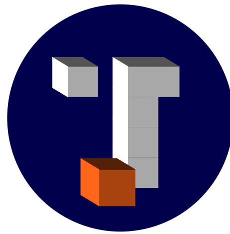
UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN



PROYECTO FIN DE CARRERA

TRAZADO DE LLAMADAS A FUNCIONES DE BIBLIOTECA DINÁMICA

JUAN CÉSPEDES PRIETO

Mayo de 2003

Título:
Trazado de llamadas a funciones de biblioteca dinámica

Autor:
Juan Céspedes Prieto

Tutor:
Joaquín Seoane Pascual

Departamento:
Ingeniería de Sistemas Telemáticos

Miembros del tribunal:

Presidente:

Vocal:

Secretario:

Calificación:

Madrid, de de 2003.

Resumen:

El objetivo general del proyecto es facilitar la depuración de programas de usuario por medio del estudio de trazas de su comportamiento. Los eventos de interés a trazar son las llamadas a funciones de biblioteca dinámica, las llamadas al sistema, así como las señales recibidas. En particular se realiza una herramienta para sistemas POSIX, de código abierto y libre distribución, que mejora otras herramientas disponibles, siendo su principal novedad el trazado de llamadas a bibliotecas dinámicas.

Palabras clave:

Depuración, trazado, biblioteca dinámica, llamada al sistema, señal, *breakpoint*, ELF, `ltrace`, `ptrace`, `strace`

Summary:

The aim of the project is to simplify the debugging of user programs by tracing their behaviour. The events that will be traced are dynamic library calls, system calls, and signals received. An open-source and freely redistributable POSIX systems tool is developed for this purpose, with the dynamic library call tracing being its main breakthrough.

Keywords:

Debugging, trace, dynamic library, system call, signal, *breakpoint*, ELF, `ltrace`, `ptrace`, `strace`

Índice general

1. Introducción **6**

1.1. Objetivos 9

1.2. Estado del arte 10

1.2.1. `GDB` 11

1.2.2. `strace` 13

1.2.3. `electric-fence` 14

1.2.4. *Purify* 14

1.2.5. `libtrace` 15

2. Historia de `ltrace` **16**

2.1. Instalación de un sistema operativo 16

2.2. Instalación de *GNU/Linux* 17

2.3. Disquetes de rescate 18

2.4. `miniutils` 19

2.5. `minilibc` 20

2.6. Depuración de `minilibc` 21

2.7. El nacimiento de `ltrace` 22

3. Diseño **23**

3.1. Conceptos previos 23

3.2. Módulos del programa 28

3.2.1. Inicialización 29
3.2.2. Ejecución del programa 30
3.2.3. Autómata de estados 32
3.3. Ejecución de ficheros *ELF* 33
3.4. Control del ejecutable a trazar 40
3.4.1. Uso de una biblioteca adicional 40
3.4.2. Control del ejecutable mediante el uso de un programa externo 42
3.4.3. Ejecución de una llamada al sistema 43
3.4.4. Recepción de un *breakpoint* 44

4. Implementación 47
4.1. Elección del entorno de desarrollo 47
4.2. Estructura general 49
4.2.1. Ficheros distribuidos 51
4.2.2. Contenido de la distribución 51
4.2.3. Configuración y compilación del código fuente 53
4.2.4. Ficheros de Debian 53
4.3. Inicialización 54
4.3.1. `ltrace.c` 54
4.3.2. `options.c` 55
4.3.3. `read-config-file.c` 56
4.4. Ejecución 56
4.4.1. `proc.c` 57
4.4.2. `execute-program.c` 57
4.4.3. `elf.c` 58
4.5. Autómata 58
4.5.1. `wait-for-something.c` 59

4.5.2. `process-event.c` 60
4.6. Funciones comunes 61
4.6.1. `breakpoints.c` 62
4.6.2. `dict.c` 63
4.6.3. `debug.c` 65
4.6.4. `output.c` 65
4.6.5. `display-args.c` 66
4.6.6. `demangle.c` 66
4.6.7. `summary.c` 67
4.7. Parte dependiente del sistema operativo 67
4.7.1. Lista de funciones 68
4.7.2. Particularidades de *GNU/Linux* 70

5. Manual de usuario 71
5.1. Ejemplo de sesión con `ltrace` 71
5.2. Opciones de ejecución 73
5.3. Ficheros de configuración 81
5.4. Ejecución de programas 86
5.4.1. Modo de funcionamiento 87
5.4.2. Información mostrada a la salida 88

6. Conclusiones 92
6.1. Colaboradores 93
6.2. Líneas de trabajo futuras 93
6.2.1. Portabilidad 94
6.2.2. Eficiencia 95

Presupuesto 96

Apéndices 99

A. GNU GENERAL PUBLIC LICENSE 100

B. Glosario de términos 109

Bibliografía 118

Índice de figuras

1.1. La mascota de GDB 11

2.1. Tamaño aproximado ocupado por un “Hello, world” 18

3.1. Lista de módulos 28

3.2. Detalle de cada módulo 29

Capítulo 1

Introducción

Uno de los campos en los que más tiempo y dinero se invierte hoy en día es el del desarrollo de aplicaciones informáticas. Prácticamente todas las empresas relacionadas con las nuevas tecnologías tienen un departamento de desarrollo de *software* encargado de la creación o innovación de programas. Pero también en las empresas "tradicionales" tienen cabida este tipo de desarrollos, especialmente en las más grandes.

En la elaboración de cualquier programa informático han de seguirse fundamentalmente las siguientes fases:

1. Captura de requisitos
2. Diseño
3. Implementación
4. Depuración
5. Mantenimiento

Las dos últimas, contrariamente a lo que se suele pensar, son las más laboriosas y en las que más tiempo se invierte.

Los trabajos de Boehm [Boe81] en los años 80, corroborados más recientemente por nuevos experimentos y mediciones de Kaplan [Kap95], demuestran que el coste de la corrección de errores de programación aumenta de forma dramática cuanto más tarde son detectados.

Una solución para asegurar que los errores no aparecerán en estas fases tardías llega de la mano de la verificación formal de los programas que trata de probar matemáticamente la corrección de los mismos. Sin embargo, el incipiente estado de investigación en que se encuentra esta nueva rama de la programación hace que únicamente resulte aplicable a determinados programas, pequeños y sencillos.

Por lo tanto, a pesar de que siempre seguirán apareciendo errores por muchos controles de calidad que se apliquen, no es extraño que se dediquen esfuerzos al desarrollo de herramientas que al menos faciliten —y por tanto abaraten— el proceso de detección y corrección de los mismos, esto es: los depuradores.

Dentro de estas herramientas podemos englobar los depuradores al estilo de `gdb` [S$^+$00], que permiten tener un control exhaustivo del flujo de control del programa según se va ejecutando, así como del contenido de la memoria y de los registros en cada momento. Sin embargo, estos depuradores generalmente no permiten depurar los programas de los que no se disponga el código fuente.

Pero es necesario también incluir entre estas herramientas otras que nos proporcionan información sobre el flujo y desarrollo del programa al mismo tiempo que éste se ejecuta, y que normalmente no necesitan disponer del código fuente para llevar a cabo su cometido.

Dicha información es, habitualmente, más limitada que la que son capaces de proporcionarnos los depuradores tradicionales, pero a menudo resulta más que suficiente para hacerse una idea precisa de qué está fallando y su obtención se realiza de forma más rápida y sencilla. Nos estamos refiriendo, naturalmente, a herramientas del estilo de `strace` [Akk].

Se trata de una potente utilidad que muestra todas las llamadas al sistema que realiza un proceso según se va ejecutando, con los parámetros que pasa a estas llamadas y los valores de retorno que devuelve el núcleo. Tan solo con esa información resulta fácil, de un vistazo, ver si un programa no funciona porque está pasando un puntero no válido a una función o si es que está tratando de conectarse a una dirección IP equivocada, etc.

Sin embargo, `strace` no es suficiente debido a una limitación fundamental que, lamentablemente, es intrínseca a su diseño: únicamente muestra las llamadas al sistema.

La experiencia de muchos años de desarrollo y depuración de *software* nos revelaba como enormemente atractiva la posibilidad de que una herramienta muy similar a `strace` fuera capaz de mostrar, igualmente, las llamadas a funciones de biblioteca, donde muchas veces se localizan (y se podrían detectar así fácilmente) importantes errores. Basta para hacerse una idea de lo que decimos recordar

los no por viejos y conocidos menos cuantiosos problemas derivados del empleo descuidado de las funciones `malloc()` y `free()`, encargadas de reservar y liberar memoria, y que son —como es bien sabido— llamadas a funciones de una biblioteca (la `libc` [Fre01]) y no llamadas al sistema.

En este proyecto se estudiará, pues, la posibilidad de llevar a cabo este tipo de depurador tan poco habitual y del que no tenemos constancia que exista nada parecido hasta el momento: una herramienta que vaya mostrando qué es lo que hace un programa mientras éste se está ejecutando, incluyendo llamadas al sistema y llamadas a bibliotecas, todo ello sin necesidad de haberlo compilado de ninguna manera especial (ni tan siquiera de tener accesible el código fuente) y, naturalmente, sin que las manipulaciones que lleve a cabo la herramienta interfieran para nada en el funcionamiento normal del programa.

Acerca del nombre: `ltrace`

Todo proyecto de programación tiene un nombre para poder distinguirlo de los demás, y este no podía ser menos.

En la elaboración del proyecto se barajaron distintas posibilidades, pero teniendo en cuenta la función principal y el tipo de nombres que se suelen usar en este tipo de herramientas, se llegó a la conclusión de que hacía falta un nombre con las siguientes características:

- Sencillo, formado por una sola palabra, de entre 3 y 10 caracteres.
- Escrito en inglés, con la posibilidad de que sea la combinación de varias palabras, o siglas o alguna abreviatura.
- Fácil de recordar y a ser posible de pronunciar en varios idiomas.

Dada la función principal del programa (trazado de llamadas a funciones de biblioteca), lo más directo sería llamarlo “*library call tracer*” o alguna combinación de estas palabras, y teniendo en cuenta la similitud con “`strace`”, utilidad que responde a la descripción de “*system call tracer*”, se resolvió que el mejor nombre para esta herramienta fuera “`ltrace`”.

El “otro” `ltrace`

Desgraciadamente, algún tiempo después de que se decidiera el nombre del programa y cuando ya estaba distribuyéndose y empezando a ser usado en muchos

lugares del mundo, se descubrió que había otro programa con el mismo nombre (`ltrace`) [Sol],

Este otro programa, sin embargo, es distinto en finalidad y en funcionamiento a la herramienta descrita en este Proyecto Fin de Carrera, y teniendo en cuenta de que ya estaba muy extendido, se decidió que no merecía la pena realizar un cambio de nombre.

Este "otro" *LTRACE* es una aplicación de desarrollo para ayudar a la depuración de programas escritos en *lex* y en *yacc*. Usándolo se permite la ejecución de un programa *lex* con la particularidad de que se muestra por la salida estándar una sentencia que contiene el número de línea, las órdenes y los argumentos que se usan después de cada regla y antes de que se ejecuten las sentencias correspondientes a esa regla.

Hoy en día, sin embargo, el *LTRACE* del que hablamos en esta sección es muy poco conocido, y es este proyecto lo que se conoce por el nombre de "`ltrace`" casi en exclusiva, al menos entre los usuarios de variantes del sistema operativo *GNU/Linux*.

1.1. Objetivos

El principal uso que se le puede dar a un proyecto de esta envergadura es, evidentemente, su uso como herramienta de depuración. Con él se podrá tener un control exhaustivo de las tareas que va ejecutando cada programa que queramos investigar, de manera que cualquier comportamiento anómalo pueda ser detectado y corregido de manera satisfactoria.

Más concretamente, lo que se quiere conseguir con este proyecto es tener un programa capaz de ir mostrando lo que van realizando otros programas mientras estos se ejecutan.

En particular, se han de capturar y mostrar tres tipos de acciones realizadas por otros procesos:

- **Señales que reciben**

  Se ha de indicar el momento exacto en que un proceso recibe cada señal, el nombre de esta señal y si causa la terminación del programa.

- **Llamadas al sistema que realizan**

Es deseable que, en este aspecto, el comportamiento de `ltrace` sea similar al de `strace`, indicando cuándo se realiza cada llamada al sistema, el nombre de esta llamada, los argumentos recibidos por ésta y, por último, el valor de retorno de la llamada, esto es, lo que devuelve la función, indicando entre otras cosas si fue satisfactoria o no.

- **Llamadas a funciones de biblioteca dinámica**

  Este es el punto principal, la razón de ser del proyecto (ya que las dos acciones anteriores ya existen en otros proyectos similares como `strace`).

Además de estos objetivos, queremos lograr que el programa sea lo menos intrusivo que se pueda, y que dentro de lo posible los programas a estudiar se comporten exactamente de la misma manera si están siendo trazados que si no estuvieran siéndolo, para procurar tener una visión más objetiva de cómo se comportan en realidad.

1.2. Estado del arte

El campo de la depuración de programas en sistemas POSIX [Lew91] no es demasiado extenso. Pese a que existen algunos depuradores (DBX, notablemente en sistemas BSD; SDB en System V; XDB; ... [Gil92]) se podría decir que el único depurador, líder en el mercado y omnipresente independientemente del procesador, arquitectura o sistema operativo empleado es sin duda `gdb` [S+00], quedando los demás relegados a nichos cada vez más reducidos.

Se cuenta, también, con varios *front-end*s a `gdb` que ofrecen distintos niveles de velocidad, amigabilidad y facilidad de uso. Entre estos destacaremos de forma especial `ddd` [Fou00] y `xxgdb` [oNSCL]. Cygnus liberó hace unos años otro front-end, llamado `Insight` [SS].

En otro aspecto o campo de la depuración, no cubierto por la familia de editores tipo `gdb` anteriores, tenemos `strace` [Akk] o su equivalente en Solaris, `truss` [Mic98]. Este tipo de herramientas resulta bastante útil a la hora de indagar sobre lo que está haciendo un programa al mismo tiempo que se ejecuta, incluso sin necesidad de tener el código fuente. Si dijéramos que el paradigma de los depuradores que comentábamos en el párrafo anterior se puede definir en un "ver, parar y tocar", el de éstos que ahora comentamos se limita a un "ver las llamadas al sistema", que pese a resultar en principio un tanto lacónico, día a día se revela como de gran utilidad por la facilidad y rapidez con que se obtiene la información, suficiente en muchos casos para encontrar el error buscado.

Finalmente, merecen especial mención algunas bibliotecas dinámicas que sirven para ayudar al programador a escribir código libre de errores. Ejemplos de este tipo de herramientas son `electric-fence` y `purify`. El procedimiento habitual que se emplea aquí es sustituir las llamadas a biblioteca o al sistema más conflictivas de los programas por las que provee la biblioteca de depuración, que a la vez de realizar una función similar a la llamada sustituida llevan a cabo algún tipo de control, registro o prueba especial sobre los datos que se les pasa con el ánimo de detectar errores en estos.

Como es lógico, esas comprobaciones extra ralentizan el programa, pero en el momento en el que se cree que éste se encuentra ya libre de errores, se vuelven a utilizar las llamadas originales, bastando para ello con no enlazar la biblioteca de depuración con el resto del programa.

Seguidamente pasaremos a mostrar con mayor detalle este tipo de herramientas de forma individualizada, mencionando en cada caso las carencias que presentan y que el presente Proyecto Fin de Carrera pretende llenar.

1.2.1. GDB

`GDB` es la herramienta de depuración por excelencia. Fue creada por la *Free Software Foundation* [Fou] a finales de los años 80.

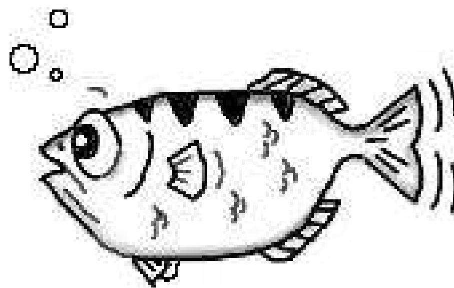


Figura 1.1: La mascota de GDB

`GDB` es el depurador del *Proyecto GNU* [GNU], y permite ver lo que ocurre en el “interior” de otro programa mientras se está ejecutando, o lo que un programa estaba haciendo en el momento en que dejó de funcionar como se esperaba.

Para extraer la mayor cantidad de información sobre ese “interior”, y con ello el esperado beneficio de descubrir la mayor cantidad de errores, el programa debe haber sido compilado con información de depuración apropiada para el depurador

que se va a utilizar. Normalmente, para añadir esa información extra basta con activar una opción en la línea de órdenes del compilador (en caso de utilizar el compilador `gcc`, esta opción es la “`-g`”).

`GDB` puede llevar a cabo cuatro tipos principales de funciones para ayudar en la detección de fallos de programación:

- Ejecutar un programa, especificando cualquier cosa que pueda afectar a su comportamiento.
- Hacer que el programa se pare cuando se cumplan determinadas condiciones.
- Examinar el programa parado, teniendo acceso a los registros, variables internas y estado de la memoria.
- Efectuar cambios en el programa que se ejecuta, sin necesidad de recompilarlo ni tan siquiera de ejecutarlo de nuevo, para poder experimentar en la corrección de fallos de programación.

Con `gdb` se pueden depurar programas escritos en C, C++, Pascal, Fortran, Modula-2, Java... y muchos otros lenguajes de programación. Estos programas pueden ejecutarse en la misma máquina en la que ejecutamos GDB (modo nativo) o en otra máquina (modo remoto). GDB funciona correctamente en la mayor parte de los sistemas UNIX y en algunas versiones de Microsoft Windows.

Este tipo de depuración es, ciertamente, el que mayor información permite obtener del programa, pero con el inconveniente de tener que compilarlo de una forma especial que añade información extra para la posterior depuración. Como inconveniente añadido, es posible además que esa información de depuración haga que el programa sólo sea legible con determinados depuradores, mientras que otros o bien no la entenderán o bien fallarán al encontrarse con ella.

Evidentemente, un programa compilado con información de depuración ocupa más espacio en disco, aunque al precio al que se encuentran ahora los discos duros esto no resulta especialmente preocupante.

Una consecuencia lógica de lo explicado hasta aquí es que depurar de esta manera programas de los que no se dispone de código fuente resulta muy difícil de realizar, y no solo porque no se puedan recompilar activando la información de depuración, sino también porque ésta hace referencia en muchos casos al código fuente y el depurador espera, por tanto, encontrarlo a la hora de interpretarla.

1.2.2. `strace`

`strace` es una utilidad que muestra las llamadas al sistema que va realizando otro proceso o programa mientras éste se va ejecutando, de manera transparente y sin necesidad de hacer nada especial con el programa a trazar: no se precisa tener acceso al código fuente, ni tampoco compilarlo de ninguna forma especial.

La ejecución del programa seguramente más simple dentro de un sistema UNIX, `true` (que no hace nada salvo terminar inmediatamente con un código de retorno satisfactorio), nos muestra éste resultado a través de `strace`, donde podemos hacernos una clara idea del tipo de información que nos ofrece: llamadas al sistema, parámetros que se le pasan, y valores de retorno:

```
execve("/bin/true", ["/bin/true"], [/* 31 vars */]) = 0
uname({sys="Linux", node="gizmo", ...}) = 0
brk(0)                                  = 0x804afc8
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=63314, ...}) = 0
old_mmap(NULL, 63314, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40012000
close(3)                                = 0
open("/lib/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\275Z\1"..., 1024) = 102
4
fstat64(3, {st_mode=S_IFREG|0755, st_size=1103880, ...}) = 0
old_mmap(NULL, 1113636, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40022000
mprotect(0x4012a000, 32292, PROT_NONE)  = 0
old_mmap(0x4012a000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x10
7000) = 0x4012a000
old_mmap(0x40130000, 7716, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONY
MOUS, -1, 0) = 0x40130000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
x40132000
munmap(0x40012000, 63314)               = 0
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=519856, ...}) = 0
mmap2(NULL, 519856, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40133000
close(3)                                = 0
brk(0)                                  = 0x804afc8
brk(0x804bfc8)                          = 0x804bfc8
brk(0)                                  = 0x804bfc8
brk(0x804c000)                          = 0x804c000
exit_group(0)                           = ?
```

Lógicamente, para poder llevar a cabo estas funciones es necesaria la ayuda del núcleo, que mediante una llamada al sistema especial, presente en casi todos los sistemas POSIX, se encarga de recabar la información que se muestra sobre el proceso.

El mayor inconveniente de `strace` es evidente: se limita a mostrar las llamadas al sistema, con lo que si los errores se encuentran en alguna otra parte del programa no se detectarán o, en general, aunque se propaguen hasta reflejarse en una llamada posterior, habrá que rastrear hacia atrás para encontrar el lugar exacto en el que se produjo el error.

1.2.3. `electric-fence`

Un ejemplo clarísimo para cualquier programador que ilustra a la perfección lo dicho en el párrafo que precede a esta sección es el tipo de errores que se encuentran día a día con las funciones de reserva y liberación de memoria, `malloc()` y `free()`, que no son llamadas al sistema sino a la biblioteca estándar de C, la *libc*.

Con estos errores, `strace` sirve de escasa o nula ayuda, ya que no proporciona ninguna información de ellas, y rastrear los errores hasta la llamada al sistema a la que finalmente recurren estas funciones al más bajo nivel, `brk()`, resulta en general más complicado y costoso que recurrir a otros métodos, quizás un tanto "pedestres" pero más eficaces, como la depuración con `printf()` o la simple inspección detallada del código.

Es aquí donde `electric-fence` y otras utilidades similares encuentran su campo: suelen ser bibliotecas que se enlazan con el programa sustituyendo alguna función, en este caso las `malloc()` y `free()` de la libc, por otras propias que ayudan a diagnosticar accesos a zonas no reservadas previamente o ya liberadas.

¿Qué inconveniente presentan? Uno importante: hay que escribir nuevas funciones para todas y cada una de las llamadas que se desee inspeccionar.

Si se deseara ver todas y cada una de las que realiza un programa a cualquier función de biblioteca, la tarea se volvería, sencillamente, imposible, ya que existen infinitas bibliotecas con las que un programa puede enlazar.

1.2.4. *Purify*

Rational Purify es un producto comercial de *IBM*, que ha sido durante años el estándar en la detección de errores para los *UNIX* distribuidos por *Sun* (*Solaris*) y *HP* (*HP-UX*).

Actualmente hay versiones de *Rational Purify* para todas estas plataformas, y además para *SGI* y para *Microsoft Windows*.

En la versión para *UNIX*, permite identificar errores de memoria en las aplicaciones, al igual que `electric-fence`, sin ser necesario el tener que recompilar el programa y de una manera muy intuitiva y fácil de usar.

En la versión para *Windows*, permite detectar errores de acceso a memoria en *Visual C/C++* y muchos otros aspectos relacionados con el acceso a memoria y el *garbage collection* en *Java*, *VB.NET* y en *C#*.

1.2.5. `libtrace`

Entre los años 1996 y 1997, Roger Espel Llima escribió un programa al que llamó `libtrace` con la finalidad de hacer algo similar a lo que se consigue con este proyecto, esto es, mostrar las llamadas a las funciones de la `libc` que realiza un programa mientras este se está ejecutando.

El funcionamiento, sin embargo, es bastante distinto; `libtrace` es menos intrusivo en los programas, no necesita controlarlos desde el núcleo sino que utiliza otra biblioteca distinta en la que se redefinen las funciones que queremos mostrar, y esta nueva biblioteca se interpone entre el programa a estudiar y la biblioteca real, para indicar el nombre de las funciones que se utilizan y sus argumentos y a continuación llamar a la función deseada de la `libc`. A cambio de esto se utilizan técnicas especiales del enlazador dinámico estándar de la `libc`: el uso de la variable de entorno `LD_PRELOAD`, lo cual es algo más frágil y más propenso a errores que lo que se pretende realizar en este proyecto.

