Compilación, carga y depuración de programas

Laboratorio de Sistemas

Gorka Guardiola Múzquiz, Enrique Soriano

GSYC

1 de abril de 2019

Elf

- *Executable and Linkable Format*, `https://refspecs.linuxfoundation.org/`
- Format binario de fichero para ejecución nativa en Linux

```
$ file `which ls`
/bin/ls: ELF 64-bit LSB shared object, x86-64,
  version 1 (SYSV), dynamically linked,
  interpreter /lib64/ld-linux-x86-64.so.2,
  for GNU/Linux 3.2.0,
  BuildID[sha1]=bf40cb84e7815de09fa792a097061886933e56fa,
  stripped
```

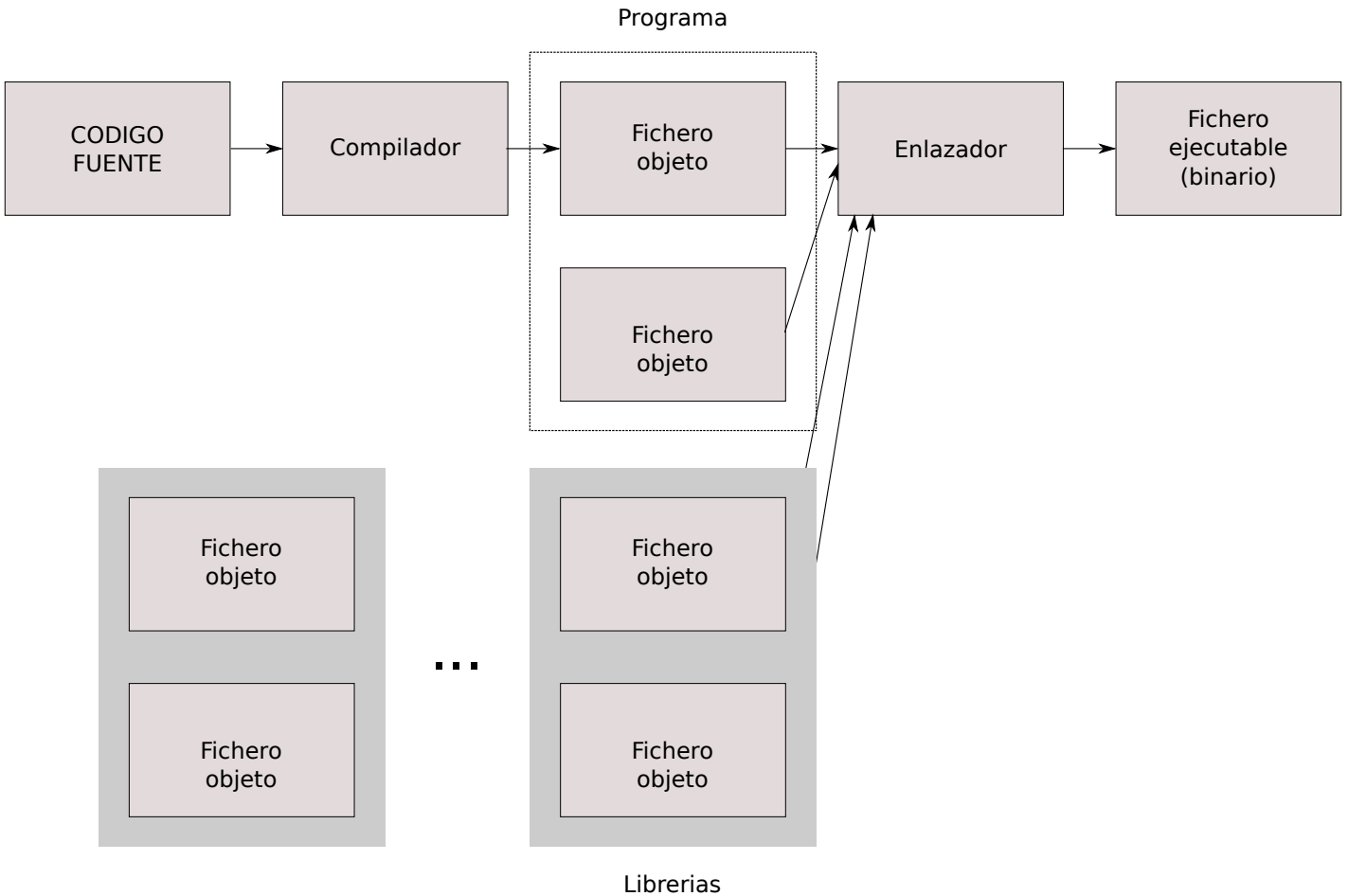
Librerías

- A veces un programa utiliza funciones ya programadas
- Se llaman librerías (o bibliotecas en su traducción estricta de *libraries*)
- Las funciones, variables, etc. de estas librerías pueden ir dentro del binario final (enlazado estático)
- O pueden ir en un fichero binario separado (enlazado dinámico, librerías dinámicas)
- En realidad tiene que ver con cuándo se resuelven los símbolos, que veremos a continuación

