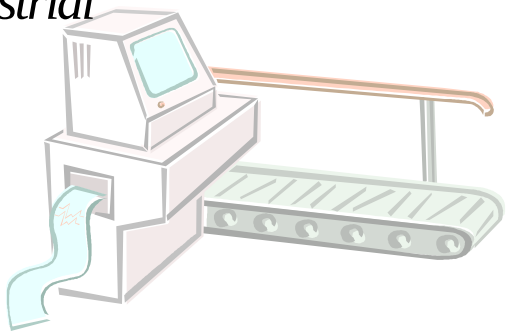
Página Web: http://webct08.uah.es
e-mail: eliseo.garcia@uah.es
Despacho: N-245
Departamento de Automática

Práctica 6

Desarrollo de programas y acceso a dispositivos en GNU/Linux

Laboratorio de Informática Industrial



Herramientas para el desarrollo de software

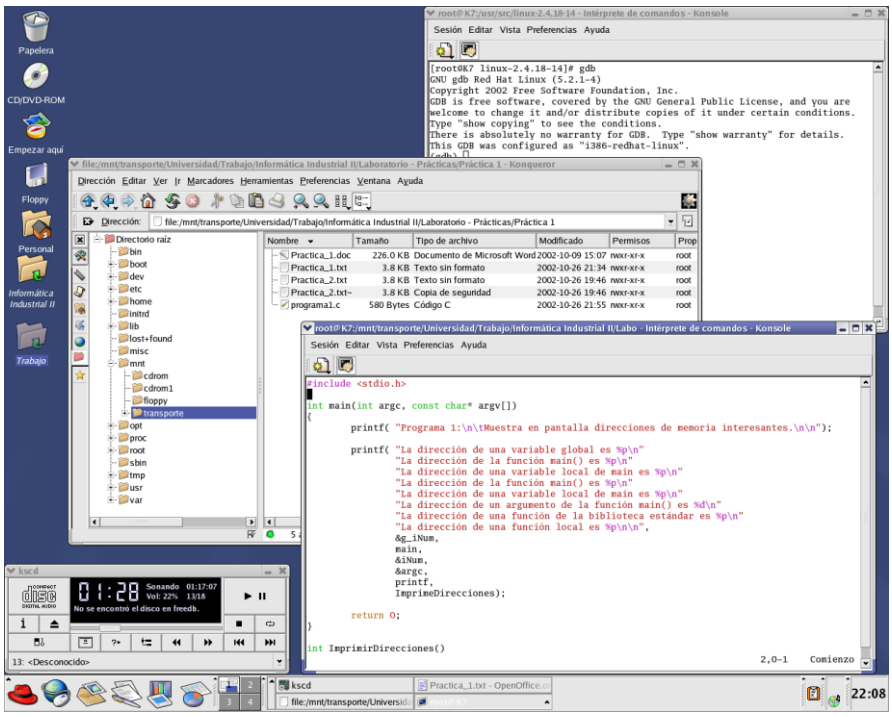
Introducción

Muchas aplicaciones industriales se realizan para ser ejecutadas sobre maquinas que poseen amplios recursos informáticos en cuanto a memoria y procesador, y que deben gobernar hardware complejo. Es el caso de maquinas gobernadas por los llamados PC industriales, como pueden ser, por ejemplo, las maquinas que gobiernan cadenas de montaje.

Tradicionalmente UNIX ha sido uno de los SS.OO. mejor dotados para el desarrollo de software. Cuenta con un conjunto de herramientas de desarrollo con bastante solera. No en vano el exitoso lenguaje C nació fuertemente ligado a UNIX. El hecho de conservar compatibilidad hacia atrás permite a los desarrolladores centrarse más en aspectos de diseño que en la forma de usar las herramientas.

Desde que a finales de los '80 las utilidades GNU de la *Free Software Foundation* alcanzaran un nivel de madurez suficiente, se han establecido como un estándar de facto para el desarrollo de software y el mantenimiento del sistema, y han trasgredido las fronteras del propio UNIX que las vio nacer. En la actualidad se dispone de ellas en casi cualquier plataforma informática de entidad suficiente. Además, exite una familia de procesadores ARM que admiten Sistemas Operativos basados en este entorno.

El compilador cruzado de C **`gcc`** es con total seguridad su aplicación principal. Está portado a más de 10 tipos diferentes de procesador, y adaptado para generar código para muchos SS.OO. La posibilidad de contar con un potente compilador de C compatible ANSI/ISO ha permitido migrar a su vez toda una legión de aplicaciones de soporte: diferentes tipos de intérpretes, utilidades de interfaz de usuario, editores de texto, analizadores, ensambladores, bibliotecas de programación, etc. Diferentes *front-ends* para **`gcc`** permiten desarrollar aplicaciones en Objective-C, C++, Pascal, Fortran y Java.



Gracias a la existencia gratuita y acompañada de código fuente de estos programas se ha propiciado la aparición de otros muchos proyectos de software de diferente envergadura entre los que podemos citar el núcleo Linux, el S.O. FreeBSD (creados ambos al estilo UNIX), el sistema de ventanas XFree86, los gestores GNOME y KDE, etc.

Como ya sabemos Linux se ha arropado con una gran cantidad de programas para formar el S.O. GNU/Linux, que puede llegarnos a través de cualquiera de sus distribuciones. En esta práctica veremos el desarrollo básico de programas haciendo uso de algunas aplicaciones GNU. Estas con el tiempo han desplazado a las que existían con anterioridad ya que incorporan mayor funcionalidad y respetan en gran medida la compatibilidad hacia atrás.

Edición de textos en UNIX

La edición de textos es uno de los pilares básicos sobre los que necesariamente se asienta la codificación de nuevos programas. En UNIX existen varias utilidades para la creación de texto plano, entre las que se podrían citar los editores **`EMACS`**, **`vi`**, **`joe`**, y el editor de flujos **`sed`**.

En general todos ellos tienen una concepción y un diseño muy antiguos para los gustos modernos. Posiblemente ninguno triunfaría si apareciera de nuevas en la actualidad. Sus principales bazas son la compatibilidad hacia atrás y su potencia como editores que desde el punto de vista de la funcionalidad superan en gran medida al resto de editores de texto ASCII modernos. Permiten el trabajo en terminales de texto y gracias a los diferentes portes, en ventana de X-Window.

El editor `vi`

El editor **`vi`** es uno de los más populares dentro del mundo UNIX. En la actualidad se suele usar una versión más moderna compatible llamada **`vim`**. Aunque las posibilidades del editor son muy variadas, sólo veremos las necesarias para realizar una edición básica.

Para el desarrollo de software sería recomendable que el editor permanezca cargado en un terminal independiente, para evitar tener que estar entrando y saliendo continuamente. Si nos encontrásemos en una consola de texto, sería buena idea tener otras abiertas para la compilación y para la depuración a las que se pudiera conmutar mediante la combinación de teclas `[Ctrl]+[Alt]+[F1~F4]`. Como alternativa podríamos tener varias ventanas abiertas en una sesión de X-Window.

Para entrar en el editor, simplemente tecleamos **`vi`** seguido del nombre del archivo que queramos editar. Si el archivo no existiera, en el momento de guardarlo se creará uno nuevo:

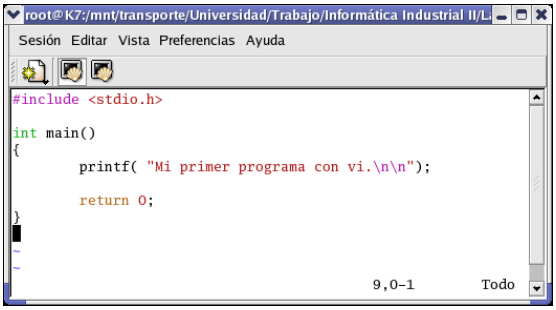
```
$ vi programa1.c

~
~
~
~                           VIM - VI Mejorado
~
~                            versión 6.1.165
~                       por Bram Moolenaar et al.
~           Vim es código abierto y se puede distribuir libremente
~
~                    ¡Ayude a los niños pobres de Uganda!
~           escriba  :help iccf<Intro>     para más información
~
~           escriba  :q<Intro>             para salir
~           escriba  :help<Intro> o <F1>   para obtener ayuda
~           escriba  :help version6<Intro> para info de la versión
~
~
~
~
~                                                          0,0-1         Todo
```

Vemos una pantalla de presentación, en la que se nos muestra un pequeño mensaje de ayuda. Si necesitáramos más ayuda, podemos conseguirla pulsando la tecla de función [F1]. En **`vi`** carecemos de menús de usuario por lo que deberemos introducir todos los comandos mediante el teclado. El primer concepto que debemos aprender es que **`vi`** funciona en dos estados: *modo inserción*, y *modo órdenes*.

El modo órdenes es el que usaremos para introducir comandos dirigidos al programa, como copiar cortar y pegar texto, salvar el documento, buscar y reemplazar, desplazamiento, etc. Existe un tipo de

comandos cortos para especificar las opciones más habituales en los trabajos de edición de texto. Estos no generan eco según son escritos, por lo que debemos prestar especial atención al teclearlos. Tampoco necesitan que se presione [Intro] al finalizar su escritura. En caso de que se insertaran incorrectamente, el editor hará sonar un pitido para comunicárnoslo. Hay otro tipo de comandos más largos que sí presentan eco por pantalla y necesitan que se presione [Intro] al final. Para introducirlos es necesario comenzar por el carácter [:].



El modo inserción es el que se utilizará para escribir texto. Desde el modo de órdenes se accede a inserción introduciendo la letra [i]. Se puede usar las teclas del cursor para desplazarse por el texto, y las teclas de borrado para eliminar caracteres si el teclado las incorpora. Para volver al modo de órdenes, hay que presionar la tecla de escape [Esc].

Paso a paso

El editor **`vi`** arranca en modo de órdenes, por lo que pasaremos al modo de inserción pulsando la tecla [i]. Vemos que la columna izquierda de caracteres del terminal está rellena con tildes [~]. Esto lo hace **`vi`** para indicarnos que esas líneas están sin rellenar, y evitar así la ambigüedad que presentaría el ver una línea vacía de la que no sabemos si contiene caracteres no imprimibles.

Si insertamos un programa en C como el que aparece en la figura, veremos que la sintaxis se va coloreando automáticamente según se escribe, siempre que tengamos a **`vi`** configurado para obtener tal funcionamiento. Esto es así porque `vi` reconoce por la extensión del archivo que se trata de un programa en lenguaje C. Inserte por ejemplo el siguiente programa:

```
#include <stdio.h>

int main()
{
     printf(  "Mi primer programa con vi.\n\n");

     return 0;
}
```

Podemos salvar el contenido en cualquier momento volviendo a modo comando mediante [Esc], e introduciendo `:w`. De esta forma se salvará el documento con el nombre que se hubiera pasado como argumento al arrancar el editor. Si quisiéramos salvarlo con cualquier otro nombre, se lo podríamos indicar inmediatamente a continuación de esta misma orden. Por ejemplo `:w programa_corto.c` salvaría el texto con el nombre `programa_corto.c`. Introduciendo [i]volveremos de nuevo a modo inserción.