Capítulo 2

Historia de `ltrace`

Este proyecto no comenzó a desarrollarse con un fin determinado, sino que como casi todos en el mundo del *software libre*, fue evolucionando poco a poco hasta llegar a lo que es hoy. En este capítulo analizaremos la historia del proyecto, desde sus comienzos hasta lo que es hoy en día.

2.1. Instalación de un sistema operativo

La instalación de un Sistema Operativo comienza siempre con el arranque, y ese arranque puede hacerse de tres maneras diferentes, dependiendo de los medios de que se dispongan en cada caso:

- **Arranque de disquete**

  Es la opción más tradicional, la manera en la que se iniciaba casi siempre un proceso de instalación de un sistema operativo hasta hace pocos años.

- **Arranque de CD**

  Los *Compact Disc* de datos ofrecen una posibilidad muy atractiva en este aspecto, y es la posibilidad de que un ordenador pueda arrancar directamente desde ellos, cargando desde *CD* el sistema operativo.

  En el caso de los *PC*, que son con diferencia los ordenadores personales más usados, los *CD*s con soporte de autoarranque son relativamente modernos y esta opción solo la utilizan sistemas operativos de los últimos años.

- **Arranque de red**

Consiste en hacer uso de las ventajas que proporcionan algunas arquitecturas y, en general, algunos componentes de *hardware* para posibilitar que el programa de arranque del sistema se encuentre en una máquina remota y se traiga por medio de algún protocolo de red (`TFTP` es el más común).

Este tipo de arranque es muy habitual en máquinas *Sun* y algunas *Alpha*, y en menor medida en *PC*s con tarjetas de red que permitan este tipo de arranque.

2.2. Instalación de *GNU/Linux*

A mediados de 1996 empezó a extenderse el uso de *GNU/Linux* en la Universidad; en aquellos tiempos, la oferta de distribuciones y de aplicaciones era mucho más reducida de lo que es ahora, y el proceso más laborioso a la hora de usar este sistema operativo era sin duda la instalación.

Hoy en día hay muchas maneras de comenzar la instalación de un sistema operativo, incluyendo las tres más habituales que se han enumerado en la sección anterior, pero hace unos años casi la única alternativa viable era la instalación a partir de disquetes de arranque que inicializaban el sistema y se preparaban para instalar el resto desde la red, desde CD o desde algún otro medio.

El uso de disquetes siempre ha conllevado un gran inconveniente, y es el hecho de que éstos son intrínsecamente poco fiables, muy propensos a fallos, y si combinamos esto con la poca capacidad que tienen (comparado con un disco duro o con un CD), es fácil llegar a la conclusión de que lo ideal es usar el mínimo número de disquetes imprescindible para llevar a cabo cada labor.

Las distintas distribuciones de *GNU/Linux* han ido evolucionando en el número de disquetes necesarios para iniciar la instalación, pero su número siempre ha estado aproximadamente entre los 2 y los 5.

Para intentar mejorar el proceso de instalación de una distribución en concreto (*Debian GNU/Linux*), en 1996 empezamos a trabajar en la posibilidad de modificar el contenido de los disquetes de arranque para intentar que la instalación se pudiese llevar a cabo usando tan solo uno en lugar de los cinco habituales.

Para ello, se empezó intentando conseguir un núcleo más pequeño pero con soporte para la mayor parte de los dispositivos habituales, y cuando se hubo conseguido un núcleo adecuado, se siguió trabajando con las utilidades de los disquetes de instalación, para intentar dejar solamente lo imprescindible e intentar que lo necesario ocupara lo mínimo posible.

2.3. Disquetes de rescate

En la mayor parte de las distribuciones de la época (y aún hoy), los disquetes de instalación tenían un doble propósito: además de para poder instalar el sistema, sirven como *disquetes de rescate*, es decir, una manera de poder tener acceso al sistema y a unas cuantas aplicaciones básicas con la finalidad de realizar reparaciones, en caso de resultar imposible por los medios tradicionales.

En un disquete de rescate debe incluirse el núcleo del sistema operativo, junto con tantas herramientas de administración como sea posible, para permitir configurar el acceso a los discos, a la red, la reparación de ficheros dañados, etc.

El proyecto, que se inició como el intento de tener un arranque para un proceso de instalación con un solo disquete, pasó en este punto a ser la elaboración de un disquete de rescate en el que tuvieran cabida tantos programas como fuera posible.

El principal problema con el que nos encontramos aquí es que un disquete tiene muy poco espacio disponible, al menos comparándolo con el tamaño que suelen tener las aplicaciones.

Como ejemplo, un disquete que tuviera simplemente el núcleo y un programa que mostrara por pantalla el ya famoso "`Hello, world`", construido con las herramientas habituales (no optimizadas en espacio), ocuparía aproximadamente lo indicado en la Figura 2.1.

Núcleo (`linux-2.4.20`)	700K
Biblioteca estándar (`libc-2.3.1`)	1200K
Ejecutable	4K
TOTAL	**1904K**

Figura 2.1: Tamaño aproximado ocupado por un "Hello, world"

Esto quiere decir que el mínimo tamaño necesario para un disquete que contenga lo necesario para ejecutar el núcleo de *Linux* y de imprimir por pantalla el mensaje "`Hello, world`!" es de más de 1900K, más de un 30 % superior al tamaño máximo de un disquete estándar (1440K).

Por lo tanto, es evidente que no se puede simplemente ir copiando tantas aplicaciones y bibliotecas como consideremos necesarias, ya que el tamaño ocupado se dispara con facilidad.

La parte que más ocupa es sin duda la biblioteca estándar, pero afortunadamente hay maneras de reducir su tamaño considerablemente, incluyendo solo las funciones que vayan a ser usadas por los programas del disquete; esto se puede

conseguir de manera casi automática con el paquete "`libc6-pic`", incluido en *Debian GNU/Linux*.

Otra posibilidad es incluir programas compilados *estáticamente* en lugar de dinámicamente, con lo que no es necesario incluir una copia completa de las bibliotecas usadas sino que cada programa incluye dentro de su código las funciones que necesite.

De esta manera se logran reducir el uso en varios cientos de K's, lo que da bastante margen para incluir muchas utilidades en el disquete.

2.4. miniutils

Entre la lista de programas que suele haber en un disquete de arranque, tenemos herramientas para la configuración de la red, para la creación y reparado de particiones, editores, etc; pero aparte de todas estas, hay muchas otras en las que no solemos pensar por estar demasiado acostumbrados a tenerlas, y son las más habituales de usar en una *shell*; es lo que el proyecto *GNU* llama las "*GNU coreutils*", e incluyen programas como el "`ls`", "`cat`", "`cp`", "`chmod`", y un largo etcétera.

Estos programas son cada uno bastante sencillo, y no ocupan demasiado (entre unos 10 y 40 K cada uno), pero su tamaño podría reducir considerablemente si se reescribieran pensando en conseguir programas simples, sin muchas funcionalidades, y reutilizando la mayor parte del código posible para no tener funciones repetidas que se usen en varios programas. De esa manera, se podría lograr tener muchos más programas en un disquete de arranque.

Por lo tanto, este fue el giro natural que dio el proyecto en este punto: escribir un programa con la funcionalidad de muchos otros con la finalidad de que el tamaño total sea mucho más pequeño.

Este subproyecto se llamó en un principio "`joker`", aunque pronto cambió de nombre pasando a llamarse "`miniutils`", y siguió evolucionando de manera independiente.

En 1998 se logró tener un ejecutable que en poco más de 50K incluía la funcionalidad de estos 40 programas:

```
cat      chmod    cmp      connect  cp       df
false    fdisk    ftpget   ftpput   grep     gunzip
halt     help     http     hwaddr   ifconfig kill
```

```
ln        ls        lsmod     mkdir     mkswap    mount
mv        ping      reboot    repart    reset     rm
rmdir     route     sleep     swapoff   swapon    sync
touch     true      umount    xtar
```

Como comparación, baste decir que si miramos el original de cada uno de esos programas de manera independiente y sumamos su tamaño, ocupan 1656K (frente a 50K), lo que significa un ahorro en el tamaño de un 97 %.

2.5. minilibc

El siguiente paso lógico después de haber reducido el tamaño de los programas consiste en hacer algo al respecto con el tamaño de las librerías de que estos dependen, ya que al fin y al cabo esto es lo que más espacio suele ocupar en un disquete, y dado que es un código que utilizan todos los programas, hay que trabajar en conseguir un código eficiente, limpio y pequeño.

El resultado es algo llamado “minilibc”, que es un sustituto de la mayor parte de las funciones de la biblioteca estándar, la *GNU libc*, pero no solo eso, sino que además se incluye un programa que actúa como compilador, llamando de manera adecuada al gcc y que permite usar directamente esta nueva biblioteca, sin tener que incluir ninguna directiva en los programas ni en la manera de llamar al compilador.

Desgraciadamente, la tarea de reescribir una biblioteca es mucho más ardua que la de reescribir unas cuantas utilidades, ya que el número de funciones que incluye una biblioteca estándar es muy elevado (por encima de 2000 en la glibc-2.3.1).

Los objetivos principales a la hora de elaborar esta minilibc fueron:

- Minimizar en tamaño
- Uso eficiente de la memoria en tiempo de ejecución
- Escritura de solo las funciones necesarias para los programas que se vayan a utilizar
- Funcionalidad mínima exigible para que estos programas funcionen

Con estos requisitos en mente se elaboró una biblioteca, con sus dos versiones (estática y dinámica), usando el cargador dinámico de bibliotecas de la propia `glibc`.

Curiosamente, y a diferencia de lo que pudiera resultar más natural, los programas a incluir en el disquete de arranque ocupaban menos si se compilaban todos estáticamente que si se compilaban dinámicamente y se incluía la biblioteca dinámica y el cargador dinámico.

Con todo esto, el tamaño total de un disquete de rescate perfectamente utilizable en la mayor parte de los casos se redujo drásticamente; el tamaño total de las `miniutils` del apartado anterior se incrementó en solo 10K al incluir las funciones de biblioteca.

2.6. Depuración de `minilibc`

La elaboración de una biblioteca dinámica es una tarea complicada, no tanto por lo que supone escribir las funciones en sí (muchas de ellas son triviales), sino lo que se refiere a la integración, la construcción de los objetos, y, sobre todo, la depuración del código para asegurarse de que siempre funciona como se espera.

A finales de 1996 se comenzó a incluir en el código de `minilibc` soporte para permitir el depurado de las funciones, consistente en modificar de determinada manera la implementación de cada función a ser depurada para que mostrara en un fichero, cada vez que era llamada, el nombre de dicha función, la lista de los argumentos que se le pasan, y el valor de retorno.

Este depurado fue sofisticándose y su uso en el código se fue simplificando cada vez más hasta llegar a un punto, en marzo de 1997, en el que tan solo era necesario añadir una línea determinada a cada función para activar la depuración en esta.

Sin embargo, aún eso se podía superar; al menos en teoría, es posible mostrar información de depurado de cada llamada a cada función sin necesidad de modificar las funciones, sino interceptando de alguna manera la llamada a éstas.

Este es en realidad el comienzo de `ltrace`, con la funcionalidad que tiene hoy en día.

2.7. El nacimiento de `ltrace`

`ltrace` comenzó su existencia el 26 de marzo de 1997, momento en el que se vio que haciendo uso de una biblioteca dinámica adicional para modificar determinadas partes de un programa, se podría tener acceso a todos los puntos en los que llamara a cualquier función. El funcionamiento era el siguiente:

- `ltrace` examina el ejecutable, descubriendo los puntos en los que se hacen llamadas a funciones de biblioteca y tomando nota de ellos
- Se ejecuta el programa, añadiendo una biblioteca dinámica a las que carga automáticamente por medio del uso de `LD_PRELOAD`
- Esta biblioteca dinámica toma el control nada más comenzar a ejecutarse el programa a trazar, y cambia el código de comienzo de cada una de las funciones a las que llama el programa para que antes de nada ejecuten las rutinas de depuración que permiten mostrar las llamadas a las funciones cuando estas se van realizando

Para hacer que funcionara `ltrace` de esta manera, había que conseguir que la biblioteca dinámica nueva, extra, tomara el control antes de que el programa hiciera ninguna llamada a otras funciones. Esto se logró en un principio exportando la función `__setfpucw()`, que es llamada al principio del programa por todos los ejecutables enlazados dinámicamente con la biblioteca `libc5` de *GNU/Linux*, y más adelante se cambió utilizando código en la sección "`.ctors`" de la biblioteca, que es más portable y no depende del uso de dicha función.

Por otra parte, había otras razones que aconsejaban intentar conseguir el mismo efecto de una manera distinta, sin utilizar una biblioteca dinámica que tomara el control; la más importante de ellas era que al estar ejecutando código de `ltrace` en una biblioteca enlazada dinámicamente, ese código lo ejecutaba el mismo proceso a trazar, y había que tener mucho cuidado para que el resultado se comportara de la misma manera antes y después de trazarlo.

Para intentar minimizar el intrusismo del código de `ltrace` en los programas, en junio de 1997 se optó por otra manera de tener acceso a las llamadas que realiza un ejecutable, que es el que se usa desde entonces y aún hoy: usando otro proceso que tiene el control del proceso a trazar, y que para su ejecución en determinados puntos utilizando *breakpoints*, al igual que hacen todos los depuradores del mercado.

Capítulo 3

Diseño

En este capítulo mostraremos los factores estudiados para escribir el programa, teniendo en cuenta las distintas alternativas que se presentaban y la solución adoptada para cada una de las partes de que consta.

3.1. Conceptos previos

A lo largo de las siguientes secciones se utilizarán una serie de términos acerca de los cuales es conveniente dar unas definiciones y descripciones previas, para facilitar la lectura y compresión de aquí en adelante.

Sistema Operativo

El sistema operativo es el conjunto de programas básicos y utilidades que hacen que pueda funcionar un ordenador. Hay varios tipos de sistemas operativos, distribuidos por diferentes compañías, y es habitual que estén incluidos en el precio de un ordenador nuevo.

Un sistema operativo típico incluye la infraestructura necesaria para hacer uso eficiente de los recursos del sistema, ofrece la posibilidad de ejecutar aplicaciones externas e incluye una serie de aplicaciones básicas para manejo del sistema, tales como editores, compiladores, juegos, etc.

Ejemplos de sistemas operativos de uso común son:

- Debian GNU/Linux

- RedHat Linux
- Microsoft Windows

Núcleo

El *núcleo* de un sistema operativo es probablemente la parte más importante de éste; es un programa que se ejecuta en modo privilegiado y con acceso a todo el sistema; se encarga de asignar adecuadamente los recursos necesarios a cada proceso, gestiona el acceso al *hardware* y la comunicación entre procesos.

El *núcleo*, además, proporciona una capa de abstracción para que cada proceso no necesite conocer los detalles de cada máquina en las que pueda ejecutarse, de tal manera que, por ejemplo, los programas solo tienen que saber cómo decirle al *núcleo* que abra un fichero, en lugar de saber los detalles de cada sistema de ficheros en los que pueda estar almacenado.

Programa

Un *programa* es una secuencia de instrucciones, almacenadas en cualquier medio, que pueden ser interpretadas y ejecutadas por un ordenador.

Este término se usa tanto para referirse a un programa escrito (un documento, escrito en algún lenguaje de programación) como a su correspondiente versión electrónica que se almacena o ejecuta en un ordenador.

Un programa puede ser cualquier aplicación, herramienta o utilidad, como un juego, una aplicación que hace uso de una base de datos o una calculadora.

Proceso

Cada uno de los estados por los que pasa un programa mientras se está ejecutando.

Un proceso se compone de varias partes:

- Código del programa (parte de la memoria que contiene las instrucciones que lee el procesador), que puede ser compartido por otros procesos que estén ejecutando el mismo programa
- Datos privados de la ejecución del programa

- Estado del procesador, en particular el contenido de los registros

Además, puede tener asociados otros, datos como el identificador del proceso (*PID*), la lista de ficheros abiertos, los límites de uso de recursos asignados por el sistema operativo, lista de los procesos hijo, los manejadores de las señales...

En algunas plataformas, un proceso puede tener varias *threads*, o hilos de ejecución independientes.

Un sistema operativo multitarea puede ejecutar varios procesos concurrentemente, o en "paralelo", y permite a los procesos crear otros procesos, a los que se les llama "hijos".

Función

Al igual que su homólogo en matemáticas, una función consisten en la ejecución de un algoritmo que requiere una serie de argumentos y devuelve un valor.

Sin embargo, a diferencia de en las funciones matemáticas, en este caso una función puede devolver valores diferentes cada vez que se la llame, aunque se llame con los mismos argumentos, y puede tener efectos laterales, realizar labores diferentes de la simple manipulación de los argumentos para obtener un resultado.

Un *procedimiento* es una función que no devuelve ningún valor, sino que solo tiene efectos laterales. Sin embargo, en muchos lenguajes de programación, como por ejemplo en *C*, no existen procedimientos, todo son funciones, algunas de ellas devuelven un valor de un tipo especial, "`void`", que indica que esa función en concreto no devuelve ningún resultado.

Biblioteca

Una biblioteca es un conjunto de funciones y datos almacenados en uno o más ficheros, en forma compilada, para ser enlazados por otros programas.

Las bibliotecas son una de las maneras más habituales de reutilización del código. Normalmente, el sistema operativo o el compilador proporciona una serie de bibliotecas estándar con funciones básicas, como son las de entrada y salida de datos, operaciones matemáticas, de cadenas, etc; pero también puede haber bibliotecas diseñadas con una función específica, como la animación de figuras en tres dimensiones.

Hoy en día se utilizan funciones de biblioteca prácticamente para todo; es posible que al menos el 90 % de las funciones llamadas en un programa cualquiera pertenezcan a una biblioteca (del sistema o no). Las funciones de bibliotecas del sistema se encargan de todos los trabajos que suponen cálculos o comunicación con el "exterior", tales como entrada/salida por pantalla, en disco, o comunicación entre procesos o entre máquinas.

Biblioteca estática

Una biblioteca estática es aquella especialmente diseñada para que las funciones de la misma que utilice un programa sean incluidas en éste en el momento de compilarlo, de manera que el programa resultante incluye parte de la biblioteca y no es necesaria ninguna otra parte a la hora de ejecutarlo.

Biblioteca dinámica

La manera más habitual de usar bibliotecas en un programa no es incluyendo todo el código necesario dentro de cada programa, sino hacer que el código compartido forme parte de uno o varios ficheros independientes que puedan ser reutilizados por varios programas. Estos ficheros con el código compartido son lo que se conocen como una *biblioteca dinámica*.

Señal

Una señal es un mensaje enviado entre dos procesos o entre el núcleo del Sistema Operativo y un proceso. Las señales se usan para indicar la existencia de un evento externo no esperado, como la terminación forzada de un proceso por parte de un usuario, usando la orden "`kill`". A cada señal le corresponde un número único asociado con ella, y cada proceso puede comportarse de manera diferente con la llegada de cada señal: pueden ignorarla, o ejecutar una función para tomar las medidas apropiadas.

Llamada al sistema

Siempre que un proceso necesite interactuar con otros procesos, o con el *hardware* del sistema, necesita hacer partícipe de ese deseo al núcleo del sistema operativo para que sea éste el que realice las gestiones necesarias; esto se hace mediante

las llamadas al sistema.

Las llamadas al sistema son la manera que tienen los procesos para poder comunicarse con el núcleo.

En la mayor parte de las arquitecturas esto no es simplemente una manera para hacer que los procesos no tengan que preocuparse de los detalles de acceder a determinados recursos, sino una obligación, ya que el procesador impide físicamente que determinados procesos puedan acceder directamente al *hardware*.

En *Linux*, por ejemplo, existen unas 250 llamadas al sistema, con funciones tan variadas como el control de ficheros, de la memoria, de los procesos, acceso a todos los dispositivos, etc.

Normalmente, los procesos no hacen uso directamente de llamadas al sistema (el proceso de realizarlas suele ser bastante dependiente tanto del núcleo como de la arquitectura en la que nos encontremos), sino que ejecutan funciones de bibliotecas estándar proporcionadas por el sistema operativo, y son estas funciones las que se encargan de la tarea de comunicarse con el sistema.

Formato de ejecutables

Para que un programa empiece a ejecutarse (y pase, por tanto, a ser un proceso en memoria), este programa tiene que estar almacenado en el disco de una manera que el sistema operativo sea capaz de entender, para poder cargarlo en memoria y comenzar su ejecución.

Y para ello hay varios estándares de maneras de almacenar esta imagen en memoria, dependiendo del sistema operativo que tengamos. Los más habituales son "`.com`", "`.exe`" (típicos en sistemas MS-DOS y *Microsoft Windows*), "`a.out`" (el clásico en sistemas *UNIX*) y "`ELF`" (el más usado hoy en día en casi todos los sistemas).

ELF

ELF significa "*Executable and Linkable Format*", y es un formato para almacenar objetos binarios, tanto ejecutables como bibliotecas o ficheros intermedios de una compilación.

El estándar *ELF* es el más seguido por la mayor parte de los sistemas operativos modernos; en concreto, es seguido por los basados en los núcleos de *Linux*, que son el objeto de nuestro estudio.

3.2. Módulos del programa

En la elaboración del diseño de un proyecto de *software* es importante definir claramente las partes de las que consta, así como la manera exacta en la que estas partes han de funcionar para que el resultado se comporte como se espera.

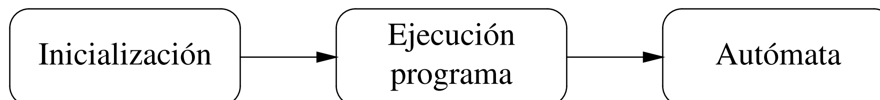


Figura 3.1: Lista de módulos

A muy grandes rasgos, la figura 3.1 muestra los tres grandes módulos en los que se divide este proyecto, que son:

- **Inicialización**

  Comienzo del programa, inicialización de variables, lectura de las opciones y de los ficheros de configuración.

- **Ejecución del programa**

  Inspección del programa a trazar, extracción de la lista de las funciones que utiliza y preparación de éste para su ejecución controlada.

- **Autómata de estados**

  Este es el corazón del proyecto. Consiste en un bucle que espera hasta que suceda un evento relacionado con el proceso o procesos trazados que tenga que ser contemplado; estos eventos pueden ser, por ejemplo, la recepción de una señal, la llamada a una función o la ejecución de una llamada al sistema.

El flujo de control entre cada uno de los tres módulos es secuencial: cada vez que se ejecuta `ltrace`, se pasa primero por el módulo de inicio, para continuar en la ejecución y por último toma el control el autómata.

La figura 3.2 muestra estos módulos con detalle, indicando su estructura interior, donde se puede ver que cada módulo está dividido en varios submódulos y las relaciones entre ellos.

Además de los tres módulos principales, el programa tiene que comunicarse con el exterior, y esto lo hace recibiendo información (datos de entrada) de los ficheros de configuración, las opciones con las que se ejecuta y el programa o programas que queremos trazar, y utiliza otro módulo para mostrar los mensajes de salida.

