

Herramientas de desarrollo

Para el desarrollo de software en un lenguaje de programación se requiere de un conjunto de herramientas conocido como *toolchain* que contiene al menos:

- **Compilador:** genera código fuente Encargado de traducir *programas fuente* en *assembly*.
- **Ensamblador (assembler):** Genera *archivos objeto* (binarios) en un formato dado (ej: ELF). ELF es un formato de los archivos objetos, en caso de linux y algunos linux es este. Otro sistema como windows tiene varios formatos.
- **Enlazador (linker):** Combina archivos objetos y *bibliotecas* para generar ejecutables o *bibliotecas*.

Nota: En el caso de los lenguajes C/C++ un programa fuente se *pre-procesa* para *expandir las macros y directivas* incluidas. los #include #define

Dados los archivos fuente:

```
/* main.c */
#include <stdio.h> /* for printf() */

extern char* hello(void);

int main(void)
{
    printf("%s\n", hello());
    return 0;
}
```

y

```
/* hello.c */
#define hi "Hello world"

char *hello(void)
{
    return hi;
}
```

El siguiente diagrama muestra el proceso de compilación de la aplicación mediante el comando `gcc -o hello main.c hello.c`. Este comando genera el programa (archivo ejecutable) `hello` desde los programas fuente `main.c` y `hello.c`.

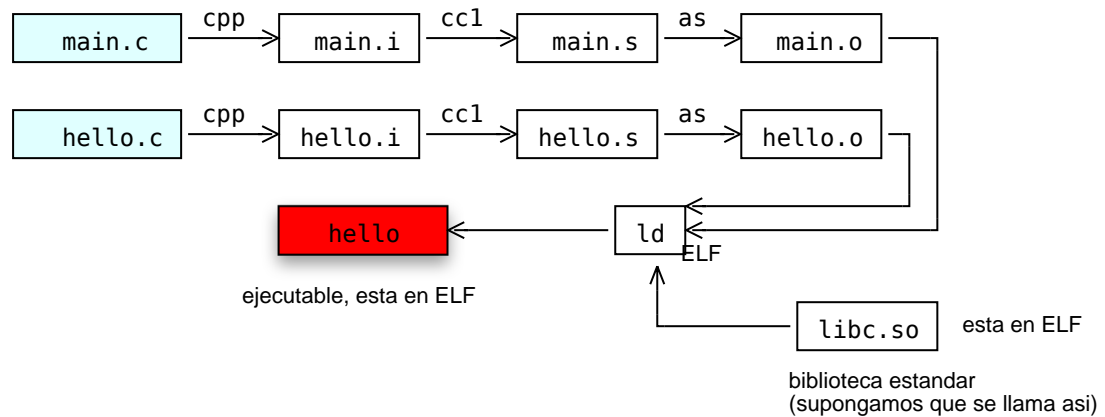


Figura 1: *Pasos en la compilación de un programa C.*

Como se puede apreciar en la figura 1, la compilación de un programa C consiste de varios pasos producidos por el comando `gcc`, el cual es un *driver* del proceso:

1. *Pre-procesamiento*: Se procesan las directivas del programa fuente y se *expanden* las macros encontradas. La salida es un archivo temporal (generalmente con extensión `.i`).
2. *Compilación*: El compilador (`cc1`) traduce de C a *assembly*.
3. *Assembler*: El *ensamblador* (`as`) produce los *archivos objeto* (binarios) correspondientes en el formato soportado por el SO como *ELF* en sistemas GNU-Linux).
4. *Enlazado (linking)*: El *linker* (`ld`) enlaza (combina y resuelve las referencias externas) de los archivos objeto y bibliotecas (en este ejemplo, la *biblioteca estándar* `libc.so` y genera el *ejecutable* final.

Estos pasos pueden ser controlados mediante el comando `gcc`. Los siguientes *flags* permiten detener en proceso en sus diferentes etapas:

- `-E` : Parar luego de la etapa de pre-procesamiento.
- `-S` : Parar luego de generar el *assembly*.
- `-c` : Parar luego de generar el archivo objeto (`.o`), es decir, no invocar al *linker*.