Una vez terminado, podemos salvar y salir introduciendo el comando `ZZ` (dos zetas mayúsculas seguidas). Esto mismo también se puede hacer introduciendo `:wq` o `:x`. Si sólo quisiéramos salir sin guardar las modificaciones, insertaríamos `:q`.

Para copiar y cortar la línea donde se encuentra actualmente el cursor introduciremos `yy` y `dd` respectivamente. Si quisiéramos que estas órdenes afectaran a varias líneas a partir de la posición actual del cursor, se antepondría el número delante quedando `<núm>yy` para copiar `núm` líneas, y `<núm>dd` para cortarlas. Si no se especifica otra cosa todas las operaciones de copiado de texto se realizan sobre un buffer por defecto. Para pegar la información que contiene se usa la orden `p` que pega detrás del cursor, o `P` que lo pega desde delante.

El editor **vi** dispone de un buffer adicional por cada letra, lo que permite tener varios textos independientes copiados al mismo tiempo. Para indicar explícitamente el buffer donde queremos que se introduzca la información, antepondríamos "<letra> (comillas seguido de una letra del abecedario ASCII) a las operaciones de copiado o cortado. Por ejemplo, si colocamos el cursor en la línea del `main()` e introducimos `6yy`, se copiarán las seis líneas que hay a continuación. Ahora podemos colocar el cursor en cualquier otra parte del documento e insertar `P` para pegar el buffer. Si quisiéramos cortar el cuerpo de la función a un buffer específico, pondríamos el cursor en la línea del `printf()` e introduciríamos la orden "a3dd que copia tres líneas al buffer *a*, y las elimina del documento. Ahora colocando el cursor en otra posición, podemos recuperar el buffer *a* insertando "ap ó "aP.

A menudo interesa copiar o cortar segmentos de texto cuyo principio y fin se encuentren en posiciones arbitrarias, y no necesariamente al principio o al final de una línea. Para ello podemos actuar sobre selecciones marcadas. Para iniciar una marcación en la posición actual usaremos la orden m<letra_m> donde <letra_m> es cualquier letra del abecedario ASCII. Luego llevaremos el cursor a la posición final donde introduciremos y`<letra_m>, lo que marcará el final de la selección, copiándola al buffer por defecto. El carácter ` es el acento grave que se obtiene presionando [`] mas un espacio. Ahora podemos pegar el texto en cualquier parte usando las órdenes `p` y `P`. Si quisiéramos que la selección se introdujera en un buffer concreto, antepondríamos la secuencia "<letra> a la operación de copia y de pegado.

Para buscar una palabra, insertaríamos /<palabra> si la búsqueda es hacia alante, y ?<palabra> si es hacia atrás. Por ejemplo estando el cursor al final del texto, el introducir `?primer`, lo colocará al principio de la ocurrencia de la palabra *primer* encontrada en la cadena del argumento de `printf()`.

También disponemos de la posibilidad de deshacer la última acción introduciendo `u`.

Nota: Aunque en estas prácticas se asuma el trabajo con **vi**, cada alumno es libre de usar el editor que prefiera. Se recomienda en ese caso usar uno que permita el coloreado de la sintaxis como las versiones modernas de **vim**. El editor **KWrite** de KDE, o el entorno de desarrollo integrado **KDevelop** podrían ser buenas alternativas.

Compilación de programas

Como ya sabemos, la compilación consiste de forma genérica en la traducción de un programa expresado en un lenguaje de alto nivel, a código objeto que contiene código máquina. En la fase final de generación, se ha de insertar la información necesaria para que pueda ser cargado en memoria y ejecutado cuando sea invocado. Este proceso lo realiza el cargador del S.O. (*loader*), encargado de adaptar el código que había generado el compilador a código directamente ejecutable por la CPU, reservando para ello los recursos necesarios.

El pre-procesador cpp (C Pre-Processor)

El preprocesador de C es el encargado de resolver las *directivas del preprocesador de C*. Estas son las líneas de texto que comienzan con el carácter # y van seguidas por una palabra clave, que se utilizan para realizar operaciones sencillas de selección, sustitución, e inclusión de texto sobre los archivos fuente de C. El preprocesador no conoce exactamente la sintaxis del lenguaje C, lo único que hace son operaciones sobre el texto, sin afectar eso sí a los comentarios de C ni a las cadenas de texto que estén delimitadas por comillas. La misión final es la de pasar al compilador un archivo de texto con un programa en C puro con el que se obtendrá una vez compilado, el archivo de código objeto.

Operaciones del preprocesador:

Inclusión de archivos.- La inclusión consiste en sustituir una línea del estilo:

```
#include "ruta"
```

por el contenido del archivo cuya ruta se especifica en `ruta`. Si no se especifica una ruta absoluta, el preprocesador intentará formarla posponiendo `ruta` a los directorios que estén en el directorio de trabajo, los directorios *include* estándares del compilador, y los directorios del PATH. Se preprocesarán recursivamente todos los archivos contenidos. Una variación está en indicar la ruta entre ángulos:

```
#include <ruta>
```

La diferencia estriba en que no se buscará el archivo en el directorio de trabajo.

El uso más habitual es la inclusión de archivos de cabecera del lenguaje C, aunque no se ponen restricciones para incluir cualquier otro tipo de archivo.

Definición de macros.- En lenguaje C llamamos macro a una secuencia de texto que es sustituida por otra, realizando tal vez alguna transformación básica sobre las mismas. La forma de crear una nueva macro es mediante el uso de `#define` seguido del nombre de la macro y opcionalmente el texto alternativo. Por ejemplo, la siguiente macro crea un equivalente de 32 al que se le podrá llamar `MAX_NUM` en cualquier parte del texto que aparezca hasta el final del archivo:

```
#define MAX_NUM 32

int g_num = MAX_NUM * 10;
```

La operación que se realiza es buscar todas las ocurrencias de `MAX_NUM` y substituirlas por la cadena que aparece a la derecha de la definición. En este caso `g_num` será iniciada con el valor de 32 * 10 = 320, pero si la macro hubiera sido:

```
#define MAX_NUM 22 + 10
```

`g_num` se iniciaría con 22 + 10 * 10 = 122. A menudo para evitar comportamientos inesperados cuando la macro está formada por una cadena con varios términos, se agrupan entre paréntesis:

```
#define MAX_NUM (22 + 10)
```

Una de las mayores utilidades de las macros es su comportamiento como pseudo-funciones. Para ello permiten indicar una lista de argumentos entre paréntesis y separados por comas. A continuación se muestra una macro de un único argumento:

```
#define CUBO(n) (n*n*n)

int g_num = CUBO(3);
```

En este caso el preprocesador entiende que 3 es el argumento de la macro, por lo que asigna a la variable el valor (3 * 3 * 3) = 27. La sustitución de caracteres que realiza cada macro se llama expansión de macro. Recordemos que la macro sólo hace sustituciones de texto, sin saber el significado que tendrá en C su expansión, por lo que la siguiente expresión:

```
int g_num = CUBO(2 + 1);
```

equivale a asignar a `g_num` el valor (2 + 1 * 2 + 1 * 2 + 1) = 7. Para asegurar que se multiplique el resultado de la evaluación de los argumentos, deberemos rescribir la macro encerrándolos entre paréntesis como aparece a continuación:

```
#define CUBO(n) ((n)*(n)*(n))
```

con lo que la asignación quedará con el valor de ((2 + 1) * (2 + 1) * (2 + 1)) = 27.

Para usar varios argumentos, hay que separarlos por comas como aparecen en la macro `MIN`:

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

La definición de una macro puede afectar al cuerpo de las siguientes macros. A continuación se muestra un programa que utiliza a su vez otras macros:

```
#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE
#define EPSILON (1e-3)

#define ABS(a) ((a) < 0 ? -(a) : (a))
#define IGUAL(a, b) (ABS((a) - (b)) > EPSILON ? FALSE : TRUE)

int main()
{
    float fNum1, fNum2;

    printf("\nIntroduzca un número Real: ");
    scanf("%g", &fNum1);

    printf("\nIntroduzca otro número Real: ");
    scanf("%g", &fNum2);

    if(IGUAL(fNum1, fNum2))
        printf("\nLos números %g y %g son prácticamente iguales.\n\n", fNum1, fNum2);
    else
        printf("\nLos números %g y %g son diferentes.\n\n", fNum1, fNum2);

    return 0;
}
```

El archivo preprocesado excluyendo la parte de `stdio.h`, quedará:

```
int main()
{
    float fNum1, fNum2;

    printf("\nIntroduzca un número Real: ");
    scanf("%g", &fNum1);

    printf("\nIntroduzca otro número Real: ");
    scanf("%g", &fNum2);

    if(((((fNum1) - (fNum2)) < 0 ? -((fNum1) - (fNum2)) : ((fNum1) - (fNum2))) > (1e-3) ? 0 : !0))
        printf("\nLos números %g y %g son prácticamente iguales.\n\n", fNum1, fNum2);
    else
        printf("\nLos números %g y %g son diferentes.\n\n", fNum1, fNum2);
```

```
    return 0;
}
```

Ahora será tarea del copilador optimizar la expresión resultante de la expansión de las macros.

Compilación condicional.- Existen ocasiones en las que interesa reutilizar el mismo archivo fuente para generar código ligeramente diferente, dependiendo de ciertas condiciones previas. Podremos habilitar la compilación de texto bajo la condición de la existencia o ausencia de una macro. Para ello contamos también con la posibilidad de definir macros sin cuerpo.

La directiva `#ifdef MACRO` sirve para incluir el texto que aparece a continuación hasta encontrar un `#endif` en el caso de que `MACRO` esté definida. Para incluir el texto en el caso de que no esté definida, usaremos `#ifndef` en su lugar.

El siguiente fragmento de código muestra el uso de estas directivas.

```
#include <stdio.h>
#define NDEBUG
#define INFO

int Divide(int a, int b)
{
#ifndef NDEBUG
    if(b == 0)
    {
        perror("El divisor es 0!!!\n");
        return 0;
    }
#endif

#ifdef INFO
    printf("El resultado de la división es %d\n", a / b);
#endif

    return a / b;
}
```

El archivo que se le pasará al compilador excluyendo la cabecera queda:

```
int Divide(int a, int b)
{
    printf("El resultado de la división es %d\n", a / b);

    return a / b;
}
```

Existen tres macros reservadas por el compilador que pueden ayudar a generar mensajes de depuración más informativos: `__FILE__`, `__func__`, `__LINE__`.

`__FILE__` es una cadena con el nombre del archivo que se está compilando.

`__func__` es una cadena con el nombre de la función que se está compilando.

`__LINE__` es un entero que contiene el número de línea que se está compilando.