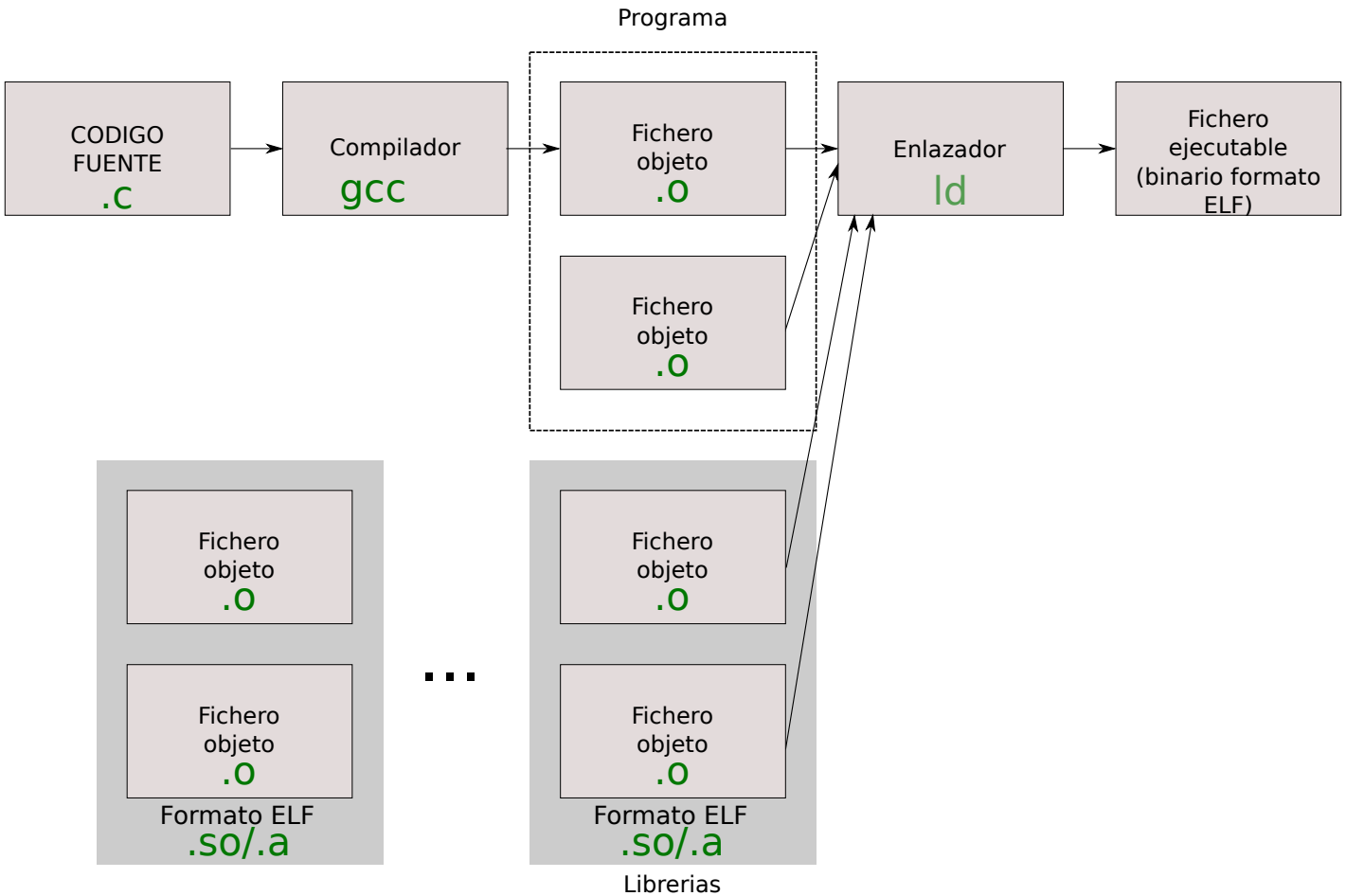
Varios ficheros

- A veces un programa está compuesto de varios ficheros o módulos
- Similar a una librería, pero sin empaquetar

Compilación + enlazado



Compilación + enlazado (ejemplo C)



Código relocalizable

- El código máquina que genera el compilador, puede ser relocalizable o no
- Si es relocalizable, todos los saltos, direcciones de memoria, etc. son relativos
- Se puede poner en varios sitios el programa en memoria y funcionaría

```
JMP   0x344c1cd3
vs
JMP    START+0x34
```

Abstracción de proceso

- El sistema operativo le da al programa que ejecuta memoria que empieza en 0
- Todos los programas creen que tienen toda la memoria (memoria virtual, HW + Sistema operativo)

