

El pipeline de compilación

Escuela Técnica Superior de Ingeniería Informática
Universidad de La Laguna

Abril, 2022

Esquema de la lección

Introducción

El pipeline de compilación

Etapas de la toolchain

Resultados del pipeline de compilación

Introducción a la toolchain GNU

Ejemplo de utilización de la toolchain de compilación

Creando un archivo de objeto reubicable

Explorando la tabla de símbolos

Segundo módulo con funciones

Construyendo el archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

Conclusiones

A partir de aquí

1 Introducción

2 El pipeline de compilación

Etapas de la toolchain

Resultados del pipeline de compilación

3 Introducción a la toolchain GNU

4 Ejemplo de utilización de la toolchain de compilación

Creando un archivo de objeto reubicable

Explorando la tabla de símbolos

Segundo módulo con funciones

Construyendo el archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

5 Conclusiones

6 A partir de aquí

Introducción (I)

- La construcción de un programa ejecutable en el procesador requiere de un conjunto de herramientas que se denomina toolchain de compilación.
- La toolchain de compilación necesita conocer el contexto en el que el futuro programa se ejecutará: arquitectura, sistema operativo, ABI, entre otras cosas.



Application Binary Interface

Introducción (II)

Por ejemplo:

- La toolchain GNU para desarrollar programas sobre sistemas Unix. Esta toolchain no es una, sino que hay muchas dependiendo del **target** del programa, por ejemplo, tenemos la toolchain de GNU para crear programas MIPS32 para Linux bajo el ABI O32, y ordenamiento Little Endian.
- Toolchains para construir programas para Windows. Existe una gran variedad con partes intercambiables. Por ejemplo, Visual Studio IDE permite configurar partes de la toolchain, podemos elegir el compilador de C++: MSVC, CLANG, o el sistema para gestionar el proceso de compilación CMake, MSBuild.
- Toolchain para compilar y cargar programas en el microcontrolador ESP32. Se trata de un conjunto de herramientas específicas para un microcontrolador concreto ESP32.

Etapas de la toolchain

- ➊ Preprocesador. Su objetivo es eliminar características del código que mejoren su legibilidad
- ➋ Generador de código: Su objetivo es convertir instrucciones de alto nivel a código ensamblador
- ➌ Ensamblador: Su objetivo es traducir las instrucciones de lenguaje ensamblador a instrucciones en lenguaje máquina.
- ➍ Linker: Su objetivo es unir bloques de código objeto separados para formar un ejecutable único.

El preprocesador

- Eliminar comentarios reemplazándolos con un espacio en blanco
- Expandir cualquier definición macro
- Eliminar cualquier “line break”, a menos que sea en literales.
- Reemplazar las líneas “include” por el contenido del fichero respectivo
- Recomponer las líneas que el programador ha dividido para mayor claridad
- Tokenizes keywords y construir la tabla de símbolos que se debe utilizar para la generación de código.

Ejemplo de preprocesamiento

```
/* incr.c
Preprocessor example
*/

#include "incr.h"

main() {

    int x = 0; /* declare an integer x */

    /* increment the value 2 and assigns to x */
    x = \
        incr(2);

}
```

Figura: Programa ANSI C, con comentarios, saltos de línea y macros. Extraído de [1]

```
/* incr.h
Defines increment macro
*/

#define incr(n) n + 1
```

Figura: Macro utilizada en el programa anterior. Extraído de [1]

El preprocesador del GNU C

Introducción

El pipeline de compilación

Etapas de la toolchain

Resultados del
pipeline de
compilación

Introducción a la toolchain GNU

Ejemplo de utilización de la toolchain de compilación

Creando un archivo
de objeto reubicable

Explorando la tabla
de símbolos

Segundo módulo con
funciones

Construyendo el
archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

Conclusiones

A partir de aquí

- El compilador de C de la toolchain GNU es GNU C. Como hemos comentado la primera fase en el trabajo de la toolchain es el preprocesamiento y GNU C la realiza llamando automáticamente al preprocesador de C, otro programa que se denomina **cpp**.
- El usuario también puede llamar explícitamente al preprocesador de C que es lo que se muestra a continuación.