Archivos objeto

Un archivo objeto (`.o`) es un binario en el formato definido por la *ABI* del sistema. En sistemas GNU-Linux el formato es *ELF*, en Mac-OS es *Mach-O* y en MS-Windows son los formatos *Portable Executable (PE)* y *Common Object File Format (coff)* (ver [PE format](#)).

Si bien estos formatos difieren en detalles, un archivo objeto o ejecutable tienen las siguientes partes (denominados secciones y/o segmentos).

- Un *header* que describe el tipo de archivo, arquitectura, *entry point*, etc.
- Tipos de secciones:
 - *Data*: Valores de constantes y variables globales. Estas secciones pueden a su vez estar divididas en *datos inicializados* y *no inicializados*
 - *Text*: Instrucciones de programa
 - *Symbol table*: Tabla de símbolos que relacionan *identificadores* (funciones, variables, constantes, etc) con sus *direcciones de memoria*.
 - *Reallocation entries*: Direcciones en el programa de las *referencias externas*, es decir aquellos símbolos definidos en otros módulos (archivos objeto o bibliotecas). Estas direcciones generalmente corresponden a los operandos de instrucciones (`call f`, `load/store d`, ...) que referencian a *símbolos externos* y que luego deberán ser *resueltos por el linker*.

Enlazado (linking)

El *linker* (o *link-editor*) es el encargado de producir los binarios (ejecutables y bibliotecas) en base a la *composición* de varios archivos objeto y bibliotecas.

Esta combinación de archivos binarios consiste de dos pasos:

1. Concatenar cada archivo objeto y *recalcular* las direcciones de las variables, constantes y funciones.
2. Resolver las direcciones de las instrucciones tipo `call address`. Estas instrucciones son *reallocation entries*. Algunas pueden ser *referencias externas* (definidas en otro archivo objeto o biblioteca).

Dados los programas de ejemplo de arriba, el módulo `main.o` contiene dos referencias externas: `hello` y `printf`. Estos símbolos aparecen como *reallocation entries*. Es posible ver con `objdump -d main.o` el *disassembly* de las instrucciones en la sección de texto.

markup

```
Text section:
```

```
_main:
```

```
00000000    ...
```

```
00000018    call 0x18          ; call hello  (UNRESOLVED)
```

```
0000001c    call 0x1c          ; call printf (UNRESOLVED)
```

```
Reallocation Entries:
```

```
address      type      symbol
```

```
0x0000001c    call32    _printf
0x00000018    call32    _hello
```

Las instrucciones `call <address>` están sin resolver: `address` apunta a ellas mismas o 0, dependiendo de la plataforma. Estas instrucciones aparecen en la tabla de *reallocation entries* para que el linker las resuelva.

Se puede ver que en los archivos objeto `main.o` y `hello.o`, las funciones `_main` y `_hello` tienen ambas dirección 0.

El comando `gcc -o myprog main.o hello.o` invoca al *linker* el cual enlaza los archivos objetos (y la biblioteca estándar) y genera el *ejecutable* `myprog`.

Al analizar la sección *text* de `myprog` (con `objdump -d myprog`) se puede observar que las direcciones de las instrucciones `call` se han resuelto.

```
...
_main:
...
00003f50    call _hello      ; call 0x3f68
00003f54    call 0x10003f74 ; stub for _printf
...
...        ret
_hello:
00003f68    ...
...        ret
```

markup

El *linker* combinó ambos módulos, recalculó direcciones (`_hello` tiene una dirección a continuación de la última instrucción de `_main`) y resolvió la dirección del `call` a la dirección de comienzo de `_hello`.

Se debe notar que también se resolvió la dirección de la invocación a `_printf` la cual está definida en la *biblioteca estándar* (de *enlace dinámico*). A continuación se describen los detalles sobre *bibliotecas* y *dynamic linking*.

Bibliotecas

Una biblioteca es una colección de módulos (código y datos) *reusables*.

Una *biblioteca* generalmente contiene un conjunto de archivos objeto (`.o`). Pueden ser de dos tipos:

1. Para *enlazado estático*
2. De *enlazado dinámico (shared)*

Una biblioteca de enlazado estático consiste en un *contenedor* (formato *archiver* en sistemas tipo UNIX) de archivos objetos. Un contenedor incluye un *índice* con los nombres de los archivos objeto que contiene.

