

# 深入淺出 Hello World

理解 Linux 上運作 Hello World 的種種機制

**Jim Huang ( 黃敬群 / "jserv" )**

Website: <http://jserv.sayya.org/>

Blog: <http://blog.linux.org.tw/jserv/>

**Open Razzmatazz Laboratory**

Nov 25, 2006

老師說…

---

一分耕耘  
一分收穫



# 基本想法



- ◆ 紿你魚吃，也教你釣魚
- ◆ 以「實驗」觀點去理解Linux
- ◆ 處處留心皆學問、落花水面皆文章
- ◆ Geek/Hacker與一般的  
user/programmer的分野並不大  
– 只是專注的範疇與態度有異

# 注意

- ◆ 本議程針對x86硬體架構，至於ARM與MIPS架構，請另行聯絡以作安排
- ◆ 簡報採用創意公用授權條款(Creative Commons License: **Attribution-ShareAlike**)  
發行
- ◆ 議程所用之軟體，依據個別授權方式發行
- ◆ 系統平台
  - Ubuntu Edgy (development branch)
  - Linux kernel 2.6.17-5
  - gcc 4.1.2 (pre-release)
  - glibc 2.4

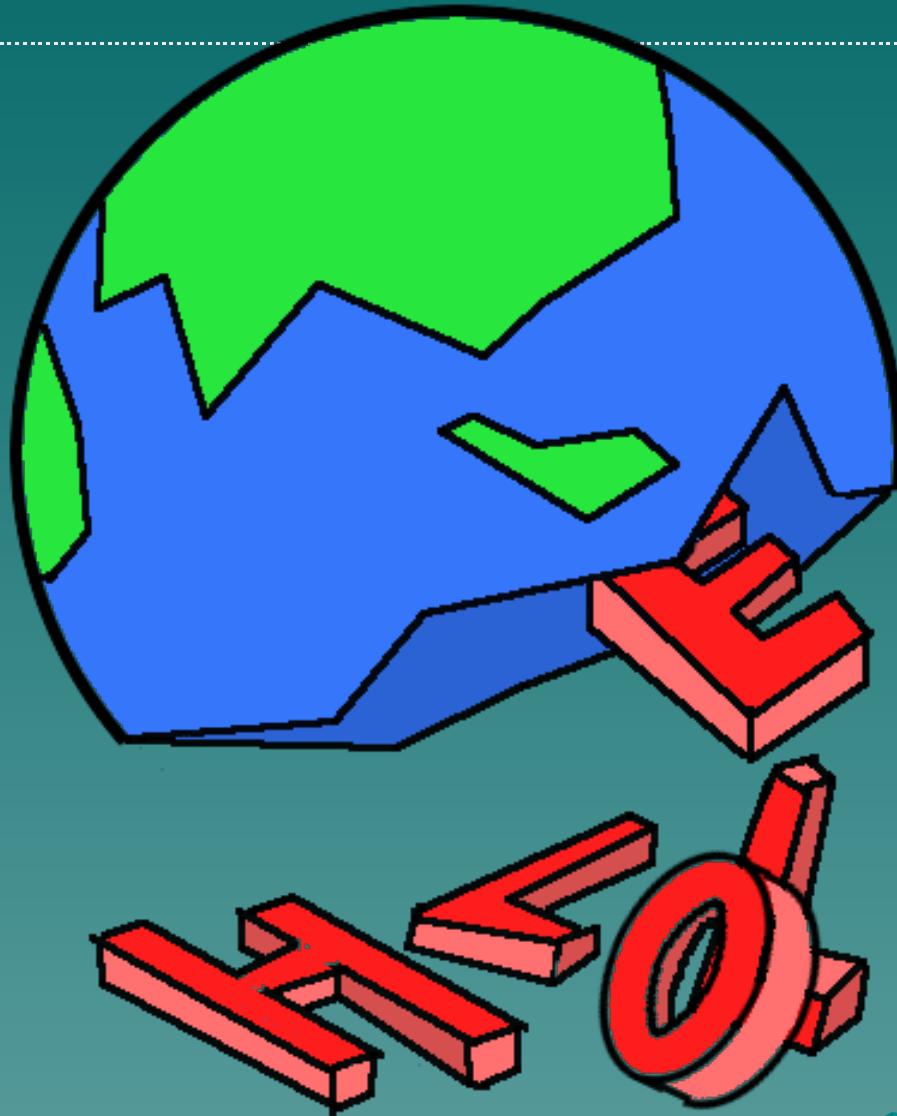


# Agenda

---

- ◆ “Hello World”人人會寫，可是又如何運作？
- ◆ 我們的平台與工具
- ◆ 奠定基礎概念
- ◆ 邁入新紀元：Orz Programming 2.0