Ejemplo de preprocesamiento (2)

```
$ cpp incr.c incr.cpp
```

Figura: Línea de comando para lanza el preprocesador cpp. La salida es el programa tras la fase de preprocesamiento que en este ejemplo se denomina incr.cpp. Extraído de [1]

```
# 1 "incr.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "incr.c"
# 1 "incr.h" 1
# 2 "incr.c" 2
```

```
main() {

    int x = 0;

    x = 2 + 1;

}
```

Optimización (1)

El propósito del paso de optimización es intentar maximizar o minimizar algún atributo del código ejecutable. Por ejemplo:

- Tamaño del código binario (binary image)
- Velocidad de ejecución
- Utilización de memoria
- Consumo de energía

Optimización (2)

- En un ordenador normal es muy posible que lo que se desee sea optimizar la velocidad. No tenemos problemas con el tamaño de memoria ni respecto al código ni a los datos. Tampoco consideramos crítico un posible mayor consumo de energía.
- En los sistemas empujados, por ejemplo un procesador que vaya dentro de un teléfono móvil, el tamaño del código es crítico porque la memoria en chip suele ser pequeña, por lo cual requeriremos que el código binario ocupe poco espacio. El consumo de energía también será crítico porque no queremos que nos descargue la batería.

Optimización (3)

Introducción

El pipeline de compilación

- Etapas de la toolchain
- Resultados del pipeline de compilación
- Introducción a la toolchain GNU
- Ejemplo de utilización de la toolchain de compilación
 - Creando un archivo de objeto reubicable
 - Explorando la tabla de símbolos
 - Segundo módulo con funciones
 - Construyendo el archivo ejecutable
 - La sección de código
 - Tablas de reubicación
 - La sección de datos
 - El ejecutable

Conclusiones

A partir de aquí

Level	Option	Comment
0	-O0	No optimisation. This is the same as omitting the -O option
1	-O1 or -O	Performs optimisation functions that do not require any speed/space trade-off. The size of the binary image should be smaller and its execution speed faster compared to the -O0 level. Furthermore the compilation can be shorter
2	-O2	Performs additional optimisation (over that of -O1) that do not require any speed/space trade-off. Executions speed should be faster without increasing the size of the image. Compilation times could be slower
2.5	-Os	Performs optimisation aimed at reducing the size of the generated code. In some cases this can also improve performance because there is less code to execute
3	-O3	The aim of this level of optimisation is to increase execution speed. It does this at the cost of increasing the binary image size. In some cases the increase in code can have the effect of reducing performance
3	-Ofast	Performs optimisation that disregards standards compliance

Figura: Opciones de optimización GCC. Extraído de [1]

Ejemplo de optimización (1)

Para hacer la optimización del programa efectiva, el compilador tiene que hacer ciertas suposiciones sobre el programa completo.

Si nuestro código no satisface alguna de las suposiciones del compilador, el resultado podría ser erróneo para ese nivel de optimización

El compilador necesita saber que funciones y variables pueden ser accedidas por librería y runtimes fuera de la unidad optimizada en tiempo de linkado.

```
#include <stdio.h>

#define N 100000

int main()
{
    long i, j, k = 0;

    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            k += (i - j);

    printf("k = %ld\n", k); /* k = 0 */
}
```

Figura: Ejemplo de optimización. Contenido de opt.c. Extraído de [1]

```
$ gcc -O0 -o opt0 opt.c
$ gcc -O1 -o opt1 opt.c
```

Figura: Compilación de opt.c para niveles O0 y O1. Extraído de [1]

Ejemplo de optimización (2)

```
$ time ./opt0
k = 0

real    0m31.729s
user    0m31.702s
sys     0m0.000s
```

Figura: Tiempo de ejecución para nivel 0. Extraído de [1]

```
$ time ./opt1
k = 0

real    0m9.080s
user    0m9.065s
sys     0m0.004s
```

Figura: Tiempo de ejecución para nivel O1. Extraído de [1]

Podemos ver que el código optimizado se ejecuta en un tercio del tiempo del código sin optimizar.

