Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

Informática I

Departamento de Ingeniería Electrónica

Recursos de programación avanzados en Linux

Interacción del Sistema Operativo y las herramientas de programación. Desarrollo de aplicaciones

Índice general

Índice de figuras

Índice de cuadros

Listings

Parte I

Programamos sobre un Sistema Operativo ¿Lo sabía?

Capítulo 1

Estructura de un programa en Linux

1.1. Introducción

Cualquier curso de programación comienza con el paradigmático ejemplo “Hola Mundo”. Es como una tradición. Comparado con los programas de aplicación habituales, muchos de los cuales emplean interfaz gráfica de modo de darle un “look and feel” muy intuitivo, amigable, y en ocasiones muy agradable a la vista, el pobre “Hola mundo”, parece muy poco interesante. Sin embargo es muy útil para comprender los conceptos generales de programación y entorno que existen en su trasfondo, y que son los mismos que cualquiera de los demás programas.

1.2. Programa fuente

```c
#include <stdio.h>
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Listing 1.1: *hola.c*: Bautismo de fuego de cualquier programador

La primer línea del programa es una directiva para el compilador que le indica a éste la inclusión de un archivo denominado ***stdio.h***. Este archivo, contiene definiciones de funciones, macros y variables. En particular, necesitamos incluirlo ya que entre las funciones definidas en el mismo se encuentra ***printf***, que estamos invocando en nuestro programa y cuyo código objeto se encuentra en la librería estándar de C denominada ***libc***.
Todos los programas escritos en C deben incluir una función main, la que representa el punto de entrada (“entry point”), es decir, la primer instrucción que ejecutará el computador al cargar el programa en la memoria. En el lenguaje C se ha tomado como convención darle obligatoriamente este nombre a la función que se ejecutará al inicio del programa. En nuestro sencillo ejemplo la función main no toma ningún parámetro ya que este programa no se ejecutará con argumentos desde la línea de comandos. Por este motivo se coloca void (vacío) como argumento, al solo efecto de ser mas específicos ya que de no haber colocado nada entre los paréntesis no habría diferencia a efectos de la compilación. Esta función devuelve un entero. En el caso de la función principal del programa este entero se le devuelve al proceso padre que es quien ha invocado su ejecución, en este caso el shell del sistema operativo. Por

convención se retorna un número de 8 bits, que será 0 en caso de terminación normal del proceso, un valor n tal que $0 < n < 128$, para procesos que hayan terminado con alguna anomalía, y $n > 128$ para procesos terminados por medio de alguna señal[1]. Las líneas encerradas entre las llaves componen el programa en sí, que no hace otra cosa que imprimir Hola Mundo!, en la consola desde la que se invocó el proceso.

1.3. El compilador

Para compilar cualquier programa en Linux utilizamos el universalmente difundido GNU C Compiler (***gcc***) y sus herramientas asociadas que generalmente vienen en el paquete ***binutils***. Para compilar nuestro pequeño programa tipeamos desde la consola el siguiente comando:

```
1 $ gcc -c -o hola.o hola.c
```

Listing 1.2: Compilación de nuestro programa fuente

La opción -c le indica al ***gcc*** que compile solamente, es decir que genere un archivo objeto. Si no la incluimos el ***gcc*** seguirá automáticamente a la siguiente etapa, invocando al linker, para producir directamente el programa ejecutable. En ésta etapa del proceso de aprendizaje no es conveniente incurrir en el confort del automatismo, y en cambio trabajar las diferentes etapas para poder seguir de cerca cada pequeño detalle que será un gran aporte a nuestro conocimiento. La opción -o sirve para indicarle a ***gcc*** el nombre del archivo de salida (o de output). En este caso lo llamamos igual que el de entrada pero terminándolo en .o en lugar de .c, para darle la nomenclatura estándar de los archivos objeto. El comando file de Linux nos puede dar alguna idea de las características de este archivo objeto:

```
1 $ file hola.o
2 hola.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Listing 1.3: Averiguando que tipo de "file" es hola.o

El archivo objeto generado por ***gcc***, es del tipo reubicable. Un archivo objeto es reubicable cuando contiene referencias simbólicas de variables y/o funciones a direcciones relativas contenidas dentro de la Unidad de Compilación o fuera de ella. Al poder invocar variables o funciones externas a la Unidad de Compilación permite utilizar otros archivos objetos obtenidos a partir de otras unidades de compilación para proveer al linker de las interfaces a las rutinas para que pueda construir el programa ejecutable.
Una referencia simbólica en nuestro caso consiste en almacenar la dirección numérica correspondiente a una etiqueta que identifica a una variable o a una función. Por ejemplo: que dirección ocupará dentro del archivo la etiqueta main, o que habrá que colocar como dirección destino en la instrucción que efectúa la llamada a ***printf***. La Unidad de Compilación no es otra cosa que lo que estamos compilando, es decir, nuestro programa. Entonces como nuestro programa posee referencias simbólicas para acceder a ***printf*** y lograr imprimir Hola Mundo! en la pantalla, es entonces reubicable.
El archivo objeto contiene en su encabezado información para la reubicación. El Linker reemplazará la información simbólica con la información de dirección actual en el momento de construir el archivo binario ejecutable. En nuestro ejemplo, la llamada al código de la función ***printf***, será resuelto por el Linker ya que el ***gcc*** no conoce la dirección de comienzo de ***printf***, ya que este código no está definido en nuestro pequeño programa. El Linker la obtendrá del encabezado de la librería estándar de C, libc, contenida en el archivo /lib/x86_64-linux-gnu/libc-2.13.so para un kernel 3.2.0-3-amd64, o /lib32/libc-2.13.so para un kernel 3.2.0-3-686 (de 32 bits).

[1] ¿Recuerdan la orden kill? Kill envía una señal a través del Kernel al proceso que deseamos terminar o controlar. Es uno de los tantos ejemplos de señales. CTRL-C es otro con efecto similar.

La otra característica saliente de nuestro archivo objeto es el formato **ELF**[1] de 32 bits que contiene una tabla de símbolos que no ha sido removida. Se puede remover la tabla de símbolos con el programa strip.

```
$ strip hola.o
$ file hola.o
hola.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), stripped
```

Listing 1.4: Quitando etiquetas a hola.o

Como vemos el resultado del comando es la remoción (strip) de la tabla de símbolos.
No es este el único síntoma visible. Queda a cargo del lector mirar con ls -las el tamaño del archivo hola.o, antes de la orden strip, y después de la misma, para verificar que se reduce el tamaño de este archivo en unos 300 bytes. Esto es lógico ya que estamos quitando información de su encabezado.
El formato **ELF**puede variar en función del tipo de procesador que se trate: la salida de nuestros ejemplos corresponde a una PC de escritorio de arquitectura IA-32, y un S.O. Debian Lenny de 32 bits. Por lo tanto es **ELF**32.
Con el comando objdump podemos comprobar varias cosas adicionales del archivo **ELF**. En el siguiente vuelco de pantalla hemos utilizado las opciones -h, para que presente los headers de las diferentes secciones del archivo objeto hola.o, -r para que imprima las entradas de reubicación del archivo objeto hola.o, y t para que imprima la tabla de información simbólica de dicho archivo. captionsetup[lstlisting]format=listing,labelfont=white,textfont=white, singlelinecheck=false, margin=0pt, font=bf,footnotesize

```
$ objdump -hrt hola.o

hola.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000025  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000005c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000005c  2**2
                  ALLOC
  3 .rodata.str1.1 0000000c  00000000  00000000  0000005c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000001f  00000000  00000000  00000068  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  00000000  00000000  00000087  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 hola.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .rodata.str1.1 00000000 .rodata.str1.1
00000000 l    d  .note.GNU-stack 00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     F .text  00000025 main
00000000         *UND*  00000000 puts


```

```
31 RELOCATION RECORDS FOR [.text]:
32 OFFSET   TYPE              VALUE
33 00000012 R_386_32          .rodata.str1.1
34 00000017 R_386_PC32        puts
```

Listing 1.5: Volcando el contenido de hola.o con objdump

Vemos que hola.o tiene 5 secciones. Antes de seguir, veamos que son las secciones en un archivo **ELF**. Básicamente la organización de un archivo objeto en formato **ELF**contiene un encabezado en donde está el mapa de organización del archivo, y luego secciones en donde se agrupan, instrucciones que componen el programa, las variables que hemos definido en el programa, y eventualmente listas de símbolos y demás elementos que hacen a los aspectos del lenguaje en el que se escribió el programa original.
Cuando pasemos la fase del linker, el archivo ejecutable también tendrá el formato **ELF**, aunque con una estructura algo diferente. En particular las secciones en un objeto, se denominan segmentos en el ejecutable derivado del mismo. Aclarado este concepto, vamos a estudiar el contenido de cada sección:

.text , que contiene el código máquina compilado que imprime “Hola Mundo!” en la consola. El programa cargador de Linux (loader) toma esta sección y la copia en el segmento de código del proceso (que es el área de memoria en la que se carga la secuencia de instrucciones que componen el programa).

.data , que en este caso está vacío ya que nuestro sencillo ejemplo no hemos definido entidades en nuestro programa que se guarden en esta sección: variables globales y variables locales estáticas inicializadas. De otro modo aquí encontraríamos los valores iniciales de aquellas variables globales o locales estáticas que hemos definido en nuestro programa, con sus correspondientes valores iniciales definidos al momento de su declaración, las que serán copiadas con esos valores en el segmento de datos del proceso.

.bss , que también en este caso está vacía ya que nuestro sencillo ejemplo no tiene ninguna variable no inicializada (de hecho no tiene variables). De otro modo aquí encontraríamos aquellas variables locales o globales no inicializadas en nuestro programa al momento de su declaración. En tal caso indicaría cuantos bytes se deben allocar[2] y completar con ceros, en el segmento de datos del proceso, además de los ya alojados para la sección .data.

.rodata , que contiene la string ”HolaMundo!\n“, etiquetada como Read Only. En general las áreas de datos de los programa no son Read Only ya que los datos en general es de esperar que puedan modificarse durante la operación del programa. Sin embargo hay datos como las constantes de programa que definimos como macros con la directiva #define, o strings de mensajes que no es de esperar que se modifiquen que el compilador pone en una sección aparte para diferenciarlas de los datos comunes. Luego dependiendo del Sistema Operativo, pueden copiarse en el segmento de datos del proceso o en el segmento de código.

.comment , contiene 0x1f bytes de comentarios cuya relación no podemos determinar ahora ya que sencillamente no hemos escrito comentario alguno en nuestro código (tan sencillo es.....). Mas adelante determinaremos de donde proviene este valor.

A esta altura es oportuno proponer una primer experiencia para el lector: Compilar nuevamente el archivo hola.c, pero agregándole la opción -g para que el ***gcc*** incluya en el archivo objeto hola.o la información simbólica que, en caso de requerirse debug del programa, permita al debuger recorrer el texto del archivo fuente, en lugar de pasar por el código assembler que de otro modo sería el único que podría determinar. Una vez recompilado con -g, compara el tamaño resultante del nuevo archivo hola.o, y por sobre todas las cosas re ejecutar el comando objdump con idénticas opciones y comparar con el listado en este documento. ¿Hay cambios?

[2] O alojar, reservar espacio para. Sucede que esta como otras ciencias, la programación ha sido desarrollada en países de habla anglosajona, y en ocasiones los profesionales para simplificar la interpretación de los textos originales en Inglés, muchas veces castellanizamos la palabra sin cambios. De este modo todos saben de que se habla.

Volviendo a nuestra salida de objdump, podemos observar que aparece una tabla de símbolos, en la que el símbolo main ocupa el offset 0 y puts figura *UND* (por Undefined). Esta tabla indica como se efectuarán la reubicaciones en la sección .text de las referencias efectuadas a secciones externas. El primer símbolo reubicable corresponde a la string "Hola Mundo!\n" contenida en la sección .rodata, y el segundo símbolo reubicable, puts, corresponde a una función de la librería libc, que se ha generado como resultado de llamar a ***printf***.
Para entender mejor el contenido de hola.o, conviene analizar su código assembler. Para ello empleamos en ***gcc*** la opción -S para que genere el código assembler resultante, y -o – para que en lugar de guardarlo en hola.s, lo presente en la consola.

```
1
2 $ gcc -S hola.c -o -
3         .file   "hola.c"
4         .section        .rodata
5 .LC0:
6         .string "Hola mundo!"
7         .text
8 .globl main
9         .type   main, @function
10 main:
11         leal    4(%esp), %ecx
12         andl    $-16, %esp
13         pushl   -4(%ecx)
14         pushl   %ebp
15         movl    %esp, %ebp
16         pushl   %ecx
17         subl    $4, %esp
18         movl    $.LC0, (%esp)
19         call    puts
20         movl    $0, %eax
21         addl    $4, %esp
22         popl    %ecx
23         popl    %ebp
24         leal    -4(%ecx), %esp
25         ret
26         .size   main, .-main
27         .ident  "GCC: (Debian 4.3.2-1.1) 4.3.2"
28         .section        .note.GNU-stack,"",@progbits
```

Listing 1.6: Salida assembler de la compilación del archivo hola.c con -S

Podemos ver algunas cuestiones interesantes. Una es que en lugar de invocar a ***printf*** el código assembler llama a la función puts de la libc, como vemos en la línea que figura resaltada, utilizando la instrucción call, y pasando previamente justo en la instrucción anterior la etiqueta .LCO, que es el inicio de la string "Hola Mundo!\n" que presentará por la consola, y que también hemos resaltado en los dos puntos del programa en los que aparece. Además podemos ver de donde proviene la sección .comment en el objdump anterior: La ante última línea, es introducida automáticamente por el compilador. Volviendo a puts, el compilador debe poner en el campo de la instrucción call que corresponde a la dirección, el valor relativo al comienzo del programa que ocupará el código de puts cuando se arme todo el bloque de código. Cabe preguntarse como sabe cual es este valor. La respuesta es muy simple (y algo dramática): no lo sabe. Tal vez el comando nm nos pueda dar alguna pista.

```
1 $ nm hola.o
2 00000000 T main
```

```
3           U puts
```

Listing 1.7: Salida de nm del archivo hola.o

El comando nm lista los símbolos declarados en el encabezado de un archivo objeto, asumiendo que éste está en formato **ELF**. Básicamente la salida tiene tres columnas: En la primera de ellas se tiene el valor que colocará el compilador en reemplazo de la etiqueta en el momento de crear el archivo objeto y que corresponde a la distancia en bytes que habrá desde el lugar que ocupa esta etiqueta respecto del primer byte del segmento en el que se cargará nuestro programa. En la segunda columna se tiene el tipo de etiqueta (los diferentes significados de cada letra se obtienen con man nm), que determina si es mayúscula que la etiqueta corresponde a un símbolo Global / Estático, o minúscula si es una etiqueta local. En el caso de main, la T indica que está contenida en la sección Text, es decir, código. La tercer columna es la etiqueta definida en el programa. Lo mas importante: la etiqueta puts, figura con una U, que significa Undefined. Estas son las cosas que no puede resolver el compilador ya que trabaja sobre el archivo fuente especificado.

1.4. El Linker

El linker entonces es un programa que toma uno o más archivos objetos y los combina (linkea) en un archivo ejecutable. Los archivos objetos pueden provenir de diferentes archivos fuente compilados en nuestro proyecto, y también de librerías de objetos externas. Tal el caso de puts.
Llegó el momento de analizar como se genera un programa ejecutable a partir del objeto hola.o. Leyendo por encima las man pages de ***ld***, y asumiendo que no sería muy diferente de lo que requiere ***gcc***, nuestro primer intento es:

```
1  $ ld -o hola hola.o -lc
2  ld: warning: cannot find entry symbol _start; defaulting to 00000000080481a4
3  ls -las
4  total 20
5  4 drwxr-xr-x 2 alejandro alejandro 4096 dic 28 01:08 .
6  4 drwxr-xr-x 5 alejandro alejandro 4096 dic 26 18:33 ..
7  4 -rwxr-xr-x 1 alejandro alejandro 1932 dic 28 01:08 hola
8  4 -rw-r--r-- 1 alejandro alejandro   74 dic 26 19:22 hola.c
9  4 -rw-r--r-- 1 alejandro alejandro  860 dic 27 17:56 hola.o
10 $ ./hola
11 bash: ./hola: No existe el fichero o el directorio
```

Listing 1.8: Salida de nm del archivo hola.o

No hemos generado un archivo que pueda ser reconocido como ejecutable por el sistema operativo. Por eso el mensaje de error.
Regla Nº 1: Nunca lea por encima las man pages.
Regla Nº2: los primeros intentos rara vez nos llevan a buen puerto (en especial si violamos la Regla Nº1).
En algunas plataformas el uso de un linker es bastante trivial. Pero en los sistemas operativos “Unix like” (tal el caso de Linux), la estabilidad se suele pagar con algunas complejidades. Esto hace que el pasaje del rótulo main que está al inicio del programa a un punto de entrada binario para ser tomado por el sistema, no resulte una tarea trivial. Esta es probablemente una de las razones por las cuales el compilador ***gcc*** invoca por default en forma automática al linker. Para evitarnos algunas complejidades.
Esto es muy saludable cuando estamos en un ambiente profesional en donde ya hemos pasado por la etapa formativa, y ahora formamos parte de un equipo de profesionales especializados en desarrollo, contexto en el cual la productividad

es lo importante, y por ende echamos mano de cualquier simplificación posible, seguros de tener los conocimientos necesarios que nos permitan tomar el control cuando las simplificaciones no funcionen (lo cual ocurre mas a menudo de lo que el lector pueda imaginar).
Justamente es en la etapa formativa en donde las complejidades no deben esquivarse ya que junto con ellas se esquiva el conocimiento. Y esto nos dejaría muy mal preparados para la etapa profesional.
Entonces, hay que entender algunas cosas, antes de continuar.
El ***gcc*** arma un llamado a un módulo de GNU utils llamado ***collect2***, que hace las veces de linker. Para saber como lo arma, podemos observarlo incluyendo en la compilación la opción -v. Esta opción hace que el ***gcc*** imprima por stderr los comandos que arma para ejecutar las diferentes etapas de compilación, además de información sobre versiones y demás.Probemos entonces:

```
1 $ gcc -o hola hola.o -v
2 Using built-in specs.
3 Target: i486-linux-gnu
4 Configured with: ../src/configure -v --with-pkgversion='Debian 4.3.2-1.1' --with-
      bugurl=file:///usr/share/doc/gcc-4.3/README.Bugs --enable-languages=c,c++,fortran,
      objc,obj-c++ --prefix=/usr --enable-shared --with-system-zlib --libexecdir=/usr/lib
      --without-included-gettext --enable-threads=posix --enable-nls --with-gxx-include-
      dir=/usr/include/c++/4.3 --program-suffix=-4.3 --enable-clocale=gnu --enable-
      libstdcxx-debug --enable-objc-gc --enable-mpfr --enable-targets=all --enable-cld --
      enable-checking=release --build=i486-linux-gnu --host=i486-linux-gnu --target=i486-
      linux-gnu
5 Thread model: posix
6 gcc version 4.3.2 (Debian 4.3.2-1.1)
7 COMPILER_PATH=/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/
      usr/lib/gcc/i486-linux-gnu/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-
      linux-gnu/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-linux-gnu/
8 LIBRARY_PATH=/usr/lib/gcc/i486-linux-gnu/4.3.2/:/usr/lib/gcc/i486-linux-gnu/4.3.2/:/
      usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../lib/:/lib/../lib/:/usr/lib/../lib/:/
      usr/lib/gcc/i486-linux-gnu/4.3.2/../../../:/lib/:/usr/lib/
9 COLLECT_GCC_OPTIONS='-o' 'hola' '-v' '-mtune=generic'
10  /usr/lib/gcc/i486-linux-gnu/4.3.2/collect2 --eh-frame-hdr -m elf_i386 --hash-style=
       both -dynamic-linker /lib/ld-linux.so.2 -o hola /usr/lib/gcc/i486-linux-gnu
       /4.3.2/../../../../lib/crt1.o /usr/lib/gcc/i486-linux-gnu/4.3.2/../../../../lib/
       crti.o /usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o -L/usr/lib/gcc/i486-linux-gnu
       /4.3.2 -L/usr/lib/gcc/i486-linux-gnu/4.3.2 -L/usr/lib/gcc/i486-linux-gnu
       /4.3.2/../../../../lib -L/lib/../lib -L/usr/lib/../lib -L/usr/lib/gcc/i486-linux-
       gnu/4.3.2/../../.. hola.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-
       needed -lgcc_s --no-as-needed /usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o /usr/lib/
       gcc/i486-linux-gnu/4.3.2/../../../../lib/crtn.o
```

Listing 1.9: Salida de gcc cuando se compila con opción -v (verbose)

Bastante complejo por cierto. Nuestra parte de interés se reduce a COLLECT_GCC_OPTIONS. Es decir a la última parte. Si lo depuramos eliminando rutas redundantes y simplificando los paths, nos queda el siguiente comando:

```
1 $/usr/lib/gcc/i486-linux-gnu/4.3.2/collect2 --eh-frame-hdr -m elf_i386 --hash-style=
      both -dynamic-linker /lib/ld-linux.so.2 -o hola /usr/lib/crt1.o /usr/lib/crti.o /
      usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o -L/usr/lib/gcc/i486-linux-gnu/4.3.2 -L/
      usr/lib/gcc/i486-linux-gnu/4.3.2 -L/lib -L/usr/lib hola.o -lgcc --as-needed -lgcc_s
       --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i486-
      linux-gnu/4.3.2/crtend.o /usr/lib/crtn.o
```

Listing 1.10: línea simplificada (¿?) para linkeo invocando a ld...