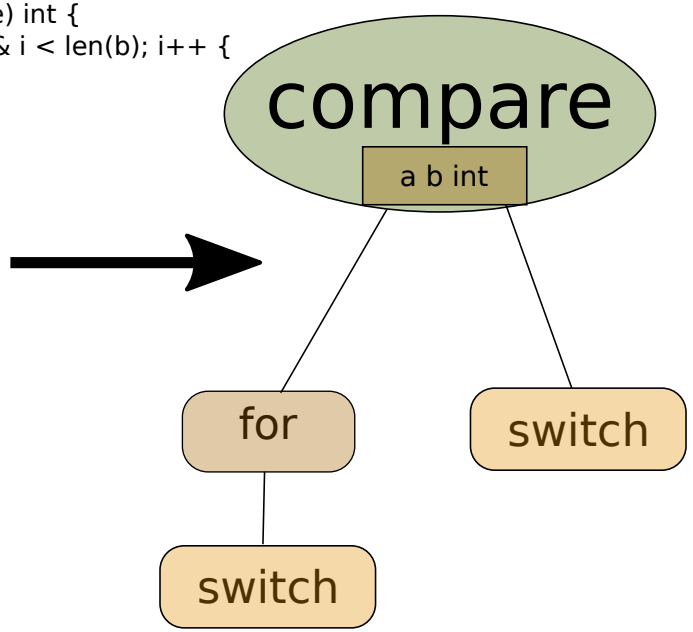
Abstracción de proceso

- Cuando el programa esté ejecutando (proceso), tiene que saltar a direcciones concretas
- Imaginemos que tenemos una llamada a función
- En algún momento esta llamada a función pasa de ser simbólica `CALL print` a concreta `CALL 0x234c2d34`
- Esto se llama resolver el símbolo y pasa con todos los identificadores

Compilador

- El objetivo del compilador es traducir del lenguaje de alto nivel (C, go, C++) a código máquina
- Para ello construye un árbol de sintaxis (AST) y luego lo traduce

```
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) > len(b):
        return 1
    case len(a) < len(b):
        return -1
    }
    return 0
}
```