Introducción

El pipeline de compilación

Etapas de la toolchain

Resultados del
pipeline de
compilación

Introducción a la toolchain GNU

Ejemplo de utilización de la toolchain de compilación

Creando un archivo
de objeto reubicable

Explorando la tabla
de símbolos

Segundo módulo con
funciones

Construyendo el
archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

Conclusiones

A partir de aquí

- En la fase de ensamblado el compilador va a generar una imagen binaria que representa a los datos y al código del programa.
- En los programas destinados a ser ejecutados por el procesador directamente, el código del programa quedará representado por instrucciones máquina de la arquitectura a la que va destinado.
- La mayoría de los compiladores traduce el lenguaje de alto nivel a un programa en lenguaje ensamblador como paso intermedio. Luego el programa ensamblador es traducido a instrucciones máquina por una herramienta especializada.

Ejemplo de Ensamblado (1)

Figura: Contenido de decl.c. Extraído de [1]

```
int main() {  
    int x = 42;  
    return x;  
}
```

Figura: Compilación de decl.c parando en nivel de ensamblado.
Extraído de [1]

Ejemplo de Ensamblado (2)

```
.file    "decl.c"
.text
.globl  main
.type   main, @function
main:
pushl   %ebp
movl     %esp, %ebp
subl     $16, %esp
movl     $42, -4(%ebp)
movl     -4(%ebp), %eax
leave
ret
.size    main, .-main
```

```
.ident   "GCC: (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3"
.section .note.GNU-stack,"",@progbits
```

Figura: Código ensamblador. Extraído de [1]

Ejemplo de Ensamblado (3)

```
$ as -o decl.o decl.s
```

Figura: Ejecutar ensamblador para traducir instrucciones de ensamblador a instrucciones objeto (.o). Extraído de [1]

```
$ gcc -o decl decl.o
```

Figura: Ejecutar el linker para completar el proceso de compilación. Extraído de [1]

```
$ ./decl
```

```
$ echo $?
```

```
42
```

Figura: Ejecutar el binario y comprobar el código devuelto. Extraído de [1]

Linker o enlazador (1)

- Los programas grandes normalmente se descomponen en módulos por conveniencia. Estos módulos se compilan por separado y luego se **enlazan** en un ejecutable único.
- El linker es la herramienta de la toolchain que realiza la parte final convirtiendo uno o varios módulos compilados en un único ejecutable.
- La ventajas de organizar el programa en módulos son muchas. La principal es la reutilización del código (librerías) pero también acelera los tiempos de compilación pues solo es necesario recompilar el módulo afectado.

El programas pueden compilarse en librerías, para que las funciones y los datos puedan ser reutilizados por otros programas. Hay dos variedades de librerías:

- **Librerías estáticas** (sufijo .a en los sistemas Unix)
Formarán parte del ejecutable gracias a la acción del linker.
- **Librerías dinámicas** (sufijo .so en los sistemas Unix) No
formarán parte del ejecutable, sino que serán cargadas en
memoria por el cargador del sistema operativo cuando un
proceso las invoque en tiempo de ejecución (salvo que ya
estuvieran cargadas).

Archivos OBJ y Archivos ejecutables

- Observamos en el pipeline de compilación que se generan ciertos archivos: el compilador produce archivos de “**objeto reubicable**” y el linker produce archivos “**ejecutables**”. Además tenemos el caso específico de las librerías.
- Los archivos ejecutables y las librerías dinámicas son utilizados por el **cargador** del sistema operativo para crear y modificar el proceso en memoria.
- Los archivos de objeto reubicable permiten modularizar la construcción de los programas, de forma que las librerías y los ejecutables pueden construirse a partir de módulos representados por estos “archivos objeto”.

Archivos ELF

- Los archivos Objeto y los archivos ejecutables deben tener un formato soportado por el sistema operativo. En el caso de los sistemas Unix actualmente el formato más utilizado es el denominado **ELF (Executable and Linker Format)**
- Tipos de archivos ELF:
 - **Módulos reubicables o archivos de objeto reubicable.** Son producidos por el compilador. Utilizados por el linker de la toolchain.
 - **Archivos ejecutables.** Producidos por el linker a partir de archivos de objeto reubicable. Utilizados por el cargador del SO.
 - **Librerías compartidas o dinámicas.** Producidas por el linker y utilizadas por el cargador del SO.