Que funciona perfectamente generando el programa ejecutable hola, y que ahora si puede ejecutarse sin errores.
Lo mismo ocurre si en lugar de ***collect2*** utilizamos ***ld*** (queda a cargo del lector reemplazar ***ld*** por ***collect2*** para comprobarlo), que es el linker "oficial" por llamarlo de algún modo a la luz de la aparición del misterioso ***collect2***.
Es interesante discutir algunas de las opciones empleadas por gcc para invocar a ***collect2***. En principio el uso de la opción -dynamic-linker, denota que se trabajará con librerías de código dinámicas. Esta opción trabaja con un argumento que es el archivo que trabajará como linker dinámico en tiempo de ejecución. En nuestro caso el argumento de la línea de comandos es -dynamic-linker /lib/ld-linux.so.2. Este módulo es el linker dinámico por default de linux, que se encarga de cargar el programa ejecutado desde el prompt y resolver en tiempo de carga las dependencias con código de librerías dinámicas necesario.
Por si no queda claro que significa una librería dinámica, veamos las opciones con que contamos:
Linkeo estático: El linker coloca todo el código máquina adentro del ejecutable. Cuando linkeamos de esta forma todas las referencias se resuelven en tiempo de linkeo. Resultado: un archivo ejecutable con todo lo necesario (sin ninguna dependencia a resolver en el momento de su ejecución, pero muy voluminoso.
Linkeo Dinámico: En este modo algunas bibliotecas pueden ser compartidas. Es decir que el código de las mismas no va a aparecer dentro de nuestro ejecutable. Pero entonces vamos a necesitar que esas bibliotecas formen parte del sistema. En este modo hay algunas referencias que se resuelven al momento de cargar el programa para su ejecución. Este último es el caso en que se tiende a trabajar debido a que se genera código mucho mas compacto. Este es el caso de nuestro ejemplo.
Ampliaremos el tema librerías en la 2.
Aparecen en la línea de argumentos varios archivos objeto que corresponden a otras Unidades de Compilación que se necesitan incluir para que el linker pueda convertir a nuestro sencillo Hola Mundo en un ejecutable ELF, y determinar su punto de entrada. Estos son:
/usr/lib/crt1.o
/usr/lib/crti.o
/usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o
/usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o
/usr/lib/crtn.o
Estos objetos son imprescindibles para conectar nuestra etiqueta main con el punto de entrada "físico" por llamarlo de algún modo distintivo de nuestro programa. Son Unidades de Procesamiento que vienen con el sistema operativo y se utilizan junto con los objetos del paquete ***binutils***, para generar las aplicaciones.
Si se mira sus contenidos mediante el comando ***objdump***, siempre con las opciones -hrt, es posible ver que *crt1.0*, y *crti.o* tienen relación directa con main. En particular *crt1.o* tiene a main en su tabla de símbolos como "U" (undefined), y *crti.o*, tiene la etiqueta **`_start`**.
Para nas detalle en el Apéndice A se listan las salidas del mencionado comando para cada uno de los archivos.
Puede verse que los otros tres complementarios. Pero los dos mencionados tienen relación directa con el punto de entrada del programa (el cual se suele encontrar como entry point, su nombre original, en inglés).
Ahora tenemos un ejecutable que funciona, y hemos logrado desentrañar lo complejo del proceso de compilación y linkeo. Queda para el lector ejecutar el comando ***`objdump -hrt hola`***, para ver la diferencia entre la salida del objeto hola.o y del ejecutable hola. Tal vez esto de una dimensión de la cantidad de información que ha agregado el linker, tal vez irrelevante para nuestro programa, pero imprescindible para que este pueda ser ejecutado en Linux.
Idéntica conclusión se obtiene comparando sus tamaños con el comando ***`ls -las hola*`***

```
1 $ ls -las hola*
2 total 24
3 4 drwxr-xr-x 2 alejandro alejandro 4096 dic 30 16:53 .
```

```
4 4 drwxr-xr-x 5 alejandro alejandro 4096 dic 26 18:33 ..
5 8 -rwxr-xr-x 1 alejandro alejandro 6272 dic 30 13:45 hola
6 4 -rw-r--r-- 1 alejandro alejandro   74 dic 26 19:22 hola.c
7 4 -rw-r--r-- 1 alejandro alejandro  856 dic 30 16:53 hola.o
```

Listing 1.11: Salida de nm del archivo hola.o

Veamos que ocurre si ejecutamos ***nm*** sobre el programa ya linkeado. La salida es mucho mas amplia que la que se obtiene cuando se lo ejecuta sobre el objeto.

```
1 $ nm hola
2 080494b4 d _DYNAMIC
3 08049588 d _GLOBAL_OFFSET_TABLE_
4 0804848c R _IO_stdin_used
5          w _Jv_RegisterClasses
6 080494a4 d __CTOR_END__
7 080494a0 d __CTOR_LIST__
8 080494ac D __DTOR_END__
9 080494a8 d __DTOR_LIST__
10 0804849c r __FRAME_END__
11 080494b0 d __JCR_END__
12 080494b0 d __JCR_LIST__
13 080495a8 A __bss_start
14 080495a0 D __data_start
15 08048440 t __do_global_ctors_aux
16 08048320 t __do_global_dtors_aux
17 080495a4 D __dso_handle
18          w __gmon_start__
19 0804843a T __i686.get_pc_thunk.bx
20 080494a0 d __init_array_end
21 080494a0 d __init_array_start
22 080483d0 T __libc_csu_fini
23 080483e0 T __libc_csu_init
24          U __libc_start_main@@GLIBC_2.0
25 080495a8 A _edata
26 080495b0 A _end
27 0804846c T _fini
28 08048488 R _fp_hw
29 08048274 T _init
30 080482f0 T _start
31 080495a8 b completed.5706
32 080495a0 W data_start
33 080495ac b dtor_idx.5708
34 08048380 t frame_dummy
35 080483a4 T main
36          U puts@@GLIBC_2.0
```

Listing 1.12: Salida de nm del archivo hola

Vemos que se han agregado una cantidad importante de referencias y etiquetas. Ahora las etiquetas marcadas con **`U`** tienen la información acerca de la Unidad de Compilación desde la cual se han obtenido y completado las referencias. En el caso de la referencia a ***puts*** que ya había aparecido al correr ***nm*** sobre el objeto, figura **`ptus@@GLIBC_2.0`**, que identifica quien contiene la definición de esa etiqueta, en este caso ***Gnu LIBC***, lo que significa que la etiqueta es

externa.
Vale también la pena analizar la etiqueta remarcada en el listado que aparece en el ejecutable. Esta etiqueta referencia a la etiqueta **_start** que en general es el punto de entrada que el linker define para nuestro programa y que como vemos se ha definido a partir de Unidades de Compilación externas a nuestro programa hola.

1.5. Carga y ejecución

En los sistemas "UNIX like", o mas específicamente en aquellos que adhieren al estándar POSIX, existe claramente definida una jerarquía de procesos basada en lo que se conoce como parentesco. El árbol de procesos comienza con ***init***, cuyo ***PID*** es siempre 1 (Process ID = número de 16 bits mediante el cual se identifica unívocamente a un proceso).
Cada vez que se enciende una terminal o se abre una ventana de sesión ***init*** crea una instancia de ***/bin/login*** que será encargado de validar el user ID y la contraseña que el usuario que abrió una terminal ingresará en la misma para su ingreso al sistema. Si los datos ingresados por el usuario son válidos, ***login*** ejecuta las scripts de inicio personalizadas para el usuario y crea una instancia del programa shell definido para ese usuario, por lo general ***bash*** (Bourne Again SHell). ***bash***, es entonces el responsable de presentar al usuario el prompt ('$'para usuarios comunes, '#'para root).
Esto puede verificarse mediante el comando ***ps*** ejecutado en una consola de modo texto:

```
1 $ ps -f
2 UID          PID    PPID  C STIME TTY          TIME CMD
3 500         6602    3459  0 10:56 tty1     00:00:00 bash
4 500         6686    6602  0 10:58 tty1     00:00:00 ps -f
```

Listing 1.13: Salida de ps -f del archivo hola.o

Podemos observar que ***bash*** (PID=6602) es un proceso hijo del que tiene PID=3459. A su vez el comando ***ps -f*** ejecutado en esa misma consola de texto resulta ser hijo de ***bash*** ya que en sus datos figura el PID de ***bash*** en la columna PPID (Parent PID). La pregunta es ¿quien es el proceso 3459?. Con la opción ***e*** del comando ***ps*** para listar todos los procesos del sistema (no los de la consola que por default son los que se listan), mas los buenos oficios del filtro ***grep***, tenemos:

```
1 $ ps -ef  grep 3459UID PID PPID C STIME TTY TIME CMD500 3459 1 1 09:37 tty1 00:00:00 /bin/login500 6602 3459 0
    10:56 tty1 00:00:00 bash500 6686 6602 0 10:58 tty1 00:00:00 grep 3459
```

Listing 1.14: Salida de ps -ef |grep 3459 del archivo hola.o

aquí aparece ***login***, cuyo padre es ***init*** (PPID=1). De modo que al tipear en el prompt del sistema ***./hola***, estamos haciendo que ***bash*** cree un proceso hijo (que en principio es una réplica de sí mismo), acción que se lleva adelante invocando a la función ***fork ()***, al que luego reemplaza por la imagen de hola, acción que se logra mediante la familia de system calls ***exec()***. Un viaje... pero aún no es momento de emprenderlo.
Podemos comprobarlo mediante algunos comandos del sistema operativo. El comando ***strace*** permite trazar las llamadas a system calls del proceso que recibe como argumento y las señales recibidas por este. Tipeando:

```
1 $ strace -i hola
```

Listing 1.15: comando strace con opción -i de un ejecutable (en este caso, hola)

, hacemos que además de proveer la información provista por el comando, agregue el valor del puntero de instrucciones del procesador al inicio de la línea (opción ***-i***).
Omitimos volcar en este documento la salida de ***strace***, debido a su extensión. El lector la puede ver en su propia instalación.
La primer línea de su extensa salida es:

```
[b7f79424] execve("./hola", ["hola"], [/* 32 vars */]) = 0
```

Listing 1.16: Línea de la salida de strace -i del archivo hola referida a la invocación de *exceve*

Tipeando ***man execve*** puede verse que el primer argumento es el nombre del archivo binario que contiene el código por el que se debe reemplazar la réplica del proceso padre que invocó a ***fork ()***.
El segundo argumento es la lista de argumentos que se ingresan por línea de comandos. Como es de práctica es ***char * argv []***, y el tercero es la lista de variables de entorno que se deseen pasar al nuevo programa.
Si ***man execve*** funciona correctamente no regresa al programa invocador.
Las dos líneas finales del listado generado por ***strace*** son:

```
[b7f63424] write(1, "Hola mundo!\n"..., 12) = 12
[b7f63424] exit_group(0)                  = ?
```

Listing 1.17: Línea de la salida de strace -i del archivo hola referida a la invocación de *exceve*

La función ***write ()*** es producto del código invocado desde ***puts*** y que no hace mas que escribir en *stdout* (o sea en la pantalla), y ***exit ()*** es la salida del programa al proceso padre (es decir a ***bash***).
Para agregar algunos detalles interesantes el comando siguiente provee información de interés:

```
$ readelf -l hola

Elf file type is EXEC (Executable file)
Entry point 0x80482f0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004a0 0x004a0 R E 0x1000
  LOAD           0x0004a0 0x080494a0 0x080494a0 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x0004b4 0x080494b4 0x080494b4 0x000d0 0x000d0 RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
```

```
22    02   .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.
          version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
23    03       .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
24    04       .dynamic
25    05       .note.ABI-tag
26    06
```

Listing 1.18: Comando readelf (man readelf)

La salida del comando ***readelf*** permite ver la estructura completa del programa, que ahora tiene un formato similar al del archivo objeto pero con la vista de un ejecutable en lugar que la de un objeto.
La principal diferencia está en que lo que en el encabezado de un archivo objeto se denominan secciones en el encabezado de un ejecutable se denominan Segmentos. Esto puede verse claramente indicado en la salida de ***readelf***.
La salida se divide claramente en dos partes: El encabezado **ELF**y los segmentos. Ambas están relacionadas como veremos a continuación.
El tercer Program header,

```
1   02   .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.
         version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
```

Listing 1.19: Tercer program header de la salida de *readelf* hola

corresponde al segmento de código que es la sección **.text** generada en el programa objeto por el compilador. La columna offset define el lugar en el que inicia el segmento relativo al comienzo del programa, en nuestro caso 0x00000000, la columna **`Virtual Address`**[3] indica **`0x08048000`**, al igual que la Columna correspondiente a la **`Physical Address`**.
El valor de la columna **`FileSiz`** indica el tamaño del segmento en el archivo en disco y el de la columna **`MemSiz`** indica el tamaño que ocupará el segmento en la memoria del sistema. El valor de la columna **`Align`**, establece la alineación que llevará en la memoria (es decir a partir de que múltiplo de valores de address el Sistema Operativo cargará a ese segmento). El segmento de código correspondiente a la sección **.text** de nuestro programa, y los datos de solo lectura que el archivo **ELF**organiza en la sección **.rodata** se cargarán entonces en este segmento a partir de la dirección virtual **`0x08048000`**, alineado a direcciones múltiplo de 0x1000, es decir con sus 12 bits menos significativos en '0'[4]. Además de acuerdo a las necesidades aparecidas durante el proceso de linkeo es probable que en este segmento se carguen secciones de código y datos de solo lectura que incluya el linker en el **ELF**ejecutable. Tratándose entonces de código y datos para lectura solamente no es extraño que en la columna **`Flg`** de la salida del comando ***readelf***, este segmento esté marcado **`R`** (Read only), y **`E`** (Executable) en lugar de **`W`** (Writable).
El cuarto header corresponde al segmento **.data** del proceso y su interpretación responde a las mismas reglas del caso anterior. Veámoslo:

```
1   03       .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
```

Listing 1.20: Cuarto program header de la salida de *readelf* hola

La diferencia de tamaño entre el archivo (**`FileSiz`**) y la imagen en memoria (**`MemSiz`**) se debe a que se incluye en este segmento la sección **.bss**.

[3]La dirección Virtual es propia de Linux y puede ser la misma para todos los programas, ya que le asigna a cada uno un espacio virtual que luego el procesador traduce a direcciones físicas diferentes en el momento de cargarlo en memoria para su ejecución. Por eso el campo Dirección física que arroja ***readelf*** no tiene significado ya que esta dirección será calculada en tiempo de ejecución, por el procesador.

[4]Durante el proceso de traducción de dirección virtual a dirección física el procesador organiza a la memoria del sistema en páginas iguales de 4Kbytes de tamaño. De esto surge la necesidad de alinear a 0x1000.

No obstante en este programa dicha sección no contiene nada y se llena con ceros. Al igual que en el caso anterior se alinea a 0x1000, y como corresponde a un segmento de datos se marca **W** (Writable).
El quinto header (**DYNAMIC**) corresponde al proceso de linkeo (recordemos que el programa ***collect2*** realizó un linkeo a librerías dinámicas.

```
1  04     .dynamic
```

Listing 1.21: Quinto program header de la salida de *readelf* hola

En general los sistemas Linux poseen un file system mapeado en RAM directamente en tiempo de boot (no está presente en el disco rígido), denominado **/proc**.
En éste se almacena información de suma utilidad y que es administrada solamente por el kernel. Los usuarios sin embargo pueden ver su contenido y muchas veces obtener información sumamente útil de él.
En particular el kernel construye un directorio para cada proceso cuyo nombre es el PID del proceso, y dentro de este un árbol de archivos y directorios con información sumamente útil e interesante.
En particular el directorio */proc/[PID_de_nuestro_proceso]/maps* contiene información muy similar a la obtenida de la salida de ***readelf***.
Claramente, el árbol de directorio de cada proceso existe en **/proc**, solo durante la vida del proceso. En nuestro caso ésta es efímera ya que todo lo que hacemos es presentar la leyenda ”Hola mundo“, y terminar el programa. ¿Como podemos hacer para detener nuestro programa de modo tal de darnos el tiempo para explorar el directorio?. Una opción es debugearlo. Eso creará una instancia que estará disponible en memoria y con un PID asociado durante todo el tiempo que queramos.
Una opción (rústica pero muy potente) es utilizar el programa ***gdb*** (GNU DeBuger).
La secuencia de comandos necesarios y su salida es la siguiente:

```
1  $ gdb
2  GNU gdb 6.8-debian
3  Copyright (C) 2008 Free Software Foundation, Inc.
4  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5  This is free software: you are free to change and redistribute it.
6  There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
7  and "show warranty" for details.
8  This GDB was configured as "i486-linux-gnu".
9  (gdb) file hola
10 Reading symbols from /home/alejandro/InfoI/hola...done.
11 (gdb) break main
12 Breakpoint 1 at 0x80483b2
13 (gdb) run
14 Starting program: /home/alejandro/InfoI/first/hola
15
16 Breakpoint 1, 0x080483b2 in main ()
17 Current language:  auto; currently asm
18 (gdb)
```

Listing 1.22: Ejecución de *gdb*

Los tres comandos que se ejecutan una vez dentro de ***gdb*** son los que hemos resaltado en negrita, y que por orden de ejecución se encargan de cargar el ejecutable hola en el contexto de ***gdb***, colocar un breackpoint en la etiqueta main (es decir en el punto de inicio del programa), y ejecutarlo.

Obviamente al ejecutar el programa este se detendrá en el breackpoint hasta que continuemos ejecutándolo en forma completa o paso a paso.
Lo importante aquí es que el proceso está cargado en memoria y detenido, lo cual nos permite explorar su entrada en /proc. Pero antes debemos tener su PID.
Para ello la siguiente secuencia nos permite alcanzar el resultado:

```
$ ps -C hola -o pid
  PID
 5398
```

Listing 1.23: Salida de ps -C del programa hola

El comando ***ps*** como ya vimos lista los procesos de acuerdo con determinados criterios que podemos acotar mediante las opciones disponibles (como siempre ***man ps*** ampliará la información de las opciones del comando). En este caso ***-C*** indica que presente los procesos cuyo nombre coincida con la cadena que se tipea a continuación, en nuestro caso *hola*, y ***-o*** permite customizar la salida colocando a continuación el nombre de la(s) columnas(s) que deseamos presentar a la salida.

```
$ cat /proc/5398/maps
08048000-08049000 r-xp 00000000 08:02 3145866    /home/alejandro/InfoI/hola
08049000-0804a000 rw-p 00000000 08:02 3145866    /home/alejandro/InfoI/hola
b7dc5000-b7dc6000 rw-p b7dc5000 00:00 0
b7dc6000-b7f1b000 r-xp 00000000 08:02 294942     /lib/i686/cmov/libc-2.7.so
b7f1b000-b7f1c000 r--p 00155000 08:02 294942     /lib/i686/cmov/libc-2.7.so
b7f1c000-b7f1e000 rw-p 00156000 08:02 294942     /lib/i686/cmov/libc-2.7.so
b7f1e000-b7f23000 rw-p b7f1e000 00:00 0
b7f23000-b7f24000 r-xp b7f23000 00:00 0          [vdso]
b7f24000-b7f3e000 r-xp 00000000 08:02 278572     /lib/ld-2.7.so
b7f3e000-b7f40000 rw-p 0001a000 08:02 278572     /lib/ld-2.7.so
bff2b000-bff40000 rw-p bffeb000 00:00 0          [stack]
```

Listing 1.24: Contenido de la entrada maps del proceso correspondiente al programa hola en /proc

Aquí vemos el segmento de código en la primer línea, con las direcciones a partir de las cuales se ha papeado, y a continuación el segmento de datos (contiene las secciones **.data .bss** y el ***heap***).
A continuación una serie de líneas correspondientes a las librerías dinámicas con las que el linker enlazó nuestro breve código para poderlo convertir en algo ejecutable dentro de Linux, y en el final el stack. Este último no tiene correspondencia alguna con el formato **ELF**.

1.6. Finalización

Cuando nuestro programa ejecuta dentro de main la función ***return***, devuelve el control a una de las funciones de la librería dinámica contra la que se linkea nuestro programa para poder ejecutarse, y dicha función invoca a la system call ***exit ()*** pasándole el mismo argumento que lleva ***return*** en su código.
Por su parte ***exit ()***, actúa sobre el proceso padre del que ejecuta ***hola***, el cual está a su vez ejecutando una system call denominada ***wait ()***. Como resultado el proceso padre estará bloqueado en esa función esperando (waiting) a que el

proceso hijo, en nuestro caso ***hola***, finalice su ejecución.
De este modo nuestro proceso devolverá al sistema operativo de manera ordenada todos los recursos que se le han asignado y su padre (en este caso el proceso shell) quedará liberado en lo que a este proceso hijo se refiere.
Podemos mediante el comando ***strace***, desentrañar en parte el proceso de finalización, tipeando lo siguiente:

```
1 $ strace -e trace=process -f sh -c "hola; echo $?" > /dev/null
2 execve("/bin/sh", ["sh", "-c", "hola; echo 0"], [/* 32 vars */]) = 0
3 clone(Process 6142 attached
4 child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0
      xb7da96f8) = 6142
5 [pid  6142] execve("./hola", ["hola"], [/* 32 vars */] <unfinished ...>
6 [pid  6141] waitpid(-1, Process 6141 suspended
7  <unfinished ...>
8 [pid  6142] <... execve resumed> )      = 0
9 [pid  6142] exit_group(0)               = ?
10 Process 6141 resumed
11 Process 6142 detached
12 <... waitpid resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0) = 6142
13 --- SIGCHLD (Child exited) @ 0 (0) ---
14 waitpid(-1, 0xbf856c98, WNOHANG)       = -1 ECHILD (No child processes)
15 exit_group(0)                          = ?
```

Listing 1.25: Salida de strace para el proceso hola

El comando ***strace*** es sumamente potente, y por lo tanto involucra una cantidad de conceptos que aún no estamos en condiciones de abordar. Sin embargo, podemos ver en su salida todas las llamadas al sistema operativo que hizo nuestro proceso. Esto es debido al empleo en el comando de la opción ***-e trace=process***, que le indica que se presente a la salida del comando todas las system calls y señales involucradas.
La opción ***-f*** sirve para especificar allí el comando que queremos ejecutar, y que el proceso ***strace*** siga las system calls y señales del proceso especificado y de sus procesos hijos. En nuestro caso lo que hemos hecho es ejecutar una instancia del shell (***sh***) con la opción ***-c*** para que el comando lo tome de la string de caracteres contigua en lugar del *stdin* (teclado).