En sistemas tipo UNIX se puede generar una biblioteca estática con el comando

```
ar rcs libmylib.a file1.o file2.o ...
```

estos son los archivos objeto que quiero meter en la biblioteca.

markup

El comando `ar` es por *archiver*. Hacer `man ar` para más detalles.

Generalmente una biblioteca `B` se nombra de la forma `libB.a` .

Continuando con el ejemplo anterior, es posible crear `libhello.a` con

```
ar rcs libhello.a hello.o
```

markup

y crear el ejecutable `myprog` con

```
gcc -o myprog -L $PWD -l hello
```

con el `-L` es el path de búsqueda, porque el linker no lo busca en el directorio que estas.

`-l` es una opción del linker para linkear con la biblioteca, en este caso `hello`

markup

El flag `-L $PWD` adiciona el directorio corriente al *path* de búsqueda de `ld` .

Enlazado estático y dinámico

El enlazado estático tiene como ventaja que el ejecutable resultante es auto-contenido, es decir que todas sus dependencias están incluidas en el archivo.

La desventaja es que si todas las aplicaciones tendrían incluidas las bibliotecas de uso común (como la estándar), ese código estaría replicado en cada aplicación, consumiendo mayor espacio en el sistema de archivos y en la memoria cuando estén en ejecución.

Los sistemas operativos modernos soportan *shared libraries* o ** que son enlazadas dinámicamente, es decir en tiempo de ejecución del proceso. Estas bibliotecas también se conocen como *dynamic linking libraries (DLLs).*

La ventaja es que durante la ejecución se carga una única copia en memoria de la biblioteca que será enlazada se *comparte* por todos los procesos que la usen. Esto hace que los archivos ejecutables son más pequeños y no hay código repetido de las bibliotecas de uso común.

Como desventaja es que se produce una pequeña sobrecarga en ejecución ya que se debe realizar el enlazado antes de comenzar la ejecución de cada proceso. Veremos que esta sobrecarga se puede reducir significativamente con *lazy linking*.

Es posible generar la biblioteca `libhello.so` de enlazado dinámico haciendo

```
posicion independi code (fpic) markup  
gcc -c -fpic hello.c  
gcc -shared -o libhello.so hello.o  
shared esta diciendo que vamos a generar una biblioteca de enlace dinamico
```

El primer comando genera un archivo objeto con *position independent code*. Esto es requerido ya que la biblioteca se enlazará con diferentes direcciones de memoria base a cada proceso.

Es posible que una biblioteca tenga sus dos versiones: estática y dinámica. Generalmente el linker prefiere la versión dinámica a menos que se especifique el nombre específico (ej: `hello.a`).

Cuando se realiza *dynamic linking* el linker genera una función para invocar a la *biblioteca del linker (ld.so)* que determina cuáles son las bibliotecas dinámicas requeridas, las carga (bajo demanda) y resuelve los símbolos externos del programa.

Muchos sistemas operativos modernos realizan *lazy linking*, es decir el enlazado a una función se realiza en su primera invocación. Esto reduce la latencia inicial en la ejecución del proceso.

Con *lazy linking* el linker genera por cada *función externa* una pequeña rutina (*stub*) en el ejecutable la cual es responsable de resolver ese símbolo de función en su primera invocación. El linker resuelve estáticamente las invocaciones a estos *stubs*.

Por ejemplo, en una llamada a `printf` se invoca a `printf_stub` . Cada *stub* usa su entrada correspondiente en la *Global Offset Table(GOT)* que contiene las direcciones de las direcciones de las funciones a resolver. Esta tabla tiene las direcciones reales de las funciones que vamos a llamar.

Inicialmente esas entradas apuntan a una instrucción especial del *stub* que invocará a la función de biblioteca del linker para resolver ese símbolo en particular. En el retorno, la entrada en la *GOT* se reemplaza por la dirección de la función resuelta. En las próximas invocaciones al stub, éste *saltará* a la función ya resuelta.

Este proceso se muestra en las siguientes figuras para la función `printf`.

El linker genera la *GOT* y los stubs. El stub correspondiente a `printf` se denota aquí como `printf_stub`.

En el programa, en cada llamada a `printf` se compiló a una llamada a `printf_stub`.

Inicialmente, la `GOT[_printf]` apunta a una instrucción de `printf_stub` que invocará al *linker*.