A continuación se muestra un ejemplo de uso:

```
#define INFO
#include <stdio.h>
```

```
int Divide(int a, int b)
{
#ifndef NDEBUG
    if(b == 0)
    {
        printf("### ERROR: El divisor es 0!!! ###\n\tArchivo: %s; Función: %s, Línea: %d\n",
            __FILE__, __func__, __LINE__);
        return 0;
    }
#endif

#ifdef INFO
    printf("El resultado de la división es %d\n", a / b);
#endif

    return a / b;
}
```

La compilación condicional facilita la portabilidad de aplicaciones C a diferentes plataformas, ya que permite adaptar el código fuente a las necesidades específicas de la máquina. También es útil para incluir sólo una vez los archivos de cabecera, en la confección del programa que se le pasa al compilador.

Como una forma de asegurar que un programa se está ejecutando correctamente, podemos usar la macro `assert()`, que detendrá la ejecución del programa si su argumento es falso.

La idea de `assert()` es funcionar a modo de cortafuegos, deteniendo la ejecución allí donde falle un punto de comprobación que el programador haya puesto. En el caso de que se haya definido la macro `NDEBUG`, `assert()` se expande a una expresión `void`.

En el siguiente programa se usa la macro `assert()` en el mismo contexto en el que antes usábamos la compilación condicional.

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

int Divide(int a, int b)
{
    assert(b != 0);

    return a / b;
}

int main(int argc, char* argv[])
{
    int num1, num2;

    if(argc < 3)
    {
        printf("Uso:\n\t%s <numero1> <numero2>\n\n", argv[0]);

            return 0;
    }

    num1 = atoi(argv[1]);
    num2 = atoi(argv[2]);

    printf("%d / %d = %d\n", num1, num2, Divide(num1, num2));

    return 0;
```

```
}
```

En el caso de pasarse un denominador nulo a la función `Divide()`, la salida del programa sería similar a:

```
$ ./programa4
### ERROR: El divisor es 0!!! ###          Archivo: programa4.c; Función:
Divide, Línea: 9
```

El programa tradicionalmente usado para preprocesar archivos se llama **cpp**. Para preprocesar el archivo `programa.c`, y generar el archivo `programa.pp`, usaríamos la siguiente orden:

```
$ cpp -o programa.pp programa.c
```

El compilador **gcc** no depende de preprocesadores externos, y permite una funcionalidad idéntica a la orden anterior usando el modificador `-E`, el cual ordena que sólo se realice el preprocesado del archivo de entrada:

```
$ gcc -E -o programa.pp programa.c
```

En ambos casos se utiliza el modificador `-o` seguido del nombre del archivo que se desea crear.

El compilador de C gcc (GNU C Compiler)

El programa **gcc** es la utilidad que emplearemos para generar ejecutables a partir de código C. Desde este podremos preprocesar directivas, traducir el C a ensamblador, ensamblar, y enlazar, todo en uno y de forma transparente. La sintaxis es:

```
gcc [opciones] fichero...
```

Por ejemplo, edite el siguiente programa, y llámelo `programa0.c`:

```
#include <stdio.h>

int main()
{
        puts("Este es un programa de C.\n");

        return 0;
}
```

Para compilarlo tan solo será necesario introducir lo siguiente:

```
$ gcc programa0.c
```

lo que creará un archivo ejecutable llamado `a.out` que podremos cargar de la forma habitual:

```
$ ./a.out
Este es un programa de C.
```

El formato de salida del compilador es compatible con el S.O. para el que se esté compilando. En el caso de GNU/Linux el formato será ELF, que es comprendido por el programa cargador del sistema.

Si queremos especificar un nombre distinto al que se toma por defecto, lo podemos especificar con el modificador `-o` de la siguiente forma:

```
$ gcc -o mensaje programa0.c
```

Lo que indica que el nombre del archivo de salida sea `mensaje` en lugar de `a.out`.

Se puede pedir a **gcc** que intente optimizar en velocidad el código que va a generar. Para ello usaremos los modificadores `-O1` para una optimización aceptable, `-O2` para una optimización más agresiva, `-O3` permite además insertar funciones en línea según convenga. El tiempo que se emplea de más en la etapa de compilación es despreciable, comparando con la mejora de velocidad que se obtiene. En el desarrollo de software empotrado a menudo es preferible optimizar para obtener un tamaño de código pequeño, mediante el modificador `-Os`. Este modificador incluye todas las optimizaciones de `-O2` que no incurran en generación de código adicional. Sucede con frecuencia que el código generado con este modificador es incluso más rápido que el que se hubiera generado con `-O2`, ya que la posterior gestión de cachés en tiempo de ejecución es más eficiente. Las optimizaciones con `-O2` pueden hacer que el código generado no sea fácilmente depurable, ya que los depuradores a nivel de código fuente pueden confundirse con la forma que adopta el código producido. Por ejemplo `-O2` conlleva implícitamente la omisión del puntero de marco de pila, lo que tiene como ventaja la disponibilidad de un registro de propósito general adicional en la CPU, pero impide que el depurador encuentre con facilidad la localización exacta de las variables locales.

Compile el siguiente código llamándole `programa1.c` y observe los resultados obtenidos para las distintas optimizaciones:

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

//------------------------ EstadisticaDeNotas -----------------------------
// Calcula la función de distribución de las calificaciones de los alumnos.
// Entradas:
//    notas - Array con todas las notas que se van a considerar.
//    numNotas - Número de alumnos.
//    fdd - dirección de un array de 11 elementos (de 0 a 10) que será
//          rellenado con la función de distribución.
// Comentario: Por claridad no se incorpora código de detección de errores.
void EstadisticaDeNotas(const int notas[], int numNotas, int fdd[])
{
    const int *itNotas = notas; // Iterador de notas

    memset(fdd, 0, 11 * sizeof(fdd[0])); // Limpia el array de la FDD

    while(numNotas--)
    {
        int nota = *itNotas++;

        if(nota >= 0 && nota <= 10)
            fdd[nota]++;
    }
}

int main()
{
    static int notas[500];
    int fdd[11];
     int i;
```

```
    // Se rellena el array con las notas aleatorias de 500 alumnos
    for(i = 0; i < 500; i++)
        notas[i] = rand() % 11;

    EstadisticaDeNotas(notas, 500, fdd);

    // Mostrar la estadística:
    for(i = 0; i <= 10; i++)
        printf("%d  \t= %d\n", i, fdd[i]);

    return 0;
}
```

Los programas generados contienen el código de las funciones de biblioteca `rand()` y `printf()`, sobre las que **`gcc`** no realiza optimización alguna. Si quisiéramos que **`gcc`** sólo generase el código objeto correspondiente a este único archivo sin enlazarlo con ningún otro, usaríamos el modificador `-c`:

```
$ gcc -c -o programa1_sin.o programa1.c
$ gcc -O1 -c -o programa1_O1.o programa1.c
$ gcc -O2 -c -o programa1_O2.o programa1.c
$ gcc -Os -c -o programa1_Os.o programa1.c
$ ls -l *1_*.o
```

Al tratarse de un programa tan pequeño, el compilador tiene muy pocas opciones para optimizar. No obstante vemos que `-Os` generará un código compacto.

Hasta ahora todas las funciones utilizadas pertenecen a la biblioteca estándar de C `libc.a`. El compilador sigilosamente le ha estado comunicando al enlazador que la utilizara. Sin embargo en ocasiones se hace necesario enlazar nuestro código con otras bibliotecas que contienen funciones adicionales. En UNIX los archivos de biblioteca suelen tener un nombre que comienza por `lib` y acaba en `.a`, formando entonces cadenas del tipo `lib<nombre>.a` con el objetivo de que sean fácilmente identificables. Si quisiéramos indicar que se enlace una biblioteca adicional, usaríamos el modificador `-l` seguido del nombre de la misma. Para que no se enlacen automáticamente las bibliotecas estándar, usaremos los modificadores `-nostdlib` o `-nodefaultlibs`. En tales casos deberemos especificar explícitamente que se enlace al menos `libgcc.a` mediante `-lgcc`, lo que incorpora las funciones internas que utiliza el compilador a la hora de generar código.

En el siguiente programa se utiliza una función de cálculo de errores `erf()` que toma como argumento un `double` y devuelve otro con el error calculado. La operación que hace internamente es irrelevante para este ejemplo. El cuerpo de esta función se encuentra en la biblioteca `libm.a`, que contiene todas las funciones matemáticas básicas:

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

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

    if(argc < 2)
    {
        printf("Uso:\n\t%s <numero>\n\n", argv[0]);

            return 0;
    }

    num = atof(argv[1]);
```

```
    printf("La función de error de %lg es %lg\n", num, erf(num));
}
```

Para enlazarlo con la biblioteca usaremos el siguiente comando:

```
$ gcc -o programa3 -lm programa3.c
```

En ocasiones resulta útil pasar definiciones a los programas desde la línea de órdenes con la que invoca al compilador. Con ello no existe la necesidad de definir las macros en los archivos a compilar. Para incluir la definición, debe ser pospuesta al modificador `-D` de **`gcc`**.

En el programa del `assert()` anterior podríamos haber definido la macro `NDEBUG` directamente como parámetro de **`gcc`**, `y quitar la línea de que lo define al principio del archivo`:

```
$ gcc -o programa4 -DNDEBUG programa4.c
```

Si se produjera un error, ahora actuaría el mecanismo por defecto del sistema, es decir, se detendría el programa en ejecución y se mostraría el mensaje de error correspondiente:

```
$ ./programa4 300 0
Excepción de coma flotante
```

En lugar del que aparecía si no se hubiera definido:

```
$ gcc -o programa4 programa8.c
$ ./programa4 300 0
programa8: programa8.c:6: Divide: Assertion `b != 0' failed.
Abortado
```

Podemos pedir al compilador que nos muestre advertencias por varios errores potenciales que pudiera detectar. En este sentido **`gcc`** llega hasta muy lejos. Lo más normal es que habilitemos todas las posibles advertencias mediante el modificador `-Wall`.

Si quisiéramos habilitar la compatibilidad estricta con ANSI C del año '89, usaríamos el modificador `-ansi`. También podremos usar `-std=c99` para compatibilidad con ISO C99, aunque en la actualidad no está plenamente soportado. Para evitar que se permitan extensiones que no están en el estándar, usaremos `-pedantic`.

Depuración de programas con `gdb` (GNU De-Bugger)

Los depuradores son utilidades que ayudan a eliminar fallos de los programas en tiempo de ejecución, una vez que se han eliminado los fallos de sintaxis en tiempo de compilación. Por lo general se suele asumir que los fallos de los programas se encuentran únicamente en el código, ya sea por un diseño deficiente o una codificación incorrecta. La verdad es que eso no siempre es así. A menudo los errores están presentes en los documentos o en los datos externos que maneja el programa, normalmente porque no se adapten a un convenio estipulado. Esto no exime de la responsabilidad de desarrollar programas robustos que puedan reponerse ante datos de entrada anormales.

La aplicación `gdb` se ha erigido por méritos propios como uno de los mejores depuradores a nivel de código fuente existentes. Es apto tanto para depurar código como datos. Aunque se trate de un programa que funciona en consola de texto, está preparado para comunicarse con otros programas que le

pueden proporcionar una interfaz más amigable. Uno de estos programas es el depurador ddd, que se utilizara en el laboratorio. En el Anexo 1 se ofrece un ejemplo de depuración de un programa mediante gdb.

Llamadas al sistema

Introducción

Una de las misiones del sistema operativo es abstraer a las aplicaciones del hardware al que nunca deberían acceder directamente, a excepción de las áreas de memoria *mapeada* en el espacio lógico de los procesos. El objetivo es poder administrar el hardware entre las aplicaciones.

Los sistemas estilo UNIX emplean un mecanismo de comunicación con el hardware que considera a los dispositivos como ficheros pertenecientes a un sistema de archivos. De esta forma se puede disponer de una interfaz normalizada que permite interactuar con ellos haciendo uso de un reducido número de funciones estándar. Además esto permite beneficiarse de algunas características interesantes de los archivos como el control de seguridad contra accesos no autorizados. Casi todos los SS.OO. modernos han adoptado algún método similar para acceder al hardware.

Dispositivos

Una de las características más destacables en UNIX es que los dispositivos se encuentran visibles a los usuarios a través del directorio `/dev`. No todos los dispositivos que dispone el S.O. representan objetos físicos. Hay algunos que están creados puramente por software, pero tienen una finalidad muy similar que la que tendría un dispositivo físico. En otras ocasiones existen dispositivos que aíslan del acceso a otros dispositivos de más bajo nivel, comportándose como una capa de abstracción intermedia.

La comunicación con los dispositivos se realiza a través de controladores de dispositivo (*device drivers*). Estos son pequeños programas que se encargan de mostrarnos los dispositivos a través de una interfaz estándar en forma de llamadas al sistema.

Para listar los dispositivos soportados de forma nativa por el sistema, introducimos la orden:

```
$ ls -l /dev
```

No todas las entradas se corresponden con dispositivos reales, sino que son más bien puntos de entrada para dispositivos potenciales. Como se puede ver en la lista, a cada dispositivo le corresponden dos números: uno mayor y otro menor, que utiliza el S.O. para reconocerlos internamente.

Existen dos tipos de dispositivos:

- **Dispositivos de tipo carácter**.- Todos aquellos cuyas lecturas o escrituras se pueden realizar mediante un flujo de bytes de datos serie. Ejemplos: puertos serie y paralelo, terminales, y tarjetas de sonido.
- **Dispositivos de modo bloque**.- Representan a dispositivos que realizan lecturas y escrituras en bloques indivisibles de tamaño fijo. Permiten el acceso aleatorio a cualquier dato. Una lectura exige que se recupere todo el bloque de información que la contiene. Una escritura parcial a cualquier parte de un bloque implica que se debe escribir todo el bloque, por lo que previamente debería haber sido leído. El ejemplo más característico es un controlador de disco.

Los dispositivos pueden ser tratados como archivos ordinarios en muchos sentidos, al compartir las mismas llamadas al sistema básicas.

Una orden muy útil en el manejo de archivos es `dd` destinada a copiar partes de ficheros, y realizando en ocasiones ciertas transformaciones sobre ellos. La diferencia fundamental entre `cp` y `dd` es que esta permite especificar el intervalo de origen y de destino de datos a ser copiados. Además permite especificar el número de bytes correspondientes a un bloque, útil para dispositivos de tipo bloque.

Los argumentos más habituales son:

- **if, of** - Especifican el archivo de entrada y de salida para la copia. Si no se incluyen se toman los flujos estándar de entrada y de salida respectivamente.
- **bs** - Tamaño del bloque en bytes. Si no se especifica se toma por defecto 512, habitualmente usado en disqueteras y algunos sistemas de archivo de disco duro.
- **count** - El número de bloques a copiar. Si no se especifica, se copia hasta el final del archivo de entrada.
- **skip, seek** - Se saltan el número de bloques especificados del archivo de entrada y el de salida respectivamente antes de empezar la copia.
- **conv** - Expecifica, entre otros, si queremos que se trunque (finalice) el fichero de salida tras los bloques copiados. Por defecto, hace el truncado. Para evitar este comportamiento, se debe poner el valor notrunc (conv=notrunc).

Como ejemplo, se va a crear un archivo de texto mediante la orden `cat`, la cual tomará datos de la entrada estándar y serán redirigidos a un archivo de texto.

```
$ cat > Salida.txt
Esto es un archivo de prueba.
No tiene más objeto que ser usado posteriormente con dd.
Para terminar la entrada estándar es suficiente con presionar simultáneamente
[Ctrl]+[D].
```

Ahora mediante `dd` se copia todo el archivo excepto los 5 primeros bytes:

```
$ dd if=Salida.txt of=SalidaRecortada.txt skip=5 bs=1
```

El contenido del nuevo archivo es:

```
$ cat SalidaRecortada.txt
es un archivo de prueba.
No tiene más objeto que ser usado posteriormente con dd.
Para terminar la entrada estándar es suficiente con presionar simultáneamente
[Ctrl]+[D].

$ ls -l
-rwxr-xr-x    1 david    root          175 nov 21 21:31 SalidaRecortada.txt
-rwxr-xr-x    1 david    root          180 nov 21 21:30 Salida.txt
```

A continuación se comentan algunos de los controladores de dispositivo más ilustrativos:

/dev/mem

Administración de accesos a memoria del ordenador.

Las direcciones de bytes se interpretan como direcciones físicas de memoria.

/dev/kmem

Idéntico a `/dev/mem` pero administrando la memoria virtual del núcleo en lugar de la física.

/dev/port

Similar a `/dev/mem` pero para el acceso a puertos de E/S en x86.

/dev/ram0 - ... - /dev/ram19

Permite el acceso a discos RAM. Se trata de particiones que se alojan en memoria, en lugar de discos duros. Su contenido desaparece cuando cesa la alimentación.

La ganancia en velocidad es inapreciable ya que Linux cachea automáticamente todos los accesos de lectura y escritura al disco duro.

/dev/null

Cualquier cosa que se copie aquí desaparecerá. Es como un agujero negro.

Puede ser útil si queremos deshacernos de un flujo de datos, por ejemplo la salida estándar de un programa:

```
$ ls > /dev/null
```

La lectura desde `/dev/null` siempre ofrece un Fin de Archivo (`EOF`):

```
$ cp /dev/null archivo_vacio.txt
$ ls -l
-rwxr-xr-x    1 david    root            0 nov 10 19:25 archivo_vacio.txt
```

/dev/zero

En cierto sentido es lo contrario a `null`. Es un dispositivo del que siempre se leen ceros.

La siguiente orden crea un archivo de 1kB de ceros.

```
$ dd if=/dev/zero of=ArchivoVacio1 count=2
```

Para crear un archivo de 160 bytes iniciados a 0 usaremos la orden:

```
$ dd if=/dev/zero of=ArchivoVacio2 count=10 bs=16
```

/dev/full

Imita el funcionamiento de un archivo que está lleno. Cualquier intento de escritura devuelve el error ENOSPC.

/dev/random

Genera números aleatorios basados en el comportamiento del usuario. Si éste pasa mucho tiempo sin interactuar con el sistema, el dispositivo dejará de generar números.

Para mostrar 32 bytes aleatorios en hexadecimal usaremos:

```
$ hexdump -n 32 /dev/random
0000000 6dac 1ae2 2b45 f390 c8d0 60cd 4bdc b22d
0000010 c4ad 18e1 a34f 06ab c30d d01d fc5c a3c0
```

/dev/urandom

Genera secuencias de números pseudo-aleatorios. No es necesario que el usuario interactúe con la máquina.

Para crear un archivo de 2kB de información pseudo-aleatoria, usaremos:

```
$ dd if=/dev/urandom of=aleatorio bs=2k count=1
```

/dev/fd0 - /dev/fd1 - /dev/fd2 - ...

Representa al dispositivo de disco blando (disquetes). El formato es detectado automáticamente.

Si quisiéramos obtener la imagen de un disquete y copiarla a un archivo `disco.img`, introduciríamos:

```
$ dd if=/dev/fd0 of=disco.img bs=1440k
```

/dev/hda - /dev/hdb - /dev/hdc - ...

Unidades de disco IDE.

/dev/hda1 - /dev/hda2 - /dev/hda3 - ... - /dev/hda63

Corresponden a particiones dentro de la unidad de disco *a*. Las particiones para otros discos se especifican de la misma forma.

Los primeros cuatro números 1 ~ 4 corresponden a particiones primarias.

Los números en el intervalo 5 ~ 63 corresponden a particiones lógicas.

/dev/sda - /dev/sdb - /dev/sdc - ...

Son los discos duros SCSI.

/dev/cdrom - /dev/cdrom1 - /dev/cdrom2 - ...

Unidades de CD-ROM.

Para obtener la imagen ISO de un CD podemos usar el siguiente comando:

```
$ cp /dev/cdrom ./imagen.iso
```

/dev/loop0 - /dev/loop1 - /dev/loop2 - ...

Dispositivos que permiten implementar sistemas de archivos virtuales dentro de archivos de datos ordinarios.

/dev/parport0 - ... - /dev/parport3

Puertos paralelo del equipo.

/dev/lp0 - ... - /dev/lp3

Hasta cuatro impresoras conectadas a los correspondientes puertos paralelo.

Si tenemos una impresora post-script conectada al puerto paralelo, podemos imprimir en ella mediante:

```
$ cp Documento.ps /dev/lp0
```

También se le podría enviar un archivo de texto:

```
$ cat ArchivoTexto.txt > /dev/lp0
```

/dev/ttyS0 - ... - /dev/ttyS3

Puertos serie del equipo del estilo 8250/16450/16550, correspondientes a las direcciones 0x3f8, 0x2f8, 0x3e8, y 0x2e8, respectivamente.

Los nombres de DOS equivalentes son COM1: ~ COM4:.

/dev/tty1 - ... - /dev/tty63

Consolas de texto a pantalla completa.

/dev/pts/0 - /dev/pts/1 - /dev/pts/2 - ...

Consolas de texto virtuales creadas habitualmente para ejecutarse en ventanas de un sistema X-Window como hace xterm.

Podemos determinar en qué consola estamos escribiendo consultando los procesos mediante:

```
$ ps
 PID TTY          TIME CMD
1139 pts/3    00:00:00 bash
1678 pts/3    00:00:00 ps
```

En este caso se trata de la consola `/dev/pts/3`, correspondiente a un terminal de X-Window.

Podemos enviarle texto mediante la siguiente orden introducida desde otra consola:

```
$ cat > /dev/pts/3
```

Lo que tecleemos será redireccionado hacia el terminal virtual 3, hasta que termine `cat`.

/dev/psmouse

Puerto PS/2 del ratón.

/dev/psaux

Puerto PS/2 auxiliar.

/dev/mouse - /dev/input/mouse0 - /dev/input/mice

Representa al ratón, que estará conectado a alguno de los puertos disponibles.

/dev/input/js0 - ... - /dev/js0 - ...

Joysticks analógicos.

/dev/mixer - /dev/sequencer - /dev/dsp - /dev/midi00 - /dev/audio - ...

Dispositivos relacionados con el sonido: Mezclador, secuenciador, procesador digital, midi, y reproductor de ondas, respectivamente.

/dev/fb0 - ...

Buffer de marco de vídeo. Permite leer y escribir directamente la memoria de video. Esta característica es útil para poder representar gráficos sin la necesidad de un pesado sistema X-Window. Para poder utilizarlo es necesario configurar al núcleo en el momento del arranque.

/dev/rtc

Permite consultar el reloj de tiempo real. Habitualmente consultado a través de las funciones de biblioteca destinadas a tal fin.

Si se dispone del código fuente del núcleo se puede ver la lista completa de dispositivos en el archivo de texto:

```
/usr/src/linux-2.4/Documentation/devices.txt
```

Como desarrolladores de software es posible que queramos crear nuestro propio dispositivo personalizado. Para ello es necesario crear una entrada en el sistema de archivos que vincule un nombre a los números de dispositivo mayor y menor.