1.7. Conclusiones

El objeto de esta guía es que se entienda la complejidad oculta detrás de cualquier aplicación y los recursos del sistema operativo que entran en juego, para la ejecución de una aplicación cualquiera, independientemente del lenguaje en el que ésta se haya programado. Lo que aquí vimos ocurre tanto para un programa escrito en C como para una applet escrita en Java, o scritp escrita en Phyton. El proceso es siempre el mismo.

Capítulo 2

Bibliotecas de código en Linux

2.1. Que es una Biblioteca

2.1.1. Definición

Una biblioteca de programas no es otra cosa que un archivo que contiene código y datos compilados y funcionales, que serán incorporados a otros programas cuando estos los requieran.

2.1.2. Porqué usar Bibliotecas

Las Bibliotecas de código facilitan la reutilización de código evitando tener que re escribirlo cada vez que lo necesitamos. Esto además de mejorar nuestra productividad como programadores, es una herramienta muy potente para facilitar el trabajo en equipo, permitiéndonos abstraernos de la implementación de un determinado código y enfocarnos solo en su utilización. Normalmente el código de una Biblioteca es de uso general, como por ejemplo *printf*. Esta función tomada como ejemplo, es muy general, y como sabemos nos permite acceder a la pantalla para imprimir en ella lo que se nos ocurra y con el formato que necesitemos. Cada vez que la invocamos en nuestros programas, no tenemos que preocuparnos ni siquiera de conocer su código. Todo lo que tenemos que tener es el archivo de la Biblioteca que tiene el código de *printf* presente en nuestro sistema e indicarle al linker en donde está ese archivo para que pueda resolver las llamadas que hacemos desde nuestro código a *printf* . En términos generales necesitamos tener instalados en nuestro sistema, los archivos que contienen las funciones de código empaquetadas en formato de Biblioteca, y el archivo cabecera (típicamente con extensión *.h*) con la definición de la función. En el caso de *printf*, nuestro programa necesita del archivo *stdio.h* que contiene, entre otros, el prototipo de la función *printf*. Resumiendo, para escribir el código de una Biblioteca necesitamos dos clases de archivo:

- Prototipos de las funciones (archivo/s cabecera .h)
- Definición de las funciones (archivos/s fuente .c)

2.1.3. Ejemplos de trabajo

A lo largo de esta guía vamos a trabajar con estos archivos fuente. Archivo *holalib.h*:

```
1 /* Header para holalib.c
2 */
3 #ifndef HOLA_H
4 #define HOLA_H
5 # include <stdio.h>
```

```
6 void hola (void);
7 #endif
```

Listing 2.1: Salida de strace para el proceso hola

Notar que incluimos una compilación condicional aprovechando los buenos oficios del preprocesador.
Normalmente los proyectos informáticos incluyen múltiples archivos fuente, de modo de compilar por separado y unirlos en un ejecutable por medio del linker.
En esas condiciones si se incluyen archivos headers que duplican includes, podemos arribar a errores de linkeo por re definición de elementos (en especial si se definen estructuras).
Para evitarlo chequeamos si ha sido definida una macro (en este caso **HOLA_H**). Si no fue definida, la definimos y luego definimos todos los prototipos y macros que necesitemos. De este modo, si otro archivo fuente del proyecto incluye este archivo header, entonces encontrará definida la macro **HOLA_H**, y salteará la re definición de los prototipos y macros evitándonos posibles errores de compilación.

Archivo *holalib.c*:

```
1 /* Programa holalib.c
2 Ejemplo para generar una Biblioteca.
3 Funcion: hola
4 Imprime en stdout Hola Mundo!
5 */
6 #include "holalib.h"
7 void hola (void)
8 {
9    printf ("Hola Mundo!\n");
10 }
```

Listing 2.2: Salida de strace para el proceso hola

Observar en holalib.c que no hay función `main()`;
Y *test_holalib.c* que será el programa de prueba de nuestra función de Biblioteca

```
1 /* Programa de prueba para nuestra Biblioteca holalib
2 */
3 # include "holalib.h"
4 int    main (void)
5 {
6   hola ();
7   return 0;
8 }
```

Listing 2.3: Salida de strace para el proceso hola

2.2. Clasificación

2.2.1. Tipos

1. Estáticas
2. Compartidas (Shared)
3. De carga dinámica (DL, del inglés Dynamic Loaded)

En el código que compone los programas objeto de una Biblioteca (independientemente del tipo que esta sea), nunca se tiene una función *main*, ya que el código que incluimos en las Bibliotecas es para ser invocado desde programas de aplicación los cuales necesitan indispensablemente una función *main*. Por lo tanto, el código que compone una Biblioteca, no representa un programa que pueda ejecutarse en forma directa, sino que será invocado desde programas que dispongan de las llamadas adecuadas a ellos.

2.2.2. Bibliotecas Estáticas

Son simples colecciones de programas objeto agrupados en un único archivo cuyo nombre típicamente finaliza en '.a'.
Al compilar un programa que utiliza código contenido en una Biblioteca estática, en el momento de realizar la fase de enlace (link), se copia en nuestro programa el código objeto de la Biblioteca (luego lo verificaremos observando el tamaño en bytes final de nuestro programa), es decir que no es necesario distribuir nuestro programa con la Biblioteca ya que el código que utilizamos de ella se encuentra incrustado en nuestro nuevo programa.
Para generar una Biblioteca estática utilizamos el utilitario *ar* (empaqueta archivos objetos en un único archivo). Antes de ello obviamente debemos haber generado el programa objeto, compilando con la opción *-c*.

```
1  alejandro@notebook:~/InfoI/second$ gcc -c holalib.c
2  alejandro@notebook:~/InfoI/second$ ls -las
3  total 24
4  4 drwxr-xr-x 2 alejandro alejandro 4096 feb 14 12:45 .
5  4 drwxr-xr-x 6 alejandro alejandro 4096 feb 14 12:23 ..
6  4 -rw-r--r-- 1 alejandro alejandro  209 feb 14 12:29 holalib.c
7  4 -rw-r--r-- 1 alejandro alejandro   44 feb 14 12:25 holalib.h
8  4 -rw-r--r-- 1 alejandro alejandro  836 feb 14 12:45 holalib.o
9  4 -rw-r--r-- 1 alejandro alejandro  121 feb 14 12:32 test_holalib.c
10 alejandro@notebook:~/InfoI/second$ ar rcs libhola.a holalib.o
11 alejandro@notebook:~/InfoI/second$ ls -las
12 total 28
13 4 drwxr-xr-x 2 alejandro alejandro 4096 feb 14 12:46 .
14 4 drwxr-xr-x 6 alejandro alejandro 4096 feb 14 12:23 ..
15 4 -rw-r--r-- 1 alejandro alejandro  978 feb 14 12:46 libhola.a
16 4 -rw-r--r-- 1 alejandro alejandro  209 feb 14 12:29 holalib.c
17 4 -rw-r--r-- 1 alejandro alejandro   44 feb 14 12:25 holalib.h
18 4 -rw-r--r-- 1 alejandro alejandro  836 feb 14 12:45 holalib.o
19 4 -rw-r--r-- 1 alejandro alejandro  121 feb 14 12:32 test_holalib.c
```

Listing 2.4: Salida de strace para el proceso hola

El comando ar se ejecutó con tres opciones: *rcs*.
r indica que reemplace al elemento previamente existente con el mismo nombre. En nuestro caso si existía dentro de la Biblioteca *libhola.a* un elemento llamado *holalib.o*, *ar* lo pisa por este nuevo.
c indica que cree el archivo de Biblioteca especificado si no existiere.
s le indica actualizar el índice de la Biblioteca una vez inserto el archivo solicitado.
Si queremos listar los archivos (objetos) que componen la Biblioteca, el comando es:

```
1 alejandro@notebook:~/InfoI/second$ ar -t libhola.a
2 holalib.o
3 alejandro@notebook:~/InfoI/second$
```

Listing 2.5: Salida de strace para el proceso hola

Tenemos todo en orden. Cabe un solo comentario antes de continuar. Las Bibliotecas en general llevan en su nombre el prefijo “lib”(por el término original del idioma inglés que las identifica: ***library***). Por lo tanto, a los efectos del linker, el nombre de la Biblioteca es lo que sigue a “lib” y precede a “.a”

```
1 alejandro@notebook:~/InfoI/second$ gcc -g -o demo test_holalib.o -L. -lhola
2 alejandro@notebook:~/InfoI/second$
```

Listing 2.6: Salida de strace para el proceso hola

aquí vemos que llamando simplemente *hola* a nuestra Biblioteca es suficiente para el linker. Hemos comprobado (no sin sufrir considerablemente) que si no se antepone “lib” al nombre del archivo que contiene la Biblioteca, el linker (que tiene su sensibilidad, no vayan a creer) se ofusca y devuelve un mensaje de error abortando el proceso de generación de nuestro ejecutable.
Además es necesario comprender como funciona este proceso a la perfección para evitar dolores de cabeza y sacarle máximo provecho. Al respecto ya vimos en la Guia001, que *gcc* arma una lista de opciones para invocar al linker (contenido por el archivo *ld* aunque en los mensajes utilice el alias *collect2*). Cuando queremos enviar en la línea de gcc opciones que son para el linker éstas van al final. De otro modo, *gcc* (otro miembro honorario del clan de los utilitarios sensibles) sencillamente se ofuscará y enviará al linker las cosas que éste no necesita o no comprende con los consiguientes dolores de cabeza para nosotros, los humildes mortales.
El switch *-L* le indica al linker en donde debe buscar la Biblioteca. En nuestro caso lleva ’.á continuación ya que la Biblioteca *hola*, está en el mismo directorio en el que se invoca a *gcc*. Por su parte *-l*, indica que debe incluirse en la compilación la Biblioteca cuyo nombre se indica a continuación, en nuestro caso, *hola*. El linker buscará *libhola.a*, en el directorio de trabajo y sacará de allí los programas objeto que incluirá en el ejecutable.
Tal vez el lector se esté preguntando “¿Y porque cuando uso *printf* no le tengo que decir al *gcc* que Biblioteca utilizar y donde se encuentra?”. Buena pregunta. Para responderla debemos recordar que el lenguaje C trabaja con un set de funciones contenidas en Bibliotecas denominadas habitualmente Bibliotecas estándar de C. Estas ya le son conocidas, y para buscarlas existe en el sistema variables de entorno que el gcc y el paquete gnutils al momento de su instalación configuran adecuadamente.
Para agregar el resto de las librerías se dispone de una variable de entorno en la cual podemos agregar la ruta absoluta a nuestro archivo de Biblioteca. La variable es *LD_LIBRARY_PATH*, y se maneja con las mismas reglas que la variable de entorno *PATH* : Se incluyen en ella las rutas separadas por el caracter ’:’. Por default su valor es nulo, como podemos ver:

```
1 alejandro@notebook:~/InfoI/second$echo $LD_LIBRARY_PATH
2
3 alejandro@notebook:~/InfoI/second$
```

Listing 2.7: Salida de strace para el proceso hola

Queda a cargo del lector echar un vistazo a los directorios contenidos por la variable *LD_LIBRARY_PATH* de su sistema, y ver los archivos allí contenidos para entender que buscando en ese par de directorios el linker encontrará lo necesario para resolver llamadas a las Bibliotecas estándar. Listemos el directorio para observar los tamaños de los diferentes archivos y de paso ejecutemos nuestro programa *demo*, linkeado estáticamente a la Biblioteca *hola*.

```
1 alejandro@notebook:~/InfoI/second$ ls -las
2 total 40
3 4 drwxr-xr-x 2 alejandro alejandro 4096 feb 16 18:02 .
```

```
4 drwxr-xr-x 6 alejandro alejandro 4096 feb 14 12:23 ..
8 -rwxr-xr-x 1 alejandro alejandro 6367 feb 16 18:02 demo
4 -rw-r--r-- 1 alejandro alejandro   21 feb 16 11:30 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 16 07:58 holalib.h
4 -rw-r--r-- 1 alejandro alejandro  836 feb 16 15:47 holalib.o
4 -rw-r--r-- 1 alejandro alejandro  978 feb 16 15:47 libhola.a
4 -rw-r--r-- 1 alejandro alejandro  121 feb 14 12:32 test_holalib.c
4 -rw-r--r-- 1 alejandro alejandro  772 feb 16 15:47 test_holalib.o
alejandro@notebook:~/InfoI/second$ demo
Hola Mundo!
alejandro@notebook:~/InfoI/second$
```

Listing 2.8: Salida de strace para el proceso hola

Observaciones: El archivo *libhola.a* es ligeramente mayor que *holalib.o*, debido a la información que necesita agregar un archivo de Bibliotecas para construir un encabezamiento con:

- El índice de archivos de programas objeto.
- Los nombres de las funciones contenidas en dichos programas objeto.
- El offset relativo dentro del archivo Biblioteca del inicio de cada función.

De todos modos tengamos en cuenta que esta simple Biblioteca contiene por ahora un solo programa objeto, de modo que a medida que se le agreguen mas programas objeto su tamaño será la suma de los tamaños de los programas objeto mas un pequeño plus debido a este encabezado que necesita.
El archivo demo por su parte contiene el encabezado *ELF*, y el código que implementa la función *hola*, extraído del archivo *libhola.a*.
Finalmente comprobamos que implementa nuestra valiosa función de imprimir “Hola Mundo!” por stdout.

2.2.3. Bibliotecas compartidas (Shared)

Estas Bibliotecas no incorporan el código de sus funciones al programa que las invoca, sino que resuelven las referencias en el momento de arranque del programa en cuestión (load time).
Recordemos de la Guia001, que el módulo */lib/ld-linux.so.2* es el dynamic loader de Linux, que se encarga de cargar un programa ejecutable en la memoria. Es, entonces, el responsable de resolver en ese momento las referencias a funciones contenidas en las Bibliotecas compartidas, realizadas por el programa en proceso de carga.
Si en el momento de la carga de nuestro programa no están cargadas en memoria las Bibliotecas necesarias, el dynamic loader cargará simultáneamente a memoria el programa en sí que va a componer el proceso disparado por el usuario, junto con las Bibliotecas que necesita. Una vez cargados en memoria todos los módulos, resolverá las referencias, dejando al programa listo para su ejecución.
Una vez instalada la Biblioteca en la memoria del sistema, si ejecutamos otro programa que invocará algún(os) programa(s) objeto incluido(s) en la Biblioteca, directamente se enlaza con la instancia de la Biblioteca residente en memoria para resolver las direcciones a donde efectuar las llamadas desde el programa invocador.
En las implementaciones de Linux tienen algunas prestaciones adicionales que las hacen mas cómodas y atractivas, como por ejemplo, actualizar la Biblioteca y seguir soportando a los programas que usan versiones mas viejas de éstas. Sin embargo al ser tan versátiles nos hacen “pagar” sus ventajas con algunas complejidades. Es así. Bienvenidos a la Ingeniería...
Este apartado trata acerca de esas complejidades. Manos a la obra.
Las Bibliotecas compartidas tienen un nombre al que se denomina *“soname”* que en general difiere del nombre del archivo, que la contiene. Este nombre se asigna mediante el parámetro *-soname* que se pasa al linker *ld* en el momento

de su construcción. Este nombre es parte del header *ELF* del archivo. Lo podemos comprobar con el conocido comando *objdump*, usando la opción *-p* para que presente información privada o adicional del archivo.En algunos casos no hay nada mas que presentar y *-p* no agrega información al listado, en otros casos si la hay.

```
alejandro@notebook:/usr/lib$ objdump -p libhighgui.so.1 | grep SONAME
  SONAME       libhighgui.so.1
alejandro@notebook:/usr/lib$ objdump -p libhighgui.so.1.0.0 | grep SONAME
  SONAME       libhighgui.so.1
alejandro@notebook:/usr/lib$
```

Listing 2.9: Salida de strace para el proceso hola

Como vemos el archivo *libhighgui.so.1* tiene en este caso un *soname* igual a su nombre de archivo, pero el *libhighgui.so.1.0.0* tiene el mismo soname que el anterior, que obviamente difiere de su nombre de archivo.
El *soname* en general lleva el prefijo *"lib"* seguido del nombre de la Biblioteca y un sufijo compuesto por la string *"so"* seguida de un '.'y un número de versión. Este número se incrementa cuando cambia la Biblioteca.
Como toda regla existe alguna excepción. En general las Bibliotecas de mas bajo nivel del C vienen sin el prefijo *"lib"*.
Sin embargo es de práctica que el prefijo del *soname*, sea el nombre del directorio en el que irá a parar la Biblioteca una vez instalada en el *file system*.
En general el nombre del archivo (también llamado *real name*) se compone del *soname*, seguida de '.', un número de versión, y opcionalmente '.'y un número de release.
El *soname* es el nombre que utiliza el linker. Si no tiene incluido el número de versión, se lo incluye. La clave del manejo de estas Bibliotecas está en tener separados *soname* y nombre de archivo. Los programas cuando listan internamente las Bibliotecas que necesitan, listan directamente el *soname* de c/u. En cambio cuando desarrollamos una Biblioteca solo nos remitimos a crear un archivo con el nombre de acuerdo a las reglas establecidas y a lo sumo incluir el número de versión. Nada mas.
Al instalar la Biblioteca, debemos ejecutar el programa */etc/ldconfig*, que se encarga de crear el *soname*, escribirlo en el encabezado *ELF*, y setear el cache */etc/ld.so.cache*.
Enlazando el soname con el nombre de la Biblioteca
Pero ¿como se enlaza el soname con el nombre de la Biblioteca?
El programa */etc/ldconfig*, se encarga de crear los enlaces simbólicos correspondientes. Comprobémoslo con un ejemplo vivo: libhighgui.so

```
alejandro@notebook:/usr/lib$ ls -las libhighgui*
260 -rw-r--r-- 1 root root 258414 sep 12  2008 libhighgui.a
  4 -rw-r--r-- 1 root root   1536 sep 12  2008 libhighgui.la
  0 lrwxrwxrwx 1 root root     19 jun 18  2009 libhighgui.so ->
libhighgui.so.1.0.0
  0 lrwxrwxrwx 1 root root     19 jun 18  2009 libhighgui.so.1 ->
libhighgui.so.1.0.0
172 -rw-r--r--1 root root 170024 sep 12  2008 libhighgui.so.1.0.0
```

Listing 2.10: Salida de strace para el proceso hola

Vemos que el archivo de la Biblioteca contiene el *soname* el número de versión y el número menor (*release*). Los *soft links* se crean al instalar la Biblioteca.
Para instalar, basta con un copy. Por lo general las Bibliotecas se instalan en */lib* o */usr/lib*. Si no son parte del sistema pueden ir en */usr/local/lib*. Cada vez que se carga un programa el módulo *loader* del sistema operativo (en general */lib/ld-linux.so.[versión]*) en nuestro caso:

```
alejandro@notebook:/usr/lib$ ls -las /lib/ld-linux.so.2
0 lrwxrwxrwx 1 root root 9 ene 23 2009 /lib/ld-linux.so.2 -> ld-2.7.so
alejandro@notebook:/usr/lib$
```

Listing 2.11: Salida de strace para el proceso hola

Baja desde disco a memoria el contenido del archivo binario ejecutable interpretando el formato *ELF* y demás delicias ya conocidas.
Una vez hecho esto, se encarga de buscar las Bibliotecas compartidas invocadas en el programa (si las hubiera), y cargar en memoria aquellas que aún no estuviesen presentes, para finalmente resolver en forma dinámica (es decir en tiempo de carga para su ejecución del programa) las referencias hechas a las Bibliotecas.
Ya está el código del programa en memoria RAM. Para saber en que directorios buscar las Bibliotecas, lee el archivo */etc/ld.so.conf*:

```
alejandro@notebook:/usr/lib$ cat /etc/ld.so.conf
include /etc/ld.so.conf.d/*.conf
/usr/local/cuda/lib
alejandro@notebook:/usr/lib$
```

Listing 2.12: Salida de strace para el proceso hola

Aquí encuentra un *path* concreto y un *include* donde se especifica leer todos los archivos terminados en *".conf"* del directorio */etc/ld.so.conf.d/*. En la consola listar el contenido de /etc/ld.so.conf.d/:

```
   alejandro@notebook:/usr/lib$ ls -las /etc/ld.so.conf.d/
total 28
 4 drwxr-xr-x   2 root root  4096 ene 23  2009 .
12 drwxr-xr-x 137 root root 12288 feb 18 12:59 ..
 4 -rw-r--r--   1 root root    64 oct 13  2008 i486-linux-gnu.conf
 4 -rw-r--r--   1 root root    44 oct 13  2008 libc.conf
alejandro@notebook:/usr/lib$ cat /etc/ld.so.conf.d/i486-linux-gnu.conf
# Multiarch support
/lib/i486-linux-gnu
/usr/lib/i486-linux-gnu
alejandro@notebook:/usr/lib$ cat /etc/ld.so.conf.d/libc.conf
# libc default configuration
/usr/local/lib
alejandro@notebook:/usr/lib$
```