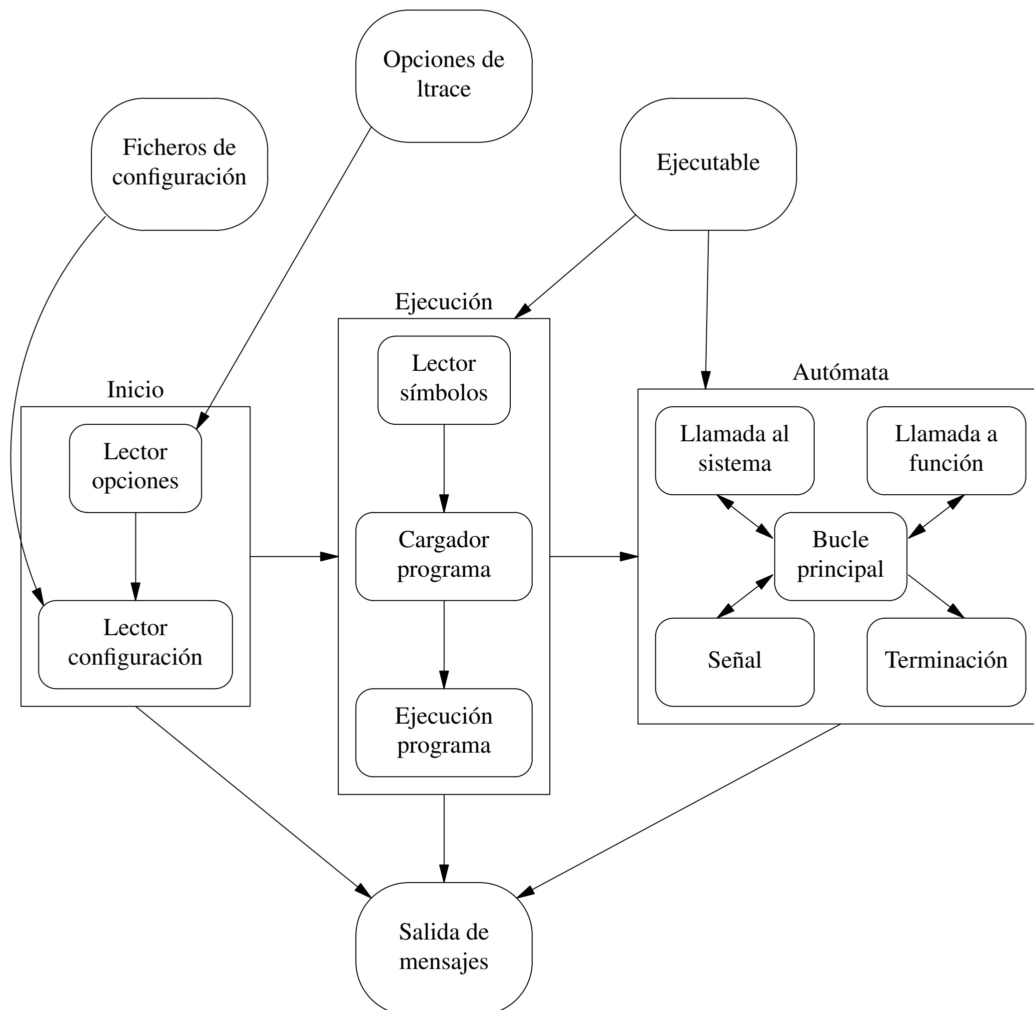


Figura 3.2: Detalle de cada módulo

3.2.1. Inicialización

Este módulo contiene las rutinas comunes que se han de ejecutar siempre, cada vez que se inicia el programa.

Durante la inicialización se llevan a cabo las siguientes tareas:

- Preparación del programa en caso de interrupción
- Inicialización del módulo de salida de mensajes
- Proceso de las opciones (apartado 3.2.1)

- Proceso de los ficheros de configuración (apartado 3.2.1)

Proceso de las opciones

De la línea de órdenes se han de interpretar las distintas opciones que afectan a diversos aspectos del funcionamiento del programa.

Las opciones son muchas y no tienen cabida en este apartado, ya que están todas detalladas en el manual de usuario (capítulo 5.2, página 73).

Proceso de los ficheros de configuración

Los ficheros de configuración de `ltrace` tienen un único propósito: decidir de qué manera se mostrarán los argumentos de las llamadas a funciones ejecutadas por los programas a trazar.

Para ello, es necesario saber para cada función el número de argumentos que requiere, el tipo de cada argumento (esto es, la manera en la que deben mostrarse) y el tipo del valor de retorno.

En la elaboración del proyecto se sopesaron diferentes maneras de especificar esta información en los ficheros de configuración, hasta optarse finalmente por la que hay actualmente y está descrita en el capítulo 5.3, página 81 del manual de usuario.

3.2.2. Ejecución del programa

Una vez que se ha inicializado todas las partes comunes del código, hace falta preparar la ejecución del programa que hemos de trazar. Para ello, hay que realizar tres tareas fundamentales: averiguar las funciones que hemos de trazar, preparar el programa para trazar dichas funciones, y comenzar a ejecutarlo de manera controlada.

Lector de símbolos

La lista de funciones de biblioteca dinámica que pueden ser ejecutadas desde un programa en concreto se encuentran dentro del código de dicho programa (esto es, forman parte del ejecutable) y se puede extraer de diversas maneras; las más

habituales son utilizando algún programa especialmente escrito para mostrar dicha información, como pueden ser:

- `nm`

  Muestra los símbolos de que consta un programa, tanto etiquetas globales como locales y tanto definidos (esto es, que forman parte del código del programa) como no definidos, que son las funciones y variables externas que forman parte de una biblioteca dinámica y tienen que ser enlazadas en el momento de la ejecución para que el programa pueda funcionar correctamente.

- `objdump`

  Esta herramienta está diseñada para mostrar toda la información que se puede extraer de un objeto (entendiendo como objeto un módulo compilado, un ejecutable, una biblioteca o un fichero *core*). Por lo tanto, `objdump` puede no solo mostrar los símbolos contenidos dentro de un objeto, sino también las cabeceras del programa, las secciones en las que se divide, la información de depurado y de reubicación, etc.

En este proyecto en concreto, no se va a usar ninguna de las herramientas anteriormente citadas para extraer la lista de símbolos, sino que se accederá directamente al ejecutable desde el programa para averiguar la información necesaria. Hay dos maneras de hacer esto:

- Escribiendo el código necesario para extraer dicha información directamente, esto es, incluyendo dentro del proyecto todo lo necesario para que dicha información se encuentre dentro de los programas a trazar.
- Utilizando una biblioteca externa que nos proporcione dicha información, tal como `GNU libbfd` [Cha91].

Hemos optado por usar la primera opción, esto es, escribir directamente el código necesario en lugar de utilizar una biblioteca externa, porque se cubrían perfectamente nuestras necesidades de una manera más sencilla, ocupando el programa resultante menos espacio y ejecutándose de manera más deprisa.

Como inconveniente cabe destacar que de la manera actual solo hay soporte para un tipo de ficheros, concretamente el estándar *ELF* [Com95]. Sin embargo, en la práctica esto no es un gran problema ya que el formato *ELF* es utilizado en la práctica totalidad de los sistemas *POSIX* del mercado.

Cargador de programa
Ejecución del programa

Estas dos partes podemos englobarlas aquí en una sola, ya que están muy relacionadas: consisten en cargar el programa a trazar en memoria, prepararlo para interceptar las llamadas cuando se realicen, y darle control al mismo.

La técnica a usar para interceptar las llamadas a funciones requiere una explicación detallada de cómo funciona el proceso de ejecución y enlazado de ejecutables en sistemas *ELF*, y se verá en las secciones 3.3 y 3.4.

3.2.3. Autómata de estados

Una vez que estamos ejecutando el programa (o programas) a trazar, el flujo de control de `ltrace` es sencillo: consiste en esperar a que se reciba algún evento y actuar en consecuencia.

La lista de eventos que se pueden recibir de un programa trazado son las siguientes:

- **Recepción de señal**

  El programa ha recibido una señal; hemos de averiguar qué señal ha sido, mostrar por la salida su nombre y a continuación hacer que el programa continúe con normalidad.

- **Llamada al sistema**

  Si está activada la opción para trazar llamadas al sistema, mostrarla por la salida.

  Si es una llamada que cambia la imagen del proceso en memoria (tal como `execve()`, hay que volver a leer la tabla de símbolos y preparar la intercepción de llamadas.

  Si se trata de una llamada que crea un proceso nuevo (`fork()`, `clone()`, etc), tenemos que asegurarnos de que pueda ejecutarse normalmente, y si está activada la opción correspondiente, comenzar a trazar también al proceso hijo.

- **Terminación de proceso**

  Se muestra por pantalla una indicación de que el proceso ha terminado y su causa. Si es no quedan más procesos por trazar, se termina el programa.

- **Comienzo o fin de la ejecución de una función trazada**

  En caso de que no esté desactivado por medio de las opciones de `ltrace`, se ha de mostrar por la salida la información del nombre de la función invocada, junto con los posibles argumentos, y los valores de retorno cuando esta función termine.

3.3. Ejecución de ficheros *ELF*

Para poder entender la forma en la que `ltrace` detecta y muestra las llamadas a funciones de biblioteca que realiza un programa es preciso explicar antes los detalles concretos bajo los que se realizan dichas llamadas, puesto que sólo así se comprenderán las sutiles modificaciones que introduce `ltrace` en el proceso a fin de obtener toda la información de las mismas.

En primer lugar hay que advertir que solamente consideraremos el caso de los ejecutables enlazados dinámicamente y en formato *ELF32*. Únicamente este formato es soportado por `ltrace`, aunque esta no es una limitación tan estricta como parece ya que este es el sistema usado por la gran mayoría de los sistemas *POSIX* de hoy en día. Las arquitecturas más modernas, sin embargo, utilizan otra variante de *ELF*: el formato *ELF64* que es muy similar al *ELF32* y no sería difícil adaptar el programa para que lo soportara; sin embargo, si no se ha hecho es porque hasta el día de hoy no hay soporte para ninguna de las arquitecturas en las que se utiliza ese nuevo formato de ejecutables.

Un fichero *ELF* incluye una cabecera con una serie de datos, entre los cuales hay unos campos en los que se identifica el tipo de fichero, la arquitectura en la que funciona, el tamaño de las palabras (32 ó 64 bits, que es lo que distingue a *ELF32* de *ELF64*), etc. En concreto, un fichero *ELF* puede ser de cuatro tipos:

- Fichero ejecutable
- Fichero reubicable (código objeto resultado de una compilación)
- Objeto compartido (biblioteca dinámica)
- Imagen de la memoria de un proceso ("*core*")

Aquí nos interesaremos por los ficheros ejecutables.

Hay dos tipos de ejecutables: los compilados *estáticamente* y los compilados *dinámicamente*. Estos últimos no pueden ejecutarse directamente sin ayuda de

otros ficheros en el sistema: necesitan una serie de *bibliotecas dinámicas* para poder funcionar. Y esto es así porque utilizan funciones que están definidas en estas bibliotecas.

Podríamos decir que un ejecutable enlazado dinámicamente es un ejecutable que se termina de enlazar en tiempo de ejecución y no en tiempo de compilación. Está enlazado "a medias" porque todas las llamadas a funciones de biblioteca se dejan sin resolver. En el ejecutable tan solo se apunta que "aquí va una llamada a función" y no es hasta que el programa se ejecuta que se localiza en qué zona de memoria se halla esa función y se resuelve esa dependencia, escribiendo sobre la imagen del proceso a dónde tiene que saltar para ejecutarla.

`ltrace` puede monitorizar programas binarios que estén enlazados tanto estática como dinámicamente, pero en los primeros tan solo se puede obtener información relativa a las llamadas al sistema y no a las de funciones de biblioteca. La razón de esta limitación se deduce del procedimiento de carga de los ejecutables que seguidamente comentaremos y resultará por sí mismo evidente al término de esta explicación.

En los enlaces dinámicos, como hemos dicho, se recurre a un enlace tardío que, además, retrasa la ejecución del programa. Esto es simplemente necesario: la ventaja de los binarios enlazados dinámicamente es que no hay que incluir dentro de *todos* y *cada uno* de los ejecutables que hacen uso de una biblioteca el código de ésta. En su lugar, este código se incluye una única vez en el sistema, dentro del fichero de biblioteca pertinente, con el consiguiente ahorro de espacio en disco. Cuando se ejecuta un programa que necesita de esa biblioteca, ésta se lleva a memoria, y allí se termina de enlazar con el proceso. Una ventaja adicional es un ahorro, también, de espacio en memoria usada, ya que si muchos procesos necesitan el código de una misma biblioteca, basta con cargar ésta una única vez en memoria y dirigir adecuadamente a los ejecutables hacia esa zona. (Esto último lleva consigo una cierta complicación que resuelve el núcleo, puesto que es preciso que la misma zona de memoria física aparezca varias veces en el espacio de memoria lógico de cada proceso).

Evidentemente, esta tarea de enlace tardío que comentamos tiene que hacerla alguien, y ese alguien es el cargador dinámico. Cuando el núcleo del sistema operativo detecta que se quiere ejecutar un programa (a través de la llamada al sistema `execve()`), empieza por analizar qué tipo de fichero se está tratando de ejecutar. En función de cada tipo tendrá que actuar de una manera o de otra.

En el caso que nos ocupa, al comprobar que se trata de un fichero *ELF* enlazado dinámicamente, no lo ejecuta directamente, sino que en su lugar ejecuta el enlazador dinámico, indicándole a éste el programa que el usuario desea ejecutar,

así como todos los parámetros que se le habían dado inicialmente a este programa.

Como ejemplo, veamos uno de los programas más sencillos que podemos crear, el famoso “Hello, world!”:

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

int
main(void) {
  printf("Hello, world!\n");
  exit(0);
}
```

Este programa, una vez compilado de la manera habitual, dará lugar a un ejecutable *ELF* con una serie de características que describiremos a continuación.

Para empezar, a no ser que se lo indiquemos explícitamente al compilador, este programa dará lugar a un ejecutable enlazado dinámicamente, como nos muestra la salida de la orden “`file`”:

```
> file hello
hello: ELF 32-bit LSB executable, Intel 80386, versio
n 1 (SYSV), for GNU/Linux 2.0.30, dynamically linked
(uses shared libs), not stripped
```

Aquí puede verse el tipo de fichero que es el resultado de la compilación: es un ejecutable *ELF32* para *Linux/i386*, y está enlazado dinámicamente, tal como puede verse en “`dynamically linked (uses shared libs)`”.

El hecho de que esté enlazado dinámicamente significa que el núcleo al ejecutarlo carga en memoria otro fichero, el *enlazador dinámico*, que se encarga de preparar el programa para que pueda ejecutarse y de enlazarlo con todas las bibliotecas que utilice. Este enlazador y la lista de las bibliotecas pueden verse aquí:

```
> ldd ./hello
        libc.so.6 => /lib/libc.so.6
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2
```

El enlazador dinámico es en realidad `/lib/ld-linux.so.2`, y eso es algo que se puede comprobar experimentalmente: resulta equivalente ejecutar “`./hello`” que “`/lib/ld-linux.so.2 ./hello`”.

Ahora bien, ¿cómo sabe el núcleo que el enlazador dinámico es precisamente "`/lib/ld-linux.so.2`"? Ese enlazador es el que se usa actualmente en ejecutables compilados con la versión 2 de la biblioteca *GNU libc*, pero no siempre ha sido así, y es posible que cambie en el futuro. Es el propio ejecutable "`hello`" el que incluye dentro de una de las secciones *ELF* el nombre del enlazador dinámico que debe invocarse.

Esta es la primera vez que nos referimos a una de las secciones *ELF*. A partir de ahora nombraremos unas cuantas más con cierta asiduidad. Todas ellas están explicadas en el estándar *ELF* [Com95].

La sección que lleva esta información se ha bautizado allí como `.interp`, y puede leerse de la siguiente manera:

```
> objdump -s -j .interp hello

hello:     file format elf32-i386

Contents of section .interp:
 80480f4 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048104 2e3200                               .2.
```

Veamos ahora cuál es la tarea del enlazador dinámico. Éste se encarga, entre otras cosas, de resolver todas las referencias a llamadas externas que pueda tener nuestro ejecutable.

Recordemos nuestro programa, "`hello.c`":

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

int
main(void) {
  printf("Hello, world!\n");
  exit(0);
}
```

Para empezar, observamos que en el código se utilizan tres nombres de funciones: `main()`, `printf()` y `exit()`, pero no todas se utilizan de la misma manera: la función `main()` está definida en nuestro programa, pero las otras dos no lo están. Son parte de la "biblioteca estándar", y a la hora de ejecutar este programa, el sistema tiene que encontrar su implementación en otro fichero.

Todas estas funciones, junto con los nombres de las variables y alguna cosa más forman la "tabla de símbolos" del ejecutable; para consultar esa tabla y ver la

lista de todos los símbolos de un ejecutable cualquiera podemos utilizar el programa "nm". En realidad, este programa muestra mucha más información de la que nos interesa ahora mismo; un extracto de su salida es el siguiente:

```
> nm ./hello
         U exit
0804835c T main
         U printf
```

Aquí se puede ver la lista de los símbolos que hay en el ejecutable. En nuestro caso, esos símbolos son la lista de funciones que aparecen en el código, y vemos las tres que hemos enumerado anteriormente. Se observa claramente la diferencia entre la única función definida (`main()`) y las no definidas (`exit()` y `printf()`): los símbolos definidos incluyen la dirección de memoria donde se encuentran, mientras que los que no lo están no incluyen dirección alguna y se distinguen por la letra "U" (*Undefined*) en la salida que muestra `nm`.

Vamos a ver ahora cómo se invoca en cada momento una función de biblioteca, por ejemplo `printf()`. El compilador traduce esta llamada a código máquina. Si nos remitimos al ensamblador de la familia *Intel x86*, que es seguramente el más conocido, el código será una instrucción similar a esta: "`call 0x804828c`". Es una instrucción correspondiente al código del programa, y por lo tanto la encontraremos en la sección `.text` del ejecutable, y a su vista podríamos pensar que en la dirección de memoria `0x804828c` es donde se encuentra la función `printf()`, pero esto sería así solo en el caso de que la función `printf()` estuviera incluida en el código del ejecutable, esto es, que estuviera compilado estáticamente.

Pero este no es el caso. Aquí el `call` apunta a otra dirección dentro de una sección distinta del ejecutable *ELF*: la denominada `.plt`, abreviatura de *Procedure Linkage Table*, y popularmente conocida como "el trampolín".

Veamos el aspecto que muestra una parte de ese `.plt` que incluya la dirección `0x804828c` que nos ocupa (para ver la sección íntegramente, ejecútese la orden "`objdump -d -j .plt hello`"):

```
 804828c:      ff 25 e0 94 04 08      jmp    *0x80494e0
 8048292:      68 08 00 00 00         push   $0x8
 8048297:      e9 d0 ff ff ff         jmp    0x804826c
 804829c:      ff 25 e4 94 04 08      jmp    *0x80494e4
 80482a2:      68 10 00 00 00         push   $0x10
 80482a7:      e9 c0 ff ff ff         jmp    0x804826c
```

Como se ve, en el `.plt` se repite un mismo patrón. Lo seguimos con el primer caso, integrado por las 3 primeras líneas: se salta a la dirección contenida en

una posición de memoria (se salta a la dirección que se guarda en la dirección `0x80494e0`, no a la propia `0x80494e0`), se guarda un valor en la pila (`0x0`), que además es único en todo el `.plt` y va incrementándose en entradas subsiguientes (`0x8`, `0x10`, ...) y, por último, se salta a la dirección `0x804826c`, la misma en todas las entradas del `.plt`.

En un principio, si ya en la primera línea de cada entrada hay un `jmp`, un salto, las otras dos no parecen tener demasiado sentido. Sin embargo, el salto de ese `jmp` es indirecto, concretamente al contenido de la dirección `0x80494e0`. Y esa dirección es parte de otra sección del ejecutable, el `.got` (*Global Offset Table*).

El `.got` es una sección que contiene un puntero por cada una de las funciones de biblioteca dinámica que utiliza el programa. Y lo más peculiar de esta sección es que en principio está inicializada para que cada puntero contenga la dirección de la instrucción inmediatamente posterior al `jmp` que encontramos en la entrada de cada símbolo en el `.plt`, esto es, a la instrucción `push`. En el caso que nos ocupa, la dirección de memoria en el `.got` correspondiente a `printf()` es la `0x80494e0`, y el contenido de la dirección `0x80494e0` es `0x8048292`. Esto hace que el flujo de ejecución continúe con el `push` y luego con el segundo `jmp`. El valor introducido en la pila gracias al `push` sirve como identificador de la función que se debe resolver —que debe resolver el enlazador dinámico—, y tras él se ejecuta el `jmp` a la dirección `0x804826c`, que es la misma en todas las entradas del `.plt` y conduce al código del enlazador dinámico (el `/lib/ld-linux.so.2` aludido anteriormente), que se encarga de buscar la dirección de la función a la que se quería saltar (`printf()`), pone esa dirección en la entrada del `.got` (en la dirección `0x80494e0`), para que la siguiente vez que se llame a la función ya esté resuelta, y se procede con la ejecución de la función.

Resumiendo: cuando en un programa aparece una llamada a una función de biblioteca, si el programa está compilado dinámicamente, sucede lo siguiente:

- Todas las llamadas a esa función en el programa aparecen como llamadas a una dirección del `.plt`.
- Esta dirección del `.plt` contiene un salto indirecto al contenido de una dirección correspondiente al `.got`.
- Si la función no está aún resuelta, la entrada del `.got` redirige el flujo del programa al enlazador dinámico, que la resuelve y actualiza el `.got` para futuras llamadas a esta misma función.

- Si la función ya se ha resuelto anteriormente, del `.got` pasa directamente al código de la función dada en la biblioteca.

Con esto ya tenemos claro cómo se comporta un programa en lo que se refiere a la ejecución de funciones y podemos ver las alternativas que existen para llevar a cabo nuestro cometido.

Dado que lo que queremos es interceptar todas las llamadas a las funciones de biblioteca, tenemos tres opciones:

- Sustituir el enlazador dinámico para tener el control cada vez que se intente resolver un símbolo, y modificar su comportamiento para que se vuelva a llamar en posteriores ocasiones.
- Averiguar cuándo pasa el control del programa por el comienzo de cada una de las funciones a trazar.
- Hacer lo mismo con las direcciones del `.plt` correspondientes a cada función.

En la práctica, la primera opción resulta demasiado complicada y lenta, y no añaden ninguna ventaja sobre las demás, por lo que se puede descartar sin problema.

La segunda y la tercera opción sugieren dos maneras distintas de afrontar el problema, y son sustituyendo o modificando el comienzo de cada función de la biblioteca o la entrada correspondiente al `.plt` para cada función. Las dos opciones son equivalentes y permiten conseguir exactamente el mismo resultado, pero la experiencia nos indica que es bastante más sencillo trabajar en el `.plt` que en las bibliotecas, por tres razones:

1. El código de las bibliotecas no está disponible desde el comienzo de la ejecución del programa, sino que son cargadas por el enlazador dinámico cuando son necesarias. Por lo tanto, no se podría acceder a dicho código en todo momento y sería muy difícil hacerlo en el momento apropiado.
2. No se conoce *a priori* la dirección de memoria en la que comenzará cada función, porque las bibliotecas son *reubicables*, esto es, el enlazador puede cargarlas en cualquier posición de memoria, luego habría que esperar a que el enlazador las cargara para extraer su dirección de los datos internos del propio enlazador.

3. Un mismo ejecutable puede utilizar funciones de muchas bibliotecas diferentes, que se cargarán en lugares de la memoria diferentes, con lo que habría que acceder a varios rangos de direcciones de memoria distintos complicando mucho la situación.

3.4. Control del ejecutable a trazar

Una vez que sabemos qué es lo que hay que monitorizar, debemos decidir cómo hacerlo.

Se han considerado dos alternativas para llevar a cabo el trazado de las llamadas: modificando el código del programa y utilizando otra biblioteca para dejar nuestras rutinas de control, o bien utilizando un programa externo que se comunique con el programa a trazar por medio de *breakpoints*.

Cada una de las opciones es perfectamente viable, y tiene sus ventajas e inconvenientes, que serán mostrados a continuación.

3.4.1. Uso de una biblioteca adicional

Si de alguna manera logramos introducir código en el espacio de usuario del proceso a trazar, se podría sustituir la primera instrucción de la ejecución de cada una de las funciones de biblioteca por una llamada a una función nuestra que se encargue de averiguar de qué función se trata y los argumentos que se le pasa, para mostrar de alguna manera esta traza y posteriormente reanudar el comportamiento normal del programa.