```
$ mknod ./mi_dispositivo c 42 0
```

En este caso se crea una nueva entrada para un dispositivo de tipo carácter al que se accederá mediante el nombre `mi_dispositivo` del directorio actual.

Llamadas al sistema para el manejo de archivos

La biblioteca estándar de C proporciona funciones para que las aplicaciones puedan leer y escribir archivos. Tal es el caso de las conocidas `fopen()`, `fclose()`, `fread()`, y `fwrite()`, que sirven para: abrir o crear, cerrar, leer, y escribir archivos respectivamente.

Los sistemas UNIX cuentan además con otras funciones de más bajo nivel con una funcionalidad equivalente. Se trata de las llamadas al sistema `open()`, `close()`, `read()`, y `write()`. Utilizan descriptores de dispositivo en lugar de punteros a estructuras de tipo FILE. Llamada al sistema equivalente a `fflush()` es `fsync()`.

A continuación se presenta la ayuda de estas órdenes:

open - abre y posiblemente crea un fichero

SINOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *camino, int flags);
```

DESCRIPCIÓN

La llamada al sistema `open()` se utiliza para convertir una ruta en un descriptor de fichero (un pequeño entero no negativo que se utiliza en las operaciones de E/S posteriores como en `read()`, `write()`, etc.).

Esta llamada crea un nuevo fichero abierto, no compartido con ningún otro proceso. El apuntador de lectura/escritura del fichero se configura para que apunte al principio del fichero. `flags` es uno de `O_RDONLY`, `O_WRONLY` u `O_RDWR` que, respectivamente, piden que la apertura del fichero sea solamente para lectura, solamente para escritura, o para lectura y escritura.

`flags` puede también ser la combinación, con el operador de bits OR (|), de una o más de las siguientes macros:

`O_CREAT`

Si el fichero no existe, será creado.

`O_EXCL`

Cuando se combina con `O_CREAT`, es un error que el fichero ya exista, y `open()` fallará.

VALOR DEVUELTO

`open` devuelve el nuevo descriptor de fichero, o -1 si ha ocurrido un error (en cuyo caso, se da un valor apropiado a `errno`).

Observe que `open` pueden abrir ficheros especiales de dispositivo, pero no puede crearlos; emplee `mknod` en su lugar.

close - cierra un descriptor de fichero

SINOPSIS

```
#include <unistd.h>

int close(int fd);
```

DESCRIPCIÓN

`close` cierra un descriptor de fichero de forma que ya no se refiera a fichero alguno y pueda ser reutilizado.

VALOR DEVUELTO

`close` devuelve 0 en caso de éxito y -1 si ocurre algún error.

read - lee de un descriptor de fichero

SINOPSIS

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

DESCRIPCIÓN

`read()` intenta leer hasta `nbytes` bytes del fichero cuyo descriptor de fichero es `fd` y guardarlos en la zona de memoria que empieza en `buf`. Si `nbytes` es cero, `read()` devuelve cero y no tiene otro efecto. Si `nbytes` es mayor que `SSIZE_MAX`, el resultado es indefinido.

VALOR DEVUELTO

En caso de éxito, se devuelve el número de bytes leídos (cero indica fin de fichero), y el indicador de posición del fichero avanza este número de bytes. No es un error si este número es menor que el número de bytes pedidos; esto puede suceder por ejemplo porque ahora mismo haya disponible un número menor de bytes (quizás porque estamos cerca del fin-de-fichero, o porque estamos leyendo de una interconexión, o de una terminal), o porque `read()` ha sido interrumpido por una señal. En caso de

error, se devuelve -1, y se pone un valor apropiado en `errno`. En este caso se deja indeterminado si el indicador de posición del fichero (si lo hay) cambia o no.

write - escribe a un descriptor de fichero

SINOPSIS

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t num);
```

DESCRIPCIÓN

`write()` escribe hasta `num` bytes en el fichero referenciado por el descriptor de fichero `fd` desde el búfer que comienza en `buf`. POSIX requiere que un `read()` que pueda demostrarse que ocurra después que un `write()` haya regresado, devuelva los nuevos datos. Observe que no todos los sistemas de ficheros son conformes con POSIX.

VALOR DEVUELTO

En caso de éxito, se devuelve el número de bytes escritos (cero indica pues que no se ha escrito nada). En caso de error, se devuelve -1 y se pone un valor apropiado en `errno`. Si `num` es cero y el descriptor de fichero se refiere a un fichero regular, se devolverá 0 sin que se cause ningún otro efecto. Para un fichero especial, los resultados no son transportables.

En aquellos archivos que soporten el acceso aleatorio es de utilidad el uso de `lseek()`:

lseek - reposiciona el puntero de lectura/escritura de un fichero

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPCIÓN

La función `lseek()` reposiciona el puntero del descriptor de fichero `fildes` con el argumento offset de acuerdo con la directiva `whence` como sigue:

`SEEK_SET`
: El puntero se coloca a offset bytes.

`SEEK_CUR`
: El número de bytes indicado en offset se suma a la dirección actual y el puntero se coloca en la dirección resultante.

`SEEK_END`
: El puntero se coloca al final del fichero más offset bytes.

La función `lseek()` permite colocar el puntero de fichero después del final de fichero. Si después se escriben datos en este punto, las lecturas siguientes de datos dentro del hueco que se forma devuelven ceros (hasta que realmente se escriban datos dentro de ese hueco).

VALOR DEVUELTO

En el caso de una ejecución correcta, `lseek()` devuelve la posición del puntero resultante medida en bytes desde el principio del fichero. Si se produce un error, se devuelve el valor `(off_t)-1` y en `errno` se coloca el tipo de error.

En realidad las llamadas a sistema vistas hasta este punto no son suficientes para controlar cualquier tipo de hardware.

Es necesaria la existencia de una llamada especial para poder comunicarnos con todos aquellos dispositivos que no encajan completamente en el modelo de flujos de E/S de UNIX.

ioctl - controlar dispositivo

SINOPSIS

```
#include <sys/ioctl.h>

int ioctl(int d, int peticion, ...)
```

DESCRIPCIÓN

La función `ioctl` manipula los parámetros subyacentes de ficheros especiales. En particular, muchas características operacionales de los ficheros especiales de caracteres (verbigracia las terminales) pueden controlarse con llamadas a `ioctl`. El argumento d debe ser un descriptor de fichero abierto.

Una petición de `ioctl` tiene codificada en sí misma si el argumento es un parámetro de entrada o de salida, y el tamaño del argumento `argp` en bytes. En el fichero de cabecera `<sys/ioctl.h>` se definen macros empleadas al especificar una petición de `ioctl`.

VALOR DEVUELTO

En caso de éxito, se devuelve cero. En caso de error, se devuelve -1 y se pone en `errno` un valor apropiado. Los argumentos, valores devueltos y semántica de `ioctl` varían según el controlador de dispositivo en cuestión.

Para conocer los comandos disponibles de los dispositivos soportados por el núcleo, podemos consultar el código fuente de los mismos.

Por ejemplo para expulsar un `cdrom` del lector se debe usar una llamada `ioctl()`.

En el archivo `/usr/src/linux-2.4/include/linux/cdrom.h` podemos encontrar los códigos reconocidos, entre los que se encuentra la siguiente línea:

```
#define CDROMEJECT          0x5309 /* Ejects the cdrom media */
```

A continuación se muestra el programa `eject_cd.c` que utiliza este código `ioctrl()`.

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    char* szDev = "/dev/cdrom";

    if (argc > 1)
         szDev = argv[1];

    /* Abre un descriptor de archivo especificado en la línea de comandos. */
    int fd = open (szDev, O_RDONLY);

    if (fd == -1)
         return -1;

    puts("Sacando la bandeja del CD-ROM.");

    /* Saca el CD-ROM. */
    ioctl (fd, CDROMEJECT);

    /* Cierra el descriptor de archivo. */
    close (fd);

    return 0;
}
```

```
$ gcc –o eject_cd eject_cd.c
```

Para poder utilizar la función debemos asegurarnos que el CD-ROM esté desmontado.

```
$ umount /dev/cdrom
$ ./eject_cd
Sacando la bandeja del CD-ROM.
```

Nota: Existe otro programa similar al creado llamado `eject` que incorpora una funcionalidad añadida.

Para poder hacer un seguimiento de las llamadas al sistema que realicen nuestras aplicaciones, podemos usar el programa `strace`:

```
$ strace ./eject_cd
execve("./eject_cd", ["./eject_cd"], [/* 32 vars */]) = 0
uname({sys="Linux", node="K7", ...})    = 0
brk(0)                                  = 0x8049614
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70382, ...}) = 0
old_mmap(NULL, 70382, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40013000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220Y\1"..., 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1395734, ...}) = 0
old_mmap(0x42000000, 1239844, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000
mprotect(0x42126000, 35620, PROT_NONE)  = 0
old_mmap(0x42126000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x126000) = 0x42126000
old_mmap(0x4212b000, 15140, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4212b000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40025000
munmap(0x40013000, 70382)               = 0
open("/dev/cdrom", O_RDONLY)            = 3
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40013000
write(1, "Sacando la bandeja del CD-ROM.\n", 31Sacando la bandeja del CD-ROM.
) = 31
ioctl(3, 0x5309, 0x4202b21d)            = -1 EBUSY (Device or resource busy)
close(3)                                = 0
munmap(0x40013000, 4096)                = 0
_exit(0)                                = ?
```

La primera llamada al sistema que se realiza es `execve()` encargada de cargar la imagen del archivo `eject_cd` en el mapa de memoria del proceso.

La llamada a `uname` se utiliza para que la aplicación disponga de información de la plataforma sobre la que se está ejecutando.

También podemos consultarla desde la línea de comandos:

```
$ uname -a
Linux K7 2.4.18-17.8.0custom #9 sab nov 23 00:39:22 CET 2002 i686 athlon i386
GNU/Linux
```

Nos Informa del nombre y versión del S.O., del nombre de la máquina, la fecha, y el tipo de procesador sobre el que se ha compilado.

A continuación aparecen una serie de llamadas destinadas a cargar las bibliotecas dinámicas de funciones de C, y mapearlas (`mmap`) en el espacio de usuario.

Las últimas líneas corresponden a las llamadas realizadas por nuestra aplicación.

Apréciese que la llamada a `puts` internamente es traducida por una llamada a `write`, con un descriptor de archivo de 1 (`stdout`).