Módulos reubicables

Introducción

El pipeline de compilación

Etapas de la
toolchain

Resultados del
pipeline de
compilación

Introducción a la toolchain GNU

Ejemplo de utilización de la toolchain de compilación

Creando un archivo
de objeto reubicable

Explorando la tabla
de símbolos

Segundo módulo con
funciones

Construyendo el
archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

Conclusiones

A partir de aquí

- Los módulos o archivos reubicables son los popularmente conocidos como archivos `.o` `.obj`, producidos por el compilador.
- Contiene el código (instrucciones máquina) y los datos del programa, así como información sobre la ubicación en memoria de estos contenidos.
- La información no es completa, sino parcial, porque hay que esperar a la fase de composición del ejecutable para conocer la ubicación final.
- Para la ubicación de los elementos del programa se establece una primera descomposición en **secciones**

Secciones ELF

Hay dos tipos de secciones:

- Las que contienen la información directa, como por ejemplo la sección **.text** que incluye el código binario representando cada instrucción del programa o la sección **.data** que representa las variables globales inicializadas.
- Las que contienen metadatos sobre elementos del programa, como por ejemplo la sección **.bss** que representa variables globales que deben inicializarse a cero.

Direcciones de los elementos del programa en los módulos reubicables

- El compilador, a partir de las directivas y el lenguaje compilado, establece a qué sección pertenece cada elemento del programa, y cuál es el ordenamiento de estos elementos dentro de la sección.
- La dirección virtual de los diferentes elementos del programa en los módulos reubicables, se proporciona de forma relativa al comienzo de la sección que le corresponda, ya que tras la actuación del compilador todavía se desconoce la dirección virtual de comienzo de cada sección.

Ejecutables ELF

- La acción del linker al producir un ejecutable pasa por asignar direcciones virtuales de comienzo a las diferentes secciones, y por tanto, los diferentes elementos del programa, datos e instrucciones, ya pueden tener direcciones virtuales asignadas.
- Las secciones son agrupadas en **segmentos ELF**. Cada segmento es una área de memoria contigua compuesto de 0 o más secciones.

La toolchain GNU (I)

Introducción

El pipeline de compilación

Etapas de la
toolchain
Resultados del
pipeline de
compilación

Introducción a la toolchain GNU

Ejemplo de utilización de la toolchain de compilación

Creando un archivo
de objeto reubicable

Explorando la tabla
de símbolos

Segundo módulo con
funciones

Construyendo el
archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

Conclusiones

A partir de aquí

Herramientas básicas que forman la toolchain GNU:

- **Binutils:** Conjunto de programas para operaciones con archivos OBJ, por ejemplo el linker **ld**, o el ensamblador **as**.
- **GCC:** sistema de compilación que incluye compiladores para muchos lenguajes de programación.
- **Depurador:** El sistema **gdb** permite la ejecución por línea de programa o por instrucción máquina de un programa compilado.

La toolchain GNU (II)

Herramientas complementarias que forma la toolchain GNU:

- Automatización de los procesos de compilación de proyectos: **Make**, **Autoconf**, **Automake**, **Libtool**.
- Librería básica C **Glibc**
- Procesador de macros **m4**
- Construcción de parsers **Bison**

Herramientas de GNU Binutils relacionadas con los archivos ELF

- La generación de los archivos se realiza con GNU as (assembler) que es un compilador de lenguaje ensamblador para la toolchain, y GNU ld que es linker de la toolchain GNU.
- La inspección de los archivos ELF se puede realizar con las utilidades **readelf** y **objdump**

Ejemplo: Creando un archivo de objeto reubicable (I)

Módulo p1. Estará compuesto por el archivo p1.s que contiene un programa en lenguaje en ensamblador. Este programa hace uso del símbolo `f_print` que no está definido aquí y por eso se ha marcado con la directiva al compilador `.extern`.

```
1  .extern f_print
2  .data
3  cad1: .ascii "Hello, world!\n"
4  cad1_length: .word . - cad1
5  .text
6  .global __start
7  __start:
8      la $a0,cad1
9      lw $a1,cad1_length
10     jal f_print
11     li $a0,0
12     li $v0,4001 #Exit syscall
13     syscall
```

Ejemplo: Creando un archivo de objeto reubicable (II)

Compilamos el módulo p1, usando un compilador:

```
mipsel-linux-gnu-as -mips2 -o p1.o p1.s
```

Estamos usando el compilador **GNU as** de la toolchain GNU para el target MIPS de 32 bits Little Endian con el ABI O32. El resultado es el archivo objeto reubicable ELF que hemos nombrado como p1.o. Lo comprobamos con la herramienta objdump, que también pertenece a las binutils de GNU.