Los pasos a seguir para lograr esto son los siguientes:

- Escribir el código con forma de biblioteca dinámica, esto es, introduciendo todo dentro de un fichero de determinadas características y de tipo “*shared object*”. En la implementación de esta alternativa, se decidió llamar a la nueva biblioteca “`libtrace.so`”.
- El código de inicialización y sustitución de instrucciones ha de estar situado en la sección `.init` de esta biblioteca. La sección `.init` de cualquier objeto *ELF* tiene la particularidad de que es lo primero que se ejecuta, antes de que comience el control normal del programa.

- Para conseguir que el programa incluya automáticamente el código de nuestra biblioteca al ejecutarse, hemos de hacerlo definiendo la variable de entorno `LD_PRELOAD`. El enlazador dinámico `/lib/ld-linux.so.2` comprueba si esta variable está definida y si lo está carga en memoria la biblioteca indicada en ella.
- La tarea de interceptar las llamadas pertinentes requiere poder escribir en el código en memoria, concretamente en la sección `.plt`, pero desgraciadamente esa zona de memoria es parte del código del programa, con lo que por defecto está protegida de manera que no se pueda escribir en ella. Por lo tanto, para lograrlo es necesario hacer uso de `mprotect()` para permitir la escritura.
- Para no interferir con el programa a trazar, el código a incluir en la biblioteca ha de estar escrito de una manera muy cuidadosa, procurando no hacer mucho uso de la pila ni reservar mucha cantidad de memoria, ni gastar mucho tiempo de CPU, ni usar ninguna señal ni ningún descriptor de fichero (tarea que complica especialmente la comunicación con el exterior). Además, conviene ser conservador y limitar al máximo las llamadas al sistema que se hagan.

En definitiva, resulta bastante difícil escribir una librería con estas características. Pero no todo acaba ahí; esta alternativa tiene una serie de desventajas que la desaconsejan:

- Depende en gran medida del enlazador dinámico, y de que éste se comporte de la manera adecuada a la hora de añadir una biblioteca mediante el uso de `LD_PRELOAD` y a la hora de ejecutar las secciones `.init` de las bibliotecas cargadas en el orden y en el momento preciso.
- Ejecutar código en el espacio de memoria del propio proceso a trazar puede causar interferencias no deseables con éste, que pueden hacer que se comporte de manera distinta cuando está siendo trazado y cuando no lo está, causando los errores conocidos como *heisenbugs*.
- Utilizando esta manera de trazar programas hay algunas cosas de las que no podemos tener control y sería muy deseable, como son:
  - Controlar las señales que se reciben
  - Controlar las llamadas al sistema
  - Saber exactamente en qué momento se crean nuevos procesos o se ejecutan otros programas

- Tener control sobre la muerte del proceso a trazar

Por todo esto, esta alternativa, a pesar de que se utilizó en versiones antiguas de `ltrace`, acabó desechándose y en su lugar hoy en día se utiliza el control externo mediante *breakpoints* y utilizando las facilidades que el Sistema Operativo proporciona para controlar otros procesos, esto es, la llamada al sistema `ptrace()`.

3.4.2. Control del ejecutable mediante el uso de un programa externo

Como se ha dicho anteriormente, el Sistema Operativo suele proporcionar herramientas que nos permiten controlar otros procesos. En el caso de los sistemas *POSIX*, estas herramientas con la llamada al sistema `ptrace()` y, en la mayor parte de las arquitecturas, la posibilidad de insertar *breakpoints* en puntos arbitrarios de la ejecución de un programa para tener control del momento en el que dicho programa pasa por los mismos.

El proceso completo que se realiza para establecer los *breakpoints* en un programa que nos permiten tener control del mismo es:

- Examen del ejecutable para averiguar la lista de funciones de biblioteca dinámica que es necesario trazar, y la dirección correspondiente a cada una de ellas en el `.plt` (tal como ya se explicó en las secciones 3.2.2 y 3.3).
- Carga en memoria del programa y preparación de éste para ser trazado mediante el uso de la llamada al sistema `ptrace(PTRACE_TRACEME)` o bien de `ptrace(PTRACE_ATTACH)`.
- Inserción de los *breakpoints* en cada una de las entradas del `.plt` correspondientes a las funciones a trazar con `ptrace(PTRACE_PEEKTEXT)` y `ptrace(PTRACE_POKETEXT)`.
- Comienzo de la ejecución controlada del programa que estamos trazando usando `ptrace(PTRACE_SYSCALL)`.

A partir de este momento, el proceso trazado comienza su ejecución normal, con la peculiaridad de que `ltrace` tiene control de todo lo que sucede: las señales que se reciben, las llamadas al sistema que realiza, cuándo y de qué manera termina su ejecución, y los momentos en los que se para la ejecución en un *breakpoint*.

El proceso trazador, es decir, `ltrace`, se queda en este momento a la espera de que el sistema operativo le informe de cualquier cambio en el estado del proceso trazado. Y eso se hace de la misma manera que los procesos padres esperan a la terminación de sus hijos, esto es, con la llamada al sistema `wait()`.

El interfaz que proporciona `wait()` para recibir información de un proceso trazado es un tanto complicado, ya que no se diseñó con este propósito. Tradicionalmente, puede informar tan solo de estos tres tipos de sucesos relativos a un proceso:

- Terminación normal de un proceso
- Terminación de un proceso debida a una señal
- Parada de un proceso debida a una señal

Los dos primeros casos tienen la semántica habitual: informan de la terminación de un proceso y de qué manera ha sucedido.

Sin embargo, el último caso (parada debida a una señal) engloba ahora varias cosas más. Junto con la información de que el proceso ha sido parado, la llamada `wait()` proporciona el número de la señal que ha causado la parada. Pues bien, este caso tiene el mismo significado de siempre a no ser que dicha señal sea `SIGTRAP` ("Trace/breakpoint trap"); en ese caso, la parada se ha producido por el comienzo o final de una llamada al sistema, o porque el proceso acaba de llegar a un *breakpoint*.

La diferencia entre ambos casos depende del sistema operativo y de la arquitectura que estemos usando. Por ejemplo, en *Linux/i386*, si determinado valor en la pila del proceso es igual a cero, entonces se trata de un *breakpoint*; en caso contrario, dicho valor indica el número de la llamada al sistema que el proceso está ejecutando.

3.4.3. Ejecución de una llamada al sistema

En todo momento, un proceso trazado puede ejecutar una llamada al sistema. `ltrace` tiene que detectar la ejecución de éstas, ya que, a pesar de que no siempre las vaya a mostrar, es necesario que detecte determinadas llamadas, como ya se ha dicho anteriormente:

- Llamadas que cambian el código que ejecuta un proceso, como `execve()`. Estas llamadas provocan que haya que volver a leer la tabla de símbolos y a inicializar los *breakpoints* del nuevo programa.
- Llamadas que crean un nuevo proceso, como `fork()`. En este caso, hay que desactivar todos los *breakpoints* justo antes de ejecutar la llamada, y volver a activarlos al terminar ésta.

En caso de que no esté activada la opción para mostrar las llamadas al sistema, ha de ejecutarse simplemente un `ptrace(PTRACE_SYSCALL)` para continuar con la ejecución normal del proceso. Pero si está activa, hemos de escribir su nombre y sus argumentos; después, permitir que el proceso termine de ejecutarla y, por último, cuando haya acabado, escribir el valor de retorno.

Para conseguir el código numérico de la llamada al sistema que se esté ejecutando, hemos de preguntárselo al sistema. La manera específica depende de la arquitectura; en algunas, como *Linux/i386*, hay que leer el valor de un registro del procesador en concreto (`eax`). Dicho número se convierte en el nombre de la llamada en cuestión utilizando una tabla única para cada arquitectura.

En este punto, tenemos que extraer los argumentos de la información que el sistema operativo nos proporciona del proceso parado. Este punto también es muy dependiente de la arquitectura: en algunas, estos parámetros están en registros del procesador, y en otras están en la pila. Por ejemplo, en *Linux/i386*, los 5 primeros parámetros de una llamada al sistema se encuentran en los registros `ebx`, `ecx`, `edx`, `esi` y `edi`, y se ha de usar `ptrace(PTRACE_PEEKUSER)` para conseguir el contenido de dichos registros.

Una vez los tenemos los datos, se muestran por pantalla y se le indica al sistema que continúe con la ejecución del proceso (usando, una vez más, la llamada `ptrace(PTRACE_SYSCALL)`).

Cuando la llamada al sistema termina, `wait()` vuelve a indicar que el proceso ha sido interrumpido y, una vez más, hay que usar `ptrace()`, esta vez para averiguar cuál es el valor de retorno de la llamada al sistema, para indicarlo por la salida y poder continuar nuevamente el proceso con otro `ptrace(PTRACE_SYSCALL)`.

3.4.4. Recepción de un *breakpoint*

Cuando se recibe una indicación de que el programa se ha parado en un *breakpoint*, lo primero que debemos hacer es averiguar en qué dirección de memoria se

ha producido la parada, y para ello hay que consultar el contenido del contador de programa. Hacemos esto utilizando la función `ptrace(PTRACE_PEEKUSER)`, y, una vez que tengamos la dirección del *breakpoint*, tenemos que buscar en nuestra lista para ver a qué función corresponde.

Una vez tenemos el nombre de la función, hay que mostrarlo junto con los argumentos con los que está siendo llamada. Para ello, primero se mira en la información de los ficheros de configuración si sabemos cuántos argumentos hay que mostrar y de qué tipo, y a continuación hay que proceder a conseguir los argumentos en sí.

Los argumentos de una función están almacenados en distintos sitios, dependiendo de la arquitectura con la que estemos trabajando. Por ejemplo, en *i386* y en *m68k* se guardan en la pila del proceso, con lo que tendremos que usar `ptrace(PTRACE_PEEKUSER)`, mientras que en *arm*, *powerpc* y *s/390* se guardan en registros del procesador y hay que usar `ptrace(PTRACE_PEEKTEXT)`.

Ya tenemos el nombre de función a la que hemos llamado, y los argumentos que se le pasan. Para seguir, hemos de introducir en el ejecutable un nuevo *breakpoint*, en la dirección de memoria que se ejecute inmediatamente después de salir de la función, esto es, la dirección de memoria inmediatamente posterior a la cual desde la que se ha hecho la llamada. Necesitamos hacer esto para volver a tener el control en el momento en el que la función termine de ejecutarse y así poder mostrar el valor de retorno que proporcione. Esta dirección también es dependiente de la arquitectura que se esté usando.

Y por último, una vez que se ha hecho todo esto, es necesario proseguir con el programa trazado, pero dejando activo el *breakpoint* para poder volver a saber cuándo se llama a esta función en el futuro. Esto se hace de la siguiente manera:

- Eliminamos el *breakpoint* temporalmente, sustituyendo su valor por el contenido original de esa dirección de memoria antes de activarlo.
- Si es necesario (en algunas arquitecturas lo es y en otras no), se decrementa el contador de programa en el número de octetos adecuado para que vuelva a apuntar a la dirección en la que estaba situado el *breakpoint*.
- Se informa al sistema operativo de que ejecute una única instrucción y vuelva a pararse (usando `ptrace(PTRACE_SINGLESTEP)`).
- Una vez la ha ejecutado y ha vuelto a pararse el proceso, volvemos a introducir el *breakpoint* que se había eliminado anteriormente.

- Finalmente, se informa al sistema operativo de que continúe el proceso de manera normal. Para que siga interrumpiéndose tanto al llegar a un *breakpoint* como al ejecutar una llamada al sistema, hemos de utilizar `ptrace(PTRACE_SYSCALL)`.

En este punto, el proceso sigue ejecutándose normalmente, con la salvedad de que ahora hay un *breakpoint* adicional que no teníamos antes, y se llega a él justo cuando termine de ejecutarse la función que estamos trazando.

En este momento, se produce otra parada en dicho *breakpoint*, y ahora podemos averiguar el valor de retorno y mostrarlo. Después, ha de desactivarse este último *breakpoint*, decrementar el contador de programa si es necesario para que apunte a la dirección en la que estaba, y continuar la ejecución desde ese punto.

Como se ha visto, esta manera de controlar la ejecución de otros procesos tiene una gran desventaja, y consiste en que cada arquitectura tiene distintas maneras de guardar el contador de programa, las direcciones de retorno, los argumentos y los valores de retorno; las señales y las llamadas al sistema también pueden ser diferentes en unas arquitecturas y en otras y, por último, también hay distintas maneras de establecer los *breakpoints*. Por lo tanto, el código a ejecutar ha de ser distinto para las distintas arquitecturas.

Capítulo 4

Implementación

En este capítulo veremos todos los aspectos relativos a la realización física del proyecto, esto es, aspectos prácticos como ver la plataforma de programación elegida, los ficheros de los que consta el desarrollo del proyecto y la manera de implementar cada uno de los módulos del programa.

Se dará una explicación de cada una de las funciones que se encuentran en cada fichero, pero no se reproducirá aquí el código. El código completo puede encontrarse fácilmente en Internet, en la dirección

`ftp://ftp.debian.org/debian/pool/main/l/ltrace`

o en cualquier distribución completa de Debian, RedHat o SuSE en CD, o bien puede solicitarse en cualquier momento al autor.

4.1. Elección del entorno de desarrollo

Las dos principales plataformas de desarrollo para la mayor parte de las aplicaciones informáticas de hoy en día son *GNU/Linux* y *Microsoft Windows*. Por lo tanto, es razonable elegir una de estas dos opciones para el desarrollo del programa.

Se ha optado por la primera, *GNU/Linux*, pero no ciñéndonos al uso exclusivo de este sistema sino más bien al estándar *POSIX*, por las siguientes razones:

- Es una plataforma de desarrollo más cómoda y barata
- Los compiladores y bibliotecas disponibles son más flexibles y permiten un control más fino de los aspectos necesarios para la realización (control de

las funciones y bibliotecas usadas, etc)

- Es un sistema abierto, con posibilidad de conseguir información acerca de todo lo necesario para la realización del proyecto (estructura de los ejecutables y de las bibliotecas, funcionamiento de las llamadas al sistema necesarias, etcétera)

El proyecto ha sido realizado íntegramente en lenguaje *C*, debido a una serie de características que son difíciles de encontrar en otros lenguajes:

- Es un lenguaje de alto nivel y permite programación modular y estructuras de datos complejas
- Al mismo tiempo, permite un control muy fino de las estructuras de bajo nivel, para lograr controlar perfectamente el uso de los *breakpoints*, las llamadas al sistema, la estructura interna de los ejecutables *ELF*, etcétera
- Además, es un lenguaje que consigue un código muy rápido, y la velocidad es crucial en este proyecto

Las herramientas utilizadas en la programación y desarrollo del proyecto han sido las siguientes:

- Compilador: *GNU Compiler Collection* (*GCC*)
- Biblioteca estándar: *GNU Libc*
- Control de generación de ejecutables: *GNU make*
- Configurador del código: *GNU autoconf*
- Depurador: *GNU symbolic debugger* (*GDB*)

Adicionalmente, a medida que el proyecto iba tomando forma y siendo funcional, se ha ido usando para corregir fallos en él mismo, esto es, se ha usado `ltrace` para corregir fallos en el propio `ltrace`.

Software Libre

Como puede observarse, todas las herramientas de programación utilizadas en el desarrollo del proyecto son libres y protegidas por la licencia *General Public License* del proyecto GNU. La GPL proporciona libertad para copiar o adaptar un programa a las necesidades de cada uno—más aún, todo el que consiga una copia del programa hereda la libertad de poder hacer más copias, modificarlas y distribuirlas libremente.

Estas libertades se han considerado muy deseables para un proyecto de esta índole, y de hecho se decidió que el propio proyecto estuviera protegido también por la licencia GPL, principalmente para permitir de manera cómoda aportaciones a su desarrollo por parte de todo tipo de personas e instituciones que consideren beneficioso hacerlo.

Hasta la fecha, y gracias a esta decisión, han sido muchas las aportaciones a este proyecto, tanto individuales como por parte de empresas, y su ayuda ha sido crucial en la tarea de adaptar el programa para que funcione correctamente en plataformas distintas de la familia de procesadores *i386*, como se verá más adelante.

4.2. Estructura general

La distribución completa de `ltrace` consta de un total de 81 ficheros, incluyendo la documentación, el código fuente y una serie de programas necesarios para compilar este código. De todos esos ficheros, los que contienen el código en sí son 47. En total hay 4626 líneas de código fuente, una cantidad razonable para un proyecto de este índole.

Como ya se ha indicado anteriormente, el programa se ha escrito íntegramente en el lenguaje de programación C, con lo que todo el código está incluido en ficheros con extensión "`.h`", que incluyen la definición de las funciones y las estructuras de control, y ficheros con extensión "`.c`", que contienen la implementación de las funciones.

El diseño del programa es muy modular, dividido en varios ficheros con un fin muy determinado, lo que permite que sea relativamente fácil realizar cualquier modificación que sea necesaria, así como posibilitar la adaptación a cualquier arquitectura o sistema operativo de una manera sencilla, a pesar de lo complicado que resulta de por sí dada la naturaleza del proyecto.

A lo largo de las siguientes secciones iremos viendo cada uno de los módulos en los que se divide el programa, los nombres de los ficheros y el cometido de cada uno, las funciones de las que se compone cada fichero y las estructuras de control usadas en estos como medio de comunicación entre los distintos módulos.

A modo de introducción, enumeraremos a continuación las partes más importantes:

- **Inicio del programa**: Módulo muy sencillo que se encarga de iniciar el programa e ir llamando a distintas funciones de otros módulos según son necesarias, y al terminar de finalizar el programa.
- **Procesado de opciones**: Se encarga de examinar los argumentos que se le dan al programa al ejecutarlo, interpretarlos y preparar el resto de los módulos para comportarse de manera adecuada.
- **Lectura de ficheros de configuración**: Lee la configuración global, que indica cómo debe mostrar cada una de las funciones que ejecute el programa a depurar.
- **Apertura y ejecución de programas**: Aquí se encuentran las rutinas encargadas de examinar los ejecutables para averiguar qué funciones de biblioteca utilizan y los prepara para poder mostrar las llamadas según se van realizando
- **Eventos**: Bucle principal del programa. Espera a que el sistema informe de algún acontecimiento a tener en cuenta para poder procesarlo de manera adecuada. Estos acontecimientos son siempre cambios de estado de alguno de los procesos que se están examinando.
- **Salida de mensajes**: Muestra por pantalla las llamadas que van realizando los programas examinados y cualquier otra información relevante acerca de estos.

Sin embargo, no basta con todo lo anterior. Lamentablemente, hay muchas partes del programa que no pueden usarse de la misma manera en arquitecturas diferentes, especialmente el módulo de apertura y ejecución de programas, ya que las técnicas usadas para abrir un programa dependen mucho de un sistema operativo a otro, el formato de los ejecutables también cambia y por último la manera de hacer llamadas a funciones es diferente. Además, incluso dentro de un mismo sistema operativo, hay que tener en cuenta la arquitectura del procesador ya que es necesario hacer operaciones con registros y con instrucciones de código máquina.

Por todo lo anterior, aparte de los módulos nombrados es necesario que parte del código sea dependiente del sistema operativo y de la arquitectura en que esté funcionando el programa.

4.2.1. Ficheros distribuidos

El proyecto se distribuye habitualmente de manera distinta según sea la distribución de *software* que lo haga, y dentro de cada distribución suele haber una distinción clara entre las fuentes y los binarios compilados para una arquitectura en concreto.

Los ficheros que se suelen distribuir normalmente son los siguientes (suponiendo que la versión distribuida es la 0.3.31):

`ltrace_0.3.31.dsc`	Descripción de fuentes de Debian
`ltrace_0.3.31.tar.gz`	Fuentes del programa
`ltrace_0.3.31_i386.deb`	Binarios de Debian para *i386*
`ltrace_0.3.31_m68k.deb`	Binarios de Debian para *Motorola*
`ltrace-0.3.31-1.src.rpm`	Fuentes para RedHat
`ltrace-0.3.31-1.i386.rpm`	Binarios de RedHat para *i386*

En lo sucesivo hablaremos únicamente de la estructura del código fuente, esto es, del contenido del fichero `ltrace_0.3.31.tar.gz`.

Este fichero es un archivo que contiene muchos otros ficheros, varios de los cuales contienen el código fuente en lenguaje *C*, pero no todos, también hay algunos necesarios para poder compilar el proyecto, ficheros de documentación, y ficheros de soporte para las distribuciones.

4.2.2. Contenido de la distribución

En la distribución del proyecto hay varios tipos de ficheros:

- **Código fuente**

  En este proyecto, todo el código está escrito en *C*, con lo que todos los ficheros fuente tienen extensión "`.c`" o "`.h`".

  La mayor parte de los ficheros que forman parte del código fuente están en el mismo directorio, pero hay unos pocos en las partes dependientes del sistema operativo.

- **Ficheros necesarios para la compilación**

  En este proyecto se utiliza la herramienta *autoconf* del Proyecto GNU; para poder utilizar esa herramienta, se incluyen los siguientes ficheros:

```
-rwxr-xr-x    1 cespedes  38377 2002-02-24 13:17 config.guess
-rw-r--r--    1 cespedes    800 2002-03-03 00:22 config.h.in
-rwxr-xr-x    1 cespedes  28991 2002-02-24 13:17 config.sub
-rwxr-xr-x    1 cespedes  54396 2002-03-03 00:22 configure
-rw-r--r--    1 cespedes   1294 2002-03-03 00:20 configure.in
-rw-r--r--    1 cespedes   1441 2002-03-03 22:00 Makefile.in
```

- **Documentación**

  Parte de los ficheros incluidos en la distribución están ahí como ayuda al usuario o programador que examina el proyecto: forman parte de la documentación. Estos ficheros son:

```
-rw-r--r--    1 cespedes    254 2001-05-29 20:52 BUGS
-rw-r--r--    1 cespedes  15468 2003-02-04 23:24 ChangeLog
-rw-r--r--    1 cespedes  18007 2001-05-29 20:52 COPYING
-rw-r--r--    1 cespedes   4222 2002-03-03 00:15 ltrace.1
-rw-r--r--    1 cespedes   2742 2002-03-31 20:38 README
-rw-r--r--    1 cespedes    954 2002-03-27 00:20 TODO
```

  Esta documentación incluye información sobre las condiciones de uso y redistribución del programa, información general acerca de éste, de fallos conocidos y de mejoras previstas, la página de manual, y un histórico con los cambios que ha habido a lo largo de las distintas versiones, desde agosto de 1997 hasta la fecha actual.

- **Ficheros de ayuda para las distribuciones**

  Las distribuciones de *software* proporcionan sus aplicaciones divididas en una serie de *paquetes*, y cada uno de estos paquetes se distribuye de manera independiente como un fichero aparte de un formato determinado. Los formatos más habituales son el usado por Debian (un fichero con extensión "`.deb`") y el estándar de RedHat (extensión "`.rpm`").

  Para poder compilar el programa y crear el paquete necesario para una distribución en concreto, es necesaria la existencia de algunos ficheros adicionales con descripción y reglas que indican qué se ha de hacer para crearlo. Estos ficheros son los siguientes:

```
-rw-r--r--    1 cespedes  14189 2002-09-16 21:22 debian/changelog
-rw-r--r--    1 cespedes     17 2001-05-29 20:52 debian/conffiles
-rw-r--r--    1 cespedes    873 2002-03-31 19:58 debian/control
-rw-r--r--    1 cespedes   1618 2002-03-31 19:53 debian/copyright
-rw-r--r--    1 cespedes    141 2001-05-29 20:52 debian/postinst
-rw-r--r--    1 cespedes    125 2001-05-29 20:52 debian/prerm
-rwxr-xr-x    1 cespedes   1547 2002-03-03 22:07 debian/rules
-rw-r--r--    1 cespedes   1375 2002-03-03 22:01 ltrace.spec.in
```

4.2.3. Configuración y compilación del código fuente

El proyecto se ha realizado usando las recomendaciones del Proyecto GNU en cuanto a la manera de crear proyectos de *software* (esto es, los *GNU coding-standards*) y más concretamente usando su sistema de `autoconf` para la configuración del código fuente, y el uso de *GNU Make* para la compilación de los módulos.