# “Hello World” 如何運作？



# 誰不會寫 Hello World ？

---

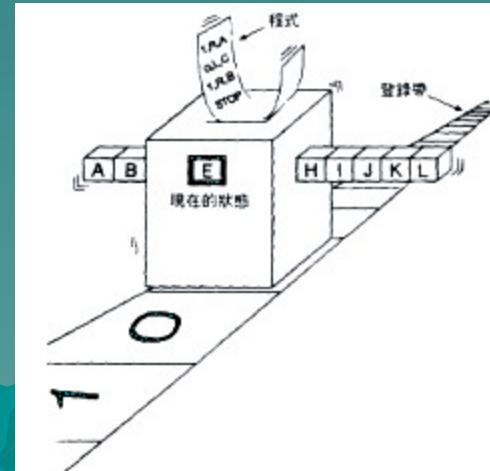
```
#include <stdio.h>

int main (int argc, char *argv[])
{
    printf ("Hello World!\n");
    return 0;
}
```

# “Hello World” 的理論基礎 (1)

## ◆ Turing Machine

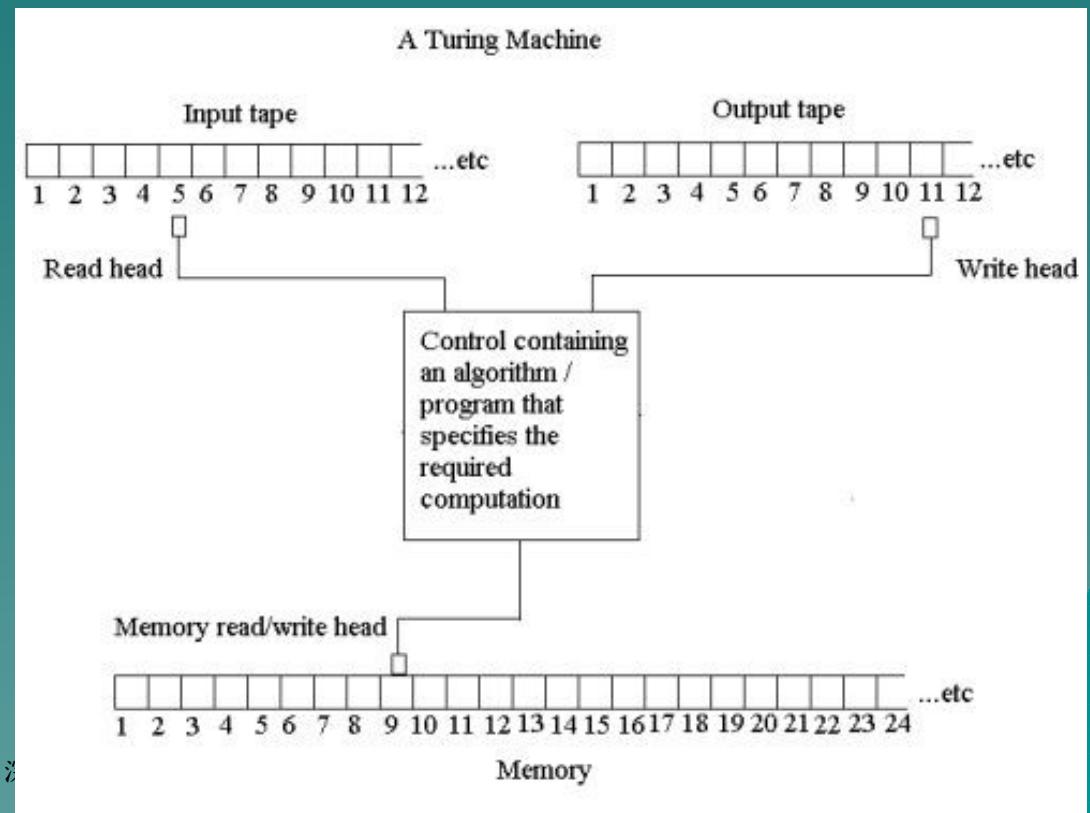
- 英國數學家Alan Turing於1936年提出的一種抽象計算模型
- 用機器來模擬人們用紙筆進行數學運算的過程：
  - ◆ 在紙上寫上或擦除某個符號
  - ◆ 把注意力從紙的一個位置移動到另一個位置
- 每階段要決定下一步的動作
  - ◆ 紙上某個位置的符號
  - ◆ 思維的狀態