Ejemplo: símbolos en un archivo de objeto reubicable (I)

objdump es una herramienta de exploración de archivos objeto reubicables general (no solo para archivos ELF). Usamos la existente en la toolchain con la que estamos trabajando.

```
mipsel-linux-gnu-objdump -tf p1.o
```

La opciones `-t` y `-f` indican respectivamente que queremos ver información sobre los símbolos incluidos en el archivo, así como la cabecera general del mismo.

Ejemplo: símbolos en un archivo de objeto reubicable (II)

```
p1.o:      file format elf32-tradlittlemips
architecture: mips:6000, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
```

```
SYMBOL TABLE:
00000000 l      d  .text  00000000 .text
00000000 l      d  .data  00000000 .data
00000000 l      d  .bss   00000000 .bss
00000000 l           .data  00000000 cad1
00000010 l           .data  00000000 cad1_length
00000000 l      d  .reginfo      00000000 .reginfo
00000000 l      d  .MIPS.abiflags 00000000 .MIPS.abiflags
00000000 l      d  .pdr   00000000 .pdr
00000000 l      d  .gnu.attributes      00000000 .gnu.attributes
00000000      *UND*  00000000 f_print
00000000 g      .text  00000000 __start
```

Ejemplo: símbolos en un archivo de objeto reubicable (III)

En la salida mostrada anteriormente:

- La última columna es el símbolo. Reconocemos por ejemplo `.text` y `.data` que son las secciones que utilizamos en el código. También `cad1` y `cad1_length` que son etiquetas en nuestra sección de datos. La etiqueta `__start` apunta al comienzo del código. Finalmente la etiqueta `f_print` es usada en nuestro código pero no está definida en él (por eso se usó la directiva `.extern`).
- La primera columna es el valor de la etiqueta (la dirección que representa). Observen que las etiquetas que representan secciones tienen dirección 0, ya que como hemos dicho no se establece la ubicación definitiva de los elementos del programa. Las etiquetas `cad1`, `cad1_length`, y `__start` tienen una dirección relativa al comienzo de la sección donde se encuentran. Por ejemplo, para `cad1_length` es `0x00000010`

Ejemplo: segundo módulo con funciones (I)

Vamos a completar nuestro programa con otro módulo a modo
de librería estática.

```
14  .data
15  prefix: .ascii "I say ... "
16  prefix_length: .word . - prefix
17
18  .text
19  .global f_print
20  f_print:
21      move $t0,$a0
22      move $t1,$a1
23      # Write syscall
24      li $a0,1 #Standard output
25      la $a1,prefix
26      lw $a2,prefix_length
27      li $v0,4004
28      syscall
29      move $a1,$t0
30      move $a2,$t1
31      li $v0,4004
32      syscall
33      jr $ra
```

Ejemplo: segundo módulo con funciones (II)

- El programa anterior define la función `f_print`. La directiva `.global` hace que el símbolo sea visible para otros módulos a la hora de procesarlo con el linker. Como ven, aquí no se define un símbolo `__start`, ya que se espera que esta librería se enlace con otro módulo que contendrá el programa principal.
- Los argumentos de la función son la dirección de una cadena, y el número de bytes de la misma (`$a0`, `$a1`). Se utilizan la llamada del sistema 4004 (`write`) en linux ABI O32, para escribir. En este caso el destino de escritura es la salida estándar (id 1).

Ejemplo: segundo módulo con funciones (III)

El pipeline de compilación

Introducción

El pipeline de compilación

Etapas de la toolchain

Resultados del pipeline de compilación

Introducción a la toolchain GNU

Ejemplo de utilización de la toolchain de compilación

Creando un archivo de objeto reubicable

Explorando la tabla de símbolos

Segundo módulo con funciones

Construyendo el archivo ejecutable

La sección de código

Tablas de reubicación

La sección de datos

El ejecutable

Conclusiones

A partir de aquí

Compilamos el código fuente p2.s para obtener el archivo objeto reubicable:

```
mipsel-linux-gnu-as -mips2 -o p2.o p2.s
```

Ejemplo: segundo módulo con funciones (IV)