Por lo tanto, se incluye un fichero "`configure.in`" en el que se especifican distintos aspectos a comprobar a la hora de compilar el código para poder adaptarlo a la arquitectura y sistema operativo que tengamos.

La herramienta `autoconf` de *GNU* es usada para transformar el fichero `configure.in` en un *script* llamado `configure`, que es el que se encarga de hacer las comprobaciones necesarias y luego generar los siguientes ficheros:

Fichero original	Fichero generado por `autoconf`
`config.h.in`	`config.h`
`Makefile.in`	`Makefile`
`ltrace.spec.in`	`ltrace.spec`

- El fichero `config.h` se incluye desde todos los módulos y les proporciona información acerca de los tipos de datos existentes y las funciones proporcionadas en la arquitectura en la que estemos trabajando
- El `Makefile` contiene las instrucciones específicas para compilar uno a uno todos los módulos y enlazarlos para dar lugar al ejecutable
- `ltrace.spec` se utiliza para generar el paquete RPM para ser usado en las distribuciones que lo requieran

4.2.4. Ficheros de Debian

Dentro de la estructura de directorios de las fuentes de `ltrace` existe un directorio llamado "`debian`" que contiene varios ficheros necesarios para la generación de los paquetes de esta distribución.

Estos ficheros son:

- **`debian/changelog`**

  Histórico de cambios en las distintas versiones distribuidas por Debian.

- **debian/conffiles**

  Lista de ficheros considerados por Debian como ficheros de configuración. Contiene un único fichero, el "/etc/ltrace.conf".

- **debian/control**

  Fichero de control con toda la información relativa al paquete, como su nombre, descripción, dependencias, nombre del desarrollador de Debian encargado de él, arquitecturas en las que funciona, etc.

- **debian/copyright**

  Nombre del autor principal y los colaboradores, copyright y licencia del programa.

- **debian/postinst**, **debian/prerm**

  Ficheros necesarios para la instalación correcta del paquete en un sistema Debian; estos ficheros son *scripts* que se ejecutan al instalar y al desinstalar el paquete, respectivamente.

- **debian/rules**

  Este fichero se ejecuta cuando se desea construir un paquete Debian; ha de tener instrucciones específicas para crear los ficheros ".dsc", ".tar.gz" y ".deb" a partir del directorio completo de fuentes.

4.3. Inicialización

Aquí veremos el inicio del programa, que consiste en la función main(), que se ejecuta siempre al inicio de todo programa escrito en C, y cómo se realiza el proceso de inicialización, en el que se procesan las opciones y se interpretan los ficheros de configuración.

4.3.1. ltrace.c

Este es el fichero que contiene el código de comienzo de la ejecución de ltrace.

main()

- Prepara el programa para dejar la terminal en un estado adecuado después de la terminación normal, e inicializa las señales `SIGINT` y `SIGTERM` para tener control de cuándo se interrumpe el proceso pulsando "`control-C`" o matándolo con `kill`, con la finalidad de poder dejar de trazar los procesos que estemos controlando antes de salir del programa.
- Inicializa la salida por pantalla.
- Llama a `process_options()` para procesar las opciones de la línea de órdenes.
- Lee los ficheros de configuración, llamando a `read_config_file()`.
- En caso de que sea necesario trazar un programa desde el principio de su ejecución, primero se carga en memoria llamando a `open_program()`, y luego se ejecuta con `execute_program()`.
- Si hay que trazar un proceso que ya se esté ejecutando, se ejecuta la función `open_pid()`.
- Por último, se entra en un bucle infinito en el que se espera a que se reciban eventos, llamando a `wait_for_something()`, para procesarlos a continuación con `process_event()`.

4.3.2. `options.c`

Aquí encontramos una función encargada de procesar las opciones con las que se ha llamado a `ltrace` en la línea de órdenes, junto con una serie de variable globales para guardar el estado de las opciones usadas en la ejecución, con la finalidad de que puedan usarse desde otras partes del programa.

process_options()

Se utiliza la función `getopt()` para ir recorriendo los argumentos que se encuentran en la línea de órdenes, y por cada opción válida se actualiza una variable global con el nombre de la opción precedida de "`opt_`". Por ejemplo, si se aparece en la línea de órdenes el argumento "`-i`", entonces la variable global "`opt_i`" estará activada y podrá consultarse desde cualquier otro fichero.

En caso de detectar una opción no válido, se muestra un mensaje de error y se sale del programa.

Si se usan las opciones “`-h`”, “`--help`”, “`-V`” o “`--version`”, que se usan para recabar determinada información general, es esta misma función la que muestra esa información en forma de mensajes explicativos y termina el programa.

4.3.3. `read-config-file.c`

En este fichero se encuentran las funciones y estructuras de control necesarias para leer e interpretar los ficheros de configuración de `ltrace`. Estos ficheros de configuración contienen una lista de nombres de funciones junto con el número y tipo de los argumentos que requieren, con el objeto de saber de qué manera deben mostrarse esas funciones cuando el programa a trazar las ejecute.

La lista de todas las funciones definidas en los ficheros de configuración se guarda en la variable global `list_of_functions`.

`read_config_file()`

Esta función va recorriendo línea a línea un determinado fichero de configuración. Cada línea es separada en *tokens* y cada uno de estos tokens es interpretado para localizar los tipos de valores que requiere cada función en concreto. Cada tipo se compara con la lista de todos los posibles tipos que se pueden usar en los ficheros de configuración, y que se encuentra en la variable `list_of_pt`.

Una vez leídas todas las líneas de cada fichero de configuración, la variable `list_of_functions` está actualizada y la función termina.

4.4. Ejecución

Una vez ha terminado la inicialización básica, hace falta estudiar los programas que deben comenzar a trazarse, cargarlos en memoria, leer la lista de funciones de biblioteca dinámica que pueden ejecutar, insertar los *breakpoints* pertinentes, y por último comenzar la ejecución del programa.

4.4.1. `proc.c`

Aquí veremos las funciones encargadas de la apertura de procesos, tanto nuevos como ya existentes, con la finalidad de proporcionar una estructura llamada “`struct process`” que contiene toda la información que vamos a necesitar por cada proceso que debamos trazar.

`open_program()`

Esta función se invoca cuando tenemos que comenzar a trazar un proceso desde el principio, esto es, hay que abrir un fichero ejecutable que está en el disco, estudiarlo y prepararlo para que pueda comenzar a ejecutarse.

Su cometido es inicializar una estructura de tipo “`struct process`”, y mediante llamadas a `read_elf()` y a `breakpoints_init()` lee la tabla de símbolos del ejecutable y prepara los *breakpoints* pertinentes para después poder activarlos.

Además, actualiza una tabla que contiene la lista de todos los procesos controlados, llamada `list_of_processes`.

`open_pid()`

Esta función es muy similar a la anterior; también inicializa una estructura de tipo “`struct process`” y la rellena con información pertinente a un ejecutable, pero esta vez el ejecutable es un proceso de la máquina.

Su funcionamiento es muy sencillo: primero, se averigua a qué fichero del disco corresponde el proceso en cuestión, llamando a la función `pid2name()` (que veremos más adelante), y seguidamente se llama a `open_program()`.

4.4.2. `execute-program.c`

Una vez se ha leído la información necesaria de un programa para comenzar a trazarlo, es necesario comenzar su ejecución controlada.

`execute_program()`

Esta es una función en principio muy sencilla; crea un nuevo proceso con la ayuda de `fork()`, informa al sistema operativo de que se ha de tener control de ese proceso, llamando a `trace_me()` (que veremos en la parte dependiente del sistema operativo), y por último ejecuta `execve()`.

La única complicación adicional que tiene está en lo referente a tratar de manera adecuada el UID y GID del proceso cuando el ejecutable tiene activados los bits de *set-uid* o *set-gid* o cuando `ltrace` es ejecutado con la opción "`-u`".

4.4.3. `elf.c`

En este fichero están todos los accesos físicos a los ejecutables para averiguar la lista de funciones que incluye.

En la versión actual funciona únicamente con ficheros de tipo *ELF32* y se accede a ellos de manera directa, sin utilizar ninguna biblioteca de apoyo como podría ser `libbfd` [Cha91].

`read_elf()`

Esta función lee la lista de símbolos de un ejecutable que corresponden con funciones no definidas (esto es, las que se encuentran en una biblioteca dinámica externa) y las incluye en una tabla de tipo "`struct library_symbol[]`". Los elementos de la tabla son el nombre de cada función y la dirección de memoria de comienzo de dicha función, esto es, la dirección de la sección "`.plt`" que corresponde a cada función.

El código de esta función es bastante intrincado, más de lo que podría ser, ya que además de leer la lista de símbolos del ejecutable, puede ser necesario leer una a una todas las bibliotecas de las que depende para ver los símbolos de cada una de éstas. Esto es necesario si se ha usado la opción "`-l`", que se usa para restringir las bibliotecas de las que queremos información.

4.5. Autómata

Hasta ahora hemos visto las partes del programa que se ejecutan siempre al principio: la inicialización y el estudio de los ejecutables a trazar para averiguar la

lista de funciones en las que hemos de incluir *breakpoints*. Una vez hecho esto, el programa ha de entrar en el bucle principal, en el que se espera indefinidamente a que se reciban *eventos* y, una vez recibidos, se procesan de manera adecuada.

4.5.1. `wait-for-something.c`

Los eventos recibidos por el programa se almacenan en una variable de tipo "`struct event`".

Cada elemento de esta estructura tiene un campo que identifica el proceso que ha provocado el evento (para poder distinguirlos en caso de que estemos trazando varios procesos simultáneamente), el tipo de evento de que se trata y una variable auxiliar que tiene información adicional, de distinta naturaleza según cuál sea el evento recibido.

Los tipos de eventos posibles son los siguientes:

Evento	**Variable auxiliar**
Señal recibida por un proceso	Número de señal
Terminación normal de un proceso	Valor de retorno
Terminación provocada por una señal	Número de señal
Comienzo de una llamada al sistema	Número de llamada
Fin de una llamada al sistema	Número de llamada
Breakpoint	Dirección de parada

`wait_for_something()`

Esta función se queda esperando a que el sistema operativo informe de un cambio en alguno de los procesos que estamos controlando, lo interpreta y devuelve un puntero a una variable de tipo "`struct event`" con información sobre dicho evento.

Antes de nada se comprueba si hay algún proceso que nos pueda enviar un evento. Si no lo hay, significa que todos han terminado y el programa termina normalmente.

Dado que la llamada `ptrace()`, que es la que se usa en este proyecto para controlar procesos, notifica a los procesos que controlan a otros a través de `wait()`, la parte principal de esta función consiste simplemente en esta línea:

```
    pid = wait(&status);
```

Una vez ejecutada, se comprueba cuál es el identificador de proceso que ha generado el evento, se mira en la lista de procesos (`list_of_processes`) para encontrar la estructura `struct process` correspondiente, se hacen las comprobaciones adecuadas para saber cuál ha sido el evento recibido, se rellena la variable de retorno y se devuelve dicha variable.

4.5.2. `process-event.c`

Una vez que se ha recibido un evento de uno de los procesos que controlamos, hay que procesarlo. Este fichero incluye la función a la que se llama para procesar los eventos y varias más auxiliares, pero no incluye ninguna estructura de control especial.

`process_event()`

La implementación de esta función es muy sencilla, ya que simplemente compara el tipo de evento recibido con cada uno de sus posibles valores, y llama a una función diferente para cada tipo de evento. Todas estas funciones son estáticas y se encargan de lo siguiente:

- **Recepción de una señal**

  Muestra por pantalla el nombre de la señal recibida, y continúa el proceso.

- **Terminación normal del programa**

  Muestra el valor de retorno del programa. Borra el proceso de la lista de procesos controlados (`list_of_processes`).

- **Terminación provocada por una señal**

  Muestra la terminación y la señal que la ha provocado. Borra el proceso de la lista `list_of_processes`.

- **Llamada al sistema**

  Llama a la función `output_left()` para mostrar el nombre de la llamada al sistema y los argumentos.

  Si la llamada provoca la creación de un proceso nuevo (como `fork()`), desactiva temporalmente todos los *breakpoints* para que el nuevo proceso no los incluya.

- **Fin de una llamada al sistema**

  Llama a la función `output_right()` para mostrar los argumentos que queden por mostrar, si es el caso, y el valor de retorno de la llamada.

  En caso de que la llamada recién ejecutada sea una que provoca la creación de un nuevo proceso (como `fork()`), vuelve a activar todos los *breakpoints* del proceso original y, si se ha usado la opción para controlar los procesos recién creados (opción `-f`), llama a `open_pid()` para comenzar a trazar al nuevo proceso.

  Si la llamada que cambia la imagen en memoria del proceso que estamos controlando (como `execve()`), vuelve a abrir el nuevo proceso y a inicializar los *breakpoints* llamando a `breakpoints_init()`.

- ***Breakpoint***

  Si un proceso controlado ha llegado a un *breakpoint* significa que o bien está a punto de comenzar a ejecutar una llamada a una función que debemos mostrar, o que acaba de terminar la ejecución de una función llamada anteriormente.

  Para distinguir en qué caso estamos, se comprueba la dirección en la que se ha parado el proceso con las direcciones de los símbolos que tenemos en `list_of_symbols` y con las direcciones de retorno que están almacenadas en una pila llamada `callstack`.

  En caso de ser el comienzo de una función, se muestra la función llamando a `output_left()` y se introduce en la pila de direcciones de retorno la correspondiente a esta función.

  Si es una terminación, se saca la dirección correspondiente de la pila, eliminando el *breakpoint* (y todos los posteriores, si los hubiera) y se muestra el final de la llamada llamando a `output_right()`.

4.6. Funciones comunes

Las funciones que aparecen en los ficheros descritos en esa sección no pueden englobarse dentro de las categorías anteriores, ya que son funciones auxiliares que se llaman desde varios lugares del programa.

4.6.1. `breakpoints.c`

Aquí veremos una serie de funciones que se utilizan en otras partes del programa (principalmente en `process_event()`) encargadas de insertar y eliminar *breakpoints* en los puntos indicados de un proceso que controlemos.

`address2bpstruct()`

Consulta la tabla de *breakpoints* activados en un proceso, y devuelve la entrada correspondiente a una dirección de memoria en concreto. Para ello hace uso de `dict_find_entry()`, para buscar una entrada en concreto en un diccionario (veremos más información sobre el uso de estos "diccionarios" a la hora de mostrar el fichero `dict.c`).

`insert_breakpoint()`

Inicializa el diccionario de *breakpoints* si es necesario (con `dict_init()`).

Lo consulta y, si la dirección que se pasa como argumento no está siendo usada en algún *breakpoint*, crea una entrada en el diccionario (con `dict_enter()`) para marcarla como usada, y habilita el *breakpoint* llamando a la función de bajo nivel `enable_breakpoint()`. En caso de que ya existiera, aumenta un contador que indica las veces que se ha hecho uso de `insert_breakpoint()` con cada dirección en concreto.

`delete_breakpoint()`

Deshace los pasos de `insert_breakpoint()`, esto es:

- Busca en el diccionario de *breakpoints* la entrada correspondiente a la dirección requerida
- Decrementa el contador de veces en uso de esa dirección
- Si ese contador ha llegado a cero, deshabilita el *breakpoint* llamando a `disable_breakpoint()`

`enable_all_breakpoints()`

Recorre el diccionario de *breakpoints* y va habilitándolos todos, uno por uno. Se usa al comienzo del programa y para volver a habilitarlos si se han deshabilitado antes debido a, por ejemplo, el uso de la llamada al sistema `fork()`.

`disable_all_breakpoints()`

Recorre el diccionario de *breakpoints* y deshabilita todos ellos. Se usa al ejecutarse una llamada al sistema que cree otro proceso, como `fork()`, y al dejar de trazar un proceso que ya existía.

`breakpoints_init()`

A esta función se la llama cuando se desea inicializar el diccionario de *breakpoints* con todas las funciones que se desean trazar de un ejecutable; por tanto, esta función es llamada cada vez que se quiere trazar un proceso distinto, o sea, al comienzo del programa y cada vez que un proceso ejecuta `execve()`.

4.6.2. `dict.c`

Esta es la implementación de un diccionario genérico de pares de claves y valores implementado con una *tabla hash*. Se utiliza para guardar distintas tablas en distintas partes del código:

- Lista de *breakpoints* activos de cada proceso trazado
- Si está activa la opción "`-c`", lista de llamadas realizadas y tiempo empleado por cada una
- Si está activa la opción "`-C`", lista de nombres de funciones en C++ codificados y su correspondiente decodificación

`dict_init()`

Crea e inicializa un diccionario nuevo. Como argumentos tiene dos funciones: una es la *función hash* que se utilizará para crear el *hash* a partir de una clave, y otra es una función para comparar dos claves en el diccionario. Devuelve un puntero a la estructura del diccionario.

`dict_clear()`

Libera el espacio usado por un diccionario que queramos dejar de utilizar.

`dict_enter()`

Introduce un dato nuevo en el diccionario, compuesto por una clave y un valor.

`dict_find_entry()`

Busca en el diccionario una entrada correspondiente a una clave dada. Devuelve el valor si existe, o `NULL` si ese dato no está en el diccionario.

`dict_apply_to_all()`

Esta función se utiliza para recorrer un diccionario y ejecutar una función con cada uno de los datos almacenados en éste. Por ejemplo, en el caso de un diccionario de *breakpoints*, se usa para habilitar o deshabilitar todos los *breakpoints* que estén en el diccionario.

`dict_key2hash_string()`

Función hash predefinida para claves consistentes en cadenas de caracteres. La función es una combinación sencilla entre todos los caracteres de la cadena que da lugar a un entero sin signo.

`dict_key_cmp_string()`

Función de comparación entre claves que sean cadenas de caracteres. Consiste en una comparación alfanumérica entre las dos claves utilizando la función `strcmp()`.

`dict_key2hash_int()`

Función hash para claves que sean números enteros. Actualmente esta función es simplemente una conversión de enteros a enteros sin signo.

dict_key_cmp_int()

Función de comparación entre claves numéricas. Consiste en la resta de las dos claves a comparar.

4.6.3. debug.c

Aquí tenemos una función para ayudar al depurado del propio `ltrace`. El depurado está activo siempre que se invoque al programa con la opción "`-d`"; si no es así, la función correspondiente no hace nada.

debug()

Imprime por pantalla una línea en la que se indica el fichero, número de línea y función en el que se ha hecho la llamada a `debug()`, junto con un mensaje especificado como uno de los argumentos.

4.6.4. output.c

Aquí tenemos las funciones principales de salida de mensajes por pantalla, que es lo que el usuario va a ver cuando se ejecuta `ltrace`.

output_left()

Se llama a esta función cada vez que se recibe un evento de comienzo de función o de llamada al sistema. Se muestra el nombre de la función, y se comprueba si tenemos información del número y tipo de parámetros que acepta de la tabla `list_of_functions`. Si es así, van mostrándose uno a uno llamando a `display_arg()`. Si no lo es, se muestran 5 argumentos tomándolos como números enteros.

output_right()

Una vez que ha terminado de ejecutarse una llamada, esta función muestra los argumentos que falten, si es que falta alguno, seguidos del valor de retorno de la función.

`output_line()`

Este programa no solo muestra llamadas a funciones, sino también señales recibidas, errores, mensajes de terminación de procesos...

Cada uno de esos mensajes se muestra por la salida gracias a esta función.

4.6.5. `display-args.c`

Los argumentos de las funciones han de mostrarse de distinta manera dependiendo del tipo que tengan; por ejemplo, el argumento de la función `umask()` se muestra como un número en octal, y el argumento de `strlen()` se muestra como una cadena de caracteres.

Los diferentes tipos de argumentos y la manera de representarse están especificados en la sección 5.3 del manual de usuario.

`display_arg()`

Esta función ha de llamarse especificando el proceso del que queremos mostrar un argumento, el número de orden del argumento en la función y su tipo. Para conseguir el argumento en sí llama a la función `gimme_arg()`, y se presenta el argumento de manera distinta dependiendo del tipo que tenga.

4.6.6. `demangle.c`

La opción "`-C`" se usa para traducir los nombres de símbolos usados internamente por C++ por otros de alto nivel fácilmente comprensibles por el usuario. En este fichero se encuentra la función encargada de hacer dicha conversión.

`my_demangle()`

Se utiliza el diccionario mostrado anteriormente para mantener una lista de símbolos ya traducidos y no tener que volver a traducirlos si vuelven a presentarse.

Para traducir cada símbolo se utiliza la función `cplus_demangle()` de la biblioteca `libiberty`.

4.6.7. summary.c

Este fichero se usa exclusivamente cuando en la línea de órdenes de `ltrace` se ha incluido la opción “`-c`”, que indica que se desea que se ejecute un proceso sin interrupción y luego se muestre un resumen con la lista de funciones llamadas, el número de veces que se ha llamado a cada una y el tiempo transcurrido por cada una de ellas.

Para conseguir esta información, se usa un diccionario en el que se guardan los nombres de las funciones, el número de veces que se ha llamado a cada una y el tiempo total transcurrido dentro de cada función. Esta información se actualiza en `output_left()` y `output_right()`.

show_summary()

Para mostrar el resumen, al finalizar la ejecución del proceso trazado se llama a esta función, que recorre el diccionario usando `dict_apply_to_all()` y rellenando una estructura que posteriormente muestra por pantalla.

4.7. Parte dependiente del sistema operativo

Todas las funciones explicadas hasta ahora son bastante independientes del sistema operativo que se esté ejecutando; solo presuponen tres cosas: que estamos en un sistema *POSIX*, que el formato de los ejecutables es *ELF32* y que la manera que proporciona el sistema operativo para acceder a los datos de otro proceso es similar a la del `ptrace()`, que se encuentra en la mayor parte de los sistemas *POSIX*.

Sin embargo, hacen falta algunas funciones más, aparte de las vistas anteriormente, que desgraciadamente no pueden escribirse sin hacer más suposiciones acerca del sistema operativo. Aquí veremos cuáles son estas funciones y cómo se implementan.

En tiempo de compilación, el sistema de `autoconf` comprueba cuál es el sistema operativo para el que debe funcionar el resultado, y proporciona un término que lo identifica (en el caso de *GNU/Linux*, este término es “`linux-gnu`”).

Posteriormente, el `Makefile` utiliza el directorio “`sysdeps/`*`término`*” (que en el caso de *GNU/Linux* será “`sysdeps/linux-gnu`”) para compilar allí la parte dependiente de la arquitectura y generar un fichero “`sysdeps.o`”

que contiene todas las funciones necesarias ya compiladas y listas para enlazarse con el resto del programa.

4.7.1. Lista de funciones

Las funciones que deben estar implementadas en el fichero “`sysdeps.o`” y su cometido son las siguientes:

- **`trace_me()`**

  Esta función ha de llamarla un proceso para que ese mismo proceso comience a ser controlado por su “proceso padre”. Esto equivale al uso de “`ptrace(PTRACE_TRACEME)`”.

- **`trace_pid()`**

  Comienza a trazar un proceso ya existente, que pasa a estar controlado por el proceso actual. Equivale a “`ptrace(PTRACE_ATTACH)`”.

- **`untrace_pid()`**

  Permite que un proceso trazado por `ltrace` continúe su ejecución normal, deje de estar controlado. Equivalente a “`ptrace(PTRACE_DETACH)`”.