Listing 2.13: Salida de strace para el proceso hola

Todos los archivos de biblioteca deben estar allí. Si al menos uno de los archivos de las Bibliotecas incluidas en el código del programa no se encuentra en estos paths no se carga el programa y se informa con un mensaje por *stderr*.
Queda por cuenta del lector el comando *cat /etc/ld.so.cache*.
El contenido de este archivo depende de la cantidad de Bibliotecas compartidas cargadas en memoria. De este modo se evita recorrer todos los directorios en busca de las Bibliotecas cargadas en memoria.
Al tratar con Bibliotecas estáticas hemos listado el contenido de la variable de entorno *LD_LIBRARY_PATH*, para saber donde buscar archivos de Bibliotecas. En el caso de las Bibliotecas compartidas, sirve para probar una nueva versión agregando allí el path para que sea nuestra versión de prueba la que primero se cargue.
Una vez finalizada la prueba podemos limpiar la variable a su estado original e instalar la Biblioteca o corregirla de

acuerdo con el estado de las pruebas efectuadas.
La variable *LD_PRELOAD* es específicamente para ello, aunque no es una variable estándar.

```
alejandro@notebook:~/InfoI/second$ gcc -fPIC -g -c -Wall holalib.c
alejandro@notebook:~/InfoI/shared$
```

Listing 2.14: Salida de strace para el proceso hola

Para compilar una librería shared, utilizaremos el mismo código fuente que en el caso de las estáticas. Aunque los comandos son diferentes.
Para compilar:

```
alejandro@notebook:~/InfoI/second$ gcc -fPIC -g -c -Wall holalib.c
alejandro@notebook:~/InfoI/shared$
```

Listing 2.15: Salida de strace para el proceso hola

Con el comando *gcc* compilamos el fuente de la Biblioteca. La opción *fPIC* indica al compilador que genere código independiente de la posición que ocupe en memoria.
Este concepto es central ya que si esa opción no se incluye el código no va a ser apto para ser enlazado dinámicamente con los programas que puedan requerir sus servicios, y pueden además existir límites en la tabla global de offsets que puede ocupar ese código dentro de una de las secciones del programa. Requiere cierto soporte a nivel de hardware que convierte a este tipo de código en procesador dependiente, pudiendo por lo tanto no ser factible bajo ciertas arquitecturas. La opción *-g* habilita la inclusión de toda la información simbólica para soporte al *debuging*. Es recomendable su inclusión. Ahora, con este comando solo hemos generado el programa objeto:

```
alejandro@notebook:~/InfoI/shared$ ls -las
total 32
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 02:22 .
4 drwxr-xr-x 3 alejandro alejandro 4096 feb 19 02:14 ..
4 -rw-r--r-- 1 alejandro alejandro  210 feb 19 02:14 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 19 02:14 holalib.h
4 -rw-r--r-- 1 alejandro alejandro 2744 feb 19 02:16 holalib.o
4 -rw-r--r-- 1 alejandro alejandro  121 feb 19 02:14 test_holalib.c
alejandro@notebook:~/InfoI/shared$
```

Listing 2.16: Salida de strace para el proceso hola

Resta generar la Biblioteca. Esta vez no recurrimos a un archivador sino al propio linker. Claro, como ya hemos visto con anterioridad a esta guía el tratamiento con el *ld* conviene dejarlo en manos del *gcc*. Sin embargo los parámetros que le vamos a pasar son todos para el linker. El *gcc* armará los argumentos de linkeo e invocará al linker. Recordemos que siempre podemos usar la opción verbose (*-v*) del *gcc* para satisfacer nuestra curiosidad de conocer los argumentos que *gcc* le arma a *ld*.

```
alejandro@notebook:~/InfoI/shared$  gcc -shared -Wl,-soname,libhola.so.1
-o libhola.so.1.0.1 holalib.o -lc
alejandro@notebook:~/InfoI/shared$ ls -las
total 32
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 02:22 .
```

```
4 drwxr-xr-x 3 alejandro alejandro 4096 feb 19 02:14 ..
4 -rw-r--r-- 1 alejandro alejandro  210 feb 19 02:14 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 19 02:14 holalib.h
4 -rw-r--r-- 1 alejandro alejandro 2744 feb 19 02:16 holalib.o
8 -rw-r--r-- 1 alejandro alejandro 6074 feb 19 02:22 libhola.so.1.0.1
4 -rw-r--r-- 1 alejandro alejandro  121 feb 19 02:14 test_holalib.
alejandro@notebook:~/InfoI/shared$
```

Listing 2.17: Salida de strace para el proceso hola

El argumento *-shared* es de tan obvia explicación que huelgan las palabras. *-Wl* indica que lo que sigue es pasado directamente al linker. Si lo que sigue a *-Wl* contiene una o mas comas, se trata de argumentos que deberán ser enviados al linker uno a uno.
En nuestro caso *-soname* y *libhola.so.1* son esos argumentos, que indican entre ambos cual es el *soname* de nuestra Biblioteca.A continuación se indican el nombre del archivo en el que se empaquetarán el(los) programa(s) objeto y la lista de archivos objeto (terminados en ".o") que se incluirá en el archivo de Biblioteca.
Allí está nuestro archivo de Biblioteca compartida. Note el lector los permisos que le dio el compilador por default.
Ahora llega el momento de ponerla disponible para los programas de aplicación que la quieran utilizar. Para ello vamos a ejecutar los pasos explicados con anterioridad:

```
alejandro@notebook:~/InfoI/shared$ sudo ldconfig -n .
alejandro@notebook:~/InfoI/shared$ ls -las
total 32
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 17:05 .
4 drwxr-xr-x 3 alejandro alejandro 4096 feb 19 02:14 ..
4 -rw-r--r-- 1 alejandro alejandro  210 feb 19 02:14 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 19 02:14 holalib.h
4 -rw-r--r-- 1 alejandro alejandro 2744 feb 19 02:16 holalib.o
0 lrwxrwxrwx 1 root      root        16 feb 19 17:05 libhola.so.1 -> libhola.so.1.0.1
8  -rw-r--r-- 1 alejandro alejandro 6074 feb 19 02:22 libhola.so.1.0.1
4  -rw-r--r-- 1 alejandro alejandro  121 feb 19 02:14 test_holalib.c
```

Listing 2.18: Salida de strace para el proceso hola

Como vemos el comando *ldconfig* se ejecuta como superusuario. ¿Porque?. Es bastante obvio. Publicar una Biblioteca compartida es una operación que requiere ser realizada solo por quien administra el equipo ya que puede afectar al resto de los usuarios[1].
Luego del comando el único rastro visible es el *soft link* creado. Pero recordemos que además trabaja en el encabezado del archivo de la Biblioteca registrando allí el *soname*, entre otras cosas.

```
alejandro@notebook:~/InfoI/shared$ ln -sf libhola.so.1 libhola.so
alejandro@notebook:~/InfoI/shared$ ls -las
total 32
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 17:06 .
4 drwxr-xr-x 3 alejandro alejandro 4096 feb 19 02:14 ..
4 -rw-r--r-- 1 alejandro alejandro  210 feb 19 02:14 holalib.c
```

[1] Si bien a esta altura puede resultar obvio, es conveniente recordar al lector que los sistemas UNIX han sido concebidos para que en un mismo computador trabajen muchos usuarios a la vez. Sus descendientes como Linux mantienen este principio, raro de entender para quienes hicieron sus primeros pasos con un computador que lleva instalado Windows.

```
4 -rw-r--r-- 1 alejandro alejandro  106 feb 19 02:14 holalib.h
4 -rw-r--r-- 1 alejandro alejandro 2744 feb 19 02:16 holalib.o
0 lrwxrwxrwx 1 alejandro alejandro   12 feb 19 17:06 libhola.so -> libhola.so.1
0 lrwxrwxrwx 1 root      root        16 feb 19 17:05 libhola.so.1 -> libhola.so.1.0.1
8 -rwxr-xr-x 1 alejandro alejandro 6074 feb 19 02:22 libhola.so.1.0.1
4 -rw-r--r-- 1 alejandro alejandro  121 feb 19 02:14 test_holalib.c
```

Listing 2.19: Salida de strace para el proceso hola

Creamos un *soft link* adicional con el *soname* sin la versión de modo que se pueda invocar en forma genérica en las líneas de compilación independientemente de la versión instalada en el sistema.
Para poder probar nuestra Biblioteca vamos a compilar el programa de test.

```
alejandro@notebook:~/InfoI/shared$ gcc -Wall -c -o test_holalib.o
test_holalib.c
alejandro@notebook:~/InfoI/shared$ gcc -o demo test_holalib.o -L. -lhola
alejandro@notebook:~/InfoI/shared$ ls -las
total 44
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 17:08 .
4 drwxr-xr-x 3 alejandro alejandro 4096 feb 19 02:14 ..
8 -rwxr-xr-x 1 alejandro alejandro 6485 feb 19 17:08 demo
4 -rw-r--r-- 1 alejandro alejandro  210 feb 19 02:14 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 19 02:14 holalib.h
4 -rw-r--r-- 1 alejandro alejandro 2744 feb 19 02:16 holalib.o
0 lrwxrwxrwx 1 alejandro alejandro   12 feb 19 17:06 libhola.so ->
libhola.so.1
0 lrwxrwxrwx 1 root      root        16 feb 19 17:05 libhola.so.1 ->
libhola.so.1.0.1
8 -rwxr-xr-x 1 alejandro alejandro 6074 feb 19 02:22 libhola.so.1.0.1
4 -rw-r--r-- 1 alejandro alejandro  121 feb 19 02:14 test_holalib.c
4 -rw-r--r-- 1 alejandro alejandro  772 feb 19 17:07 test_holalib.o
alejandro@notebook:~/InfoI/shared$
```

Listing 2.20: Salida de strace para el proceso hola

La primer línea de comando generan el programa objeto a partir del programa fuente de prueba y la segunda lo linkea con la Biblioteca.
Si las comparamos con las utilizadas para idénticos fines cuando utilizamos Bibliotecas estáticas, vemos que, como no puede ser de otro modo, no existe ni un punto ni una coma de diferencia. Este concepto es básico. Una vez generada la Biblioteca la forma de compilar aplicaciones y linkearlas a la Biblioteca es independiente del tipo de Biblioteca utilizada.
Con ejemplos tan triviales no alcanzamos a observar una diferencia notable en el tamaño del archivo binario demo, que es en definitiva nuestra aplicación. Al usar Bibliotecas tan pequeñas no es notoria la diferencia pero claramente al utilizar una Biblioteca estática el código se incrusta en la aplicación y en las compartidas se enlaza en tiempo de ejecución.
Trabajando con programas y Bibliotecas reales cuyos códigos son menos triviales que los que estamos trabajando aquí, la diferencia de tamaños de las dos versiones de demo, sería notoria.
¿Ejecutamos nuestra aplicación?

```
alejandro@notebook:~/InfoI/shared$ demo
```

```
demo: error while loading shared libraries: libhola.so.1: cannot
open shared object file: No such file or directory
alejandro@notebook:~/InfoI/shared$
```

Listing 2.21: Salida de strace para el proceso hola

¿Que es lo que ocurrió?.
Olvidamos poner nuestra Biblioteca disponible para el sistema. Y además como nunca se utilizó hasta ahora, tampoco está en el cache.

```
alejandro@notebook:~/InfoI/shared$ echo $LD_LIBRARY_PATH
/lib:/usr/lib
alejandro@notebook:~/InfoI/shared$
```

Listing 2.22: Salida de strace para el proceso hola

Los *paths* en los que el sistema buscará las Bibliotecas que utiliza el programa en el momento de su carga en memoria para ejecución son los contenidos en la variable de entorno *LD_LIBRARY_PATH*, que no tiene registrado nuestro directorio de trabajo, que es en donde está físicamente nuestra Biblioteca. Solucionamos este pequeño detalle y....

```
alejandro@notebook:~/InfoI/shared$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
alejandro@notebook:~/InfoI/shared$ echo $LD_LIBRARY_PATH
/lib:/usr/lib:.
alejandro@notebook:~/InfoI/shared$ demo
Hola Mundo!
alejandro@notebook:~/InfoI/shared$
```

Listing 2.23: Salida de strace para el proceso hola

Voila!. Ahí está funcionando.
La forma real y profesional es copiar nuestra Biblioteca y los *soft links* al directorio */usr/lib*. Puede ser al */lib* pero no es lo mas conveniente, desde el punto de vista de lo que contiene cada directorio.

2.3. Desarrollo de una Biblioteca de carga Dinámica

2.3.1. Conceptos preliminares

Estas Bibliotecas se cargan en momentos diferentes de la carga y ejecución del programa. Su principal utilidad es la implementación de módulos, o plug-ins, ya que estos elementos de software se cargan cuando se invocan durante la ejecución de un programa.
Desde el punto de vista de su formato no tienen diferencias en Linux con respecto a como se construyen Bibliotecas compartidas o programas objeto. Sin embargo hay diferencias en el código que se necesita escribir en la aplicación para trabajar con estas Bibliotecas.
Es decir que el programador de la aplicación debe incluir funciones específicas que hasta ahora en los modelos analizados no se requieren.

2.3.2. Funciones a incluir en el código de los programas

dlopen( )

```
1 void * dlopen(const char *filename, int flag);
```

Listing 2.24: Salida de strace para el proceso hola

Carga una Biblioteca y la prepara para su uso. Devuelve un identificador para esa Biblioteca que se utiliza en el resto del programa para las referencias posteriores.
const char *filename es una string con el nombre del archivo (o del *soft link*). Si proveemos el path absoluto (es decir desde / en adelante), la función busca exactamente ese archivo. Si se provee solo el nombre del archivo, lo buscará en: los directorios contenidos en la variable de entorno *LD_LIBRARY_PATH*, si no la encuentra busca en */etc/ld.so.cache*, si no está busca en */lib*, y sino en */usr/lib*. **int flag** toma los siguientes valores:

1. **RTLD_LAZY**: para resolver los símbolos no definidos como código de la Biblioteca dinámica cuando se está ejecutando.
2. **RTLD_NOW**: para resolver los símbolos no definidos antes de que *dlopen()* retorne y generar un error si no es posible resolverlos.
3. **RTLD_GLOBAL**: puede componerse mediante un operador OR con los otros valores y hace que los símbolos externos contenidos en la Biblioteca estén disponibles para las Bibliotecas cargadas consecuentemente.

dlerror( )
Retorna un string que describe el error generado por las demás funciones de manejo de Bibliotecas dinámicas.
dlsym( )

```
1 void * dlsym(void *handle, char *symbol);
```

Listing 2.25: Salida de strace para el proceso hola

Busca el valor de un símbolo presente en una Biblioteca ya abierta con *dlopen ()*. Devuelve un puntero a la función o al elemento direccionado. Los argumentos son el identificador de la Biblioteca devuelto por *dlopen ()*, y un string con el nombre del símbolo o función cuyo puntero se desea conseguir. **dlclose( )**
Cierra la Biblioteca abierta *dlopen ()* permitiendo liberar el descriptor o handle que se tenía hasta el momento para identificar a nuestra Biblioteca dinámica.

```
1 int dlclose(void *handle);
```

Listing 2.26: Salida de strace para el proceso hola

2.3.3. La aplicación

Para implementar una Biblioteca dinámica utilizamos los mismos archivos de una Biblioteca compartida. Sin cambios de ninguna índole en su procedimiento de generación, ni en su código fuente.
Lo que sí cambia es el programa invocador a las funciones de las Bibliotecas ya que para estas se ha definido un grupo de funciones específicas para el acceso.

2.3.4. Código específico

Así queda una aplicación

```
/* Programa de prueba para nuestra Biblioteca holalib */
# include "holalib.h"
# include <dlfcn.h>
# include <stdlib.h>
int     main (int argc, char **argv)
{
        void *handle;
        void (*hello) (void);
        char *error;
/*Se abre la Biblioteca y se obtiene en handle un puntero a la instancia de la misma
    */
        handle = dlopen("../shared/libhola.so.1",RTLD_LAZY);
/*Si handle es NULL, entonces se imprime el mensaje devuelto por dlerror en stderr*/
        if (!handle) {
                fprintf (stderr, "%s\n", dlerror());
                exit(EXIT_FAILURE);
        }
//En hello se guarda el puntero al label llamado hola en la Biblioteca.
        hello = dlsym (handle, "hola");
        if ((error = dlerror()) != NULL) {
                fputs(error, stderr);
               exit(EXIT_FAILURE);
        }
// Se invoca la funcion por medio de su puntero.
        (*hello) ();
        dlclose(handle);
        return 0;
}
```

Listing 2.27: Salida de strace para el proceso hola

2.4. Conclusiones

Hemos podido mediante un ejemplo sumamente trivial desarrollar las Bibliotecas de uso estándar en Linux, y observar sus características y ventajas comparativas entre los tres tipos posibles.
Las ideas fuerza que deben quedar de esta clase, tienen que ver con la importancia de tener “paquetes” de código escrito de tal forma que pueda ser reusado por las aplicaciones que desarrollemos.
Esto aumenta nuestra productividad y hace que nuestra programación esté mas libre de errores.

Capítulo 3

Herramientas para construir proyectos: make

make

3.1. Introducción

Cuando programamos proyectos mas o menos significativos normalmente utilizamos varios archivos fuentes, varios archivos headers, referencias a bibliotecas, y demás componentes.
Esto hace que en un punto se vuelva tediosa la tarea de compilar, linkear, y demás actividades conducentes a construir el proyecto para su posterior prueba detección de errores, corrección de los fuentes, y otra vez a empezar.
Claro. Si alguna vez el lector ya se enfrentó con una bonita herramienta de programación con una amigable y simpática interfaz gráfica, posiblemente esté pensando que el autor no conoce de su existencia explicando cosas tan elementales. El autor ha trabajado con esas interfaces bonitas, amigables y muy potentes en lo que hace a simplificarnos la vida. ¿Entonces para que leer este capítulo?. Como en el caso de las anteriores, cabe la misma respuesta: ***para aprender***.
Un IDE (Integrated Development Environmet) que trabaja en modo gráfico (a veces llamados entornos visuales) es la octava maravilla del mundo para los programadores de aplicaciones, ya que nos quita de encima numerosas tareas. Pero esto no es una razón para que dejemos de entender como funcionan las cosas. Finalmente una de las razones de ser de los ingenieros es entender el funcionamiento de las cosas. Y esto no es posible cuando desde el principio utilizamos herramientas que nos simplifican el trabajo ya que lo hacen a costa de esconder ciertas complejidades en las cuales por lo general se encierran conceptos importantes. El conocer estos detalles es una tarea ardua pero hace la diferencia en algunas ocasiones, aunque le doy la derecha al lector: suena muy antipático.
Luego de trabajar con estas herramientas puede continuar con el bonito IDE lleno de facilidades. Pero la diferencia fundamental será que entonces sabrá como las hace. Y cuando tenga un problema no se sentirá tan perdido como en otras ocasiones. (Al menos eso espero).

3.2. Make

`make` es una herramienta que permite ejecutar una secuencia de procesos. Utiliza un script, llamada comúnmente `makefile`. Es capaz de determinar automáticamente cuales pasos de una secuencia deben repetirse debido al cambio en algunos de los archivos involucrados en la construcción de un objeto, o en una operación, y cuales no han registrado cambios desde la última vez de modo que no es necesario repetirlos. Sus uso mas común es recompilar programas que residen en diversos archivos, pero también podemos automatizar el testing de estos programas, así como su

empaquetado y distribución. De manera sencilla lo que tenemos que hacer es definir un archivo llamado Makefile en el directorio raíz de nuestro proyecto (en realidad lo podemos poner en otro lado y hacérselo saber a **make**) y dentro de ese archivo escribimos las reglas necesarias para construir nuestro proyecto. Luego, alcanza con ejecutar en el directorio en el que hemos definido el Makefile, el comando del listado 3.1

```
$ make <regla>
```

Listing 3.1: Sintaxis del comanndo make

Parece simple.