La examinamos con objdump igual que antes para obtener los símbolos:

```
SYMBOL TABLE:
00000000 1      d  .text  00000000 .text
00000000 1      d  .data  00000000 .data
00000000 1      d  .bss   00000000 .bss
00000000 1          .data  00000000 prefix
00000000c 1          .data  00000000 prefix_length
00000000 1      d  .reginfo  00000000 .reginfo
00000000 1      d  .MIPS.abiflags 00000000 .MIPS.abiflags
00000000 1      d  .pdr   00000000 .pdr
00000000 1      d  .gnu.attributes 00000000 .gnu.attributes
00000000 g      .text  00000000 f_print
```

Ejemplo: segundo módulo con funciones (V)

- En este archivo ELF vemos que sí está definido el símbolo `f_print` en el comienzo de la sección `.text`.
- Además en la segunda columna aparece marcado con el flag “g” indicando que el símbolo estará disponible para otros módulos en el proceso de enlazado (**linker**)
- Observen también que la dirección relativa de cada símbolo se calcula independientemente del otro módulo, no puede ser de otra manera. Por ejemplo, el símbolo `__start` tiene la dirección `0x0` respecto a la sección `.text` en el módulo `p1.o`, mientras que a `f_print` le ocurre lo mismo.

Ejemplo: construyendo el archivo ejecutable (I)

Pasamos ahora a aplicar el linker:

```
mipsel-linux-gnu-ld p1.o p2.o -o p
```

- Usamos el linker ld para la toolchain GNU del mismo target.
- El linker tiene como entrada los diferentes archivos de objeto reubicable que formarán el archivo ELF ejecutable, en nuestro caso p1.o y p2.o Se genera el ejecutable ELF con nombre de archivo p.

Ejemplo: construyendo el archivo ejecutable (II)

Ahora mostramos los símbolos del archivo ELF ejecutable con la herramienta `objdump`, igual que en los anteriores ejemplos:

```
p:      file format elf32-tradlittlemips
architecture: mips:6000, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x004000f0
```

SYMBOL TABLE:

```
004000b8 l    d  .MIPS.abiflags 00000000 .MIPS.abiflags
004000d0 l    d  .reginfo   00000000 .reginfo
004000f0 l    d  .text     00000000 .text
00410150 l    d  .data     00000000 .data
00000000 l    d  .gnu.attributes 00000000 .gnu.attributes
00000000 l    df *ABS*   00000000 p1.o
00410150 l      .data 00000000 cad1
00410160 l      .data 00000000 cad1_length
00000000 l    df *ABS*   00000000 p2.o
00410170 l      .data 00000000 prefix
0041017c l      .data 00000000 prefix_length
00000000 l    df *ABS*   00000000
00418170 l      .data 00000000 _gp
00410150 g      .data 00000000 _fdata
004000f0 g      .text 00000000 __start
004000f0 g      .text 00000000 _ftext
00410180 g      .data 00000000 __bss_start
00410180 g      .data 00000000 _edata
00410180 g      .data 00000000 _end
00410180 g      .data 00000000 _fbss
00400110 g      .text 00000000 f_print
```

Ejemplo: construyendo el archivo ejecutable (III)

- Ahora la primera columna no representa la dirección relativa respecto al comienzo de la sección sino **la dirección virtual absoluta** del símbolo en la memoria. Cuando el cargador del SO cargue el archivo ejecutable, los elementos del programa se ubicarán en esas direcciones virtuales.
- El próximo curso verán en detalle qué es una dirección virtual. De momento piensen que cada proceso que se ejecuta en el sistema tiene una “visión” de la memoria y el espacio de direcciones virtuales es el conjunto de direcciones utilizadas desde el punto de vista del proceso para ubicar los elementos del programa.
- Cada instrucción tendrá asignada una dirección virtual a partir de la correspondiente al símbolo `.text`.
- En la cabecera del programa se encuentra la dirección de comienzo, que verán coincide con la del símbolo `_start`.