- **`pid2name()`**

  Dado un proceso que se esté ejecutando en este momento, esta función devuelve el nombre del fichero ejecutable con el código de dicho proceso. Esto es necesario para poder leer los símbolos de un proceso que queramos trazar usando la opción “`-p`”.

- **`enable_breakpoint()`**

  Introduce un *breakpoint* en una dirección determinada de un proceso que estemos controlando.

- **`disable_breakpoint()`**

  Desactiva un *breakpoint* que haya sido introducido anteriormente usando `enable_breakpoint()`.

- **`syscall_p()`**

  Una vez que un proceso trazado haya sido interrumpido, se utiliza esta función para averiguar si el motivo de la interrupción ha sido una llamada al sistema.

- **fork_p()**

  Dada una llamada al sistema, esta función indica si se trata de alguna que genere un proceso nuevo, tales como `clone()`, `fork()`,...

- **exec_p()**

  Indica si una llamada al sistema dada provoca que cambie la imagen en memoria del código de un proceso, esto es, que se ejecute otro programa distinto, tal como hace la llamada `execve()`.

- **get_instruction_pointer()**

  Devuelve el contenido del contador de programa de un proceso que estemos controlando.

- **get_stack_pointer()**

  Devuelve el contenido del puntero de pila de un proceso.

- **get_return_addr()**

  Esta función nos da la dirección de retorno una vez el proceso ha iniciado una llamada a una función. Esto es, devuelve la dirección por la que continuará ejecutándose el programa después de salir de la función actual.

- **gimme_arg()**

  Devuelve un argumento dado de los pasados a una función o llamada al sistema que esté ejecutando un proceso. Adicionalmente, también puede devolver el valor de retorno de una llamada (si se pide el argumento número -1).

- **umovestr()**

  Copia un rango de memoria del espacio de direcciones de un proceso trazado al del proceso trazador, es decir, al propio `ltrace`.

- **continue_after_signal()**

  Provoca la continuación normal de un proceso que ha recibido una señal.

- **continue_after_breakpoint()**

  Una vez que se ha producido un *breakpoint* en un proceso, si queremos que continúe la ejecución de manera normal pero dejando el *breakpoint* activado, hemos de llamar a esta función. Su labor es desactivarlo temporalmente y provocar la ejecución de una instrucción (con `ptrace(PTRACE_SINGLESTEP)`).

- **continue_enabling_breakpoint()**

  Poco después de haber llamado a `continue_after_breakpoint()` se interrumpirá nuevamente el mismo proceso, ya que se ha ejecutado una única instrucción de código máquina. Para continuar el proceso dejando el *breakpoint* activado hace falta este segundo paso, que se encarga de volver a activarlo y de continuar normalmente.

- **continue_process()**

  Por último, si se quiere continuar el proceso de manera normal, pero sin que haya pasado nada de lo anterior (es decir, sin que haya ningún *breakpoint* ni ninguna señal involucrada, o lo que es lo mismo, si la causa de la interrupción ha sido la ejecución de una llamada al sistema) se ejecuta esta función.

4.7.2. Particularidades de *GNU/Linux*

En el caso de *GNU/Linux*, como ya se ha dicho, el código dependiente del sistema operativo se encuentra en el directorio `sysdeps/linux-gnu`. Este directorio, a su vez, contiene varios subdirectorios, uno por cada arquitectura en la que funciona `ltrace`, esto es: `arm`, `i386`, `m68k`, `ppc` y `s390`. Esto es así porque algunas de las funciones que deben implementarse tienen que lidiar con detalles específicos de cada procesador, como puede ser el código que provoca que se genere un *breakpoint*, el nombre de los registros de la máquina, la manera de pasar argumentos a las funciones y a las llamadas al sistema, etc.

La información dependiente de la arquitectura, esto es, las funciones que están implementadas dentro del directorio de cada arquitectura en concreto son las que se encargan directamente de averiguar el contenido de registros concretos (`get_instruction_pointer()` y `get_stack_pointer()`), una para modificar un registro (`set_instruction_pointer()`) y tres más cuya implementación depende demasiado de la arquitectura que se esté usando (`gimme_arg()`, `syscall_p()` y `get_return_addr()`).

Además de estas funciones, cada arquitectura incluye tres ficheros de definiciones con información sobre la manera de usar *breakpoints* en cada arquitectura en concreto, y la lista y nombres de las señales y de las llamadas al sistema que existen en cada arquitectura.

Capítulo 5

Manual de usuario

El fin de un programa como este es posibilitar ver lo que está ocurriendo "dentro" de otro programa mientras este se está ejecutando.

Gracias a `ltrace`, podemos saber:

- Qué funciones de bibliotecas dinámicas se están ejecutando
- Qué llamadas al sistema se realizan
- Qué señales recibe el proceso

Se puede usar `ltrace` para depurar todo tipo de programas, independientemente del lenguaje de programación en el que estén escritos.

5.1. Ejemplo de sesión con `ltrace`

`ltrace` es una herramienta suficientemente sencilla como para poder empezar a utilizarla sin necesidad de haber leído antes la documentación. A continuación se mostrarán una serie de ejemplos sencillos sobre su uso.

Primeros pasos

Una vez que el programa está instalado en el sistema, proporciona un ejecutable con el mismo nombre: "`ltrace`", junto con un fichero de configuración ("`/etc/ltrace.conf`"), una página de manual y algo de documentación en

"/usr/share/doc/ltrace". El ejecutable está en uno de los directorios del sistema para uso de todos los usuarios, con lo que se puede usar simplemente escribiendo su nombre. Veámoslo:

```
> ltrace
ltrace: too few arguments
Try `ltrace --help' for more information
```

Como se ve, simplemente con ejecutarlo no hace nada, pero nos indica cómo podemos conseguir algo de ayuda para poder sacarle más provecho. Ejecutando "ltrace --help" se consigue una pantalla completa de ayuda en inglés con la manera de ejecutar el programa, la lista de las opciones disponibles y una breve indicación de para qué sirve cada una.

```
> ltrace --help
Usage: ltrace [option ...] [command [arg ...]]
Trace library calls of a given program.

  -a, --align=COLUMN  align return values in a specific column.
  -c                  count time and calls, and report a summary on exit.
  -C, --demangle      decode low-level symbol names into user-level names.
  -d, --debug         print debugging info.
  -e expr             modify which events to trace.
  -f                  follow forks.
  -h, --help          display this help and exit.
  -i                  print instruction pointer at time of library call.
  -l, --library=FILE  print library calls from this library only.
  -L                  do NOT display library calls.
  -n, --indent=NR     indent output by NR spaces for each call level nesting.
  -o, --output=FILE   write the trace output to that file.
  -p PID              attach to the process with the process ID pid.
  -r                  print relative timestamps.
  -s STRLEN           specify the maximum string size to print.
  -S                  display system calls.
  -t, -tt, -ttt       print absolute timestamps.
  -T                  show the time spent inside each call.
  -u USERNAME         run command with the userid, groupid of username.
  -V, --version       output version information and exit.

Report bugs to Juan Cespedes <cespedes@debian.org>
```

Si intentamos usar "ltrace" con un ejecutable complicado, mostrará por la pantalla una enorme cantidad de información que nos será difícil asimilar, con lo que para los ejemplos que siguen usaremos un programa sencillo, un "Hello world" (hello), que es el resultado de compilar este código:

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

int
main(void) {
  printf("Hello, world!\n");
  exit(0);
}
```

Este programa simplemente muestra por la pantalla la cadena de caracteres "`Hello, world!`" y luego se termina. Si lo ejecutamos con "`ltrace`", veremos lo siguiente:

```
> ltrace ./hello
__libc_start_main(0x08048430, 1, 0xbffffa44 <unfinished ...>
__register_frame_info(0x080494d4)                = 0x080482f4
printf("Hello, world!\n")                        = 14
exit(0 <unfinished ...>
__deregister_frame_info(0x080494d4)              = 0
+++ exited (status 0) +++
```

Aparecen por pantalla una serie de líneas, cada una de ellas indicando algo que está sucediendo en ese momento en el programa que estamos ejecutando (`hello`).

Cada una de estas líneas, excepto la última, nos indican nombres de funciones que se están ejecutando, los argumentos que se le pasan a cada una, y el valor que devuelven.

Por ejemplo, la tercera línea de salida del programa indica que se ha llamado a la función "`printf`", con un argumento que es una cadena de texto ("`Hello, world!`") y que después de ejecutarla ha devuelto el número 14.

La última línea, en cambio, no es indicativa de ninguna función que haya sido llamada por el programa, sino otro tipo de evento, que en este caso ha sido la finalización del programa. Además de indicar que el programa ha terminado, nos dice cuál ha sido el valor de retorno (0).

5.2. Opciones de ejecución

"`ltrace`" se usa principalmente para analizar el funcionamiento de un programa ya existente, que se quiera ejecutar desde el principio y del que queremos saber qué llamadas a funciones de biblioteca dinámica está realizando. Para ello, basta con escribir simplemente "`ltrace` *`programa`*", tal como se vio en el capítulo anterior. Si se quieren especificar argumentos al programa, se pueden escribir a continuación (`ltrace` *`programa arg1 arg2`*`...`)

El programa en cuestión comenzará a ejecutarse, y en el momento en que realice alguna llamada a una función de biblioteca, esta se mostrará por la pantalla, al igual que las señales que reciba, y una indicación de cuando el programa termine y cuál es su valor de retorno.

Sin embargo, se puede cambiar el comportamiento de `ltrace` indicando distintas *opciones de ejecución* delante del nombre del programa a trazar. Por lo tanto, la manera de invocar el programa es la siguiente:

```
ltrace [opciones] programa [argumentos...]
```

Las opciones de ejecución que acepta el programa usan el estilo de *GNU getopt*:

- Las opciones compuestas por una sola letra se declaran con un guión (`-`) delante de dicha letra
- Las opciones "largas", compuestas por más de una letra (una palabra) se declaran con dos guiones (`--`) delante de la palabra
- Dos opciones de una sola letra se pueden componer siempre que la primera no requiera argumentos (por ejemplo, "`-i -f`" es equivalente a "`-if`")
- Para las opciones de una letra que requieran un argumento, se puede especificar dicho argumento separado por un espacio o inmediatamente después de la opción, sin espacio (por ejemplo, "`-o file`" equivale a "`-ofile`")
- Para las opciones largas con argumento, este se puede especificar separado de la opción por un espacio o por el símbolo igual (=). Por ejemplo, "`--output=file`" y "`--output file`"

Todas las opciones se han de indicar delante del nombre de programa, ya que cualquier opción que aparezca después se entenderá como una opción para ejecutar ese programa en lugar de una opción para `ltrace`. Por ejemplo, ejecutar "`ltrace -c ls -l`" indicaría ejecutar `ltrace` con la opción "`-c`", indicando que el programa a estudiar será "`ls -l`".

El orden en el que se especifiquen las opciones carece de importancia; por ejemplo, "`ltrace -Siofile`" es equivalente a "`ltrace -o file -Si`" o a "`ltrace -i -o file -S`".

Seguidamente pasaremos a describir las opciones que se pueden especificar para ejecutar `ltrace` y cómo se puede usar cada una de ellas.

- `-a`, `--align=COL`

  Hace que los valores de retorno de las funciones estén alineados en la salida del programa a la columna especificada (o inmediatamente después de acabar los argumentos de entrada si esto no es posible).

- `-c`

  Ejecuta el programa sin mostrar ningún mensaje hasta que termina, momento en el cual se muestra una tabla en la que se indica una lista con las funciones que de biblioteca dinámica que se han ejecutado, el número de veces que se ha ejecutado cada una, y el tiempo total transcurrido en todas las llamadas a cada función.

```
> ltrace -c date
Thu Apr 24 21:54:00 CEST 2003
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- ------------------
 72.91    0.005187        5187         1 puts
  7.62    0.000542         542         1 dcgettext
  5.69    0.000405         405         1 setlocale
  3.77    0.000268         268         1 localtime
  2.64    0.000188          23         8 memcpy
  1.07    0.000076          38         2 strftime
  0.80    0.000057          57         1 clock_gettime
  0.79    0.000056          56         1 free
  0.73    0.000052          52         1 bindtextdomain
  0.62    0.000044          44         1 textdomain
  0.58    0.000041          41         1 getopt_long
  0.52    0.000037          37         1 realloc
  0.49    0.000035          35         1 __fpending
  0.49    0.000035          35         1 getenv
  0.44    0.000031          31         1 __cxa_atexit
  0.44    0.000031          31         1 nl_langinfo
  0.41    0.000029          29         1 memset
------ ----------- ----------- --------- ------------------
100.00    0.007114                    25 total
```

  En esta tabla se puede ver, por ejemplo, que el ejecutable ("`cal`") ha llamado a la función `printf()` un total de 8 veces, y que en total ha utilizado 0,030014 segundos entre las 8 veces que la ha llamado, con lo que en media ha tardado 3.751 microsegundos por cada llamada. El tanto porcentual es relativo a todas las llamadas que aparecen aquí; por lo tanto, ese valor indica que el 56,80 % del tiempo que el ejecutable ha utilizado en todas las llamadas a las funciones de biblioteca que aparecen en la tabla, lo ha consumido dentro de `printf()`.

- `-C`, `--demangle`

  Traduce los nombres de los símbolos correspondientes a funciones de C++, convirtiendo los nombres de bajo nivel usados internamente por el compilador en otros de alto nivel de manera que el usuario pueda ver claramente toda la información contenida en el nombre del símbolo, como por ejemplo el número de argumentos que requiere cada función y el tipo de estos.

```
> ltrace -C ./test-c++ >/dev/null
__libc_start_main(0x08048540, 1, 0xbffffa24 <unfinished ...>
__register_frame_info(0x080495e4)                 = 0x40064300
ostream::ls(char const *)(0x08049730, "Hello, world!\n") = 0x08049730
__deregister_frame_info(0x080495e4)               = 0x0804975c
+++ exited (status 0) +++
```

- `-d`, `--debug`

  Muestra información de depuración, o *debug*. Solo es útil para buscar posibles fallos en el propio `ltrace` o para examinar detenidamente lo que va haciendo en cada momento. Esta opción solo deberían usarla los propios desarrolladores de `ltrace`:

  ```
  > ltrace -d sync
  (más de 100 líneas de información y depurado)
  ```

  Si se repite esta opción, se incrementa el nivel de depurado.

- `-e EXPR`

  EXPR es una expresión que indica qué nombres de funciones se han de trazar. El formato de esta expresión es:

  [!]nombre1[,nombre2]...

  donde se pueden especificar la lista de nombres de funciones que se quieren examinar, o, si se usa el carácter "`!`" al principio, la lista de las que *no* se quieren mostrar. Por ejemplo, "`ltrace -e printf,scanf`" mostrará solo las llamadas a las funciones *printf()* y *scanf()* por parte del programa, mientras que "`ltrace -e !printf`" mostrará todas las llamadas a todas las funciones excepto *printf()*.

  Nótese que el carácter de exclamación suele tener un significado especial en la mayoría de las *shells*, por lo que normalmente es necesario usar el carácter de escape para poder usarlo.

  ```
  > ltrace -e printf df >/dev/null
  printf("Filesystem ")                               = 11
  printf("       ")                                   = 7
  printf(" %4s-blocks      Used Available "..., "1k") = 37
  printf(" Mounted on\n")                             = 12
  printf("%-20s", "/dev/hda1")                        = 20
  printf(" %*s %*s %*s ", 9, "2048688", 9, "1459140", 9, "589548") = 31
  printf("%*.0f%%", 3, ...)                           = 4
  printf(" %s", "/")                                  = 2
  printf("%-20s", "/dev/hda3")                        = 20
  printf(" %*s %*s %*s ", 9, "1685824", 9, "1660660", 9, "25164") = 31
  printf("%*.0f%%", 3, ...)                           = 4
  printf(" %s", "/home")                              = 6
  +++ exited (status 0) +++
  ```

- `-f`

  Esta opción indica que queremos tomar el control de todos los hijos del proceso a trazar, siguiendo todas las llamadas a `fork()`, `clone()`, etcétera.

```
> ltrace -f ./test-fork
__libc_start_main(0x08048460, 1, 0xbffffa44 <unfinished ...>
__register_frame_info(0x080494f4)                  = 0x08048318
fork()                                             = 768
[pid 767] sync( <unfinished ...>
[pid 768] sleep(1 <unfinished ...>
[pid 767] <... sync resumed> )                     = 0
[pid 767] __deregister_frame_info(0x080494f4)      = 0
[pid 767] +++ exited (status 0) +++
<... sleep resumed> )                              = 0
__deregister_frame_info(0x080494f4)                = 0
+++ exited (status 0) +++
```

- `-h`, `--help`

  Muestra una breve pantalla de ayuda con la lista de opciones disponibles y su significado.

```
> ltrace --help
Usage: ltrace [option ...] [command [arg ...]]
Trace library calls of a given program.

  -a, --align=COLUMN  align return values in a specific column.
  -c                  count time and calls, and report a summary on exit.
  -C, --demangle      decode low-level symbol names into user-level names.
  -d, --debug         print debugging info.
  -e expr             modify which events to trace.
  -f                  follow forks.
  -h, --help          display this help and exit.
  -i                  print instruction pointer at time of library call.
  -l, --library=FILE  print library calls from this library only.
  -L                  do NOT display library calls.
  -n, --indent=NR     indent output by NR spaces for each call nesting.
  -o, --output=FILE   write the trace output to that file.
  -p PID              attach to the process with the process ID pid.
  -r                  print relative timestamps.
  -s STRLEN           specify the maximum string size to print.
  -S                  display system calls.
  -t, -tt, -ttt       print absolute timestamps.
  -T                  show the time spent inside each call.
  -u USERNAME         run command with the userid, groupid of username.
  -V, --version       output version information and exit.

Report bugs to Juan Cespedes <cespedes@debian.org>
```

- `-i`

  Muestra el contenido del contador de programa en cada evento mostrado; esto permite saber desde qué posición se ha realizado cada llamada y tener una idea más clara del flujo de datos.

```
> ltrace -i ./hello >/dev/null
[08048371] __libc_start_main(0x08048430, 1, 0xbffffa44 <unfinished ...>
[08048419] __register_frame_info(0x080494d4)      = 0x080482f4
[08048443] printf("Hello, world!\n")              = 14
[08048450] exit(0 <unfinished ...>
[080483e3] __deregister_frame_info(0x080494d4)    = 0
[ffffffff] +++ exited (status 0) +++
```

- `-l`, `--library=FICH`

  Muestra únicamente las llamadas a funciones incluidas en la biblioteca especificada. Se pueden especificar varias bibliotecas (hasta un máximo de 20) usando varias veces esta opción. Por ejemplo, para ver solo las llamadas a la libc que utiliza "`xlogo`", podemos usar:

```
> ltrace -l /lib/libc.so.6 xlogo
__libc_start_main(0x08048eb4, 1, 0xbffffa44 <unfinished ...>
__register_frame_info(0x0804a880)                  = 0x08048b70
--- SIGINT (Interrupt) ---
+++ killed by SIGINT +++
```

- `-L`

  No mostrar las llamadas a funciones de biblioteca. Con esta opción, solo se mostrarán las llamadas al sistema (si está activa la opción "`-S`" y las señales recibidas por los procesos trazados).

```
> ltrace -L ping -c 3 localhost >/dev/null
--- SIGALRM (Alarm clock) ---
--- SIGALRM (Alarm clock) ---
+++ exited (status 0) +++
```

- `-n`, `--indent=NR`

  Especifica el sangrado adicional que van a tener en la salida del programa las llamadas anidadas, esto es, las llamadas a funciones que han sido realizadas dentro de otra llamada. Esta opción puede ser muy útil para averiguar el flujo interno del programa.

```
> ltrace -n 2 sync
__libc_start_main(0x08048ba8, 1, 0xbffffa44 <unfinished ...>
  __register_frame_info(0x0804ae64)              = 0x08048860
  setlocale(6, "")                               = "en_US"
  bindtextdomain("fileutils", "/usr/share/locale") = "/usr/share/locale"
  textdomain("fileutils")                        = "fileutils"
  __cxa_atexit(0x08048d34, 0, 0, 0x080497dc, 0xbffffa44) = 0
  sync()                                         = 0
  exit(0 <unfinished ...>
    ferror(0x401321e0)                           = 0
    __fpending(0x401321e0, 1, 0xbffffa44, 0x400423ee, 0x40134dd0) = 0
    __deregister_frame_info(0x0804ae64)          = NULL
+++ exited (status 0) +++
```

- `-o`, `--output=FICH`

  Guarda toda la salida en el fichero especificado en lugar de mostrarla por la salida estándar.

- `-p PID`

  Empieza a trazar un proceso ya existente, que tiene como número de proceso *PID*. Esta opción se puede repetir, para poder trazar varios procesos simultáneamente. Además, si se especifica la opción, ya no es necesario indicar un programa a ejecutar (pero puede hacerse igualmente; en ese caso, se estarán trazando dos programas simultáneamente: el correspondiente al número de proceso indicado en la opción "`-p`" y el indicado en la línea de órdenes).

  Los procesos que estén trazándose por usar la opción "`-p`" no se interrumpen cuando paramos el `ltrace`; en lugar de esto, `ltrace` vuelve a dejarlos en estado normal para que continúen su trabajo y luego termina.

- `-r`

  Muestra en cada línea una indicación relativa del tiempo transcurrido. Esto es, recoge la diferencia en segundos entre el comienzo de dos líneas sucesivas.

```
> ltrace -r ./hello >/dev/null
  0.000000 __libc_start_main(0x08048430, 1, 0xbffffa44 <unfinished ...>
  0.042956 __register_frame_info(0x080494d4)      = 0x080482f4
  0.038401 printf("Hello, world!\n")              = 14
  0.001676 exit(0 <unfinished ...>
  0.000143 __deregister_frame_info(0x080494d4)    = NULL
  0.000505 +++ exited (status 0) +++
```

- `-s LEN`

  Especifica el tamaño máximo de las cadenas que se mostrarán; si hay una cadena más larga, se truncará a este valor.

```
> ltrace -s 5 ./hello >/dev/null
__libc_start_main(0x08048430, 1, 0xbffffa44 <unfinished ...>
__register_frame_info(0x080494d4)      = 0x080482f4
printf("Hello"...)                     = 14
exit(0 <unfinished ...>
__deregister_frame_info(0x080494d4)    = NULL
+++ exited (status 0) +++
```

- `-S`

  Esta opción indica a `strace` que ha de mostrar las llamadas al sistema que realicen los procesos analizados, además de las llamadas a funciones de biblioteca. Si se usan conjuntamente las opciones "`-L`" y "`-S`", el comportamiento de `ltrace` es muy similar al de `strace` (esto es, solo muestra las llamadas al sistema).

```
> ltrace -LS sync
SYS_uname(0xbffff530)                                 = 0
SYS_brk(NULL)                                         = 0x0804a308
SYS_open("/etc/ld.so.preload", 0, 010000210574)  = -2
SYS_open("/etc/ld.so.cache", 0, 00)              = 3
                    (...)
SYS_brk(NULL)                                         = 0x0804b308
SYS_brk(0x0804c000)                                   = 0x0804c000
SYS_sync()                                            = 0
SYS_exit_group(0)                                     = <void>
SYS_exit(0 <unfinished ...>
+++ exited (status 0) +++
```

- `-t`, `-tt`, `-ttt`

  Muestra la hora exacta en la que se realiza cada una de las llamadas, con precisión de segundos si se usa la opción “`-t`”, microsegundos si se usa “`-tt`” y microsegundos mostrando la hora en formato UNIX (número de segundos transcurridos desde el 1 de enero de 1970) si se usa “`-ttt`”.