En este caso la ejecución de `ioctl()` no ha tenido éxito, y como podemos ver se nos informa del tipo de error producido (`EBUSY`).

Si la orden hubiera podido ser atendida, se habría devuelto 0.

Los descriptores de archivo que devuelve `open()` suelen ser números enteros bajos no ocupados previamente. En este caso, se ha asignado un 3 al descriptor de CD-ROM.

Practica a realizar.

1. Realizar un programa que tenga la misma funcionalidad que la orden `dd` (únicamente la opción notrunc) tal y como se ha explicado en esta práctica. Ayudarse de las páginas del manual. Optimizar la aplicación para que utilice la mínima memoria posible incluso en el caso de utilizar archivos grandes (realizar la copia por bloques).

Anexo I. Ejemplo de uso del depurador gdb

En el presente anexo, como ejemplo de uso de gdb, vamos a depurar un pequeño programa que aunque aparentemente pueda parecer correcto, contiene algunos errores típicos de tiempo de ejecución. Aquí sólo veremos un subconjunto de las posibilidades que este depurador puede ofrecer, asi como su funcionamiento en línea de comandos.

El programa tomará datos de la entrada estándar, y devolverá una estadística del uso de los caracteres que se introduzcan. Para detener la introducción de datos en UNIX se utiliza la combinación [Ctrl] + [D], que genera un carácter EOF de fin de fichero.

```c
#include <stdio.h>
#include <assert.h>

#define FALSE 0
#define TRUE 1


/*--------------------------------   EstadisticaDeEntrada   ------------------------------ *\
 * Devuelve un array con la función de distribución de los caracteres introducidos por      *
\*    el flujo de entrada estándar stdin.                                                   */
int* EstadisticaDeEntrada()
{
    int caracter;
    int fdd[256];

    memset(fdd, 0, 256); // No se inician todos los bytes del array

    while((caracter = getchar()) != EOF)
        fdd[(char) caracter] ++;

    return fdd; // Se devuelve una referencia a un área de la pila que se va a invalidar
}

/*--------------------------------   ImprimeResultados  ---------------------------------- *\
 * Imprime por pantalla los resultados de la estadística.                                   *
\* Devuelve verdadero si se ha procesado al menos un caracter, y falso en el caso contrario.*/
int ImprimeResultados(int* pFdd) // no se está pasando como un puntero constante
{
    int i, total;

    assert(pFdd);

    for(i = 0; i <= 256; i++) // Se está recorriendo un carácter más de la capacidad total
    {
        if(i == '\0')
            printf("[%d]='\0': %d\t", i, i, pFdd[i]); // No se imprime correctamente la secuencia de escape
        else if(i == '\n')
            printf("[%d]='\n': %d\t", i, i, pFdd[i]); // Se cuenta con más parámetros que los esperados
        else if(i == '\r')
            printf("[%d]='\r': %d\t", i, i, pFdd[i]);
        else if(i == '\t')
            printf("[%d]='\t': %d\t", i, i, pFdd[i]);
        else if(i < ' ')
            printf("[%d]:    %d\t", i, i, pFdd[i]);
        else
            printf("[%d]='%c': %d\t", i, i, pFdd[i]);

#ifndef NDEBUG
        fflush(stdout); // Forzar que se actualice la salida estándar stdout.
#endif
        total += pFdd[i]; // No se ha iniciado a 0
    }

    printf("\n\nCaracteres en total = %d", total);



    if(pFdd['a'] = total) // Se está haciendo una asignación en lugar de una comparación
        printf(", todos son aes.\n");
    else
        printf(", de los que el %g%% son aes.\n", pFdd['a']/ total); // La división entera genera un entero

```

```
64
65          return total;
66      }
67
68      int main()
69      {
70          int* pFdd;
71
72          pFdd = EstadisticaDeEntrada();
73
74          printf("\n\n");
75
76          if(ImprimeResultados(pFdd) == TRUE) // En C cualquier valor entero distinto de 0 es verdadero
77              printf("Se han procesado caracteres de entrada.\n");
78          else
79              printf("No se han procesado caracteres de entrada.\n");
80
81          return 0;
82      }
83
```

Si realizamos una compilación y lo ejecutamos, veremos que no está generando los resultados esperados:

```
$ gcc -o estadistica estadistica.c
$ ./estadistica
Se escriben caracteres hasta presionar [Ctrl] + [D]
. . .
```

Si en base a la respuesta del programa no somos capaces de detectar los fallos del programa, es necesario el uso de un depurador. Para poder depurar sobre el código fuente, debe ser incluida una *Tabla de Símbolos Ampliada* en la cabecera del archivo ejecutable, en la que se encuentren las equivalencias entre posiciones de memoria de variables y funciones, y los nombres que se les asignó en el programa. Esto se consigue usando el modificador `-g` de **`gcc`**:

```
$ gcc -g -o estadistica estadistica.c
$ ./estadistica
Se escriben caracteres hasta presionar [Ctrl] + [D].
```

Al contrario de lo que pudiera pensarse, el incluir estos símbolos prácticamente no ralentiza la ejecución de las aplicaciones, por lo que podremos incluirlos incluso en los casos en los que la rapidez de ejecución de alguna rutina sea fundamental.

Para entrar en el depurador, introducimos lo siguiente:

```
$ gdb estadistica
```

Tras un mensaje de presentación, se muestra el inductor de comandos del `gdb`. Desde aquí podremos insertar las consignas de depuración. Casi todos los comandos cuentan con un formato largo y otro abreviado totalmente equivalente. Para expresarlos aquí, se mostrarán como `nombre_comando{abreviación}`.

Podemos pedir ayuda en cualquier momento mediante `help{h}`.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
```

```
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

Para pedir información sobre las funciones que se utilizan, se usa `info{i} functions`:

```
(gdb) info functions
All defined functions:

File programa9.c:
int *EstadisticaDeEntrada();
int ImprimeResultados(int *);
int main();

Non-debugging symbols:
0x080482a8  _init
0x080482d0  getchar
0x080482e0  __assert_fail
0x080482f0  __libc_start_main
0x08048300  printf
0x08048310  memset
0x08048320  _start
0x08048344  call_gmon_start
0x08048368  __do_global_dtors_aux
0x080483a4  frame_dummy
0x08048630  __do_global_ctors_aux
0x08048654  _fini
```

Se nos muestran todas las funciones que se están utilizando directa o indirectamente. Primero aparecen las tres presentes en nuestro programa de las que `gdb` cuenta con toda la información de depuración necesaria. El resto pertenecen a bibliotecas que fueron compiladas sin información de depuración, por lo que no podremos depurarlas en forma de código fuente.

Para ejecutar el programa se usa el comando `run{r}`:

```
(gdb) run
```

Genera idéntica salida que si se hubiera ejecutado desde el intérprete de comandos del sistema.

Uno de los mecanismos más habituales en la depuración de código es el establecimiento de puntos de interrupción. Se trata de seleccionar una línea de código de tal forma que cuando se vaya a procesar, se detenga la ejecución a la espera de nuevas órdenes. Si quisiéramos por ejemplo detener la ejecución al principio de una función, introduciríamos `break{b}` seguido del nombre de la función. En el siguiente ejemplo se indica que se detenga la ejecución nada más comenzar `main()`:

```
(gdb) break main
Breakpoint 1 at 0x80485ec: file programa9.c, line 69.
```

También se puede indicar explícitamente la línea donde queremos que se detenga, por ejemplo en el primer `printf()` de la función `ImprimeResultados()`:

```
(gdb) break 36
Breakpoint 2 at 0x8048455: file programa9.c, line 36.
```

Para listar todos los puntos de interrupción configurados usamos `info{i} breakpoints{b}`:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080485ec in main at programa9.c:69
2   breakpoint     keep y   0x08048455 in ImprimeResultados at programa9.c:36
```

Como vemos, `gdb` numera cada punto de interrupción. De esta forma podemos referirnos a ellos de forma más sencilla. Podemos deshabilitar y rehabilitar un breakpoint mediante `disable` y `enable` seguido del número correspondiente.

```
(gdb) disable 2
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080485ec in main at programa9.c:69
2   breakpoint     keep n   0x08048455 in ImprimeResultados at programa9.c:36
```

Para borrarlos se emplea `delete{d}` seguido del número:

```
(gdb) delete 1
(gdb) info breakpoints
Num Type           Disp Enb Address    What
2   breakpoint     keep n   0x08048455 in ImprimeResultados at programa9.c:36
```

Si ejecutamos de nuevo el programa:

```
(gdb) run
Starting program: /mnt/transporte/Universidad/Trabajo/Informática Industrial
II/Laboratorio - Prácticas/Práctica 1/estadistica
Se escriben caracteres hasta presionar [Ctrl] + [D].

Breakpoint 2, ImprimeResultados (pFdd=0xbffff450) at programa9.c:36
36                  printf("[%d]='\0': %d\t", i, i, pFdd[i]);
```

Nos quedamos detenidos en la línea 36 donde establecimos el breakpoint. Podemos establecer otro breakpoint dos líneas más abajo mediante:

```
(gdb) break +2
Breakpoint 4 at 0x8048483: file programa9.c, line 38.
```

Para continuar con la ejecución, insertaremos `continue{c}`:

```
(gdb) continue
Continuing.

Breakpoint 4, ImprimeResultados (pFdd=0xbffff450) at programa9.c:38
38                  printf("[%d]='\n': %d\t", i, i, pFdd[i]);
```

Podemos solicitar que nos muestre el código fuente que estamos depurando mediante la orden `list{l} principio[, fin]`. Tanto `principio` como `fin` pueden ser nombres de funciones o números de línea. Si no se especifica `fin` se muestran 10 líneas que se encuentren el la proximidad de `principio`, 5 por delante y 5 por detrás. Si no se especifica `principio`, se tomará la línea siguiente a las que se mostraron en el último listado. Para mostrar las primeras 10 líneas más cercanas a `main()` introduciremos:

```
(gdb) list main
64
65          return total;
66      }
67
68      int main()
69      {
70          int* pFdd;
71
72          pFdd = EstadisticaDeEntrada();
73
```

Para mostrar las líneas que van desde la 68 a la 83, introduciremos:

```
(gdb) list 68, 83
68      int main()
69      {
70          int* pFdd;
71
72          pFdd = EstadisticaDeEntrada();
73
74          printf("\n\n");
75
76          if(ImprimeResultados(pFdd) == TRUE)
77              printf("Se han procesado caracteres de entrada.\n");
78          else
79              printf("No se han procesado caracteres de entrada.\n");
80
81          return 0;
82      }
83
```

En cualquier momento, podemos solicitar que se nos muestre la pila de llamadas a funciones mediante el comando `backtrace{bt}` o con `where{w}`, con los que también se indica el valor de los argumentos de las mismas.

Podemos avanzar por las líneas del programa paso a paso mediante la orden `step{s}`. Si se encontraran funciones que contasen con información de depuración, se entrará a ejecutar su código paso a paso. Si no quisiéramos este comportamiento usaríamos la orden `next{n}`. Con `next` las funciones que se encontraran serán ejecutadas como si se trataran de una única línea:

```
(gdb) n
72          pFdd = EstadisticaDeEntrada();
(gdb) s
EstadisticaDeEntrada () at programa9.c:16
16          memset(fdd, 0, 256); // No se inician todos los bytes del array
(gdb) n
18          while((caracter = getchar()) != EOF)
. . .
```

```
(gdb) backtrace
#0  ImprimeResultados (pFdd=0xbffff450) at programa9.c:38
#1  0x080485ff in main () at programa9.c:71
#2  0x420158d4 in __libc_start_main () from /lib/i686/libc.so.6
```

Backtrace será de gran utilidad en los programas que llamen a la misma función desde diferentes sitios, o de forma recursiva. De esta forma se puede determinar cómo se ha llegado hasta ese punto. Observamos que la primera función de C que se ejecuta en los programas no es `main()`, sino `__libc_start_main()`, perteneciente a la biblioteca de C.

Para salir del depurador introduciremos `quit{q}`:

```
(gdb) quit
```

Como hemos podido apreciar, aunque pusimos un punto de interrupción en la línea 38, no se generó salida alguna por el `printf()` de la línea 36, por lo que no pudimos ver su efecto. Esto es debido a que el sistema utiliza un mecanismo basado en buffer FIFO para sacar datos por la salida estándar, de tal forma que cuando se hayan acumulado suficientes, se llevarán todos de golpe a la salida con una sola operación de E/S.

Para ayudar en la depuración podríamos forzar la impresión mediante la función `fflush()` que ordena que se vacíe el buffer del archivo especificado. La función `printf()` escribe el archivo de salida estándar por lo que usaremos `stdout` en este caso. Introducimos las siguientes líneas en el código en forma de compilación condicional, de tal forma que no afecten cuando no se vaya a depurar:

```
. . .
48. #ifndef NDEBUG
49.         fflush(stdout); // Forzar que se actualice la salida estándar stdout.
50. #endif
51.        total += pFdd[i];
}
. . .
```

Poniendo ahora un punto de interrupción en la línea 51, podremos ir viendo la salida de texto según se vaya generando.

```
(gdb) b 51
Breakpoint 1 at 0x80485b8: file programa9.c, line 51.
(gdb) r
Starting program: /mnt/transporte/Universidad/Trabajo/Informática Industrial
II/Laboratorio - Prácticas/Práctica 1/estadistica
asfasdf asdf asd fasdf a
[0]='
Breakpoint 1, ImprimeResultados (pFdd=0xbffff450) at programa9.c:51
51              total += pFdd[i];
```

Como vemos, no se llega a imprimir correctamente siquiera el primer `printf()`. Deteniéndonos más en la forma en que está escrita la cadena:

```
"[%d]='\0': %d\t"
```

y la salida que provoca:

```
[0]='
```

Vemos que lo que hemos indicado es que inserte la secuencia de escape `\0` en plena cadena, lo que es interpretado por `printf()` como un carácter NUL de final de cadena. Similares casos ocurren en los `printf()` siguientes. Deberíamos haber escrito lo siguiente:

```
"[%d]='\\0': %d\t"
"[%d]='\\n': %d\t"
"[%d]='\\r': %d\t"
"[%d]='\\t': %d\t"
```

Recompilando y volviendo al depurador:

```
(gdb) b 51
Breakpoint 2 at 0x80485b8: file estadistica.c, line 51.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /mnt/transporte/Universidad/Trabajo/Informática Industrial
II/Laboratorio - Prácticas/Práctica 1/estadistica
asasda ada fadfasd


[0]='\0': 0
Breakpoint 2, ImprimeResultados (pFdd=0xbffff440) at estadistica.c:51
51              total += pFdd[i];
```

Ahora sí ha funcionado correctamente, pero si continuamos tres iteraciones más, veremos que no continúa imprimiendo lo que pretendíamos:

```
(gdb) c
Continuing.
[1]:    1
Breakpoint 2, ImprimeResultados (pFdd=0xbffff440) at estadistica.c:51
51              total += pFdd[i];
(gdb) c
Continuing.
[2]:    2
Breakpoint 2, ImprimeResultados (pFdd=0xbffff440) at estadistica.c:51
51              total += pFdd[i];
(gdb) c
Continuing.
[3]:    3
Breakpoint 2, ImprimeResultados (pFdd=0xbffff440) at estadistica.c:51
51              total += pFdd[i];
```

En apariencia el número de ocurrencias es igual al índice del array. Repasando los argumentos de `printf()`, vemos que no concuerdan con los que se habían especificado en la cadena:

```
           printf("[%d]='\\0': %d\t", i, i, pFdd[i]);
       else if(i == '\n')
           printf("[%d]='\\n': %d\t", i, i, pFdd[i]);
       else if(i == '\r')
           printf("[%d]='\\r': %d\t", i, i, pFdd[i]);
       else if(i == '\t')
           printf("[%d]='\\t': %d\t", i, i, pFdd[i]);
       else if(i < ' ')
           printf("[%d]:    %d\t", i, i, pFdd[i]);
```

Hay que eliminar un argumento de cada una y recompilar:

```
      if(i == '\0')
           printf("[%d]='\\0': %d\t", i, pFdd[i]);
       else if(i == '\n')
           printf("[%d]='\\n': %d\t", i, pFdd[i]);
```

```
        else if(i == '\r')
            printf("[%d]='\\r': %d\t", i, pFdd[i]);
        else if(i == '\t')
            printf("[%d]='\\t': %d\t", i, pFdd[i]);
        else if(i < ' ')
            printf("[%d]:    %d\t", i, pFdd[i]);
        else
            printf("[%d]='%c': %d\t", i, i, pFdd[i]);
```

No es necesario salir de `gdb` para depurar la nueva versión. Simplemente con indicarle que corra de nuevo con `run{r}`, actualizará automáticamente la tabla de símbolos interna.

Si ejecutamos el programa sin interrupciones, veremos que llegado a cierto punto, se empieza a obtener cuentas extrañas.

```
[79]='O': 0     [80]='P': 0     [81]='Q': 0     [82]='R': 0     [83]='S': 0
[84]='T': 0   [85]='U': 0      [86]='V': 0     [87]='W': 0     [88]='X': 0
[89]='Y': 0     [90]='Z': 0     [91]='[': 0     [92]='\': 0     [93]=']': 0
[94]='^': 0     [95]='_': 0     [96]='`': 0     [97]='a': 6   [98]='b': 0
[99]='c': 1073798334   [100]='d': 5   [101]='e': 0   [102]='f': 1073745699
[103]='g': 1073816760  [104]='h': -1073744199  [105]='i': 0    [106]='j': -
1073743848  [107]='k': 1073793407
```

Para no tener que avanzar por todos los elementos uno por uno, pondremos un punto de interrupción condicional también conocido como punto de vigilancia (*watchpoint*) cuando el índice vaya por 97.

```
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) break 51 if i==97
Breakpoint 4 at 0x80485b8: file estadistica.c, line 51.
(gdb) i b
Num Type           Disp Enb Address    What
4      breakpoint          keep  y      0x080485b8  in  ImprimeResultados  at
estadistica.c:51
        stop only if i == 97
(gdb) r
. . . [95]='_': 0     [96]='`': 0     [97]='a': 7
Breakpoint 4, ImprimeResultados (pFdd=0xbffff440) at estadistica.c:51
51              total += pFdd[i]; // No se ha iniciado a 0
```

Para consultar el valor de una variable, se usa el comando `print{p}`, que admite como argumento una expresión con una sintaxis similar a la de C, pero sin paréntesis. Incluso podemos aplicarles una conversión *cast* si deseamos ver el valor con algún formato distinto:

```
(gdb) print i
$1 = 97
(gdb) print (char) i
$2 = 97 'a'
(gdb) print pFdd[i]
$3 = 7
```

Esto nos indica que en el texto que se había introducido de prueba, existían 7 letras 'a'. Si lo necesitáramos, podríamos cambiar el valor de una variable:

```
(gdb) print i=400
$4 = 400
```

Continuando con la ejecución vemos que el contenido inicial del elemento 100 de `pFdd` es muy elevado:

```
(gdb) c
Continuing.
[99]='c': 1073798334
Breakpoint 5, ImprimeResultados (pFdd=0xbffff440) at estadistica.c:51
51              total += pFdd[i];
```

Esto nos induce a pensar que el array no está bien iniciado. En efecto, si consultamos la línea 16 vemos:

```
memset(fdd, 0, 256);
```

que sólo inicia a 0 los primeros 256 bytes del array; pero el array contiene 256 enteros y cada uno ocupa más de 1 byte. La solución es iniciar el array con el tamaño correspondiente:

```
memset(fdd, 0, 256 * sizeof(int));
```

Ejecutando de nuevo el programa, vemos que aún no se ha solucionado el problema de la basura. Ahora aparece en posiciones más elevadas (a partir de la 141 aproximadamente). Si depuramos la función `EstadisticaDeEntrada()`, veríamos que los datos son rellenados correctamente. Sin embargo, ejecutando desde la función `ImprimeResultados()`, vemos que llegado a cierto valor el array toma valores erróneos diferentes a los que se habían establecido. Se trata de un fallo habitual de principiante: En la función `EstadisticaDeEntrada()` se ha rellenado correctamente un array local, que es devuelto para ser reutilizado posteriormente; pero recordemos que cuando una función sale de ámbito, el contenido de su marco de pila queda anulado y puede ser sobrescrito por el de las otras funciones. Aunque este sería un error difícil de detectar con un depurador, **`gcc`** nos lo ha estado advirtiendo durante todas las compilaciones. La solución es crear el array `fdd` en un sitio que no quede fuera de ámbito durante su uso. A continuación se muestra la posibilidad de ponerlo como variable local de `main()`:

```
1.  #include <stdio.h>
2.  #include <assert.h>
3.
4.  #define FALSE 0
5.  #define TRUE 1
6.
7.
8.  /*--------------------------------   EstadisticaDeEntrada   ------------------------------ *\
9.   * Devuelve un array con la función de distribución de los caracteres introducidos por      *
10. \*    el flujo de entrada estándar stdin.                                                    */
11. void EstadisticaDeEntrada(int* pFdd)
12. {
13.     int caracter;
14.
15.     memset(pFdd, 0, 256 * sizeof(int)); // No se inician todos los bytes del array
16.
17.     while((caracter = getchar()) != EOF)
18.         pFdd[(char) caracter] ++;
19. }
20.
```