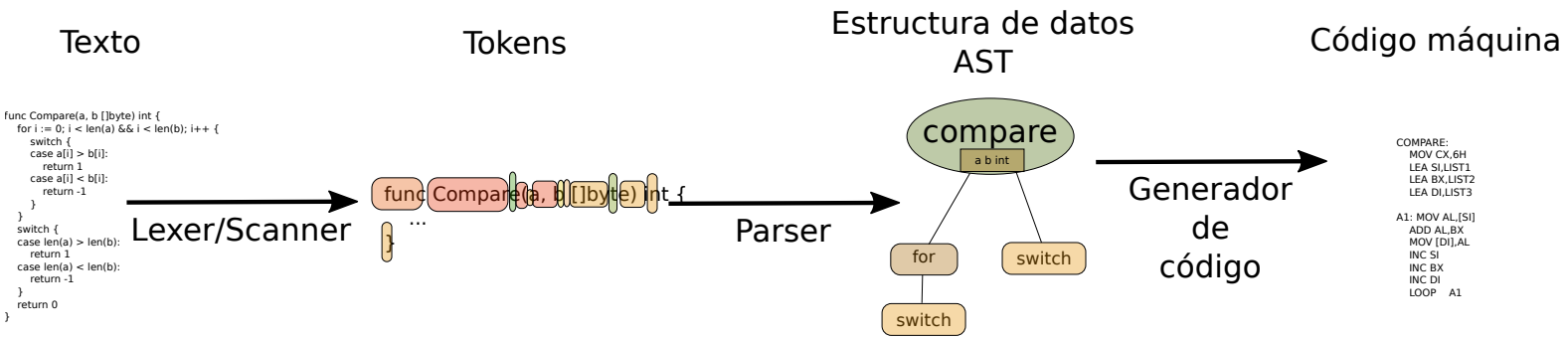



```
COMPARE:
    MOV CX,6H
    LEA SI,LIST1
    LEA BX,LIST2
    LEA DI,LIST3

A1: MOV AL,[SI]
    ADD AL,BX
    MOV [DI],AL
    INC SI
    INC BX
    INC DI
    LOOP    A1
```

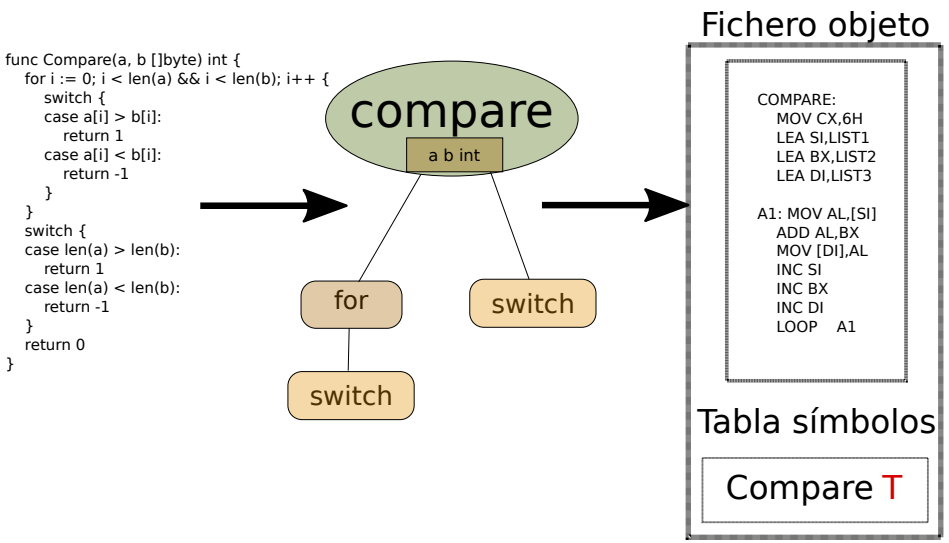
Compilador

- El objetivo del compilador es traducir del lenguaje de alto nivel (C, go, C++) a código máquina
- Para ello construye un árbol de sintaxis (AST) y luego lo traduce



Compilador

- El compilador para todos los símbolos definidos en el fichero, los resuelve
- Para todos los símbolos, crea una tabla
- Los símbolos resueltos para depurar y dar mensajes de error
- Los no resueltos para más adelante en el enlazado (con las librerías) resolverlos



Compilador

- El fichero que genera el compilador es un fichero objeto
- Código relocalizable
- Posiblemente con símbolos sin resolver

Enlazar

- Despues del enlazado, queda un fichero ejecutable (por ejemplo Elf)
- El enlazado puede suceder en
  - Tiempo de compilación: enlazado estático, el fichero ejecutable tiene todos los símbolos resueltos y es autosuficiente
  - Tiempo de ejecución: enlazado dinámico, el fichero ejecutable tiene símbolos que se resuelven al ejecutar, depende de librerías dinámicas
- En ambos casos puede haber tabla de símbolos con símbolos resueltos o no, para depurar.