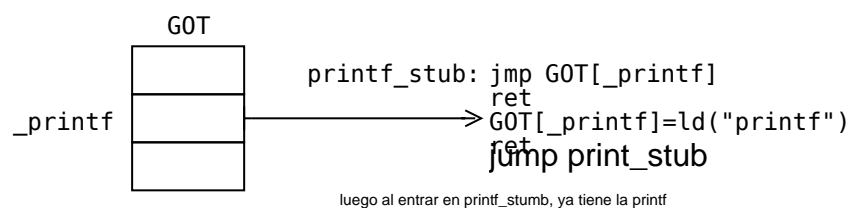


Figura 2: *GOT antes de invocar a `printf_stub`.*

`/lib64/ld-linux-*x86-64.so.2.`
Linker dinamico en kernel de linux

Esta info esta en todos los ejecutables

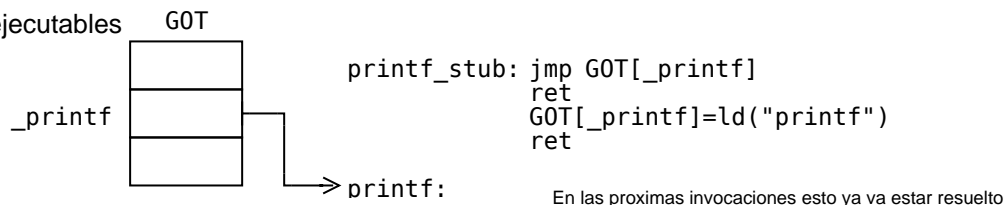


Figura 3: *GOT luego de la primera invocación a `printf_stub`.*

La figura 3 muestra que luego de la primera invocación a `printf_stub` la entrada `GOT[_printf]` apunta directamente a `printf` de la biblioteca estándar. En las siguientes invocaciones a `printf_stub` ésta salta a `printf` (ya resuelta).

La sobrecarga es que en cada invocación se requiere un salto adicional (o indirecto).

Build systems

Un sistema de desarrollo incluye herramientas de automatización de los pasos de producción del software. Para la construcción de un componente de software se debe tener en cuenta sus dependencias y los comandos a aplicar en cada paso del proceso de compilación y enlazado hasta obtener los programas y/o bibliotecas finales.

GNU build system

Este sistema^[1] se basa en la herramienta `make`, la cual se basa en una especificación de los objetos o *targets* a construir y sus dependencias. La especificación se escribe en un archivo `Makefile` y consiste en una secuencia en reglas de la forma:

markup

```
target: dependencies
<tab> build command
```

A modo de ejemplo, el programa `myprog` basado en los archivos fuente del capítulo anterior, puede especificarse en la siguiente regla:

Los makefile podemos verlo como grafos de dependencias

Makefile

```
myprog: main.o hello.o
    gcc -o $@ $^
    $@ se refiere a myprog, $^ a los .o

%.o: %.c
    gcc -c $<
    esto dice basicamente que los .o dependen de los .c
    Por ejemplo: main.c depende de main.o
```

La primera regla determina que para generar el ejecutable `myprog`, el cual depende de los archivos objeto `main.o` y `hello.o`, se construye con el comando `gcc -o $@ $^`. Para más detalles sobre las *variables* usadas ver [automatic variables](#).

El *GNU build system* incluye otras herramientas que permiten automatizar el proceso de compilación y enlazado de paquetes de software en forma independiente de la plataforma. Estas herramientas se conocen como las *autotools* que contiene los siguientes componentes:

- *Autoconf*: Crea archivos de configuración desde la especificación `autoconf.ac`.
- *Automake*: Se basa en las directivas en `Makefile.am` y `configure.ac` y genera el `Makefile.in`.


Con estas herramientas un desarrollador genera un *paquete* `my-package.tgz` (contenedor *tar* comprimido). [Aquí](#) se muestra un ejemplo de cómo crear un paquete simple.

Este paquete se distribuye y un usuario lo puede instalar siguiendo los siguientes pasos:

sh

```
# tar xzvf my-package.tgz
# ./configure && make && sudo make install
```


Referencias

[1]  GNU build system:

https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html

< Anterior

Interfaces (syscalls)

Próximo >

Procesos