3.2.1. La lógica de `make`

makefunciona teniendo siempre presente las fases de desarrollo de un programa, cuyo proceso completo podemos ver en la Figura 3.1. Este proceso genera una serie de dependencias. Cada programa objeto depende de su correspon-

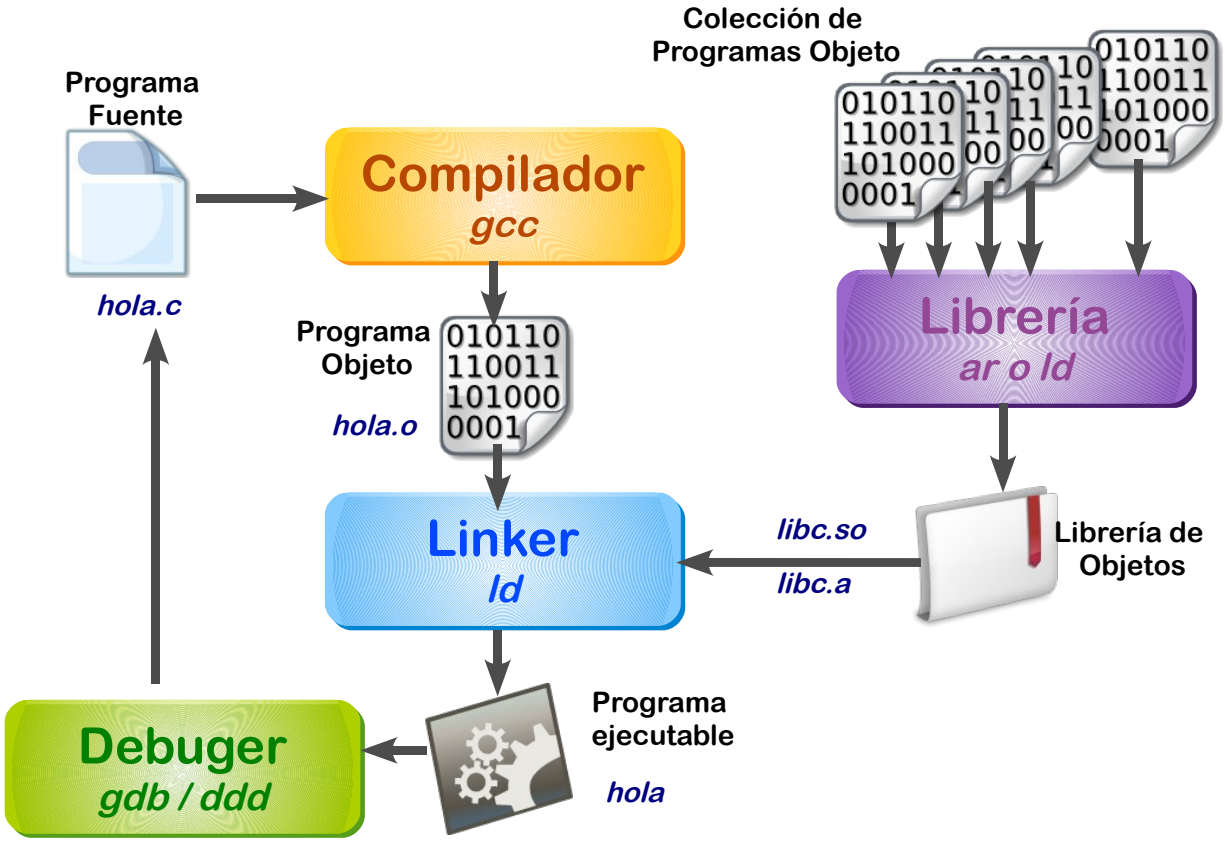


Figura 3.1: Ciclo de desarrollo de software

diente programa fuente, y headers si los hubiere. Un ejecutable dependerá de los programas objeto involucrados en

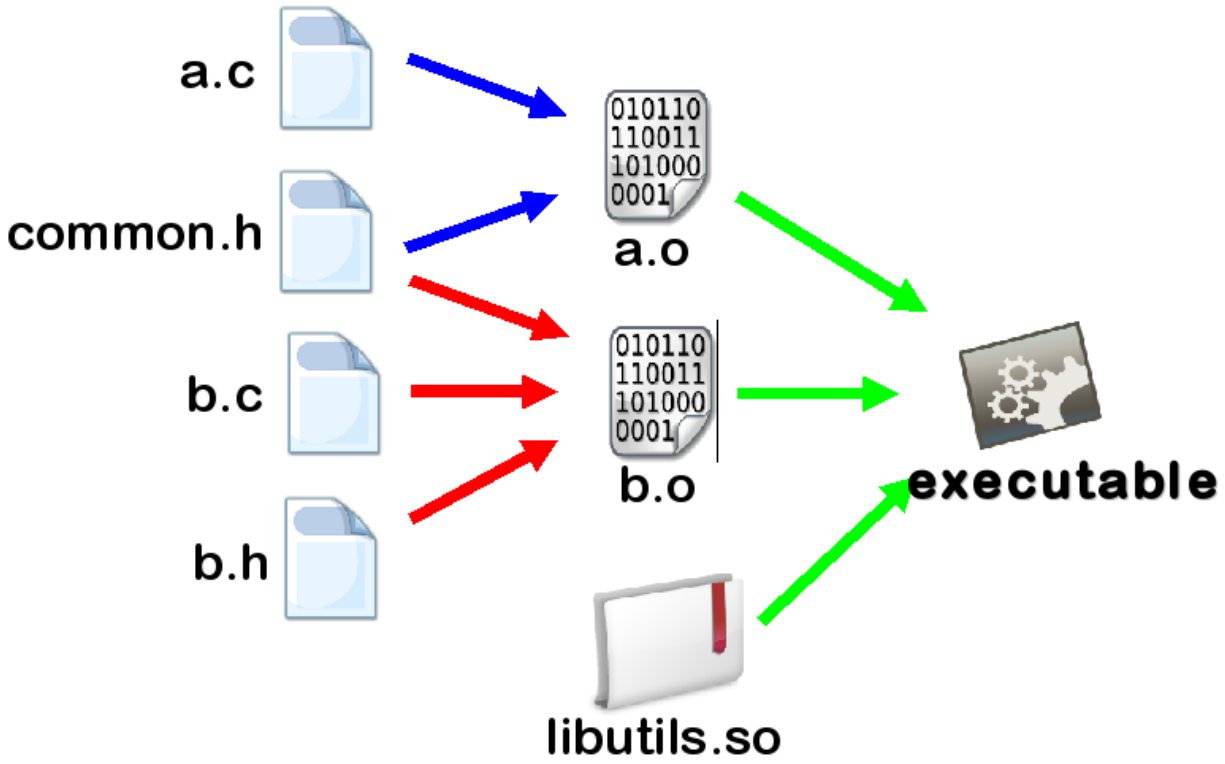


Figura 3.2: Dependencias entre los diferentes archivos de un proyecto de software

su proyecto, mas las librerías de uso en el mismo. Esta situación se observa en la Figura 3.2 En esta figura podemos ver las dependencias generadas en el desarrollo de un proyecto. Tenemos tres grupos claramente diferenciados por los colores de las flechas. A la hora de armar un `makefile` necesitamos tener claro el mapa de dependencias. Es como tener esta figura (adaptada a la realidad de nuestro proyecto) en la cabeza mientras escribimos los diferentes comandos en el `makefile`. Es necesario respetar estas dependencias ya que make asume que el script escrito en el `makefile` ha tenido en cuenta este criterio. Una vez armado y en funcionamiento, si modificamos el archivo b.c, no importa el motivo, **`make`**detectará la actualización pero en lugar de repetir todas las compilaciones y linkeos, solo recompilará al archivo b.c, lo cual modificará el objeto b.o, y por lo tanto deberá relinkear la aplicación para llegar a la versión de ejecutable que contenga los cambios hechos en b.c. Lo interesante es que no tocó el resto. Es decir lo que no cambia se deja tal como está. Pero veamos como es una primer versión de `makefile` para el caso de la Figura 3.2.

```
1 excecutable: a.o b.o
2     gcc a.o b.o -o executable -lutils
3 a.o: a.c
4     gcc -c -o a.o a.c
5 b.o: b.c
6     gcc -c -o b.o b.c
7 clean:
8     rm -f ./*.o
9     rm -f executable
```

Listing 3.2: Contenido muy básico de un `Makefile` sin ninguna inteligencia

Lo que vemos al inicio de la línea como un nombre terminado con ':'se denomina regla. Si no le especificamos nada y solo tipeamos **make**a secas, **make**asumirá que la regla a ejecutar es solo la de la primer línea con esta característica. En nuestro primer ejemplo executable.

Si la regla a continuación del carácter ':'tiene dependencias, éstas deberán corresponder dentro del `makefile` a otras reglas que se escriban a continuación de la regla dependiente. En nuestro caso excecutable es dependiente de a.o y b.o. Por lo tanto debe necesariamente existir una regla para a.o y otra para b.o escritas luego de la regla dependiente. Estas líneas están a continuación con sus respectivas dependencias.
Cada regla tiene a continuación el comando que necesita para su implementación.
Finalmente hemos escrito una regla que permita limpiar todos los archivos generados a partir de los fuentes. Esto puede resultar útil para hacer una recompilación general, especialmente al cierre del proyecto donde podemos eliminar ciertas opciones de compilación (como por ejemplo la opción -g) y por lo tanto se necesita reducir los tamaños de los ejecutables.
Si ejecutamos la sentencia del listado 3.3, se ejecuta esa regla pasando por alto las restantes, hasta resolver sus dependencias.

```
$ make clean
```

Listing 3.3: comando make ejecutando una regla específica

El formato general de cada regla es:
dependiente: dependencia
comando para generar el dependiente a partir de la dependencia.

En tutoriales varios y bibliografía que pueda consultar, puede encontrarse la siguiente terminología para lo mismo:

target: fuentes (o sources)
comando (o command)

En cualquier caso es solo a los efectos de comprender la lógica de generación de cada línea. Un detalle nada menor:

Las líneas que contienen comandos empiezan con un tabulador. De otro modo no funciona.

3.2.2. Uso de Variables

La versión anterior es la mas primitiva. El uso de variables permite por un lado resumir opciones, nombres y demás, y en especial facilitar los cambios cada vez que hay que retocar parámetros que se repiten a lo largo del `makefile`. Comencemos preparando nuestro `makefile` para poder modificarlo de manera simple en caso de usar otro compilador (puede ser el `cc`, o el `gpp`). Para ello es necesario empezar por el código del listado 3.4

```
1 CC=gcc
2 excecutable: a.o b.o
3     $(CC) a.o b.o -o executable -lutils
4 a.o: a.c
5     $(CC) -c -o a.o a.c
6 b.o: b.c
7     $(CC) -c -o b.o b.c
```

```
8  clean:
9      rm -f ./*.o
10     rm -f executable
```

Listing 3.4: Contenido muy básico de un `Makefile` con uso de variables

Vemos que la forma de usar variables se parece a la que usa el shell, salvo por los paréntesis que aquí se utilizan adicionalmente al carácter '$'para acceder al contenido. Con este criterio, podemos afianzar mas el uso de variables, como vemos en el listado 3.5.

```
1  CC=gcc
2  CFLAGS=-c -g
3  LDFLAGS=-g -lutils
4  OBJS=a.o b.o
5  excecutable: $(OBJS)
6      $(CC) $(OBJS) -o executable $(LDFLAGS)
7  a.o: a.c
8      $(CC) $(CFLAGS) -o a.o a.c
9  b.o: b.c
10     $(CC) $(CFLAGS) -o b.o b.c
11 clean:
12     rm -f ./*.o
13     rm -f executable
```

Listing 3.5: Contenido muy básico de un `Makefile` con uso de variables

3.2.3. Macros

`make`posee algunas reglas sintácticas adicionales que pueden propender, aún mas lograr un `makefile` genérico, aunque a costa de volverlo mas críptico.

```
1  CC=gcc
2  CFLAGS=-c -g
3  LDFLAGS=-g -lutils
4  OBJS=a.o b.o
5  excecutable: $(OBJS)
6      $(CC) $(OBJS) -o $@ $(LDFLAGS)
7  a.o: a.c
8      $(CC) $(CFLAGS) -o $@ $<
9  b.o: b.c
10     $(CC) $(CFLAGS) -o $@ $<
11 clean:
12     rm -f ./*.o
13     rm -f executable
```

Listing 3.6: Contenido muy básico de un `Makefile` con uso de variables y macros

$@** es una macro que contiene para cada regla el valor del target (o dependiente). En el caso de la regla principal de nuestro `makefile` **$@ = executable. En la regla siguiente: **$@** = a.o, y en la siguiente **$@** = b.o.
$< por su parte se refiere en cada regla al fuente de la misma, siempre que haya un único fuente.
En este punto observemos el aspecto del `makefile`del Listado 3.6. Es muy diferente del primero que habíamos

escrito, es mas inteligible, pero mas flexible para su modificación.
Observemos además las reglas de formación de a.o y b.o. Realmente son muy similares. De hecho con las macros que usamos hasta aquí la línea correspondiente al comando es la misma en ambos casos.

```
1 CC=gcc
2 CFLAGS=-c -g
3 LDFLAGS=-g -lutils
4 OBJS=a.o b.o
5 excecutable: $(OBJS)
6     $(CC) $(OBJS) -o $@ $(LDFLAGS)
7 .o: .c Makefile
8     $(CC) $(CFLAGS) -o $@ $<
9 clean:
10    rm -f ./*.o
11    rm -f executable
```

Listing 3.7: Contenido muy básico de un `Makefile` con uso de variables, macros, y wildcards

Observemos el listado 3.7. Aparece el carácter '%'que trabaja de wildcard, indicando el equivalente a "cualquier cosa". Entonces, la línea de la segunda regla se lee: "Cualquier cosa" que termine en ".o" que necesite para construir el ejecutable, se construye a partir de la dependencia del mismo nombre terminada en ".c".
Además al incluir `Makefile` en esa línea, **`make`**detectará incluso si cambió el `Makefile` en cuyo caso repite todo el proceso completo.
Vamos logrando un `makefile` mas general. Ya no dependemos de los nombres de un fuente determinado. Esto significa que podemos agregar fuentes a nuestro proyecto, y los va a compilar. Solo deberíamos agregar el objeto en la variable `OBJS` para que se incluyan como dependencias en ejecutable.
Hay una forma mas sofisticada que está en este listado. La opción -MM del gcc, hace que no se compilen los fuentes, pero el compilador lista las dependencias por stdout.
Lo podemos verificar en nuestro directorio de trabajo utilizado para construir los ejemplos de bibliotecas de código utilizados en el capítulo 2. Recordemos primero su contenido. Para ello se recomienda primero utilizar ls para ver el contenido del directorio.

```
alejandro@notebook:~/InfoI/second$ ls -las
total 48
4 drwxr-xr-x 4 alejandro alejandro 4096 feb 19 18:53 .
4 drwxr-xr-x 6 alejandro alejandro 4096 feb 17 01:16 ..
8 -rwxr-xr-x 1 alejandro alejandro 6367 feb 16 18:02 demo
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 20 18:19 dynamic
4 -rw-r--r-- 1 alejandro alejandro  210 feb 16 11:30 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 16 07:58 holalib.h
4 -rw-r--r-- 1 alejandro alejandro  836 feb 16 15:47 holalib.o
4 -rw-r--r-- 1 alejandro alejandro  978 feb 16 15:47 libhola.a
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 17:08 shared
4 -rw-r--r-- 1 alejandro alejandro  121 feb 14 12:32 test_holalib.c
4 -rw-r--r-- 1 alejandro alejandro  772 feb 16 15:47 test_holalib.o
alejandro@notebook:~/InfoI/second$
```

Listing 3.8: comando make ejecutando una regla específica

Ahora mediante el switch -MM del gcc veamos la salida

```
alejandro@notebook:~/InfoI/second$ ls -las
alejandro@notebook:~/InfoI/second$ gcc -MM *.c
holalib.o: holalib.c holalib.h
test_holalib.o: test_holalib.c holalib.h
alejandro@notebook:~/InfoI/second$
```

Listing 3.9: comando make ejecutando una regla específica

Aplicamos ahora esta facilidad en nuestro `makefile`.

```
1  CC=gcc
2  CFLAGS=-c -g
3  LDFLAGS=-g -lutils
4  OBJS=a.o b.o
5  SOURCES=$(OBJS:.o=.c)
6  excecutable: $(OBJS)
7      $(CC) $(OBJS) -o $@ $(LDFLAGS)
8  dependencias:
9      $(CC) $(CFLAGS) -MM $(SOURCES) > $@
10 -include dependencias
11 clean:
12     rm -f ./*.o
13     rm -f executable
```

Listing 3.10: Contenido muy básico de un `Makefile` con uso de variables, macros, y wildcards

En primer lugar hemos definido una variable `SOURCES`. Para inicializarla hemos simplemente tomado la lista de valores incluidos en `OBJS`, a los que les hemos reemplazado el sufijo ".o" por ".c", utilizando los operadores ":" e "=". Luego creamos una regla llamada dependencias que se ocupa de ejecutar al compilador con la opción -MM, tomando uno a uno los archivos que estén en la variable `SOURCES`, y envía línea por línea su salida a la variable **$@**, que corresponde a la regla dependencias.
De este modo a la salida del comando, la regla dependencia contiene la salida del compilador (que es del tipo de la listada anteriormente en nuestro directorio que hemos mostrado con `ls` en el listado3.8.
Inmediatamente debajo de esta regla se incluye dependencias, mediante la macro `-include`.Esto pondrá la lista de reglas y dependencias listadas como salida por gcc -MM.
Podemos ir en el `makefile` mas allá y por ejemplo crear una regla para la entrega del proyecto.

```
1  CC=gcc
2  CFLAGS=-c -g
3  LDFLAGS=-g -lutils
4  OBJS=a.o b.o
5  SOURCES=$(OBJS:.o=.c)
6  HEADERS=*.h
7
8  excecutable: $(OBJS)
9      $(CC) $(OBJS) -o $@ $(LDFLAGS)
10 .o: .c Makefile
11     $(CC) $(CFLAGS) -o $@ $<
12 clean:
13     rm -f ./*.o
14     rm -f executable
15 enviar:
```

```
16    tar zcvf proyecto.tar.gz $(SOURCES) $(HEADERS) Makefile
```

Listing 3.11: Contenido muy básico de un `Makefile` con uso de variables, macros, y wildcards

Para ello se agregó tan solo una variable llamada `HEADERS` para poder incluir en el archivo de entrega a los headers sin los cuales no es compilable.
Al final se incluye una regla llamada entrega. Al terminar nuestro proyecto, conviene ejecutar **`make`clean**, luego quitar las opciones de compilación de debuging e incluir tal vez alguna opción de optimización (-O0, -O1, -O2, o -O3, consultar man gcc para ver detalles). Se vuelve a ejecutar **`make`**para generar una versión compacta y optimizada, y recién luego de testearlo por última vez (dude hasta el final!), se ejecuta **make enviar**, y el resultado es un archivo proyecto.tar.gz con todo el proyecto listo para enviar.

3.3. Conclusiones

Hemos aprendido a construir nuestro proyecto utilizando una herramienta sumamente potente. Tan potente que esta guía es solo una pequeñez comparada con las posibilidades y construcciones que permite hacer.
La experiencia de uso de **`make`**indica que una vez que tenemos un `makefile` mas o menos apto para nuestro proyecto, todo lo que hacemos es simplemente cambiar el nombre de las dependencias en la variable `OBJS`, y agregar o quitar flags del compilador o del linker para luego reutilizarlo.
Si en cambio nos topamos con un entorno gráfico de programación, ahora sabremos mas acerca de como personalizarlo y sacarle provecho. No se lo cuenten a nadie. Pero esos entornos tan amigables finalmente cuando tienen que hacer las cosas importantes, es decir construir su proyecto, utillizan **gcc** y **`make`**.;)

Parte II

Recursos avanzados de programación

Capítulo 4

Señales

4.1. Introducción

Ya aprendimos a crear procesos utilizando ***fork ()***. También comprendimos (muchas veces a costa de sufrir algunos inconvenientes) los riesgos que implica tener esta "llave maestra".
Asumiendo que estos conceptos están ciertamente comprendidos, nos proponemos ahora intercomunicar esos procesos con los recursos que dispone el sistema operativo para tales fines. En tal sentido Linux ofrece los mismos mecanismos que el resto de los UNIX like.
Trataremos de ir desde los mas sencillos a los mas elaborados de modo de abordar el tema incrementalmente desde el punto de vista de su complejidad.

4.2. Ahora si: Señales

Las señales son tal vez (además de uno de los primeros) el mas simple de los mecanismos de intercomunicación de Procesos que podamos encontrar. A tal punto es su simplicidad que no transmiten datos en sí desde un proceso a otro, sino que lo que hacen en realidad es enviarle, lógicamente a través del kernel, una suerte de aviso (es decir, una señal) a un proceso determinado. Esa señal, hará que el destinatario realice una acción determinada.
El estándar POSIX define 32 señales para enviar a cada uno de los procesos junto con su comportamiento predeterminado, es decir, que es lo que se espera que haga el proceso receptor de la señal.
Además cada proceso tiene la posibilidad de modificar el comportamiento predefinido para una o mas señales (todas excepto una como veremos) reemplazando la función que le asignó por defecto el Sistema Operativo por una función propia que le resultará mas conveniente.
Finalmente POSIX define también, como podrá imaginarse, un conjunto de system calls para enviar señales, y modificar la función predeterminada que le asigna el sistema operativo a un proceso.
Para comenzar conozcamos las señales disponibles y sus funciones predefinidas. Desde cualquier consola de texto del sistema, se puede tipear el comando ***kill -l***, y se presentarán las señales disponibles, como vemos en la figura 4.1. El comando ***kill*** es el que utilizaremos desde el shell para enviar una señal a un proceso cualquiera. De hecho, el lector estará sumamente familiarizado con kill -9. La llave de escape que tantas veces nos salvó de situaciones, digamos, indeseadas. Si observamos nuevamente (pero ahora mas detalladamente) el listado de salida del comando de la 4.1, y lo combinamos con una excursión por ***man kill***, veremos que el número que va seguido al guión es el identificador de la señal. En este caso (frecuente por cierto) lo que hacemos es enviar la señal Nº 9 (es decir SIGKILL) al proceso que está complicando nuestra apacible existencia en el sistema.
El detalle de las señales de Linux y su comportamiento default se detallan en Tabla 4.1

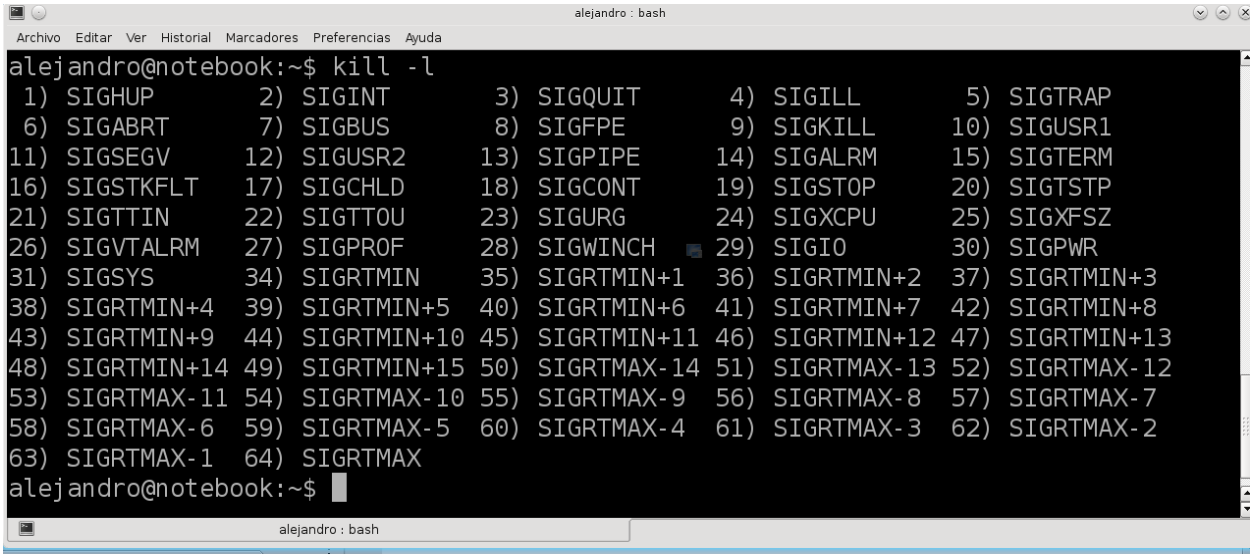


Figura 4.1: Listado de señales en Linux

Las acciones posibles definidas en la Tabla 4.1 son las siguientes e implican:

- **Terminar**: Finaliza la ejecución del proceso. Realiza las mismas actividades de la system call ***exit ()***.
- **Volcar**: (Dump), finaliza la ejecución del proceso pero con el agregado de la generación de un archivo imagen core. Esta imagen tiene fines de proveer información a un debugger como gdb, el que puede obtener de ella la información del estado del procesador y demás cuestiones inherentes al error que se produjo en el código del proceso y que motivó el envío de la señal desde el kernel al proceso para su terminación.
- **Detener**: (Stop), suspende la ejecución del proceso (no lo termina) simplemente lo coloca en estado TASK_STOPPED. Puede reasumirse mediante el envío de SIGCONT
- **Continuar**: Reasume la ejecución de un proceso detenido
- **Ignorar**: La señal no tiene efecto, el proceso receptor por defecto la descarta.

