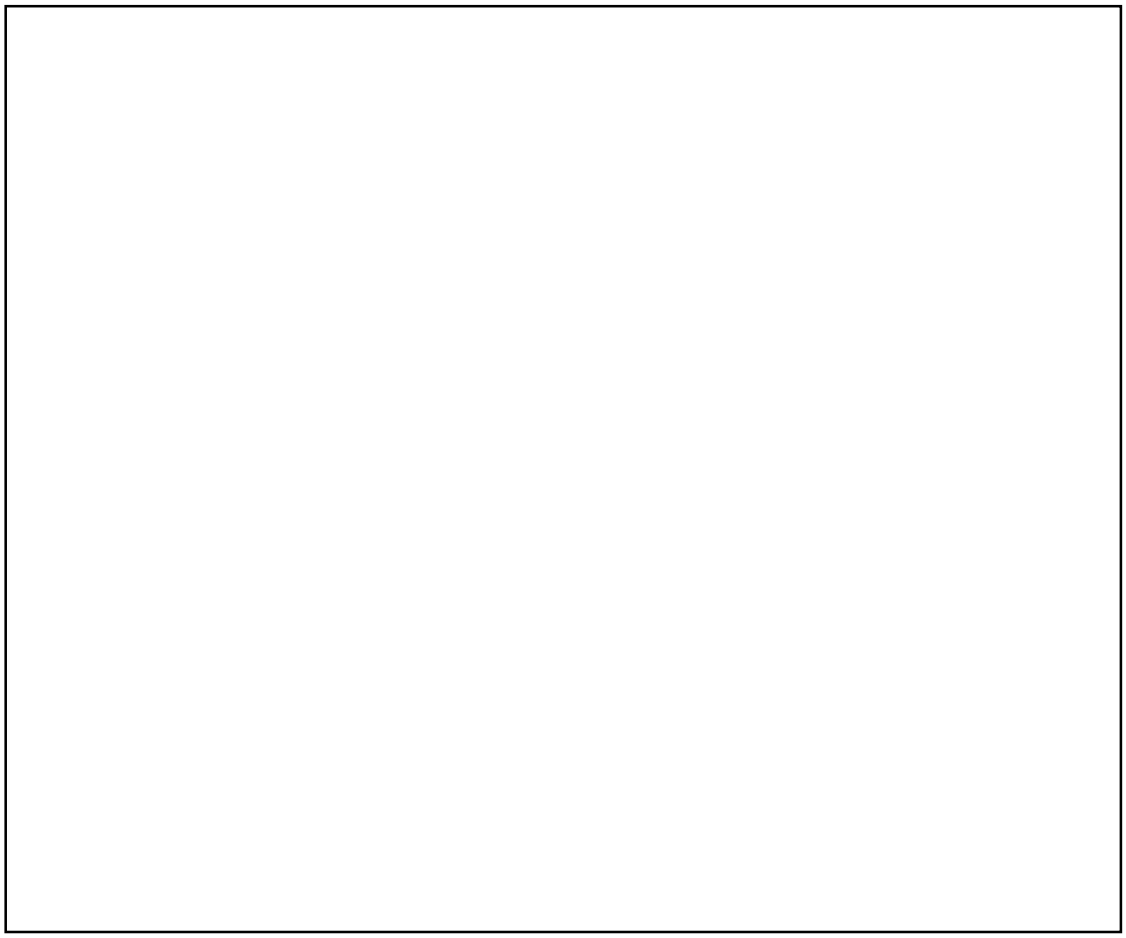
- Compilar **despara1.c** con el comando **cc despara1.c –o despara1** y ejecutar este programa para descifrar en paralelo las claves cifradas correspondientes a "**esesi**" y "**ojosi**". Comprobar que localiza las claves y anotar en la cuarta columna de la tabla 1 el tiempo (real) invertido en la búsqueda 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



- 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**).