```
21. /*--------------------------------   ImprimeResultados  ---------------------------------- *\
22. * Imprime por pantalla los resultados de la estadística.                                    *
23. \* Devuelve verdadero si se ha procesado al menos un caracter, y falso en el caso contrario.*/
24. int ImprimeResultados(int* pFdd) // no se está pasando como un puntero constante
25. {
26.     int i, total;
27.
28.     assert(pFdd);
29.
30.     for(i = 0; i <= 256; i++) // Se está recorriendo un carácter más de la capacidad total
31.     {
32.         if(i == '\0')
33.             printf("[%d]='\\0': %d\t", i, pFdd[i]);// No se imprime correctamente la
    secuencia de escape
34.         else if(i == '\n')
35.             printf("[%d]='\\n': %d\t", i, pFdd[i]); // Se cuenta con más parámetros que los esperados
36.         else if(i == '\r')
37.             printf("[%d]='\\r': %d\t", i, pFdd[i]);
38.         else if(i == '\t')
39.             printf("[%d]='\\t': %d\t", i, pFdd[i]);
40.         else if(i < ' ')
41.             printf("[%d]:    %d\t", i, pFdd[i]);
42.         else
43.             printf("[%d]='%c': %d\t", i, i, pFdd[i]);
44.
45. #ifndef NDEBUG
46.         fflush(stdout); // Forzar que se actualice la salida estándar stdout.
47. #endif
48.         total += pFdd[i]; // No se ha iniciado a 0
49.     }
50.
51.     printf("\n\nCaracteres en total = %d", total);
52.
53.
54.
55.     if(pFdd['a'] = total) // Se está haciendo una asignación en lugar de una comparación
56.         printf(", todos son aes.\n");
57.     else
58.         printf(", de los que el %g%% son aes.\n", pFdd['a'] * 100 / total); // La división entera
    genera un entero
59.
60.
61.
62.     return total;
63. }
64.
65. int main()
66. {
67.     int fdd[256];
68.
69.     EstadisticaDeEntrada(fdd);
70.
71.     printf("\n\n");
72.
73.     if(ImprimeResultados(fdd) == TRUE) // En C cualquier valor entero distinto de 0 es verdadero
74.         printf("Se han procesado caracteres de entrada.\n");
75.     else
76.         printf("No se han procesado caracteres de entrada.\n");
77.
78.     return 0;
79. }
80.
```

Ejecutando de nuevo el programa, vemos que aun aparece el último elemento con una cuenta elevada:

```
[251]='': 0     [252]='': 0     [253]='': 0     [254]='': 0     [255]='': 0
[256]='': 1107341000
```

Ya que los elementos del array `fdd` están comprendidos de 0 a 255, el máximo valor que deberíamos representar se obtendría con el subíndice `i` igual a 255, por lo que preparamos un watchpoint para que se interrumpa la ejecución cuando se sobrepase.

```
(gdb) b 32
Breakpoint 1 at 0x80484a7: file estadistica_bak.c, line 32.
(gdb) r
Starting program: /mnt/transporte/Universidad/Trabajo/Informática Industrial
II/Laboratorio - Prácticas/Práctica 1/estadistica
. . .
Breakpoint 1, ImprimeResultados (pFdd=0xbffff440) at estadistica_bak.c:32
32              if(i == '\0')
(gdb) b if i==256
Note: breakpoint 1 also set at pc 0x80484a7.
Breakpoint 2 at 0x80484a7: file estadistica_bak.c, line 32.
(gdb) disable 1
```

Los puntos de vigilancia que se ponen en variables locales, sólo pueden establecerse si se está dentro del ámbito de la función. Para ello primero se prepara un punto de ruptura en la línea 35, y después se sustituye por el watchpoint.

```
(gdb) c
Continuing.
. . . [252]='': 1108505376     [253]='': 134514687     [254]='': -1073743788
[255]='': 1073819648
Breakpoint 2, ImprimeResultados (pFdd=0xbffff440) at estadistica_bak.c:32
32              if(i == '\0')
(gdb) p i
$1 = 256
```

Como podemos comprobar, efectivamente se ha llegado a 256. La razón es simple: la condición del `for(;;)` para mantenerse dentro del bucle no está bien elegida:

```
for(i = 0; i <= 256; i++)
```

Debería sustituirse por:

```
for(i = 0; i < 256; i++)
```

Aunque ahora los datos del array se presentan en pantalla correctamente, la cuenta de caracteres es errónea. Para averiguar lo que pasa, ponemos un punto de interrupción en la primera línea que utiliza la variable acumuladora `total`, para comprobar su contenido en la primera ejecución:

```
(gdb) b 48
Breakpoint 1 at 0x80485b0: file estadistica.c, line 48.
(gdb) r
Starting program: /mnt/transporte/Universidad/Trabajo/Informática Industrial
II/Laboratorio - Prácticas/Práctica 1/estadistica
asfasd asdfa sd
```

```
[0]='\0': 0
Breakpoint 1, ImprimeResultados (pFdd=0xbffff460) at estadistica.c:48
48              total += pFdd[i];
(gdb) p total
$1 = 1107633040
```

Vemos que antes de realizar el incremento correspondiente, ya cuenta con un valor inicial, cuando debería valer 0. El fallo es que no ha sido iniciada:

```
26.     int i, total = 0;
```

Ahora notifica correctamente la cuenta de los caracteres leídos, sin embargo indica erróneamente que siempre son 'a'.

```
Caracteres en total = 5, todos son aes.
```

Para saber por qué imprime este mensaje, ponemos un punto de interrupción en la línea 55 que es donde se verifica la igualdad entre la cantidad de 'a' y el número de caracteres en total.

```
(gdb) b 55
Breakpoint 2 at 0x80485f6: file estadistica.c, line 55.
(gdb) r
Starting program: /mnt/transporte/Universidad/Trabajo/Informática Industrial
II/Laboratorio - Prácticas/Práctica 1/estadistica
asdfasgd
. . .
Breakpoint 2, ImprimeResultados (pFdd=0xbffff460) at estadistica.c:55
55          if(pFdd['a'] = total)
(gdb) p pFdd['a']
$2 = 2
(gdb) p total
$3 = 9
```

Como vemos justo antes de procesar la línea, ambas variables contienen valores distintos. Mediante el comando `step{s}` podemos avanzar una línea. Consultando de nuevo las variables, descubrimos que ahora tienen el mismo valor:

```
(gdb) step
56              printf(", todos son aes.\n");
(gdb) p pFdd['a']
$4 = 9
(gdb) p total
$5 = 9
```

La culpa de que esto sea así la tiene la propia línea 55, que en lugar de realizar una comparación, hace una asignación. Como el resultado de esta es distinto de 0, se toma como una expresión verdadera, y por eso se ejecuta la primera opción del `if()`. La solución es por tanto rescribirla como aparece a continuación:

```
if(pFdd['a'] == total)
```

Hubiera ayudado a detectar este fallo el haber especificado desde un principio que `pFdd` apunta a un array que no debería ser modificado, usando para ello el modificador `const`:

```
int ImprimeResultados(const int* pFdd)
```

Ahora resulta que no se notifica bien la proporción de 'a' que hay en el texto.

```
Caracteres en total = 15, de los que el 5.41109e-312% son aes.
```

Esto es porque en el `printf()` se está pasando una división de enteros, pero se pide representar un número en coma flotante:

```
printf(", de los que el %g%% son aes.\n", pFdd['a'] * 100 / total);
```

Simplemente con especificar que el número por el que se multiplica sea de coma flotante, se realiza una conversión automática del resto de la expresión. Para mayor seguridad, siempre se podría usar la conversión forzosa a `float`. Aprovechamos también para especificar la cantidad de dígitos que se deberían mostrar como máximo.

```
printf(", de los que el %.3g%% son aes.\n", pFdd['a'] * 100.f / total);
```

Ahora ya se produce una salida más aceptable:

```
Caracteres en total = 6, de los que el 16.7% son aes.
No se han procesado caracteres de entrada.
```

Sin embargo la última línea dice incorrectamente que no se han procesado caracteres de entrada cuando en la línea anterior se especifica que sí los ha habido. El que se haya producido esa salida induce a pensar que la condición que aparece en la línea 73 ha de ser falsa:

```
if(ImprimeResultados(fdd) == TRUE)
```

Podríamos comprobar el valor que devuelve `ImprimeResultados()` poniendo un punto de interrupción en su última línea:

```
(gdb) b 62
Breakpoint 4 at 0x804863f: file estadistica.c, line 62.
```

y ejecutando hasta llegar:

```
Breakpoint 4, ImprimeResultados (pFdd=0xbffff460) at estadistica.c:62
62          return total;
(gdb) p total
$1 = 9
```

Con lo que vemos que se está devolviendo un valor verdadero (9 en este caso). El problema está en la propia línea 73. Aunque 9 y `TRUE` sean los dos verdaderos, tienen valores numéricos distintos por lo que la comparación de igualdad da falso como resultado. Nunca deberíamos comparar con `TRUE`. Para subsanarlo podemos arreglar la expresión cambiándola por cualquiera de las dos siguientes:

```
if(ImprimeResultados(fdd))
```

o

```
if(ImprimeResultados(fdd) != FALSE)
```

El programa finalmente quedaría:

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

#define FALSE 0
#define TRUE 1

/*--------------------------------   EstadisticaDeEntrada   ------------------------------ *\
 * Devuelve un array con la función de distribución de los caracteres introducidos por      *
\*    el flujo de entrada estándar stdin.                                                   */
void EstadisticaDeEntrada(int* pFdd)
{
    int caracter;

    memset(pFdd, 0, 256 * sizeof(int));

    while((caracter = getchar()) != EOF)
        pFdd[(char) caracter] ++;
}

/*--------------------------------   ImprimeResultados  ---------------------------------- *\
 * Imprime por pantalla los resultados de la estadística.                                   *
\* Devuelve verdadero si se ha procesado al menos un caracter, y falso en el caso contrario.*/
int ImprimeResultados(const int* pFdd) // no se está pasando como un puntero constante
{
    int i, total = 0;

    assert(pFdd);
    for(i = 0; i < 256; i++)
    {
        if(i == '\0')
            printf("[%d]='\\0': %d\t", i, pFdd[i]);
        else if(i == '\n')
            printf("[%d]='\\n': %d\t", i, pFdd[i]);
        else if(i == '\r')
            printf("[%d]='\\r': %d\t", i, pFdd[i]);
        else if(i == '\t')
            printf("[%d]='\\t': %d\t", i, pFdd[i]);
        else if(i < ' ')
            printf("[%d]:    %d\t", i, pFdd[i]);
        else
            printf("[%d]='%c': %d\t", i, i, pFdd[i]);

#ifndef NDEBUG
        fflush(stdout);
#endif
        total += pFdd[i];
    }

    printf("\n\nCaracteres en total = %d", total);

    if(pFdd['a'] == total)
        printf(", todos son aes.\n");
    else
        printf(", de los que el %.3g%% son aes.\n", pFdd['a'] * 100.f / total);

    return total;
}
```

```
int main()
{
    int fdd[256];

    EstadisticaDeEntrada(fdd);

    printf("\n\n");

    if(ImprimeResultados(fdd))
        printf("Se han procesado caracteres de entrada.\n");
    else
        printf("No se han procesado caracteres de entrada.\n");

    return 0;
}
```

Cuando generemos programas en modo de depuración, sería buena idea pasar al menos los siguientes modificadores a **gcc**:

```
$ gcc -Wall -ansi -pedantic -g -o nombre_programa nombre_archivo.c
```

Los tres primeros modificadores ayudarán a encontrar potenciales fallos antes de que se genere el ejecutable.

Para generar la versión definitiva, podríamos usar:

```
$ gcc -Wall -ansi -pedantic –Os -DNDEBUG -o nombre_programa nombre_archivo.c
```