Ejemplo: la sección de código (I)

Dentro de los archivos ELF (de objeto reubicable, ejecutable, u objeto compartido) también están las instrucciones en binario (código máquina). Podemos utilizar `objdump` para obtener ese binario de la instrucción y la instrucción de la que proviene:

```
mipsel-linux-gnu-objdump -d p
```

```
p:      file format elf32-tradlittlemips
```

Disassembly of section `.text`:

```
004000f0 <__start>:
```

4000f0:	3c040041	lui	a0,0x41
4000f4:	24840150	addiu	a0,a0,336
4000f8:	3c050041	lui	a1,0x41
4000fc:	0c100044	jal	400110 <f_print>
400100:	8ca50160	lw	a1,352(a1)
400104:	24040000	li	a0,0
400108:	24020fa1	li	v0,4001
40010c:	0000000c	syscall	

Ejemplo: la sección de código (II)

Continuación del “desensamblado” del programa:

00400110 <f_print>:

400110:	00804025	move	t0,a0
400114:	00a04825	move	t1,a1
400118:	24040001	li	a0,1
40011c:	3c050041	lui	a1,0x41
400120:	24a50170	addiu	a1,a1,368
400124:	3c060041	lui	a2,0x41
400128:	8cc6017c	lw	a2,380(a2)
40012c:	24020fa4	li	v0,4004
400130:	0000000c	syscall	
400134:	01002825	move	a1,t0
400138:	01203025	move	a2,t1
40013c:	24020fa4	li	v0,4004
400140:	0000000c	syscall	
400144:	03e00008	jr	ra
400148:	00000000	nop	
40014c:	00000000	nop	

Ejemplo: la sección de código (III)

- La primera columna es la dirección virtual absoluta de la instrucción. Observen que todas las direcciones son múltiplos de 4, porque cada instrucción ocupa 4 bytes en MIPS32.
- La segunda columna es una palabra de 32 bits que representa la **instrucción máquina**.
- La tercera columna es la instrucción correspondiente en ensamblador, aunque en el caso de MIPS hay una salvedad: las pseudoinstrucciones que se traducen en una sola instrucción máquina se mantienen en la tercera columna aunque la instrucción máquina sea otra. Por ejemplo:

```
li v0,4004
```

tiene asociada la instrucción máquina 0x24020fa4, que en formato binario es:

```
001001000000000100000111110100100
```

Los 6 bits más significativos (001001) constituyen el opcode que es el que corresponde a la instrucción MIPS **addiu**. Es una instrucción de formato I, con lo que los siguientes 5 bits (00000) son el registro que actúa como operando (registro \$zero), los siguientes 5 bits (00010) son el registro destino (registro 2 o \$v0), y el resto es el valor inmediato en 16 bits que es 0x0FA4 que expresado en decimal es el esperado 4004. Así que realmente la instrucción máquina es:

```
addiu $v0,$zero,4004
```

Ejemplo: Tablas de reubicación (I)

Puesto que como hemos visto el linker ha asignado direcciones a los elementos del programa (variables, funciones), también tendrá que haber modificado en el código las referencias correspondientes a estos elementos. Por ejemplo, veamos el desensamblado del archivo de objeto reubicable p1.o:

```
p1.o:      file format elf32-tradlittlemips
```

Disassembly of section .text:

```
00000000 <__start>:
```

0:	3c040000	lui	a0,0x0
4:	24840000	addiu	a0,a0,0
8:	3c050000	lui	a1,0x0
c:	0c000000	jal	0 <__start>
10:	8ca50010	lw	a1,16(a1)
14:	24040000	li	a0,0
18:	24020fa1	li	v0,4001
1c:	0000000c	syscall	

Ejemplo: tablas de reubicación (II)

- Observen que en la instrucción `jal` la dirección de salto se ha sustituido con un `0`. Esto se debe a que en este punto, esa dirección a la que debe saltar no se ha definido aún.
- Observen también la carga de los registros `a0` y `a1`, todas las direcciones están a `0`. La razón es la misma.