```
> ltrace -t ./test-sleep
18:56:39 __libc_start_main(0x080483f0, 1, 0xbffffa34 <unfinished ...>
18:56:39 __register_frame_info(0x08049484)       = 0x080482d0
18:56:39 sleep(2)                                = 0
18:56:41 sleep(3)                                = 0
18:56:44 __deregister_frame_info(0x08049484)     = NULL
18:56:44 +++ exited (status 0) +++
```

- `-T`

  Muestra el tiempo transcurrido en el interior de cada llamada a función, aparte del resto de los datos. Este tiempo se calcula como la diferencia entre la hora del sistema vista al entrar y al salir de cada función, y se muestra en segundos, con una precisión de microsegundos (esto es, con 6 cifras decimales), a la derecha de cada función y rodeado de los símbolos “<” y “>”.

```
> ltrace -T sync
__libc_start_main(0x08048430, 1, 0xbffffa44 <unfinished ...>
__register_frame_info(0x080494d4)      = 0x080482f4 <0.000062>
printf("Hello, world!\n")              = 14 <0.000203>
exit(0 <unfinished ...>
__deregister_frame_info(0x080494d4)    = NULL <0.000054>
+++ exited (status 0) +++
```

- `-u USER`

  Ejecuta la orden especificada cambiando antes el *UID* y el *GID* del proceso a los correspondientes al usuario indicado. Esta opción solo tiene efecto cuando *ltrace* lo ejecuta el usuario `root`, y permite la ejecución correcta de binarios de tipo *set-uid* o *set-gid*.

- `-V`, `--version`

  Muestra el número de versión de `ltrace` y termina el programa.

```
> ltrace --version
ltrace version 0.3.30.
Copyright (C) 1997-2002 Juan Cespedes <cespedes@debian.org>.
This is free software; see the GNU General Public Licence
version 2 or later for copying conditions.  There is NO warranty.
```

5.3. Ficheros de configuración

Una de las primeras labores que realiza `ltrace` al empezar a ejecutarse es leer los ficheros de configuración, que especifican cómo debe mostrarse la información de cada función cuando esta la ejecuta el programa trazado.

Internamente, no hay ninguna base de datos con nombres de función y el tipo de argumentos que tiene cada una, ni se puede conseguir de otra manera que no sea leyendo directamente estos ficheros de configuración, con lo que en la práctica estos ficheros suelen ser bastante grandes, con información sobre cada una de las funciones que queremos que reconozca y que sepa interpretar a la hora de trazar un programa.

Hay dos ficheros de configuración que `ltrace` busca y procesa:

- `/etc/ltrace.conf`

  Este es el fichero de configuración global de la máquina. `ltrace` lo lee siempre, sea cual sea el usuario que lo esté ejecutando. Aquí tienen cabida las funciones globales de todas las bibliotecas del sistema. Este fichero está incluido en la distribución de `ltrace`, y actualmente contiene información acerca de 275 funciones.

- `${HOME}/.ltrace.conf`

  Este fichero es similar al fichero de configuración global. Su estructura y su sintaxis es idéntica, y se utiliza para que cada usuario pueda añadir información sobre funciones que no están en el fichero global, o para modificar la información sobre alguna que ya exista

Los ficheros de configuración se componen de varias líneas, cada una de las cuales incluye información sobre una función en concreto, como pueden ser “`malloc()`” o “`strlen()`”, incluyendo el número de argumentos que requiere y el tipo de cada uno de ellos, y además el tipo que devuelve. Su formato es muy similar al de la declaración de una función:

```
tipo_retorno nombre_función ( arg1, arg2, ... );
```

En realidad, para que `ltrace` reconozca una línea de la configuración como válida, tiene que encontrar los siguientes *tokens*:

- Una palabra que indica el tipo devuelto por la función; ha de estar entre la lista de tipos válidos (que veremos más adelante)
- El nombre de la función descrita
- Un símbolo "`(`"
- Lista de tipos separados por comas
- Un símbolo "`)`"

En cualquier caso, antes o después de cada elemento, siempre pueden aparecer uno o más espacios que son ignorados. Igualmente, todo lo que aparezca después del símbolo "`)`" es también ignorado. Y por último, cualquier línea que no se considere válida, por no encontrar los campos adecuados o porque no se reconozca algún tipo, es igualmente ignorada.

Ejemplo de líneas en el fichero de configuración:

```
; Esta línea es ignorada, aquí podría haber un comentario

void _exit(int);
addr bsearch(string, addr, uint, uint, addr);
int chmod(strig,octal);
uint strftime(+string2,uint,string,addr);
```

Como se puede ver, el aspecto es similar al de declaraciones de funciones en *C*, pero los "tipos" son distintos a los habituales.

A continuación se expone la lista de tipos que se pueden incluir en los ficheros de configuración, una explicación de cada uno y un pequeño ejemplo en el que aparezca el resultado de usar cada tipo.

- `void`

  Este es un "tipo" especial, solo se puede usar como valor de retorno o como único argumento. Si es usa como valor de retorno, indica que esta función no devuelve nada, y en la salida del programa siempre aparecerá como "= `<void>`". Si se usa como un argumento, indica que esta función no requiere argumentos. Ejemplo:

En el fichero de configuración:

```
void closelog(void);
```

En la salida de `ltrace`:

```
closelog()                                   = <void>
```

- `int`

  Indica un valor entero con signo. En la salida del programa se expresa como un número decimal, del tamaño de una palabra en la arquitectura que se use (32 bits en *i386*, lo que permite expresar números entre el -2147483648 y el 2147483647).

  En el fichero de configuración:

  ```
  int geteuid(void);
  ```

  En la salida de `ltrace`:

  ```
  geteuid()                                    = 1000
  ```

- `uint`

  Entero sin signo (*unsigned int*). Se usa para indicar números enteros que no pueden ser negativos, y en la salida se expresa como un número en decimal.

  En el fichero de configuración:

  ```
  uint alarm(uint);
  ```

  En la salida de `ltrace`:

  ```
  alarm(5)                                     = 0
  ```

- `octal`

  Al igual que "`uint`", `octal` indica el tipo para un número entero sin signo, pero especifica que ha de expresarse en octal en lugar de en decimal. En la salida, para indicar que el número está expresado en octal, se indica con un "0" delante de este.

  En el fichero de configuración:

  ```
  octal umask(octal);
  ```

  En la salida de `ltrace`:

  ```
  umask(077)                                   = 022
  ```

- `char`

  Número entero de 8 bits que se ha de representar como un carácter ASCII. Este carácter aparecerá en la salida entre comillas simples (`'`), y si es un carácter no imprimible (como un carácter de control), aparecerá su representación simbólica o en octal, precedida del carácter "\".

  En el fichero de configuración:

  ```
  char tolower(char);
  ```

  En la salida de `ltrace`:

  ```
  tolower('A')                     = 'a'
  tolower('\n')                    = '\n'
  tolower('\250')                  = '\250'
  ```

- `string`

  Una cadena de caracteres. En *C* (y en cualquier otro lenguaje de bajo nivel), una cadena de caracteres equivale a un puntero a una serie de caracteres, uno detrás del otro, con lo que el argumento se interpretará de esa manera. En pantalla se mostrarán los primeros caracteres imprimibles de esa cadena, hasta llegar al final o hasta un límite que en principio es de 32 caracteres pero que se puede cambiar con la opción "`-s`".

  En el fichero de configuración:

  ```
  string setlocale(int, string);
  ```

  En la salida de `ltrace`:

  ```
  setlocale(6, "")                 = "en_US"
  ```

- `addr`

  Indica un número sin signo que ha de expresarse en hexadecimal (precedido de "`0x`"). Normalmente es usa para indicar direcciones de memoria, ya que en la mayor parte de las arquitecturas el tamaño del bus de direcciones es el mismo que el del bus de datos (y por lo tanto una dirección de memoria ocupa lo mismo que un entero) y teniendo en cuenta que es habitual expresar las direcciones de memoria escritas en hexadecimal. En caso de tener que expresar la dirección `0x0`, esta se muestra como `NULL`.

  En el fichero de configuración:

  ```
  addr signal(int,addr);
  ```

En la salida de `ltrace`:

```
signal(13, 0x08048328)                    = NULL
```

- `file`

  Indica un tipo equivalente a un "`FILE *`" en *C*, esto es, el tipo que se usa por la mayor parte de las funciones de entrada y salida de la biblioteca estándar de *C* para referirse a ficheros abiertos. Actualmente, el comportamiento de un tipo "`file`" es el mismo que el de un tipo "`addr`", esto es, se muestra la dirección de memoria en hexadecimal.

  En el fichero de configuración:

  ```
  file fopen(string,string);
  int fclose(file);
  ```

  En la salida de `ltrace`:

  ```
  fopen("/etc/passwd", "r")              = 0x08049520
  fclose(0x08049520)                     = 0
  ```

- `format`

  Muy similar a "`string`", de hecho este argumento se trata cambiar con la opción "`-s`" exactamente de la misma manera (como un puntero a una serie de caracteres que se muestran unos a continuación de otros hasta un tamaño máximo), con la salvedad de que en este caso, los argumentos siguientes a este los interpreta tal y como los interpretaría la función `printf()` al ver esa cadena de formato. Por ejemplo, si en la cadena aparece un "`%d`" seguido de un "`%s`", se interpretarían los dos argumentos siguientes como un entero y como una cadena de caracteres, respectivamente.

  En el fichero de configuración:

  ```
  int printf(format);
  ```

  En la salida de `ltrace`:

  ```
  printf("The time is %d:%02d\n", 21, 5) = 18
  ```

- `string`*N*

  El uso de "`string`" seguido de un número entero no negativo (esto es, mayor o igual a 0) indica que el argumento es una cadena de caracteres, pero con un tamaño máximo especificado en el argumento con ese número de orden. Por ejemplo, si el valor de retorno es de tipo "`string1`", y el primer

argumento de la función es el número 20, entonces el valor de retorno se mostrará como una cadena de caracteres de tamaño máximo 20. El número 0 se refiere al valor de retorno (esto es, "`string0`" es una cadena cuyo tamaño máximo es el valor de retorno de la función).

En el fichero de configuración:

```
addr strncpy(addr,string3,uint);
```

En la salida de `ltrace`:

```
strncpy(0xbffff590, "Hello, wo", 9) = 0xbffff590
```

- `+`*`tipo`*

  Un símbolo "+" precediendo a un nombre de tipo indica que ese argumento no debe mostrarse hasta después de que la función haya terminado, esto es, que ese valor es modificado por la función y debería mostrarse su valor a posteriori. Puede usarse junto con cualquier tipo, pero solo tiene sentido junto con `string` o con `string`*`N`*.

  En el fichero de configuración:

  ```
  int snprintf(+string,format);
  ```

  En la salida de `ltrace`:

  ```
  sprintf("Feb 2003", "%s %d", "Feb", 2003) = 13
  ```

5.4. Ejecución de programas

Hay muchas maneras de ejecutar `ltrace`, dependiendo del uso que se quiera hacer de él. Podemos usarlo simplemente como curiosidad, para saber qué está haciendo un proceso en concreto en un determinado instante, o para saber cómo hace determinado programa su trabajo, o para depurar un programa que estemos escribiendo y no se comporte de la manera que esperamos, o para hacer *profiling* para intentar conseguir que un programa nuestro sea más correcto o más eficiente, o para ayudar a conseguir detalles para que un desarrollador averigüe qué falla en un programa, o para intentar averiguar la función de un determinado programa, o para estudiarlo para saber si ejecuta funciones consideradas "peligrosas" o si realiza de manera correcta acciones delicadas, o para estudiar la manera de acceder a determinado recurso que tiene un programa con objeto de copiar su funcionamiento…

Como se ve, hay muchas ocasiones en las que el uso de `ltrace` es útil, y hay muchas maneras de ejecutarlo para distintos propósitos. En este capítulo intentaremos ver los usos más habituales y cómo podemos sacar el mayor partido posible a esta herramienta.

5.4.1. Modo de funcionamiento

Básicamente, la función de `ltrace` es contactar con el Sistema Operativo para tener el control de uno o más procesos mientras se están ejecutando, de manera que reciba información sobre los siguientes acontecimientos:

- Ejecución de llamadas al sistema
- Recepción de señales
- Terminación del programa
- Entrada y salida de cada una de las funciones de biblioteca dinámica contra las que el programa esté enlazado

Para ello, en todo momento `ltrace` ha de tener el control sobre la ejecución de uno o más procesos, y esto ha de ser de manera completamente transparente a estos procesos, para evitar que se comporten de manera diferente si están siendo estudiados o no y por lo tanto evitar los errores conocidos como *heisenbugs* (errores en un programa que no se manifiestan cuando tratamos de depurarlo).

Hay dos maneras de comenzar a "tener el control" de un programa, dependiendo de la manera y el modo en el que se consigue ese control:

- Ejecutando un proceso nuevo y trazándolo desde el principio.

  Esta es la manera más habitual, y es la que se utiliza cuando ejecutamos `ltrace` con un nombre de programa en la línea de órdenes, usando algo como "`ltrace` *`nombre-de-programa`*".

- Trazando un proceso ya existente.

  Hay dos casos en los que `ltrace` comienza a controlar la ejecución de un proceso que ya existe, y los dos casos vienen determinados por el uso de sendas opciones:

- “-p”. Con esta opción se le indica específicamente a ltrace que debe comenzar a trazar un proceso determinado, que ya existe en el sistema.
- “-f”. Esta opción indica que se han de trazar todos los hijos que tengan los procesos que estemos controlando, con lo que en cuanto ltrace tenga constancia de que un proceso trazado ejecute la función fork() o alguna similar, se ha de avisar al Sistema Operativo para comenzar a trazar al nuevo proceso.

Las dos maneras difieren en cómo ltrace ha de informar al Sistema Operativo para tomar el control de los procesos, pero una vez que lo tiene, todos los procesos controlados se tratan de manera idéntica y se muestra la misma información en la salida.

En caso de que haya un solo proceso trazado, este no se identifica de ninguna manera; en cuanto hay dos o más, delante de cada línea se muestra un número de proceso para identificar el proceso al que se refiere.

5.4.2. Información mostrada a la salida

Como se ha dicho anteriormente, hay muchos eventos que pueden causar el mostrado de diversos tipos de información por parte de ltrace; estos eventos son:

- Comienzo de ejecución de una función o llamada al sistema

  En este caso se muestra el nombre de la función llamada, seguida de una serie de argumentos de entrada, separados por comas.

  Las llamadas al sistema se distinguen de las funciones de biblioteca porque comienzan siempre por “SYS_”.

  El hecho de que existan dos eventos distintos, uno para indicar el comienzo de una llamada y otro para indicar la terminación, se debe a que ambos no siempre son correlativos; por ejemplo, mientras se está realizando una llamada a una función de biblioteca puede recibirse una señal, o puede existir otra llamada anidada, etc.

  En caso de que este evento no sea inmediatamente seguido por el evento correspondiente de terminación de esta llamada, entonces la línea acaba con un “ <unfinished ...>”

```
textdomain("coreutils" <unfinished ...>
```

- Finalización de una llamada a función o al sistema

  Se terminan de mostrar los argumentos de la llamada, en caso de que faltara alguno por mostrar, y a continuación el valor de retorno de la función.

  Si este evento no ocurre inmediatamente después de la llamada a la función adecuada, entonces esta información aparece en una línea individual que comienza por "`<... función resumed>` ":

```
<... textdomain resumed> )          = "coreutils"
```

- Recepción de una señal

  Cada vez que un proceso controlado por `ltrace` recibe una señal, esta es mostrada por pantalla en una línea independiente, indicando el nombre "corto" de la señal ("`SIGINT`" o "`SIGSTOP`", por ejemplo), seguido por una breve explicación en inglés de ese nombre ("`Interrupt`" o "`Stopped`"). Una línea de recepción de una señal se indica con tres guiones:

```
--- SIGFPE (Floating point exception) ---
```

- Terminación del programa

  La terminación de un proceso controlado por `ltrace` también se indica en la salida del programa, especificando la causa de la terminación (bien por terminación normal o debido a una señal). Dicha terminación se denota con una línea que comienza por "+++":

```
+++ killed by SIGINT +++
+++ exited (status 0) +++
```

Independientemente del evento que esté mostrándose en cada momento, cada línea de la salida de `ltrace` puede venir acompañada de una serie de información adicional, dependiendo de las opciones que estén activas en la ejecución. Veamos qué tipo de información adicional puede aparecer en cada línea:

- Identificación del proceso

  En el caso de que estemos trazando más de un proceso simultáneamente (debido al uso de las opciones "`-p`" o "`-f`"), en cuanto aparezca una línea en la salida se identifica a qué proceso corresponde indicando su *PID*, mostrando por pantalla el texto "`[pid número]`". Ejemplo:

```
[pid 9070] getpid()                                    = 9070
```

- Momento en el que se ejecuta cada función

  Hay dos posibilidades para indicarle a `ltrace` que nos muestre cuándo se realiza cada función: de manera absoluta (usando la opción “`-t`”) y relativa (con la opción “`-r`”):

  `-t` Indica la hora del sistema expresando horas, minutos y segundos:

  ```
  [pid 9070] 10:56:15 getpid()                         = 9070
  ```

  `-tt` Indica la hora del sistema con horas, minutos, segundos y microsegundos:

  ```
  [pid 9070] 10:56:15.366533 getpid()                  = 9070
  ```

  `-ttt` Indica la hora del sistema indicando el número de segundos transcurridos desde el 1 de enero de 1970, y los microsegundos:

  ```
  [pid 9070] 1045562175.366533 getpid()                = 9070
  ```

  `-r` Indica los segundos y microsegundos transcurridos desde la última línea que se mostró:

  ```
  [pid 9070]   0.006044 getpid()                       = 9070
  [pid 9070]   0.004273 geteuid()                      = 1000
  ```

- Contenido del contador de programa

  La opción “`-i`” provoca que se muestre indica la dirección de memoria que se estaba ejecutando cuando sucedió el evento mostrado:

```
[pid 9070] 10:56:15 [0804833d] getpid()     = 9070
```

- Sangrado

  En caso de que se haya especificado la opción “`-n`”, se muestra un sangrado de uno o más espacios en blanco justo delante del contenido principal de la línea, para hacerlo más legible en caso de que haya llamadas anidadas:

```
[pid 9070] 10:56:15 [0804833d] getuid( <unfinished ...>
[pid 9070] 10:56:15 [400be527]     SYS_getuid() = 0
[pid 9070] 10:56:15 [0804833d] <... getuid resumed> ) = 0
```

- Tiempo dentro de la función

  Por último, usando la opción "-T" se muestra al final de las líneas de terminación de llamadas el tiempo transcurrido entre el comienzo y la terminación de dicha llamada, en segundos y microsegundos:

```
[pid 9070] 10:56:15 [0804833d] getuid()   = 0 <0.000081>
```

Capítulo 6

Conclusiones

En el curso de este Proyecto se ha realizado un profundo estudio de las soluciones de depuración existentes en la actualidad, no solo en cuanto a las herramientas disponibles sino también a los métodos y técnicas que éstas emplean.

Se ha estudiado asimismo con profusión el mecanismo de carga de bibliotecas dinámicas —y con ello el formato *ELF*— y el enlace dinámico entre éstas y el ejecutable. De esta forma se ha podido encontrar una manera de detectar qué llamadas a funciones de biblioteca realiza un programa dado, en qué momento y con qué parámetros.

Fruto de la combinación de esta técnica ideada en este proyecto y de las ya empleadas en los otros depuradores anteriormente estudiados (uso de `ptrace()`, *breakpoints*, etc…) se ha desarrollado una herramienta única hasta ahora que logra extender el popular `strace` para obtener la información, no sólo de las llamadas al sistema realizadas por un proceso y de las señales recibidas, sino también de las llamadas a funciones pertenecientes a bibliotecas dinámicas.

El programa se ha probado extensiva y satisfactoriamente, alcanzando gran aceptación internacional: ha sido incluido en las principales distribuciones de *GNU/Linux* (Debian, RedHat, SuSE, Mandrake), ha sido portado —en unas ocasiones por parte del propio autor, en otras por contribución e iniciativa de multitud de usuarios voluntarios— a otras arquitecturas, ya que inicialmente y debido a los detalles de tan bajo nivel que hay que considerar solo estaban soportados los procesadores *i386* de Intel, y ha despertado el interés y colaboración de particulares y empresas, entre las que cabe destacar la multinacional IBM.

6.1. Colaboradores

El autor principal de `ltrace` es **Juan Céspedes**, principal desarrollador del proyecto desde 1997.

Sin embargo, el programa completo, tal como es hoy en día, habría sido imposible sin la ayuda de decenas de colaboradores a lo largo del mundo, entre los cuales merecen especial consideración los siguientes:

- La **Free Software Foundation** ha contribuido en la filosofía del proyecto, la licencia, y muchas de las herramientas necesarias para desarrollarla (tales como `gcc`, `gdb`, `emacs`...).

1998 **Pat Bernie** adaptó el programa para su funcionamiento en los procesadores *ARM*.

1998 **Roman Hodek** adaptó el programa a procesadores de la familia *68000* de *Motorola*.

1998 **SiuL+Hacky** ha aportado varias ideas al desarrollo, hizo de *beta-tester* y escribió varios ensayos sobre el uso de `ltrace` [Siu], [Siu98], [Siu99].

2001 **Bert Driehuis** modificó `ltrace` para que funcionara en *BSD/OS 4.0* y en *FreeBSD*.

2001 **Silvio Cesare** cambió el manejo de ficheros *ELF* para hacerlo más robusto.

2001 **Morten Eriksen** ha corregido y mejorado en numerosas ocasiones varios aspectos del programa.

2002 **Anton Blanchard** lo adaptó para la plataforma *PowerPC*.

2002 **IBM Corporation** adaptó el programa a su plataforma *S/390*.

6.2. Líneas de trabajo futuras

Dada la envergadura del proyecto, es fácil encontrar aspectos en los que se puede trabajar y mejorar. Algunos de ellos serán enumerados a continuación.

6.2.1. Portabilidad

Existen 3 grandes limitaciones del programa en cuanto a portabilidad:

- Formato de los ejecutables
- Sistema Operativo
- Procesador

Seguidamente pasamos a describir más exhaustivamente cada una de estas limitaciones y cómo solventarlas.

Formato de los ejecutables

Actualmente, `ltrace` solo funciona en sistemas con ejecutables *ELF* de 32 bits. Esta limitación deja fuera de su alcance todos los sistemas que utilizan algún otro tipo de ejecutables (`a.out`, `.exe`, etc) y los sistemas que no sean de 32 bits, como los Alpha, S/390x, Itanium, etc.

Sistema Operativo

El único Sistema Operativo del que se tiene constancia de que `ltrace` funcione actualmente es *GNU/Linux*. Sin embargo, en el código no se utilizan demasiadas características específicas de este sistema, con lo que en teoría sería posible portarlo a cualquier otro sistema POSIX sin mucha dificultad.

El único aspecto de `ltrace` en el que se utiliza código que no se adhiere a POSIX y, por lo tanto, es dependiente del sistema operativo, es en la manera de controlar otros procesos: poder pararlos, solicitar ejecución paso a paso, insertar y eliminar puntos de ruptura (*breakpoints*), etc. Para ello se utiliza la función `ptrace()`, que no forma parte del estándar POSIX aunque sí forma parte de SVr4, SVID EXT, AT&T, X/OPEN y BSD 4.3. Sin embargo, a pesar de que su API es el mismo, la manera de interactuar con el sistema operativo es distinta, ya que esta función se usa para acceder directamente a muchas estructuras de control del sistema, y éstas son muy diferentes de unos sistemas a otros.

Sin embargo, dada la estructura del código, la parte dependiente del sistema está muy separada del resto y en principio no sería demasiado complicado portar el programa para que funcione en otros sistemas. De hecho, ya hay un grupo de trabajo tratando de portarlo a BSD/386 y a FreeBSD.

Procesador

Incluso dentro de un mismo Sistema Operativo, muchas veces hay diferencias considerables entre unas arquitecturas y otras (por ejemplo, entre una máquina Intel y una Sparc). Estas diferencias no suelen ser muy importantes para la mayor parte de los programas, e incluso pasan desapercibidas, pero `ltrace` necesita hacer uso de algunas características de muy bajo nivel que son diferentes entre unos procesadores y otros: la forma en que los argumentos se pasan a las funciones, la forma en que las funciones devuelven un valor, cómo poner puntos de ruptura (*breakpoints*) en un programa, etc.

Actualmente, `ltrace` funciona en 5 tipos distintos de procesadores: i386, arm, m68k, S/390 y PowerPC; en un futuro próximo, es previsible que funcione en S/390X, Alpha y Sparc.

6.2.2. Eficiencia

Dada la estructura interna de esta herramienta, se hacen muchos cambios de contexto con cada llamada a una función que se realiza en el programa a trazar: la diferencia en velocidad puede variar entre ser prácticamente imperceptible si el programa a trazar no realiza apenas llamadas a funciones de biblioteca y ser varios órdenes de magnitud más lento que si no se traza, en caso de programas que realizan muchas llamadas.

Desgraciadamente, la única manera de evitar que se produzcan tantos cambios de contexto consiste en que el control del proceso a trazar esté dentro del mismo proceso, y esa es una opción que ya se estudió, se usó y se rechazó anteriormente, principalmente por razones de falta de transparencia y de control.

Sin embargo, no todo está perdido: hay algo que tal vez sí se pueda hacer para mejorar la situación, y es realizar cambios no en este proyecto, sino en el núcleo del sistema operativo, específicamente en el código de *Linux*. El interfaz que proporciona `ptrace()` no es en absoluto el óptimo, y podría mejorarse considerablemente el rendimiento si se pudieran depurar otros procesos de una manera similar a la que proporciona *Solaris* con su interfaz de control utilizando el directorio `/proc`. Pero para poder lograrlo hay que realizar muchos cambios en el núcleo y no es una tarea fácil sino algo que requiere mucho tiempo y esfuerzo.

Presupuesto

Coste de los materiales

Se hace notar que en los gastos en medios técnicos no se incluyen gastos de *software*, ya que durante todo el proyecto se ha usado única y exclusivamente Software Libre.

Material de oficina	150 €
Gastos en medios técnicos (ordenadores, impresoras)	4.000 €
COSTE DE LOS MATERIALES	4.150 €

Coste de la mano de obra

Este proyecto ha sido llevado a cabo, íntegramente, por un Ingeniero Superior de Telecomunicación. Las diversas tareas llevadas a cabo en él, así como la duración estimada de las mismas, se reseñan a continuación a fin de calcular el coste de la mano de obra.

Tarea	**Horas**
Análisis del problema	10
Búsqueda de posibles soluciones	80
Primeros prototipos	200
Desarrollo del programa	180
Pruebas	250
Elaboración de la memoria	200
TOTAL	920

Un total de 920 horas, a 50 € por hora, resulta un coste de mano de obra de 46.000 €.

Presupuesto de ejecución material

Coste de los materiales	4.150 €
Coste de la mano de obra	46.000 €
PRESUPUESTO DE EJECUCIÓN MATERIAL	50.150 €

Presupuesto total

Concepto	Importe
Presupuesto de ejecución material	50.150 €
16 % de IVA	8.024 €
PRESUPUESTO TOTAL	58.174 €

Por tanto, el presupuesto total del presente proyecto asciende a la cantidad de:

CINCUENTA Y OCHO MIL CIENTO SETENTA Y CUATRO EUROS.

Madrid, 27 de mayo de 2003
EL INGENIERO PROYECTISTA

Fdo. Juan Céspedes Prieto

Apéndices

Apéndice A

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute

them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or

binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.*
> Copyright (C) *yyyy name of author*
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19*yy name of author* Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may

be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Apéndice B

Glosario de términos

386BSD Sistema Operativo libre basado en *BSD*, nacido en junio de 1992 de mano de William Jolitz y muchos otros contribuidores. Este sistema fue más adelante sustituido por *FreeBSD* y *NetBSD*.

68000 Modelo de procesadore de la firma *Motorola*. Es uno de los más usados en la universidad y tienen cabida en múltiples aparatos electrónicos gracias a su versatilidad y su bajo coste. Dentro del a familia del *Motorola 68000* hay varios ordenadores muy populares, como el *Atari* y el *Amiga*.

Alpha Familia de procesadores creada por *DEC* y que más tarde fue absorbida por *Compaq*.

API *Application Program Interface*. Conjunto de convenciones de programación que definen cómo se invoca un servicio desde un programa. (Fuente: RFC1208).

ARM *Advanded RISC Machines*. Firma líder en la industria en el desarrollo de procesadores RISC empotrados de 16 y 32 bits. Su uso se centra en comunicaciones móviles, agendas electrónicas, aplicaciones digitales multimedia y sistemas empotrados. Véase `http://www.arm.com/`.

Biblioteca dinámica Una biblioteca de funciones que se enlaza con los programas que dependen de ella en el momento en que estos se cargan en memoria para ser ejecutados, en lugar de hacerlo en la fase de compilación. Esta es la manera más habitual de usar bibliotecas hoy en día.

breakpoint Un punto en un programa que, al llegar a él, hace que el sistema lo pare y se comporte de una determinada manera útil para poder depurarlo.

brk() Llamada al sistema encargada de cambiar el tamaño del segmento de datos en memoria. Esta es la llamada que suele realizarse al pedir más memoria con una función como `malloc()`.

BSD *Berkeley Software Distribution*. Distribución de *software* para UNIX nacida en la Universidad de California en Berkeley alrededor de 1977 y que ha sido la base de muchos Sistemas Operativos comerciales, como *SunOS*, *ULTRIX*, y varios libres como *386BSD* y *FreeBSD*.

BSD/OS Versión comercial de *BSD* creado por la compañía *BSDI* en marzo de 1993.

C Lenguaje de programación diseñado por Dennis Ritchie en 1972 para la programación de sistemas del *PDP-11* y posteriormente utilizado para reimplementar *UNIX*. En 1989 fue revisado por el *American National Standards Institute* creando el *ANSI C*.

C++ Uno de los lenguajes de programación orientados a objetos más usado, diseñado originalmente por Bjarne Stroustroup en 1986 como sustituto del *C*.

clone() Llamada al sistema similar a `fork()`, encargada de crear un nuevo proceso partiendo de uno anterior, permitiendo que herede muchas de las características del padre. Esta llamada es específica de *Linux*.

Cygnus Solutions Empresa fundada en California en 1989, dedicada al desarrollo de aplicaciones de *software* libre.

Debian Sistema Operativo libre, mayor distribución de *GNU/Linux* del mercado, con más de 8710 paquetes de *software* creados por un grupo de más de 1200 desarrolladores distribuidos por todo el mundo. Véase la página principal de *Debian* en `http://www.debian.org/`.

debug Término inglés muy usado en la informática, significando depuración, búsqueda o eliminación de errores en un programa.

DIT Departamento de Ingeniería de Sistemas Telemáticos. Es uno de los departamentos más grandes e importantes de la Universidad Politécnica de Madrid (UPM). Véase `http://www.dit.upm.es/`.

ddd *Data Display Debugger*. Este programa es un *front-end* gráfico para depuradores como GDB, DBX, WDB, Ladebug, JDB, XDB, el depurador de Perl y el de Python. Es parte del proyecto GNU. Más información en `http://www.gnu.org/software`.

electric-fence Depurador de las funciones de la libc encargadas de reserva de memoria, tales como `malloc()`, `realloc()` y `free()`, escrito por Bruce Perens (`http://www.perens.com/FreeSoftware/`).

ELF *Executable and Linkable Format*. Formato de ficheros binarios introducido por *System V* y utilizado por la mayor parte de los sistemas *UNIX* en la actualidad.

fork() Llamada al sistema de *UNIX* encargada de crear un proceso nuevo, prácticamente idéntico al del proceso padre que lo ha creado.

Fortran *FORmula TRANslation*. El primero de los lenguajes de programación para aplicaciones numéricas y científicas (date de 1954). Durante mucho tiempo fue el único que existió con estas características, y hoy en día sigue siendo muy usado en algunos entornos.

free() Función de la biblioteca estándar libc en *UNIX* encargada de liberar memoria anteriormente reservada con `malloc()`, `calloc()` o `realloc()`.

FreeBSD Sistema Operativo libre basado en *BSD 4.4-lite* para ordenadores *Intel* 80386 o compatibles. Se puede descargar libremente y conseguir más información en `http://www.freebsd.org/`.

front-end Término inglés para referirse a un programa que actúa como interfaz de otro, hablando directamente con él para permitir un uso más sencillo o más intuitivo, o para proporcionar usos ligeramente distintos al inicialmente previsto.

FSF *Free Software Foundation*. Fundación sin ánimo de lucro creada en 1985 y con sede en Boston, Massachussetts, dedicada a promover los derechos de los usuarios para usar, estudiar, copiar, modificar y redistribuir *software* y documentación. En particular, la *FSF* es cuna del *Proyecto GNU*. Véase `http://www.gnu.org/fsf/fsf.html`.

función hash Función utilizada para generar índices a partir de claves en una *tabla hash*.

GCC *GNU Compiler Collection*. Compilador del proyecto *GNU*. Escrito por Richard Stallman y otros, es un compilador extremadamente portable y eficiente. Actualmente existen *front-end*s de este compilador para C, C++, Objective-C, Fortran, Java, y Ada. Es probablemente el mejor compilador de C del mercado (`http://www.gnu.org/software/gcc/`).

GDB *GNU Debugger*. Depurador del proyecto *GNU*. Permite examinar lo que ocurre en el "interior" de un programa mientras se ejecuta y modificar su comportamiento, o saber lo que estaba haciendo en el momento en que deja de funcionar. Véase `http://www.gnu.org/software/gdb/`.

GID *Group ID*. Número identificador de un grupo de usuarios en sistemas *UNIX* y compatibles.

GNU *GNU's Not UNIX*. El Proyecto GNU fue fundado en 1984 por Richard Stallman para desarrollar un sistema operativo completo similar a *UNIX* pero compuesto exclusivamente de *software* libre. El principal uso del Proyecto GNU hoy en día es en una variante en conjunción con el kernel *Linux*; esta variante es lo que se conoce como sistema *GLU/Linux*. Más información en `http://www.gnu.org/`.

GNU/Linux Variante de un sistema *GNU* que utiliza un kernel *Linux*. La mayor parte de las distribuciones de sistemas operativos libres del mercado hoy en día son sistemas GNU/Linux, aunque muchas veces se las llama simplemente *Linux*. Se puede encontrar más información sobre estos sistemas y su nombre en `http://www.gnu.org/gnu/linux-and-gnu.html`.

GPL *General Public License*. Licencia creada por la *Free Software Foundation* para su *Proyecto GNU* y usada por la mayor parte de su *software*, además de por muchos otros autores de *software libre* pasa sus programas. Esta licencia garantiza las cuatro libertades que caracterizan al *software* libre y que todas las modificaciones y distribuciones sigan siendo libres. La definición de esta licencia está en `http://www.gnu.org/copyleft/gpl.html`.

hardware Las partes físicas, tangibles o materiales de un ordenador o algún otro sistema. Este término se usa para distinguir dichas partes fijas del *software* o los datos que el ordenador ejecuta, almacena o transporta.

i386 Acrónimo de *Intel 386*. Familia de procesadores de la firma *Intel* compatible con el modelo *80386*. Este fue el primer procesador de *Intel* con bus de datos y de direcciones de 32 bits. Tiene soporte interno de multiproceso gracias a su "modo protegido" y es usado por múltiples sistemas operativos.

IBM *International Business Machines*. Compañía internacional dedicada al mundo de la informática casi desde sus inicios, tanto al *software* como al *hardware*. Creó el primer ordenador con un procesador *Intel*, hoy líderes en el mercado. `http://www.ibm.com/`.

Intel Fabricante de microprocesadores desde 1971, creadora de la familia *80x86*, que junto con *IBM* formó parte de la familia de ordenadores con más éxito. `http://www.intel.com/`.

Insight Interfaz de usuario gráfico para *GDB*, escrito por *Red Hat* y *Cygnus* y liberado en 1996 bajo la licencia *GPL*. Se puede conseguir más información sobre *Insight* en `http://sources.redhat.com/insight/`.

Itanium Primer procesador de 64 bits de la familia *Intel*. Más información en `http://intel.com/products/server/processors/server/itanium/`.

Java Lenguaje de programación simple, orientado a objetos, distribuido, interpretado, robusto, seguro, intependiente de la arquitectura y de propósito general creado por *Sun Microsystems* en 1955. Más información en la página `http://java.sun.com/`.

LaTeX Sistema de preparación de documentos creado por Leslie Lamport. LaTeX añade muchas funcionalidades a TeX a la hora de procesar textos, para permitir que el usuario pueda concentrarse en la estructura del texto en lugar de las órdenes de formato. [Lam94]

libc Biblioteca de funciones estándar para programas escritos en *C* en sistemas *POSIX*. La versión de la *libc* creada por el proyecto *GNU* se denomila `glibc`, y puede ver en `http://www.gnu.org/software/glibc/`.

Linux Implementación del kernel de un sistema operativo creado inicialmente como un hobby por un estudiante, Linus Torvalds, en la Universidad de Helsinki en Finlandia. Linus comenzó el desarrollo en 1991, momento en que se liberó la versión 0.02, y continúa hasta hoy, junto con un grupo de cientos de voluntarios a lo largo del mundo. Más información en `http://www.linux.org/`.

ltrace Herramienta para sistemas *POSIX* que permite mostrar las llamadas a funciones de bibliotecas dinámicas que va realizando un determinado programa [Cés03].

m68k Acrónimo de *Motorola 68000*. Familia de procesadores de la firma *Motorola* compatible con el modelo *68000*, de 16 y de 32 bits. Esta es una serie de procesadores muy versátiles y de bajo coste, y ha sido usado en muchos ordenadores de bajo precio como el *Apple*, el *Amiga* y el primer *Macintosh*.

malloc() Función de la biblioteca estándar libc en *UNIX* encargada de reservar una cantidad determinada de memoria para poder usarla en cualquier otro

punto del programa. Se puede cambiar el tamaño de esta zona de memoria con `realloc()` o liberarla después de su uso con `free()`.

Microsoft Windows Sistema Operativo creado por Microsoft Corporation, líder en el mercado pero muy distinto en filosofía y diseño al resto de los sistema habituales, basados en el estándar POSIX. Más información de este producto en `http://www.microsoft.com/windows/default.mspx`.

Modula-2 Lenguaje de programación de alto nivel diseñado por Niklaus Wirth en 1978. Está basado en el *Pascal*, añadiendo el concepto de *módulo* y permitiendo programación en paralelo.

Motorola Una de las empresas líderes mundiales en sistemas electrónicos avanzados, telefonía, semiconductores y servicios. Creadora de la familia de procesadores *m68k*. Véase `http://www.motorola.com/`.

Pascal Lenguaje de programación descendiente del Algol, diseñado por Niklaus Wirth alrededor de 1970. Fue diseñado con el fin de conseguir simplicidad y un lenguaje adecuado para la enseñanza.

PIC Lenguaje gráfico diseñado por Brian Kernighan, con la finalidad de describir textualmente imágenes para ser procesadas por `troff`.

PID *Process ID*. Número identificador de un proceso en sistemas *UNIX* y compatibles.

POSIX *Portable Operating System Interface*. Serie de estándares definidos por el *IEEE* para proporcionar portabilidad a nivel de aplicaciones entre las distintas variantes de ssitemas *UNIX*. El estándar "IEEE 1003.1" define el interfaz del sistema operativo; el "IEEE 1003.2" define la *shell* y las utilidades y el "IEEE 1003.4" define las extensiones de *tiempo real*. Más información en `http://std.dkuug.dk/JTC1/SC22/WG15/`.

PowerPC Procesador *RISC* diseñado conjuntamente por *Motorola*, *IBM* y *Apple*. Este procesador se usa habitualmente en sistemas empotrados y de bajo consumo como calculadoras, agendas electrónicas y ordenadores de mano.

printf() Función de la biblioteca estándar de C encargada de formatear un texto para mostrarlo por la salida estándar, normalmente la pantalla. Esta es probablemente la función más conocida y más usada en C.

ptrace() Llamada al sistema presente en muchas versiones de *UNIX* y en Linux que proporciona un método de observar y controlar la ejecución de otro proceso, y de examinar y cambiar su imagen y sus registros. Su principal

uso es para facilitar el depurado de programas y para controlar las llamadas al sistema.

Red Hat Una de las principales distribuciones de *GNU/Linux*. Hoy en día es la más usada, y *Red Hat Software* es el mayor y más reconocido proveedor de *software libre*. Su página principal es `http://www.redhat.com/`.

S/390 Línea de grandes ordenadores (*manframes*) desarrollada por *IBM*. *S/390* es la arquitectura más fiable de *IBM*. Algunas distribuciones de *GNU/Linux*, como *Debian* tienen sus distribución preparada para esta arquitectura, e incluso el propio *IBM* ofrece soporte para este tipo de adaptaciones. Véase `http://www.s390.ibm.com/`.

SDB *Debugger* de System V. Este es un depurador simbólico diseñado para estudiar programas escritos en lenguaje ensamblador, en *C* y en *FORTRAN*.

scanf() Función de la biblioteca estándar de *C* encargada de leer datos de la entrada formateados de una determinada manera y guardarlos en variables de distinto tipo.

set-gid Atributo que pueden tener algunos ficheros en sistemas *UNIX* y que consiste en que en el momento de ejecutar ese fichero, se ejecuta cambiando el *UID* efectivo al del dueño del fichero, en lugar de ser el del proceso que lo ejecuta como sería lo normal.

set-uid Atributo para ficheros en sistemas *UNIX* consistente en que a la hora de ejecutar un fichero con ese atributo, el *GID* efectivo de ese proceso pasa a ser el del grupo del fichero en lugar del *GID* del proceso padre.

shell Intérprete de órdenes, usado para darle instrucciones al *Sistema Operativo* sobre lo que debe hacer en cada momento. Las *shells* más comunes en *UNIX* son la *Bourne shell* (“`sh`”) y la *C Shell* (“`csh`”).

software Programas, procedimientos o reglas escritos para ser ejecutados por un ordenador. Esta palabra suele usarse en contraposición con *hardware*, que se refiere al dispositivo físico en el que estos programas se ejecutan.

Solaris Sistema Operativo *UNIX* desarrollado por *Sun Microsystems*. Es el nuevo nombre que tienen las versiones modernas de *SunOS* (a partir de la 4.1.2), y hoy en día es probablemente el *UNIX* más extendido del mercado. Véase `http://wwws.sun.com/software/solaris/`.

SPARC *Scalable Processor ARChitecture*. Arquitectura *RISC* diseñada por *Sun Microsystems* para su propio uso en 1985. Manteniendo su filosofía abierta,

permitió que otras compañías fabricaran procesadores *SPARC*. Su evolución y estandarización está dirigida ahora por *SPARC International, Inc.* (`http://www.sparc.com/`).

strace Programa diseñado para interceptar y registrar las llamadas al sistema realizadas por un proceso, y las señales que recibe [Akk]. Fue creado por Paul Kranenburg para *SunOS*, y posteriormente ampliado y portado a múltiples plataformas por varios voluntarios.

SunOS *SUN Operating System*. Sistema Operativo *UNIX* desarrollado por *Sun Microsystems*. Está basado en *BSD*, con algunas de las características de *SVr4* y con el sistema de ventanas *OpenWindows*. Después de la versión 4, *SunOS* pasó a llamarse *Solaris*.

SVr4 *System V Release 4*. Versión 4 de *System V*, la más conocida y usada, y en la que se basan muchis sistemas *UNIX* modernos.

System V Una de las primeras y principales versiones de *UNIX*, junto con la de *BSD*. Las siguientes versiones que fueron surgiendo,, como *SunOS*, combinaron las mejores características de los *UNIX* de *System V* y *BSD*.

tabla hash Esquema para proporcionar acceso rápido a datos. Cada dato tiene asociada una *clave*, y para identificar la posición de un dato en concreto se aplica una determinada *función hash* a la clave del dato en cuestión. Esta función proporciona un índice para encontrar el dato en la tabla.

TEX Procesador de texto basado en macros, extremadamente versátil y potente, creado por Donald E. Knuth entre 1978 y 1985, muy popular en entornos académicos y entre la comunidad informática.

truss Programa trazador de llamadas al sistema para sistemas operativos SunOS y Solaris, predecesor de `strace`.

UID *User ID*. Número identificador de un usuario en sistemas *UNIX* y compatibles.

UNIX Sistema Operativo multiusuario y multiproceso creado en 1969 por Ken Thompson y Dennis Ritchie después de que "Bell Labs" abandonara el proyecto *Multics*. En 1974 se reescribió en *C*, convirtiéndolo en el primer sistema operativo portable a otras arquitecturas. En 1991 llegó a ser el sistema operativo multiusuario de propósito general más usado del mundo.

UPM *Universidad Politécnica de Madrid*. Fundada en 1971, fue la sede de todos los estudios técnicos en Madrid y recoge hoy más de 20 centros. Más información en `http://www.upm.es/`.

X/Open Consorcio internacional de empresas que definen un entorno para permitir la portabilidad de aplicaciones, el "*X/Open Common Applications Environment*" (`http://www.opengroup.org/`).

XDB Depurador estándar para el Sistema Operativo *HP-UX*.

xxgdb Interfaz de uso del depurador `gdb` para el sistema de ventanas "X-Window System".

Bibliografía

[Akk] Wichert Akkerman. *Homepage for* `strace`*: a system call tracer*. URL: `http://www.wi.leidenuniv.nl/~wichert/strace`.

[Boe81] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[Cha91] Steve Chamberlain. *The Binary File Descriptor Library*, April 1991.

[Com95] TIS Committee. *Executable and Linking Format (ELF) Specification, version 1.2*, May 1995.

[Cés03] Juan Céspedes. Trazado de llamadas a funciones de biblioteca dinámica. Technical report, Departamento de Ingeniería de Sistemas Telemáticos, Abril 2003.

[Fou] Free Software Foundation. URL: `http://www.fsf.org/`.

[Fou00] Free Software Foundation. *DDD - Data Display Debugger*, January 2000.

[Fre01] Free Software Foundation. *The GNU C Library Reference Manual*, July 2001.

[Gil92] Daniel Gilly. *UNIX in a nutshell*. O´Reilly & Associates, 2nd edition, June 1992.

[GNU] Proyecto GNU. URL: `http://www.gnu.org/`.

[Kap95] C.R. Clark y V. Tang Kaplan. *Secrets of Software Quality: 40 Innovations from IBM*. McGraw-Hill, 1995.

[Lam94] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison Wesley Professional, second edition, 1994.

[Lew91] Donald Lewine. *POSIX Programmer's Guide*. O´Reilly, April 1991.

[Mic98] Sun Microsystems. *truss – trace system calls and signals*, July 1998. SunOS 5.8.

[oNSCL] College of Natural Sciences Computing Laboratories. xxgdb debugger. Technical report, University of Northern Iowa, Cedar Falls, IA 50614.

[S+00] Richard Stallman et al. *Debugging with GDB*. Free Software Foundation, Inc., March 2000.

[Siu] SiuL+Hacky. *About Introducing Your Own Code*. URL: `http://www.woodmann.com/fravia/siul_333.htm`.

[Siu98] SiuL+Hacky. *Ltrace. The Tool*, July 1998. URL: `http://www.woodmann.com/fravia/siullin2.htm`.

[Siu99] SiuL+Hacky. *Cracking Bajo Linux II*, February 1999. URL: `http://www.hackuma.com/textos/e-zines/set/set18.txt`.

[Sol] Lucent Technologies Technology Licensing Solutions. *ltrace (LEX tracer)*. URL: `http://www.lucentssg.com/ltrace.html`.

[SS] Red Hat Software and Cygnus Solutions. *Insight: The GDB GUI*. URL: `http://sources.redhat.com/insight/`.