Hay algunas señales bastante relevantes ya que las usamos a menudo sin saberlo tal vez, o se producen automáticamente y no percibimos su presencia. Una de ellas es **SIGKILL**, que está resaltada en la Tabla 4.1. Como ya hemos dicho, está señal es la que enviamos a un proceso cuya finalización queremos asegurar mediante el comando ***kill -9*** seguido del process ID de dicho proceso. Desde el shell podemos enviar las señales especificando su número o su nombre, de modo que es equivalente a enviar ***kill -SIGKILL***.
Esta señal tiene una particularidad: No puede ser modificado su comportamiento mediante el reemplazo de su función establecida por el kernel. Esto significa que ningún proceso puede convertirse por si mismo en NO Interrumpible. De otro modo no tendríamos forma de detener a un proceso que está tal vez fuera de control por una falla en su código, consumiendo CPU y/o memoria en exceso, o en un dead lock.
Por otra parte, **SIGTERM** (15) se envía cuando usamos el comando ***kill*** sin especificar la señal que deseamos enviar. Si bien las dos tienen la misma acción default, **SIGTERM** no actúa si el proceso está lockeado esperando un evento o actualizando un archivo por ejemplo. En cambio **SIGKILL**, es implacable. Lo termina de cualquier forma, y puede hacernos perder información si el archivo está ocupado actualizando un log o lo que fuere que lo tuviese ocupado. Lo recomendable es tratar con **SIGTERM**, y en caso de no tener resultados optar por **SIGKILL**.
Por su parte **SIGCHLD**, es de gran importancia para garantizar el correcto funcionamiento de procesos que generan

Nº	Nombre	Acción (default)	Descripción	POSIX
1	SIGHUP	Terminar	Suspende (Hang up) el control de una terminal o proceso	Si
2	SIGINT	Terminar	Interrupción desde teclado	Si
3	SIGQUIT	Volcar	Quit desde el teclado	Si
4	SIGILL	Volcar	Instrucción Ilegal	Si
5	SIGTRAP	Volcar	Breakpoint for debugging	No
6	SIGABRT	Volcar	Terminación Anormal	Si
6	SIGIOT	Volcar	Equivalente a SIGABRT	No
7	SIGBUS	Volcar	Error de Bus	No
8	SIGFPE	Volcar	Excepción de Floating-point	Si
9	**SIGKILL**	**Terminar**	**Terminación Forzada**	**Si**
10	SIGUSR1	Terminar	User Defined. Disponible	Si
11	SIGSEGV	Volcar	referencia inválida de memoria.	Si
12	SIGUSR2	Terminar	User Defined. Disponible	Si
13	SIGPIPE	Terminar	Escritura en un pipe sin lectores	Si
14	SIGALRM	Terminar	Real-timerclock	Si
15	SIGTERM	Terminar	Process termination	Si
16	SIGSTKFLT	Terminar	Coprocessor stack error	No
17	SIGCHLD	Ignorar	Proceso Hijo terminado o detenido	Si
18	SIGCONT	Continuar	Reasume la ejecución si estaba Detenido	Si
19	SIGSTOP	Detener	Detiene ejecución del proceso	Si
20	SIGTSTP	Detener	Detención del proceso enviada desde tty	Si
21	SIGTTIN	Detener	Proceso en Background requiere entrada	Si
22	SIGTTOU	Detener	Proceso en Background requiere salida	Si
23	SIGURG	Ignorar	Condición Urgente en socket	No
24	SIGXCPU	Volcar	Límite de tiempo de CPU excedido	No
25	SIGXFSZ	Volcar	Límte de tamaño de archivo superado	No
26	SIGVTALRM	Terminar	Virtual timer clock	No
27	SIGPROF	Terminar	Perfil del timer clock	No
28	SIGWINCH	Ignorar	Window resizing	No
29	SIGIO	Terminar	I/O imposible	No
29	SIGPOLL	Terminar	Equivalente a SIGIO	No
30	SIGPWR	Terminar	Falla en fuente de alimentación	No
31	SIGSYS	Volcar	System call errónea	No
32	SIGUNUSED	Volcar	Equivalente a SIGSYS	No

Cuadro 4.1: Señales en Linux

hijos a través de ***`fork ()`***. En la Tabla 4.1 algo se sugiere acerca de su funcionamiento: Cada vez que un proceso termina el kernel envía al proceso padre una señal **SIGCHLD** automáticamente. Esta señal advierte al proceso padre acerca de la terminación o detención de un proceso creado por él. Para entender la utilidad de esta señal es necesario comprender que necesita hacer el kernel para terminar un proceso. En este sentido debemos recordar que cuando un proceso ejecuta la función ***`fork ()`***, nos se duplican los espacios de memoria de código ni de datos. Cuando uno de lo procesos modifica una variable, el sistema operativo duplica automáticamente la página de memoria que contiene el dato solo para ese proceso. Este mecanismo se conoce como ***Copy-On-Write***. Por lo tanto cuando un proceso termina, el kernel eventualmente debe actualizar información en el área de control del proceso padre. Pero no puede hacerlo sin cerciorarse que el proceso padre, al ser interrumpido por el scheduler la última vez, no haya quedado en medio de una operación que también requiera actualizar información en su área de control. En tal caso actualizarla para cerrar un proceso hijo resultaría riesgoso, ya que podría sobreescribir información en uso por parte del proceso padre y que éste requeriría al ser reasumido. Para evitar esto, el kernel solo actualizará el área de control del proceso padre cuando éste esté ejecutando una función que lo mantenga bloqueado y en espera para tal fin. Esa función es ***`wait()`*** o ***`waitpid()`***. Ambas bloquean al proceso que las invoca de modo que no resultaría práctica su inserción en el flujo principal del programa, ya que el proceso padre no podría hacer nada mas hasta tanto no termine la instancia child que termina de generar, perdiéndose las ventajas de paralelización de código que muchas veces se busca con esta técnica.

Sin embargo como es posible cambiar la función que el kernel asigna por default a una o mas señales de cada proceso, podemos reemplazar la función predefinida por el kernel para la Señal **SIGCHLD** por una función propia del proceso en donde se ejecuten ***wait()*** o ***waitpid()***. De este modo el tiempo de bloqueo del proceso será mínimo ya que se invoca a la función cuando el proceso child ha concluido (prueba de ello es que estamos dentro de esta función a la que solo se ingresa si el proceso ha recibido la señal **SIGCHLD**).
Cabe destacar que si el proceso padre no espera la finalización del proceso hijo, éste último no podrá concluir ya que el kernel no ingresará al área de control del proceso padre que le permitirá terminar de actualizar toda la información que se requiere para cerrar adecuadamente al proceso hijo. El proceso hijo en esta condición quedará inconcluso y en estado Zombie, (o *defunct* como se lo suele denominar en la jerga UNIX). En este estado consume memoria y una entrada en la tabla de procesos pero no responde ni hace nada útil. Es claramente una situación que debemos evitar.
Por su parte **SIGQUIT** (3), es la señal que recibe un proceso cuando pulsamos CTRL-C desde del teclado, y **SIGSTOP** (19) cuando se pulsa CTRL-Z. Ésta última, al igual que **SIGKILL** no acepta modificación a su handler, por parte de un proceso, de modo que un proceso no puede decidir por si mismo no ser Detenible por parte de otro proceso o mediante una señal enviada desde el shell de modo teto.

4.3. System Calls para manejo de señales

Si bien las iremos estudiando mediante ejemplos a medida que avancemos en el tema, es esta sub sección, la idea es al menos enumerarlas para plantear su ámbito funcional.
Antes de comenzar es muy importante aclarar que en Linux un desarrollador dispone de toda la documentación de las system calls en el man. Generalmente en el capítulo 2 o 3. En este caso tenemos toda esta documentación en el 2.
En principio las system calls mas importantes son:

- **kill**
  Prototipo:

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

  Headers:

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

  La system call ***kill*** se usa para enviar la señal que recibe en el argumento ***sig*** a un proceso o grupo de procesos, definidos en el argumento ***pid***.
  Si ***pid*** > 0, entonces la señal ***sig*** es enviada a ***pid***. En este caso, se devuelve 0 si hay éxito, o un valor negativo en caso de producirse algún tippo de error.
  Si ***pid*** = 0, ***sig*** se envía a cada proceso que tenga el mismo GroupID (grupo de procesos) del proceso que envía la señal.
  Si ***pid*** = -1, ***sig*** se envía a cada proceso, excepto al proceso 1 (init).
  Si ***pid*** < -1, ***sig*** se envía a cada proceso en el grupo de procesos ***-pid***.
  Si ***sig*** = 0, no se envía ninguna señal pero se realiza la comprobación de errores.

- **signal**
  Prototipo:

  ```
  typedef void (*sighandler_t)(int);
  sighandler_t signal(int signum, sighandler_t handler);
  ```

Headers:

#include <signal.h>

La system call ***signal*** es utilizada por un proceso para reemplazar el handler que posee para manejar una señal (generalmente es es que le ha predefinido el sistema operativo), por otro que especifica en su propio código. De este modo colocando en el segundo argumento un puntero a la función deseada el proceso puede reemplazar el handler que recibió al ser creado por otro afín a sus propósitos. También pueden utilizarse dos macros para ignorar la señal (si ese es el comportamiento que se desea) o para utilizar el handler default en caso de querer recuperarlo: estas son **SIG_IGN** y **SIG_DFL** respectivamente.

- **sigaction**

- **pause**

4.3.1. Operaciones con señales

- **Primer ejemplo**: Modo simple de bloque de una señal.
  El listado E.1 muestra el código de un programa que se ejecuta recibiendo por línea de comandos una señal, reemplaza el handler de dicha señal y luego duerme en un lazo infinito para esperar que se envíe esa señal a la cual ignorará. Hemos empleado la macro **SIG_IGN**, que pone automáticamente un handler que ignora la señal (no tenemos que incluirlo en nuestro código.

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

int main (int argc, char * argv[])
{
  if (argc < 2 )
  {
    printf ("Cantidad insuficiente de argumentos\nUso: noint [nro-señal]\n");
    exit (1);
  }
  /**
  Modificamos el handler de la señal recibida como argumento, por otro al que
      ignore.*/
  if (signal(atoi(argv[1]),SIG_IGN) == SIG_ERR)
  {
    printf ("No se puede trapear la señal %d\n", atoi(argv[1]));
    exit (1);
  }
  while (1)
    sleep(1);
  return;
}
```

Listing 4.1: *noint.c*: Reemplaza un handler de señal recibido por línea de comando como argumento, por otro que ignore dicha señal

Para compilar, ejecutar:

```
$ gcc -o noint noint.c
```

Esta versión sumamente compacta permite al ejecutar el código comprobar algunas de las afirmaciones que hemos hecho hasta ahora. Por ejemplo, ejecutando el shell los siguientes comandos:

```
$ ./noint 9
No se puede trapear la señal 9
$
```

Verificamos que no es posible cambiar el handler que el Sistema Operativo le instala al proceso en el momento de crearlo, para la señal **SIGKILL**.

```
$ ./noint 19
No se puede trapear la señal 19
$
```

Lo mismo para **SIGSTOP**.

```
$ ./noint 2
^C^C^C
```

Y en el caso de no querer que nuestro proceso sea cancelado con la combinación CTRL-C desde el teclado, se logra ignorando **SIGQUIT**. En este caso solo podemos cancelarlo con el comando ***kill*** desde una consola de texto.

- **Segundo Ejemplo**: Reemplazo de todos los handlers de señal (que pueden reemplazarse)
  El listado 4.2 muestra el código de un programa que reemplaza todos los handlers de señal haciéndolos apuntar a una única rutina y luego duerme en un lazo infinito para esperar que se le envíen cualesquiera de las señales excepto **SIGKILL** y **SIGSTOP** las que como vimos no son reemplazables y por lo tanto lo evita en el código.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <signal.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <string.h>
8
9  int fdout;
10
11 void multisignal_hand(int senial)
12 {
13   sprintf (m,"Llegó señal:% d\n",senial);
14   printf("%s",m);       // mensaje por stdout
15   write (fdout,m, strlen(m));  // mensaje por archivo de log
16   return;
17 }
18
```

```
19 //************************************************************
20 int main(void){
21   int i;
22   fdout = open("signals.log", O_WRONLY | O_CREAT);
23   //archivo para loguear actividad
24   //instalo signal handlers
25   for (i=1;i<32;i++)
26   {
27     if (i==9 || i==19)
28       i++;
29     // las 9 y 19 son SIGKILL y SIGSTOP, no se pueden trapear
30     if (signal(i, multisignal_hand) == SIG_ERR ){
31       perror ("signal");
32       exit (1);
33     }
34   }
35   printf("Listo para comenzar a recibir señales...camon! hit me..!\n");
36   while (1)
37     sleep(1);
38   return(0);
39 }
```

Listing 4.2: ***siganaliz.c*****: Reemplaza todos los handlers de señales posibles para su análisis**

Para compilar, ejecutar:

```
$ gcc -o siganaliz siganaliz.c
```

Este ejemplo analiza las diferentes señales que le llegan a un proceso. Se envían desde una consola de texto utilizando el comando ***kill***. Es interesante verificar que este comando trabaja por default con la señal **SIGTERM** (15), es decir cuando no se le especifica la señal que debe enviar.

- **Manejo de SIGCHLD** En el listado 4.3, podemos comprobar que ocurre si no tenemos la precaución es esperar (wait) la terminación de un proceso hijo.

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main(void){
6
7     pid_t p;
8     int i;
9
10    p=fork();
11    if (!p) {
12    /*proceso child.*/
13       printf("\nproceso hijo\n\n");
14    /*Dormirá 10 segundos y terminará, saliendo del if y ejecutando return (0).En
          estas condiciones quedará en estado zombie, si el padre no terminó primero
          */
15       sleep(10);
```

```
16    }else{
17       printf("\nProceso padre\n\n");
18       for (i=0;i<50;i++){
19    /*El padre hace una demora de 50 segundos presentando en pantalla el contador y
           luego termina. Durante los últimos 40 segundos el child estará zombie.*/
20          sleep(1);
21          printf("..%d\n",i);
22       }
23    }
24    return 0;
25 }
```

Listing 4.3: *zombie.c*: Genera un proceso zombie

Para compilar, ejecutar:

```
$ gcc -o zombie zombie.c
```

Para analizar el estado de los programas se recomienda el uso de ***top*** o ***htop***, para verlo en forma interactiva, o bien ir monitoreando desde una terminal con el comando ***ps -elf | grep zombie***. Se verá a ambos procesos con su estado. El proceso padre estará en estado ***S*** (por sleeping, aunque en realidad el estado interno que el kernel le asigna se llama TASK_INTERUPTIBLE). EL proceso hijo estará en dicho estado durante los primeros 10 segundos de operación (se puede ver por el contador que imprime el padre a intervalos regulares de 1 seg. Luego de ese lapso pasará al estado **Z**, (por Zombie, aunque en realidad el estado interno que el kernel le asigna se llama EXIT_ZOMBIE), acompañado de la leyenda <defunct>.

- **Uso de wait() desde el handler de SIGCHLD** En el Listado 4.4, se espera por la terminación del proceso child reemplazando el handler de la señal **SIGCHLD** del proceso padre, por una función propia en la que se ejecuta la system call ***wait ()***.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <signal.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <string.h>
8
9  /* handler de SIGCHLD*/
10 void sigchld_hand(int senial){
11    printf("Estamos en el handler de SIGCHLD del proceso padre pid=%d....\n",getpid
          ());
12    printf("Antes del wait sobre el hijo, dormimos al proceso padre 10 seg. para
          ver que el hijo queda zombie durante ese tiempo....\n");
13    sleep(10);
14    printf("Ahora hacemos wait sobre el hijo...\n");
15    wait(NULL);
16    printf("Saliendo de handler.\nRevisar lista de procesos y ver que hijo termin
          ó....\n");
17    return;
18 }
19
```

```
20 //*********************************************************/
21 int main(void){
22   pid_t  pidhijo,pidpadre;
23   pidpadre=getpid();
24   /*instalo signal handler*/
25   if (signal(SIGCHLD, sigchld_hand) == SIG_ERR )
26   {
27     perror ("signal");
28     exit (1);
29   }
30
31   if (!fork())
32   {
33     pidhijo=getpid();
34     printf("Proc Hijo, pid=%d arrancando\n",pidhijo);
35     sleep(20);/* Tenemos 20 segundos, para visualizar con ps en otra termial que
           está durmiendo */
36     printf("Proc Hijo pid=%d terminando\n",pidhijo);
37   }
38   else
39   {
40     printf("Proc Padre, pid=%d arrancando y a dormir\n",pidpadre);
41     while (1)
42       sleep(1);
43   }
44   return(0);
45 }
```

Listing 4.4: ***waitchild.c*****: Versión mejorada. Evita que el proceso child quede zombie**

Para compilar, ejecutar:

```
$ gcc -o waitchild waitchild.c
```

Al ejecutar el programa, conviene tener una consola con top ejecutando ordenado por pid para ver rápidamente los resultados parciales a medida que evoluciona el proceso. Al principio se tiene la primer parte de la salida:

```
$ ./waitchild
Proc Padre, pid=19131 arrancando y a dormir
Proc Hijo, pid=19132 arrancando
```

Obviamente los números de proceso variarán cada vez que ejecute el programa, y se ilustran simplemente para ver la diferencia entre el proceso padre y el hijo.
Esta situación se mantiene por 20 segundos debido a la espera que el proceso child introduce en la línea 35 del Listado 4.4. Transcurrido ese tiempo, el proceso hijo ejecuta return (0) (línea 44 de Listado 4.4), y termina. A partir de ese momento el kernel entra en acción tratando de liberar todos los recursos del proceso hijo y necesita como ya se explicara acceder al área de control del proceso padre (que no es otra cosa que un área de memoria del kernel denominada en Linux *Process Descriptor*). En este momento el padre recibe desde el kernel la señal **SIGCHLD**. El kernel activa su handler que no es el predefinido sino el que nosotros agregamos (Líneas 10 a 18 del Listado 4.4). El proceso hijo ya ha sido puestoen estado EXIT_ZOMBIE por el kernel. Esta situación se evidencia mediante la línea 13 del Listado 4.4, ya que al dormir el padre por 10 segundos podemos ver al

proceso hijo en estado Zombie durante ese lapso. Trasncurrida la demora, en la línea 15 se ejecuta **`wait ()`**, y el hijo es terminado normalmente por el kernel.

Parte III

Bibliotecas y Herramientas de Documentación

Capítulo 5

Opencv

5.1. Características

OpenCV es una biblioteca open source para C/C++ para procesamiento de imágenes y visión computarizada, desarrollada inicialmente por Intel. Su primer versión estable fue liberada en 2006. En Octubre de 2009, se libera el segundo release mayor: OpenCV v2.
Su documentación no es muy exhaustiva. Mas bien la descripción de las diferentes funciones y sus principales variables tipos, y estructuras de datos. Se puede acceder en: http://opencv.willowgarage.com/wiki/.
OpenCV se encuentra disponible para los sistemas operativos Linux, Mac, y Windows, y sus funciones de acceso a recursos gráficos interactúan con las API de las GUI de cada sistema de modo de proveer un conjunto de recursos, si bien limitado en cuanto a posibilidades, 100 % portable a cada una de estas plataformas.
Tiene estructuras básicas de datos para operaciones con matrices y procesamiento de imágenes, permite visualizar datos muy sencillamente y extraer información de imágenes y videos, independientemente de los diversos formatos que soporta en cada caso. Tiene funciones de captura y presentación de imágenes.

5.1.1. Componentes y Nomenclaturas

Opencv se compone de 4 Módulos.