El archivo de objeto reubicable lleva incluido una tabla con el registro de estas situaciones. Podemos acceder a la misma con la herramienta `readelf` que también forma parte de la toolchain de GNU y está en GNU binutils:

```
readelf --relocs p1.o
```


Ejemplo: tablas de reubicación (III)

Relocation section `'.rel.text'` at offset `0x1a4` contains 5 entries:

Offset	Info	Type	Sym.Value	Sym. Name
00000000	00000205	R_MIPS_HI16	00000000	.data
00000004	00000206	R_MIPS_LO16	00000000	.data
00000008	00000205	R_MIPS_HI16	00000000	.data
00000010	00000206	R_MIPS_LO16	00000000	.data
0000000c	00000a04	R_MIPS_26	00000000	f_print

Nos muestra cinco entradas, cuatro afectan a referencias al símbolo `.data` (comienzo de la sección de datos de este módulo) y una afecta al símbolo `f_print`. Si observan la primera columna verán que el offset se corresponde a la dirección de la instrucción afectada. Por ejemplo, la instrucción `jal` está en el offset `0xc` de la sección `.text`, e implica un conjunto de 26 bits a establecer en el código de la instrucción máquina correspondiente a `jal`. Por su parte, `.data` es una dirección de 32 bits que cuando se utiliza debe cargarse en los registros con dos instrucciones de tipo inmediato, primero la parte alta y luego la parte baja, por eso hay dos sustituciones a realizar cada vez que se usa `.data`

Ejemplo: la sección de datos

También podemos observar en los archivos ELF el aspecto de las secciones de datos:

```
mipsel-linux-gnu-objdump -j .data -s p
```

```
p:      file format elf32-tradlittlemips
```

Contents of section .data:

```
410150 48656c6c 6f2c2077 6f726c64 210a0000  Hello, world!...
410160 10000000 00000000 00000000 00000000  .....
410170 49207361 79202e2e 2e200000 0c000000  I say ... .....
```

Como vemos, la sección de datos se ha compuesto a partir de las secciones correspondientes en los módulos.

Ejemplo: el ejecutable (1)

- Es importante entender que estamos observando el archivo ELF del ejecutable, no el programa en ejecución.
- La conclusión a la que debemos llegar es que el archivo ELF resultante del pipeline de compilación contiene las estructuras de datos necesarias para que el cargador pueda crear un **proceso** que cumpla con las expectativas del desarrollador durante la ejecución.

Ejemplo: el ejecutable (2)

Si intento ejecutar el archivo en mi sistema Linux x64 obtengo:

```
-bash: ./p: cannot execute binary file: Exec format error
```

- Se obtiene un error porque el cargador del sistema operativo detecta que el ELF ejecutable no es válido para la arquitectura y ABI con la que está funcionando.
- Una alternativa a llevar el archivo a un sistema MIPS real, sería utilizar un emulador con la arquitectura y ABI correcto. Por ejemplo, qemu-user para MIPS.

```
$ qemu-mipsel ./p
I say ... Hello, world!
$
```

Conclusiones

- La toolchain de compilación es un conjunto de herramientas que permite obtener archivos susceptibles de ser convertidos en procesos en ejecución por parte del cargador en un sistema operativo.
- El proceso básico incluye la creación de módulos con objetos reubicables a partir de los cuales se construye el archivo ejecutable.
- Estos archivos precisan de un formato reconocible por el sistema operativo, por ejemplo ELF.
- La asignación de direcciones virtuales definitiva se realiza al obtener el archivo ejecutable. Los módulos reubicables asignan los elementos del programa a direcciones relativas a las secciones incluidas en los mismos módulos. Estas direcciones serán modificadas finalmente por el linker para construir el ejecutable.
- El proceso de asignación de direcciones a los elementos del programa implica la modificación de referencias a las mismas en el código.

A partir de este punto...

Hay muchísimas más cosas que aprender de la toolchain. Por ejemplo:

- ¿Cómo encajan las librerías de carga dinámica en este esquema?
- La toolchain suele tener otras herramientas adicionales, por ejemplo el depurador, de los cuáles gdb es un buen ejemplo.
- Otro aspecto que no se ha visto aquí relacionada precisamente con el depurador es la inclusión de información de depuración, como las líneas del programa original en el archivo ELF.
- Otras herramientas sumamente importantes para el desarrollador son las herramientas de automatización del proceso de compilación como Make o CMake. El ejemplo presentado aquí apenas tiene un par de archivos y se ha intentado utilizar las opciones mínimas de las herramientas para dar una visión general, pero la configuración del proceso de compilación de un proyecto puede ser bastante compleja y requerir de sistemas de automatización.



Alan Holt and Chi-Yu Huang.

Compiler Toolchains, pages 151–169.

Springer International Publishing, Cham, 2018.



Igor Zhirkov.

Low-Level Programming.

Apress, 2017.