Comandos

- `nm` imprime la tabla de símbolos
- `strip` quita la tabla de símbolos (hace más pequeño el binario, más difícil la depuración)

Carga

- Para ejecutar un binario, se copia a memoria y se salta al punto de entrada (se pone el registro `PC` en el punto de entrada), se prepara la pila, etc.
- Esto se llama carga, y si el binario requiere estar en un punto de la memoria concreta (código no relocalizable), que es lo normal, hay que copiarlo ahí
- Si tiene enlazado dinámico hay que cargar también las librerías dinámicas y resolver los símbolos (esto lo hacen `ld.so` y `ld-linux.so` en colaboración con el kernel, ver *ld.so(8)*)
- Se puede ir copiando el binario cuando se necesita, esto se llama `carga en demanda`

Librerías dinámicas y objetos

- Las librerías dinámicas y estáticas y los objetos están en un formato especificado por `Elf` en Linux (es parte del mismo estándar)
- La tabla de símbolos están en `Dwarf`, un formato asociado a Elf
- Las librerías dinámicas son normalmente ficheros acabados en `.so` y las estáticas acabadas en `.a`

Dónde están las librerías dinámicas

- Al compilar, pueden tener un path absoluto, si no (simplificado, más detalles en `ld.so(8)`):
  - Se busca en la variable de entorno `LD_LIBRARY_PATH`
  - Luego en la cache `/etc/ld.so.cache`
  - Si no, `/lib`, `/usr/lib` (en algunos `/lib64`, `/usr/lib64`)
- Para saber las dependencias, `readelf -d fichero`
- La configuración de la caché y los enlaces simbólicos de las librerías dinámicas las hace `ldconfig`
- La configuración de `ldconfig` está en `/etc/ld.so.conf` y `/etc/ld.so.conf.d`

Depuración

- Tipos de errores
  - Compilación
  - Ejecución
  - Funcionales
  - Problemas de rendimiento, liberación de recursos

Depuración

- Experimentalidad vs. idea mental incorrecta
- Cuando depuramos estamos intentando ver qué hace el código
- Puramente experimental (como la física)

Depuración: estrategias

- Volcar el estado del programa (print, trazas)
- Antes y después de que pase a ser incorrecto
- Ir dividiendo en 2 el código, buena convergencia con el número de líneas ($O(ln(n))$)

Depuración: estrategias

- Instrumentar las estructuras de datos, que impriman su estado
- Tener niveles de depuración habilitables, (verbose)

Depuración: armas pesadas, el depurador

- Programa que me permite parar el programa e inspeccionarlo (ejemplo `gdb`)
- O inspeccionar un `core` (volcado de memoria de programa muerto)
- Permite desensamblarlo...

Depuración: armas pesadas, el depurador

- Sólo si no me queda más remedio
- Desensamblado
- Paso a paso (mala idea, $O(n)$, consideraciones de tiempo (multithread, red...))
- `Breakpoint`, que pare cuando llege a un punto
- `Watchpoint` que pare cuando se acceda a una variable

Profiling

- Herramienta: `profiler`, medir el uso de recursos, memoria, CPU, búsqueda de puntos calientes (hot spots)
- También se puede hacer a mano midiendo tiempo, contando pasos por un sitio o asignación y liberación de recursos
- Hay de varios tipos
- Traza estadística (cada cierto tiempo mide donde está)
- Inyecta contadores de tiempo en la funciones, contadores de asignación/liberación de recursos en memoria...
- Contadores hardware
- Lo que no se ha medido no se sabe bien (siempre hay sorpresas), medir antes de optimizar

Análisis estático: Linter

- Herramienta: `linter`, para encontrar errores y problemas automáticamente
- Hace *análisis estático* del código
- Patrones incorrectos o fácilmente erróneos de uso
- Peligro de falsos positivos
- `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`

Análisis dinámico: Valgrind

- Ayuda para encontrar errores y problemas automáticamente
- Hace análisis dinámcio del código en ejecución (instrumentado o sin instrumentar)
- Leaks, corrupción de memoria, patrones incorrectos de uso de los recursos
- `http://valgrind.org`