- **cv**. Contiene las Funciones principales de la biblioteca
- **cvaux**. Contiene las Funciones Auxiliares (experimental)
- **cxcore**. Contiene las Estructuras de Datos y Funciones de soporte para Álgebra lineal
- **Highgui**. Funciones para manejo de la GUI

Cada uno de estos está relacionado con un header.

```
1 #include <cv.h>
2 #include <cvaux.h>
3 #include <highgui.h>
4 #include <cxcore.h>    // innecesario, incluido en cv.h
```

Listing 5.1: *Opencv*: Headers para incluir en los programas

Existen convenciones para los Nombres de las Funciones y Datos. En general el formato es:
`cvActionTargetMod (...)`

- **Action:** Función core. Ej: set, create.
- **Target:** Elemento destino de la Acción. Ej: Contorno, polígono.
- **Mod :** Modificadores opcionales. Ej: Tipo de argumento.

Respecto de Parámetros de imágenes, están definidos diferentes parámetros y sus valores, como por ejemplo:

IPL_DEPTH_<bit_depth>(S|U|F)

Ejemplos de este parámetro:

```
IPL_DEPTH_8U    //imagen de 8 bits no signados.
IPL_DEPTH_32F   // imagen de 32 bits punto flotante.
```

Listing 5.2: *Opencv*: Ejemplos de definición de la profundidad de bit

5.2. Primer Ejemplo

5.2.1. Invocar funciones básicas y compilar

Nos proponemos simplemente abrir un archivo de imagen cualquiera y presentarlo en la pantalla en una ventana.

```
/**
 * \file imgproc
 * \brief Programa sencillo para abrir una imagen y presentarla en una ventana
 * \details Utilizamos la biblioteca Opencv para abrir la imagen aprovechando
 * sus funciones independientes del tipo de archivo, y también para crear una
 * ventana sencilla (sin menú), para presentarla en la GUI del sistema operativo
 * que se tenga (Linux, Windows, o MAC OS).
 * Este programa compila en Linux, mediante el comando siguiente:
 * gcc -o img img.c `pkg-config --cflags --libs opencv` -g -ggdb -Wall
 * \author Alejandro Furfaro. afurfaro@electron.frba.utn.edu.ar
 * \date 12.05.2011
*/

#include <cv.h>
#include <highgui.h>

int main()
{
  //Instancia una estructura IplImage, definida en openCV para
  //cargar la información de la imagen origen
  IplImage *image = cvLoadImage("LENA.JPG",0);
  //Creamos una ventana llamada "ejemplo1", autoajustable al
  //tamaño de la imagen que le vamos a colocar dentro
  cvNamedWindow("ejemplo1",CV_WINDOW_AUTOSIZE);
```

```
25    //Movemos la ventana al pixel 100 de la fila 100
26    cvMoveWindow("ejemplo1",100,100);
27    //Mostramos la imagen leída en la ventana creada
28    cvShowImage("ejemplo1", image);
29    //Esperamos que el usuario presione cualquier tecla...
30    cvWaitKey(0);
31    //Liberamos los recursos utilizados
32    cvReleaseImage(&image);
33    //Terminamos el programa
34    return 0;
35 }
```

Listing 5.3: Listado de código del programam de ejemplo Nº1

Para empezar a trabajar abrir con el editor de texto preferido, el archivo eje1.c que contiene el código del listado 5.3. Además se necesita una consola cuyo prompt esté posicionado en el directorio de trabajo en el que está el programa eje1.c. Para compilar, ejecutar en la consola el comando indicado en el encabezado del programa del listado 5.3, que reproducimos a continuación para mayor claridad:

```
1 gcc -oeje1 eje1.c -g -ggdb `pkg-config --cflags --libs opencv` -Wall
```

Listing 5.4: Comando de compilación del código de 5.3

5.2.2. Análisis

Los comentarios del programa listado en 5.3, son suficientes para entender que hace cada línea de código. En principio vemos lo simple del mismo, en especial la manipulación de ventanas a cargo de la biblioteca highgui, que nos independiza del manejo de los detalles que requiere el software de base. Revisemos la nomenclatura explicada anteriormente en el código ejemplo. Es muy fácil de ver cuales son funciones de esta biblioteca y cuales nó, simplemente teniendo en cuenta que las funciones Opencv comienzan con el prefijo ***cv***.

- Carga de una imagen
  `IplImage *image = cvLoadImage("Lena.bmp");`

- Crear y Ubicar una ventana
  ```
  cvNamedWindow ("ejemplo1", CV_WINDOW_AUTOSIZE);
  cvMoveWindow ("ejemplo1", 100, 100);//desde borde superior izquierdo
  ```

- Mostrar la imagen en la ventana creada
  `cvShowImage("ejemplo1", image);`

- Liberar recursos
  `cvReleaseImage(&image);`

Con respecto a la acción de cada función:

- Carga de una imagen
  `IplImage *image = cvLoadImage("Lena.bmp");`

- Crear y Ubicar una ventana
```
cvNamedWindow ("ejemplo1", CV_WINDOW_AUTOSIZE);
cvMoveWindow ("ejemplo1", 100, 100);//desde borde superior izquierdo
```

- Mostrar la imagen en la ventana creada
```
cvShowImage("ejemplo1", image);
```

- Liberar recursos
```
cvReleaseImage(&image);
```

Y finalmente respecto del Destino de la función:

- Carga de una imagen
```
IplImage *image = cvLoadImage("Lena.bmp");
```

- Crear y Ubicar una ventana
```
cvNamedWindow("ejemplo1", CV_WINDOW_AUTOSIZE);
cvMoveWindow("ejemplo1", 100, 100);//desde borde superior izquierdo
```

- Mostrar la imagen en la ventana creada
```
cvShowImage("ejemplo1", image);
```

- Liberar recursos
```
cvReleaseImage(&image);
```

5.2.3. IplImage

IplImage es **"La"** Estructura

```
typedef struct_IplImage
{
 int  nSize;
 int  ID;
 int  nChannels;
 int  alphaChannel;
 int  depth;
 char colorModel[4];
 char channelSeq[4];
 int  dataOrder;
 int  origin;
 int  align;
 int  width;
 int  height;
 struct _IplROI *roi;
 struct _IplImage *maskROI;
 void  *imageId;
 struct _IplTileInfo *tileInfo;
 int  imageSize;
 char *imageData;
```

```
21  int   widthStep;
22  int   BorderMode[4];
23  int   BorderConst[4];
24  char *imageDataOrigin;
25 }
26 IplImage;
```

Listing 5.5: Declaración de la estructura IplImage

Si bien está en la documentación de la Biblioteca, es oportuno desglosar aquí el significado de cada uno de sus miembros

- **Nsize:** sizeof (IplImage), es decir su tamaño en bytes.
- **ID:** Versión, siempre igual a 0
- **nchannels:** Número de canales. La mayoría de las funciones **OpenCV** soportan 1 a 4 canales.
- **alphaChannel:** Ignorado por **OpenCV**
- **depth:** Profundidad del canal en bits + el bit de signo opcional (`IPL_DEPTH_SIGN`).
  - **`IPL_DEPTH_8U`:** entero no signado de 8 bits.
  - **`IPL_DEPTH_8S`:** entero signado de 8 bits.
  - **`IPL_DEPTH_16U`:** entero no signado de 16 bits.
  - **`IPL_DEPTH_16S`:** entero signado de 16 bits.
  - **`IPL_DEPTH_32S`:** entero signado de 32 bits.
  - **`IPL_DEPTH_32F`:** Punto flotante simple precisión.
  - **`IPL_DEPTH_64F`:** Punto flotante doble precisión.
- **colorModel:** Ignorado por **OpenCV**. La función **CvtColor** de **OpenCV** requiere los espacios de color origen y destino como parámetros.
- **channelSeq:** Ignorado por **OpenCV**.
- **dataOrder:**
  - 0: `IPL_DATA_ORDER_PIXEL` - canales de color entrelazados.
  - 1: canales de color separados.
  - CreateImage solo crea imágenes con canales entrelazados. Por ejemplo, el layout común de colores de una imagen es: `b_00 g_00 r_00 b_10 g_10 r_10 ...`
- **origin:**
  - 0: origen extremo superior izquierdo.
  - 1: origen extremo inferior izquierdo, (estilo Windows bitmap).
- **align:** Alineación de las filas de la imagen(4 u 8). **OpenCV** ignora este campo usando en su lugar **widthStep**.
- **width:** Ancho de la Imagen en pixels.
- **height:** Alto de la Imagen en pixels.

- **roi:** Region Of Interest (ROI). Si no es `NULL`, se procesa solo esta región de la imagen.
- **maskROI:** Debe ser `NULL` en **OpenCV**.
- **imageId:** Debe ser `NULL` en **OpenCV**.
- **tileInfo:** Debe ser `NULL` en **OpenCV**.
- **imageSize:** Tamaño en bytes de la imagen. Para datos entrelazados, equivale a: `image->height * image->widthStep`
- **imageData:** Puntero a los datos alineados de la imagen.
- **widthStep:** Tamaño en bytes de una fila de la imagen alineada
- **BorderMode y BorderConst:** Modo de completamiento del borde, ignorado por **OpenCV**.
- **imageDataOrigin:** Puntero el origen de los datos de la imagen (no necesariamente alineados). Usado para desalojar la imagen.

5.2.4. Opencv gira alrrededor deIplImage

La estructura **`IplImage`** se hereda de la Librería original de Intel. Su formato es nativo, aunque **OpenCV** solo soporta un subset de formatos posibles de **`IplImage`**.
Además de las restricciones anteriores, OpenCV maneja las ROIs de modo diferente. Las funciones de **OpenCV** requieren que los tamaños de las imágenes o los de las ROI de todas las imágenes fuente y destino coincidan exactamente.
Por otra parte, la Biblioteca de Intel de Procesamiento de Imágenes procesa el área de intersección entre las imágenes origen y destino (o ROIs), permitiéndoles variar de forma independiente.
El tema es que cualquier imagen va a parar a una estructura de este tipo, incluso video. Al respecto, **OpenCV** permite visualizar videos desde dos fuentes de información: cualquier Cámara web conectada a la PC, o desde archivo formato avi, o flv. Una imagen de video se compone de cuadros de n*m pixeles. Cada cuadro se carga en una estructura **`IplImage`**

5.3. Aplicaciones y más funciones

5.3.1. Crear una imagen

El siguiente bloque de código permite crear una imagen. Su contenido inicial es vacío.

```
1 IplImage* cvCreateImage (CvSize size, int depth, int channels);
```

Listing 5.6: Función para creación de una imagen

- **size**: Tamaño en pixels del frame que va a contener la imagen:
- typedef struct CvSize. Es una estructura que contiene la dimensión de una imagen:

```
Typedef struct CvSize
{
  int width;
  int height;
}CvSize;
```

Listing 5.7: Estructura CvSize

Por su parte en general para muchas estructuras simples ocurre lo mismo, existe una función relacionada que oficia de "Constructora". En este caso:

```
 cvSize (int width, int height);
```

Listing 5.8: Estructura CvSize

- ***depth***:profundidad del pixel en bits: **`IPL_DEPTH_8U, IPL_DEPTH_32F`**.
- ***channels***: Número de canales por pixel. Sus valores posibles son 1, 2, 3 o 4. Los canales están entrelazados. El layout de datos usual de una imagen color es **`b0 g0 r0 b1 g1 r1`**.

Utilizando estas funciones tenemos los ejemplos de creación de una imagen.

```
//Crear una imagen con canal de 1 byte
IplImage* img1=cvCreateImage (cvSize(640,480), IPL\_DEPTH\_8U,1);
//Crear una imagen con tres canal de float
IplImage* img2=cvCreateImage(cvSize(640,480), IPL\_DEPTH\_32F,3);}}
```

Listing 5.9: Ejemplos de creación de una imagen con cvSize y CvSize

5.3.2. Operaciones Básicas

Es necesario además Cerrar y Clonar Imágenes. Los siguientes códigos ilustran estas operaciones con las funciones que se utilizan.

- Cerrar una imagen

```
cvReleaseImage(\&img);
```

Listing 5.10: Función para Cerrar una imagen

- Clonar una imagen

```
IplImage*img1=cvCreateImage(cvSize (640,480),IPL\_DEPTH\_8U,1);
IplImage* img2;
img2 = cvCloneImage (img1);}}
```

Listing 5.11: Código para Clonar una imagen

En la mayoría de las aplicaciones nos concentramos en cierta región de la pantalla, donde está la información que queremos procesar. Esta zona se denomina ROI, siglas de Región Of Interest. Es como una sub matriz de la matriz general. Setear u obtener la región de interés (ROI), requiere de algunas funciones en Opencv.

```
void cvSetImageROI (IplImage* image, CvRect rect);
CvRect cvGetImageROI (const IplImage* image);
```

Listing 5.12: Prototipos de las funciones para setear u obtener una ROI

Al trabajar con ROIs estamos definiendo "cajas" dentro de la Imagen. Para ello es importante la estructura **CvRect**, que define las coordenadas de la esquina superior izquierda y el tamaño del rectángulo, que arma la caja de la ROI.

```
typedef struct CvRect {
   int x;   //coordenada x de la esquina superior izquierda
   int y;   //coordenada y de la esquina superior izquierda
   int width;  //ancho del rectángulo
   int height; //alto del rectángulo
}
inline CvRect cvRect( int x, int y, int width, int height ); //inicialización
```

Listing 5.13: Estructura CvRect y su función constructora

5.3.3. Manejando pixeles

Existen algunas otras Estructuras asociadas, para manejo de pixeles. Es una estructura **CvScalar**. Es un contenedor de un arreglo de 1, 2, 3, o 4 doubles. Cada double pertenece al valor R G B y Alfa. En caso de imágenes monocromo contiene el valor en escala de gris en formato double. Su prototipo es:

```
typedef struct CvScalar {
   double val[4];
   } CvScalar;
```

Listing 5.14: Estructura CvScalar

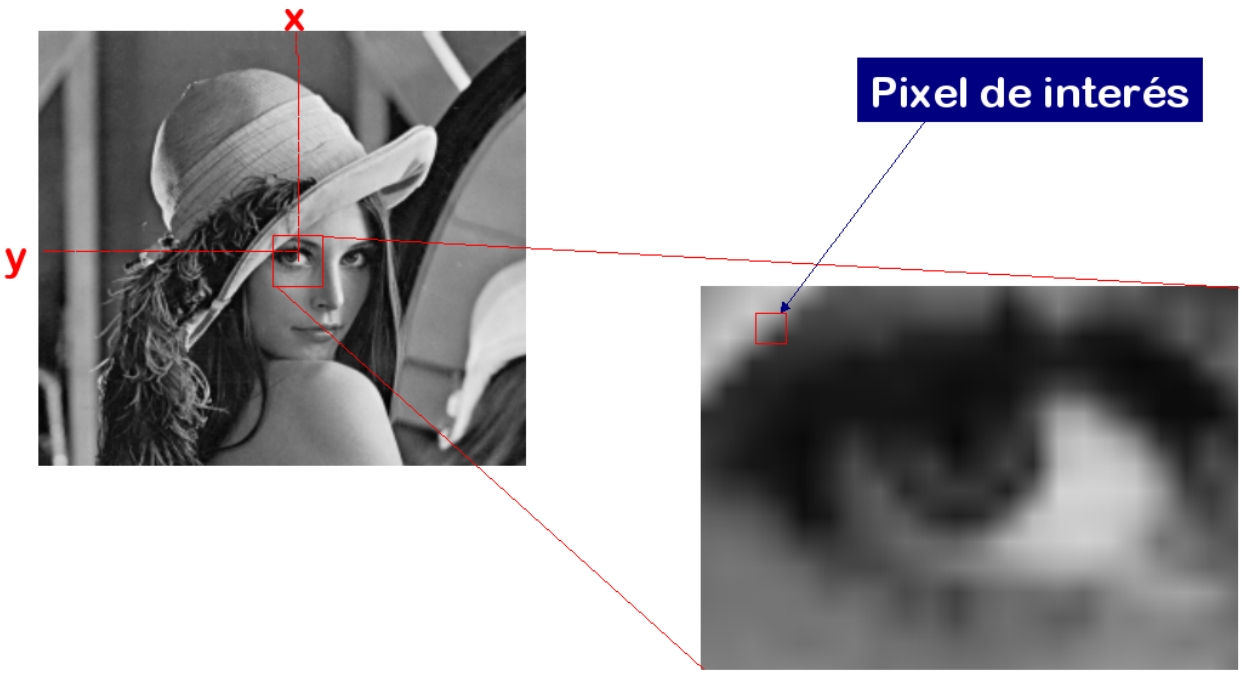
Para pixeles color y monocromáticos a continuación algunos ejemplos de código para inicializarlos en valores determinados.

```
//Inicializar val[0] con val0, val[1] con val1, etc.
  inline CvScalar cvScalar(double val0, double val1=0, double val2=0, double val3=0);
//Inicializar los cuatro elementos val[0]...val[3] con el valor val0123.
  inline CvScalar cvScalarAll(double val0123);
//Inicializar val[0] con val0,y el resto (val[1]...val[3]) con ceros
  inline CvScalar cvRealScalar(double val0);
```

Listing 5.15: Inicialización de pixeles

Obtener el valor de un pixel es fundamental a menudo para aplicar cualquier acción de transformación, procesamiento, etc. En al figura siguiente vemos su significado. La función CvGet2D es útil a estos propósitos entregando CvScalar.

```
//Prototipo de cvGet2D
```




```
CvScalar s = cvGet2D (img, row, col)
```

Listing 5.16: Prototipo de cvGet2D

- Si la imagen está en escala de grises, **`s.val[0]`** es el valor del pixel.
- Si la imagen está en color,**`s.val[0]`**,**`s.val[1]`**, y **`s.val[2]`** son respectivamente R, G,y B.
- **`Img`** es un puntero a la **`IplImage`** obtenida al abrir o crear la imagen.
- **`row`** y **`col`** con **x** e **y** del slide anterior.

5.4. Segundo Ejemplo

5.4.1. Manejando video

Fuente: archivo avi

```
/**
 * \file aviexample.c
 * \brief Programa sencillo para abrir un archivo avi y reproducirlo en una ventana
 * \details Utilizamos la biblioteca Opencv para abrir el archivo aprovechando
 * sus funciones independientes del tipo de archivo (con algunas limitaciones a
 * los tipos mpeg, y también para crear una ventana sencilla (sin menú), para
 * reproducirlo en la GUI del sistema operativo que se tenga (Linux, Windows, o
 * MAC OS).
 * Este programa compila en Linux, mediante el comando siguiente:
```

```
10  * gcc -o videmo aviexample.c `pkg-config --cflags --libs opencv` -g -ggdb -Wall
11  * \author Alejandro Furfaro. afurfaro@electron.frba.utn.edu.ar
12  * \date 22.06.2011
13 */
14 \#include <cv.h>
15 \#include <highgui.h>
16 int main( int argc, char** argv )
17 {
18   //Creamos una ventana llamada "VideoPlayer", autoajustable al
19   //tamaño de la imagen que le vamos a colocar dentro
20   cvNamedWindow("VideoPlayer", CV\_WINDOW\_AUTOSIZE);
21   //Esta función Obtiene el Identificador del dispositivo de captura, en este
22   //caso un archivo avi.
23   CvCapture* capture = cvCreateFileCapture( argv[1] );
24 /**
25  * Solicitamos las características del dispositivo mediante la siguiente
26  * función:
27  *   double cvGetCaptureProperty(CvCapture* capture, int property_id)
28  * Obtiene las propiedades del dispositivo de captura de video.
29  *    Parámetros:
30  *         capture - Estructura de captura de video.
31  *         property_id - Identificador de la propiedad que se quiere obtener.
32  *                       Puede ser alguno de los siguientes valores:
33  *             CV_CAP_PROP_POS_MSEC - Posición actual del Film en mseg. o
34  *                                    timestamp de captura de video.
35  *             CV_CAP_PROP_POS_FRAMES - Indice base-0 del frame a decodificar o
36  *                                      capturar
37  *             CV_CAP_PROP_POS_AVI_RATIO - Posición relativa del archivo de
38  *                                         video (0 - start of film, 1 - end of
39  *                                         film).
40  *             CV_CAP_PROP_FRAME_WIDTH - ancho de los frames del stream de video
41  *             CV_CAP_PROP_FRAME_HEIGHT - alto de los frames del stream de video
42  *             CV_CAP_PROP_FPS - Frame rate
43  *             CV_CAP_PROP_FOURCC - Código de 4bytes del codec
44  *             CV_CAP_PROP_FRAME_COUNT - Número de frames del archivo de video.
45  *             CV_CAP_PROP_BRIGHTNESS - Brillo de la imagen (solo para cámaras)
46  *             CV_CAP_PROP_CONTRAST - Contraste de la imagen (solo para cámaras)
47  *             CV_CAP_PROP_SATURATION - Saturación de la imagen (solo para
48  *                                      cámaras)
49  *           CV_CAP_PROP_HUE - Hue de la imagen (solo para cámaras)
50 */
51   double fps=cvGetCaptureProperty(capture,CV_CAP_PROP_FPS);
52   printf ("Frame por seg: %d\n",(int)fps);
53   getchar();
54   //Instancia una estructura IplImage, definida en openCV para
55   //cargar la información de cada frame
56   IplImage* frame;
57   while(1)
58   {
59     // Lee un frame desde el dispositivo de captura cuyo identificador obtuvo
60     //al principio
61     frame = cvQueryFrame( capture );
62     if( !frame )
```

```
63        break; //Si el puntero es NULL, falló => Sale
64      //La lectura fue correcta, por lo tanto presenta el frame en la ventana
65      //que definió
66      cvShowImage( "VideoPlayer", frame );
67      //Esta línea adapta la velocidad de reproducción. Para entenderla, comentarla
            compilar y reproducir :)
68      char c= cvWaitKey ((int)(1000/fps));
69      // De paso... si pulsa ESC sale
70      if( c == 27 )
71        break; //si está aquí se pulsó ESC
72    }
73    //O llegó al fin de la peli... o falló alguna lectura de frame...
74    //Libera recursos y sale
75    cvReleaseCapture( &capture );
76    cvDestroyWindow( "VideoPlayer" );
77    return 0;
78  }
```

Listing 5.17: Ejemplo2: Código para abrir y reproducir un archivo avi

Para empezar a trabajar abrir con el editor de texto preferido, el archivo aviexample.c que contiene el código del listado 5.17. Además se necesita una consola cuyo prompt esté posicionado en el directorio de trabajo en el que está el programa aviexample.c. Para compilar, ejecutar en la consola el comando indicado en el encabezado del programa del listado 5.17, que reproducimos a continuación para mayor claridad:

```
1 gcc -o videmo aviexample.c -g -ggdb `pkg-config --cflags --libs opencv` -Wall
```

Listing 5.18: Comando de compilación del código de 5.17

5.4.2. Análisis

¿Que hicimos?

- Creamos una ventana llamada VideoPlayer.

```
1   cvNamedWindow("VideoPlayer", CV_WINDOW_AUTOSIZE);
```

Listing 5.19: Creación de una ventana VideoPlayer

- Tomar un dispositivo de captura de Video.

```
1   CvCapture* capture = cvCreateFileCapture (argv[1]);
```

Listing 5.20: Obtención de un identificador para el dispositivo de captura

- Obtiene los Frames por segundo del video contenido en el archivo.

```
1  double fps=cvGetCaptureProperty(capture,CV_CAP_PROP_FPS);
```

Listing 5.21: Obtención de un identificador para el dispositivo de captura

- Crear un puntero a una estructura **IplImage** en donde se guardarán los frames.

```
1  IplImage* frame;
```

Listing 5.22: Instancia de una estructura IplImage para almacenar los frames leídos

- Luego entramos a un bucle infinito
- Se obtiene cada frame del avi

```
1  frame = cvQueryFrame( capture );
```

Listing 5.23: Lee un frame desde el dispositivo de captura

- Y lo mostramos (esto ya lo aprendimos)...

```
1    cvShowImage( "VideoPlayer", frame );
```

Listing 5.24: Presenta el cuadro (frame) leido en la ventana

- Finaliza cuando el puntero al frame es NULL (encontró EOF, o el archivo está corrupto).

```
1   if( !frame ) break;
```

Listing 5.25: Presenta el cuadro (frame) leido en la ventana

- Esperamos una tecla (Opencv tiene una función para esto también): Sin embargo la línea siguiente sirve fundamentalmente para sincronizar los frame a la velocidad de Frames per second obtenida.

```
1    char c = cvWaitKey(33);
```

Listing 5.26: Presenta el cuadro (frame) leido en la ventana

- Liberamos recursos

```
1    cvReleaseCapture ( &capture );
2    cvDestroyWindow ( "avidemo" );
```

Listing 5.27: Libera recursos antes de salir

Capítulo 6

Doxygen

6.1. ¿Que es doxygen?

Doxygen es una herramienta para convertir los comentarios de un programa escrito en C, C++, VHDL, PHP, C#, y Java (entre otros lenguajes), en documentación suficientemente prolija y presentable como para poser publicarse. Maneja la salida de documentación en formatos: html, LaTeX, pdf, rtf, entre otros. Y lo mas interesante es que la documentación se genera de manera automática, y a partir de los archivos fuente de modo que siempre será consistente con éstos.
Entre Ventajas de usar Doxygen podemos conntar:

1. Al escribir la documentación en el mismo archivo fuente en forma de comentarios, nos releva de mantener otra documentación separada de los fuentes.
2. La documentación se genera en forma automática agregando además diagramas de interacción entre los diferentes módulos
3. Permite documentar variables, estructuras de datos además de las funciones.
4. Es una herramienta de documentación automática. Hay otras pero su uso genera el hábito metodológico de trabajo.

6.1.1. ¿Quienes usan Doxygen?

6.2. Instalación

Hace falta instalar los siguientes paquetes

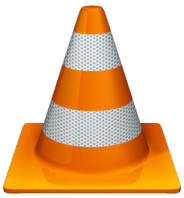
```
1 sudo apt-get install doxygen
2 sudo apt-get install graphviz
```

Listing 6.1: *Doxygen*: Comandos para instalar Doxygen

Opcionalmente (aunque no estaría de mas...)

```
1 sudo apt-get install kate
```

Listing 6.2: *Doxygen*: Comando para instalar Kate



6.3. Comenzando...

1. Escribir comentarios claros, y suficientes para describir cada función, antes de comenzar a “codearla”.
2. Una vez dentro de ella, insertar comentarios de modo de describir la operación (al menos los “big steps”)
3. Observar buenas prácticas de programación: Identar código, Usar convenciones para nombres de variables y funciones, Escribir código de manera “legible”.
4. Entender al comentario como parte escencial de la aplicación: Cada agregado o modificación debe ser documentado describiendo sus características, fecha y autor.

6.4. Configuración

Abrimos una terminal e ingresamos al directorio que contiene nuestros archivos de trabajo.
En ese directorio generamos el archivo de configuración para Doxygen.
Para ello tipear:

```
1 doxygen -g
```

Listing 6.3: *Doxygen*: Generación de un archivo de configuración genérico

6.4.1. Doxyfile

Es el archivo de texto plano utilizado para la configuración de doxygen. Todo lo que comienza con # se toma como comentario. Cada línea válida tiene el formato siguiente:
`PARAMETRO = VALOR` Si bien el archivo aparece como gigantesco al editarlo, los Parámetros de interés son los siguientes:

Parámetro	Descripción / Valor
PROJECT_NAME	Nombre del Proyecto. Ejemplo: Trabajo Práctico N^{o} 4.
PROJECT_NUMBER	Número o versión del proyecto. Ej: Ejercicio 3.4..
OUTPUT_DIRECTORY	Es el directorio a partir del cual se generará el árbol de archivos y subdirectorios (ver opción siguiente) que componen la documentación. Generalmente usamos =./doxy, de modo que se genere dentro del directorio del proyecto.
CREATE_SUBDIRS	Valor default YES. En este caso crea dos niveles de subdirectorios a partir del de documentación para almacenar, los diferentes archivos que la componen.
OUTPUT_LANGUAGE	Lenguaje en el que queremos generar la documentación. Default English.
TAB_SIZE	Es la cantidad de espacios por los que reemplazará cada Tab encontrado en el código. Default: 8.
OPTIMIZE_OUTPUT_FOR_C	Si el proyecto es enteramente escrito en C conviene activar esta opción para que Doxygen optimice la salida para este lenguaje. En tal caso es = YES.
EXTRACT_ALL	Si está en YES, doxygen agregará a la documentación todo lo que encuentre (funciones, variables, etc), aunque no se hayan documentado.
INPUT	Es el directorio del proyecto, que se toma como entrada de información para Doxygen. En caso de poner Doxyfile en el mismo directorio del proyecto (como es nuestro caso) colocamos =./
INPUT_ENCODING	Establece el sistema de codificación de caracteres con el que se generará la documentación: Si se trabaja en inux colocamos UTF-8, para Windows ISO-8859.
GENERATE_LATEX	Conviene ponerlo en NO (el default es YES) si no queremos generar la documentación en LaTeX.

6.5. Generando la documentación

Es la parte mas fácil. Solo hace falta tipear en una consola dentro del directorio del proyecto el siguiente comando:

```
1 doxygen Doxyfile
```

Listing 6.4: *Doxygen*: Comando para la generación de la documentación del proyecto

Luego, simplemente hay que abrir el archivo ./doxy/index.html con un navegador cualquiera y tenemos lista la documentación.

6.6. Preparando el código para su documentación

Básicamente la clave consiste en como armar los comentarios:

```
1 /**
2 * Estilo JavaDoc: Iniciar con '/**'
3 Apto para programas en C
4 Los asteriscos intermedios son opcionales
```

```
5  */
6
7  /*!
8  * Estilo QT
9  Los asteriscos intermedios son opcionales
10 */
11
12 ///
13 // Estilo C++
```

Listing 6.5: *Doxygen*:Estilos de comentarios por lenguaje para generar documentación

Luego, conocer algunas de las Macros que permiten mejorar y organizar la documentación general del programa.

```
1  /**
2  \file programa.c
3  \brief Este archivo contiene el programa
4         principal
5  \details Aquí nos explayamos sobre la tarea
6           que reaizará el programa o la
7           función
8  \author Alejandro Furfaro afurfaro@ieee.org
9  \date 30 de Marzo de 2011
10 \version 1.0.0
11 */
```

Listing 6.6: *Doxygen*: Comentario tipo para encabezamiento de programa

Las mismas y algunas otras Macros ayudan con la Documentación general de las funciones.

```
1  /**
2  \fn void print_time (* struc tv)
3  \bref Funcion para obtener la fecha y la hora
4        con formato.
5  \details En primer instancia se llama a gettimeofday,
6           pasando como argumento el puntero a una
7           estructura timeval (tv). Luego con strftime ()
8           se le da un formato apto para presentar un
9           time stamp con el formato
10          año-mes-día horas:minutos:segundos.
11 \param [in] * struc tv: puntero a una
12          estructura timeval (tv)
13 \return Nada (No regresa no regresa valores
14 \author Alejandro Furfaro afurfaro@ieee.org
15 \date 2011.05.08
16 \version 1.0.0
17 */
```

Listing 6.7: *Doxygen*: Comentario tipo para encabezamiento de función

6.6.1. Documentando variables

Versión compacta:

```
unsigned char Buffer [Buffersize]; //!< Buffer para recibir caracteres
```

Listing 6.8: *Doxygen*: Comentario tipo para una variable

Versión extendida

```
/**
\var unsigned char Buffer [Buffersize];
\brief Buffer para recibir caracteres
\details Aquí si es necesario podemos explayarnos
         acerca de la variable.
*/
unsigned char Buffer [Buffersize];
```

Listing 6.9: *Doxygen*: Documentación extendida para una variable

Estructuras

```
/**
\struct coordenada
\brief Variable para almacenar un par de coordenadas (x,y)
\details Aquí si es necesario podemos explayarnos acerca de la estructura.
*/
struct coordenada{
 int x; //!< Coordenada x;
 int y; //!< Coordenada y;
 };
```

Listing 6.10: *Doxygen*: Comentario tipo para documentación de una estructura

En general

```
\struct
\enum
\union
\class
\def
\typedef
```

Listing 6.11: *Doxygen*: Macro tipo para documentación de las diferentes variables y tipos de C

Appendices

Apéndice A

Apéndice A

```
$ objdump -hrt /usr/lib/crt1.o

/usr/lib/crt1.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .note.ABI-tag 00000020  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         00000024  00000000  00000000  00000054  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  2 .rodata       00000004  00000000  00000000  00000078  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .rodata.cst4  00000004  00000000  00000000  0000007c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000004  00000000  00000000  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000000  00000000  00000000  00000084  2**2
                  ALLOC
  6 .comment      0000001f  00000000  00000000  00000084  2**0
                  CONTENTS, READONLY
  7 .debug_pubnames 00000025  00000000  00000000  000000a3  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_info   0000008d  00000000  00000000  000000c8  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_abbrev 0000004b  00000000  00000000  00000155  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_line   00000027  00000000  00000000  000001a0  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .debug_str    000000dc  00000000  00000000  000001c7  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .note.GNU-stack 00000000  00000000  00000000  000002a3  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    d  .note.ABI-tag  00000000 .note.ABI-tag
00000000 l    d  .text  00000000 .text
00000000 l    d  .rodata        00000000 .rodata
```

```
00000000 l    d  .rodata.cst4    00000000 .rodata.cst4
00000000 l    d  .data   00000000 .data
00000000 l    d  .bss    00000000 .bss
00000000 l    d  .comment        00000000 .comment
00000000 l    d  .debug_pubnames         00000000 .debug_pubnames
00000000 l    d  .debug_info     00000000 .debug_info
00000000 l    d  .debug_abbrev   00000000 .debug_abbrev
00000000 l    d  .debug_line     00000000 .debug_line
00000000 l    d  .debug_str      00000000 .debug_str
00000000 l    d  .note.GNU-stack         00000000 .note.GNU-stack
00000000 l    df *ABS*   00000000 init.c
00000000 g     O .rodata          00000004 _fp_hw
00000000         *UND*   00000000 __libc_csu_fini
00000000 g     F .text    00000000 _start
00000000         *UND*   00000000 __libc_csu_init
00000000         *UND*   00000000 main
00000000  w      .data    00000000 data_start
00000000 g     O .rodata.cst4    00000004 _IO_stdin_used
00000000         *UND*   00000000 __libc_start_main
00000000 g       .data    00000000 __data_start


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000c R_386_32          __libc_csu_fini
00000011 R_386_32          __libc_csu_init
00000018 R_386_32          main
0000001d R_386_PC32        __libc_start_main


RELOCATION RECORDS FOR [.debug_pubnames]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_info


RELOCATION RECORDS FOR [.debug_info]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_abbrev
0000000c R_386_32          .debug_str
00000011 R_386_32          .debug_str
00000015 R_386_32          .debug_str
00000019 R_386_32          .text
0000001d R_386_32          .text
00000021 R_386_32          .debug_line
00000028 R_386_32          .debug_str
0000002f R_386_32          .debug_str
00000036 R_386_32          .debug_str
0000003d R_386_32          .debug_str
00000044 R_386_32          .debug_str
0000004b R_386_32          .debug_str
00000059 R_386_32          .debug_str
00000060 R_386_32          .debug_str
00000067 R_386_32          .debug_str
```

```
90 00000071  R_386_32          .debug_str
91 00000076  R_386_32          .debug_str
92 00000083  R_386_32          _IO_stdin_used
```

Listing A.1: Header ELF de /usr/lib/crt1.o

Apéndice B

Apéndice B

```
$ objdump -hrt /usr/lib/crti.o

/usr/lib/crti.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000034  2**2
                  ALLOC
  3 .init         00000022  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  4 .fini         00000013  00000000  00000000  00000058  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .comment      0000001f  00000000  00000000  0000006b  2**0
                  CONTENTS, READONLY
  6 .note.GNU-stack 00000000  00000000  00000000  0000008a  2**0
                  CONTENTS, READONLY
  7 .debug_line   000000aa  00000000  00000000  0000008a  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_info   0000008d  00000000  00000000  00000134  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_abbrev 00000012  00000000  00000000  000001c1  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_aranges 00000028  00000000  00000000  000001d8  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
 11 .debug_ranges 00000020  00000000  00000000  00000200  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 initfini.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .init  00000000 .init
```

```
00000000 l    d  .fini  00000000 .fini
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .debug_ranges  00000000 .debug_ranges
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .debug_aranges 00000000 .debug_aranges
00000000  w      *UND*  00000000 __gmon_start__
00000000 g     F .init  00000000 _init
00000000         *UND*  00000000 _GLOBAL_OFFSET_TABLE_
00000000 g     F .fini  00000000 _fini


RELOCATION RECORDS FOR [.init]:
OFFSET   TYPE              VALUE
0000000f R_386_GOTPC       _GLOBAL_OFFSET_TABLE_
00000015 R_386_GOT32       __gmon_start__
0000001e R_386_PLT32       __gmon_start__


RELOCATION RECORDS FOR [.fini]:
OFFSET   TYPE              VALUE
0000000f R_386_GOTPC       _GLOBAL_OFFSET_TABLE_


RELOCATION RECORDS FOR [.debug_line]:
OFFSET   TYPE              VALUE
0000007c R_386_32          .init
00000097 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_info]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_abbrev
0000000c R_386_32          .debug_line
00000010 R_386_32          .debug_ranges


RELOCATION RECORDS FOR [.debug_aranges]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_info
00000010 R_386_32          .init
00000018 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_ranges]:
OFFSET   TYPE              VALUE
00000008 R_386_32          .init
0000000c R_386_32          .init
00000010 R_386_32          .fini
00000014 R_386_32          .fini
```

Listing B.1: Header ELF de /usr/lib/crti.o

Apéndice C

Apéndice C

```
$ objdump -hrt /usr/lib/crtn.o

/usr/lib/crtn.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000034  2**2
                  ALLOC
  3 .init         00000004  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .fini         00000004  00000000  00000000  00000038  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .comment      0000001f  00000000  00000000  0000003c  2**0
                  CONTENTS, READONLY
  6 .note.GNU-stack 00000000  00000000  00000000  0000005b  2**0
                  CONTENTS, READONLY
  7 .debug_line   00000054  00000000  00000000  0000005b  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_info   00000065  00000000  00000000  000000af  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_abbrev 00000012  00000000  00000000  00000114  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_aranges 00000028  00000000  00000000  00000128  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
 11 .debug_ranges 00000020  00000000  00000000  00000150  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 initfini.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .init  00000000 .init
```

```
00000000 l    d  .fini  00000000 .fini
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .debug_ranges  00000000 .debug_ranges
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .debug_aranges 00000000 .debug_aranges


RELOCATION RECORDS FOR [.debug_line]:
OFFSET   TYPE              VALUE
00000033 R_386_32          .init
00000045 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_info]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_abbrev
0000000c R_386_32          .debug_line
00000010 R_386_32          .debug_ranges


RELOCATION RECORDS FOR [.debug_aranges]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .debug_info
00000010 R_386_32          .init
00000018 R_386_32          .fini


RELOCATION RECORDS FOR [.debug_ranges]:
OFFSET   TYPE              VALUE
00000008 R_386_32          .init
0000000c R_386_32          .init
00000010 R_386_32          .fini
00000014 R_386_32          .fini
```

Listing C.1: Header ELF de /usr/lib/crtn.o

Apéndice D

Apéndice D

```
$ objdump -hrt /usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o

/usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000083  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  000000c4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000008  00000000  00000000  000000c8  2**2
                  ALLOC
  3 .ctors        00000004  00000000  00000000  000000c8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .dtors        00000004  00000000  00000000  000000cc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .jcr          00000000  00000000  00000000  000000d0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .fini         00000005  00000000  00000000  000000d0  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  7 .init         00000005  00000000  00000000  000000d5  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  8 .comment      0000001f  00000000  00000000  000000da  2**0
                  CONTENTS, READONLY
  9 .note.GNU-stack 00000000  00000000  00000000  000000f9  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 crtstuff.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .ctors 00000000 .ctors
00000000 l     O .ctors 00000000 __CTOR_LIST__
00000000 l    d  .dtors 00000000 .dtors
00000000 l     O .dtors 00000000 __DTOR_LIST__
00000000 l    d  .jcr   00000000 .jcr
```

```
37 00000000 l     O .jcr   00000000 __JCR_LIST__
38 00000000 l     F .text  00000000 __do_global_dtors_aux
39 00000000 l     O .bss   00000001 completed.5706
40 00000004 l     O .bss   00000004 dtor_idx.5708
41 00000000 l    d  .fini  00000000 .fini
42 00000060 l     F .text  00000000 frame_dummy
43 00000000 l    d  .init  00000000 .init
44 00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
45 00000000 l    d  .comment       00000000 .comment
46 00000000 g     O .data  00000000 .hidden __dso_handle
47 00000000       *UND*  00000000 .hidden __DTOR_END__
48 00000000  w      *UND*  00000000 _Jv_RegisterClasses
49
50
51 RELOCATION RECORDS FOR [.text]:
52 OFFSET   TYPE              VALUE
53 00000009 R_386_32          .bss
54 00000012 R_386_32          .bss
55 00000017 R_386_32          __DTOR_END__
56 0000001c R_386_32          .dtors
57 00000034 R_386_32          .bss
58 0000003b R_386_32          .dtors
59 00000041 R_386_32          .bss
60 0000004b R_386_32          .bss
61 00000067 R_386_32          .jcr
62 00000070 R_386_32          _Jv_RegisterClasses
63 0000007b R_386_32          .jcr
64
65
66 RELOCATION RECORDS FOR [.fini]:
67 OFFSET   TYPE              VALUE
68 00000001 R_386_PC32        .text
69
70
71 RELOCATION RECORDS FOR [.init]:
72 OFFSET   TYPE              VALUE
73 00000001 R_386_PC32        .text
```

Listing D.1: Header ELF de /usr/lib/crtbegin.o

Apéndice E

Apéndice E

```
$ objdump -hrt /usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o

/usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000002a  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000006c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000006c  2**2
                  ALLOC
  3 .ctors        00000004  00000000  00000000  0000006c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .dtors        00000004  00000000  00000000  00000070  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  00000000  00000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .jcr          00000004  00000000  00000000  00000078  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .init         00000005  00000000  00000000  0000007c  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  8 .comment      0000001f  00000000  00000000  00000081  2**0
                  CONTENTS, READONLY
  9 .note.GNU-stack 00000000  00000000  00000000  000000a0  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 crtstuff.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .ctors 00000000 .ctors
00000000 l     O .ctors 00000000 __CTOR_END__
00000000 l    d  .dtors 00000000 .dtors
00000000 l    d  .eh_frame      00000000 .eh_frame
00000000 l     O .eh_frame      00000000 __FRAME_END__
```

```
37 00000000 l    d  .jcr    00000000 .jcr
38 00000000 l     O .jcr    00000000 __JCR_END__
39 00000000 l     F .text   00000000 __do_global_ctors_aux
40 00000000 l    d  .init   00000000 .init
41 00000000 l    d  .note.GNU-stack          00000000 .note.GNU-stack
42 00000000 l    d  .comment         00000000 .comment
43 00000000 g     O .dtors  00000000 .hidden __DTOR_END__
44
45
46 RELOCATION RECORDS FOR [.text]:
47 OFFSET   TYPE                VALUE
48 00000008 R_386_32            .ctors
49 00000012 R_386_32            .ctors
50
51
52 RELOCATION RECORDS FOR [.init]:
53 OFFSET   TYPE                VALUE
54 00000001 R_386_PC32          .text
```

Listing E.1: Header ELF de /usr/lib/crtend.o

Bibliografía

[1] Tools Interface Standard (TIS) Comitee, Executable and Linkable Format (ELF), Portable Formats Specification, Version 1.1