# “Hello World” 的理論基礎 (2)

## ◆ Turing Machine

- 每階段要決定下一步的動作
  - ◆ 紙上某個位置的符號
  - ◆ 思維的狀態



# “Hello World” as a Turing Machine

State	Read	Write	Step	Next Step
1	Empty	H	>	2
2	Empty	e	>	3
3	Empty	I	>	4
4	Empty	I	>	5
5	Empty	o	>	6
6	Empty	blank	>	7
7	Empty	W	>	8
8	Empty	o	>	9
9	Empty	r	>	10
10	Empty	I	>	11
11	Empty	d	>	12
12	Empty	!	>	STOP

# 思考…

- ◆ 理論上，可透過任何程式語言甚至硬體機制來實現”Hello World”
- ◆ 數論中，有些猜測敘述簡單易懂，但卻難於證明
  - 費瑪最後定理與哥德巴赫猜想
  - 費瑪最後定理(1631年)  
對於  $n=3, 4, 5, \dots$ ，方程式
$$X^n + y^n = z^n$$
沒有正整數解
- ◆ 《禮記 · 大學》
  - 「古之欲明明德於天下者，先治其國。欲治其國者，先齊其家，欲齊其家者，先修其身。欲修其身者，先正其心。欲正其心者，先誠其意。欲誠其意者，先致其知。致知在格物。」

“Cubem autem in duos cubos, aut quadratoquadratum in duos quadrato-quadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos eiusdem nominis fas est dividere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.”

(Nagell 1951, p. 252)



# 用各種語言寫 Hello World(1)

- ◆ 433 Examples in 132 (or 162\*) programming languages
  - <http://www.ntecs.de/old-hp/uu9r/lang/html/lang.en.html>
- ◆ 補充

```
/* Inline assembly */
#include <stdio.h>

char message[] = "Hello, world!\n";

int main(void)
{
    long _res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (_res)
        : "a" ((long) 4),
        "b" ((long) 1),
        "c" ((long) message),
        "d" ((long) sizeof(message)));
    return 0;
}
```

```
/* Inline BASIC */
#include "ubasic.h"
static const char program[] =
"10 GOSUB 100\n\
20 FOR i = 1 TO 10\n\
30   PRINT \"Hello World!\"\n\
40 NEXT i\n\
50 PRINT \"End\"\n\
60 END\n\
100 RETURN\n";

int main(void)
{
    ubasic_init(program);
    do {
        ubasic_run();
    } while (!ubasic_finished());
    return 0;
}
```

# 用各種語言寫 Hello World(2)

```
\u0070\u0075\u0062\u006c\u0069\u0063\u000a\u0063\u006c\u0061  
\u0073\u0073\u0020\u0055\u0067\u006c\u0079\u000a\u007b\u0070  
\u0075\u0062\u006c\u0069\u0063\u000a\u0020\u0020\u0020  
\u0073\u0074\u0061\u0074\u0069\u0063\u000a\u0076\u006f\u0069  
\u0064\u0020\u006d\u0061\u0069\u006e\u0028\u000a\u0053\u0074  
\u0072\u0069\u006e\u0067\u005b\u005d\u000a\u0020\u0020\u0020  
\u0020\u0061\u0072\u0067\u0073\u0029\u007b\u000a\u0053\u0079  
\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074\u000a\u002e  
\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u000a\u0022  
\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u0022\u002b\u000a  
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u007d\u007d
```



```
public class Ugly {  
    public static void main(String[] args) {  
        System.out.println("Hello Wor1d");  
    }  
}
```

```
public  
class Ugly  
{public  
    static  
void main(  
String[]  
args){  
System.out  
.println(  
"Hello W"+  
"or1d");}}
```

等等，這就是今天的主題？！



**Orz** programming 1.0



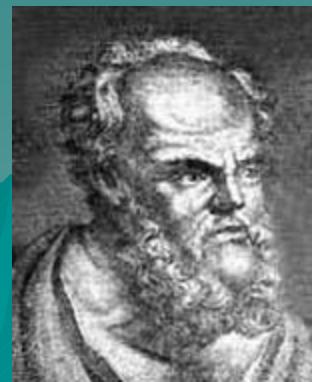
編 程 認 可



蘇格拉底說…

---

我唯一所知的  
就是我一無所知



# 我們的「平台」與「工具」





# 對抗無知的武器



## ◆ GNU Toolchain

- gcc (GNU Compiler Collection)
- binutils

```
$ apt-cache search binutils
```

**binutils** - The GNU assembler, linker and binary utilities

**binutils-dev** - The GNU binary utilities (BFD development f

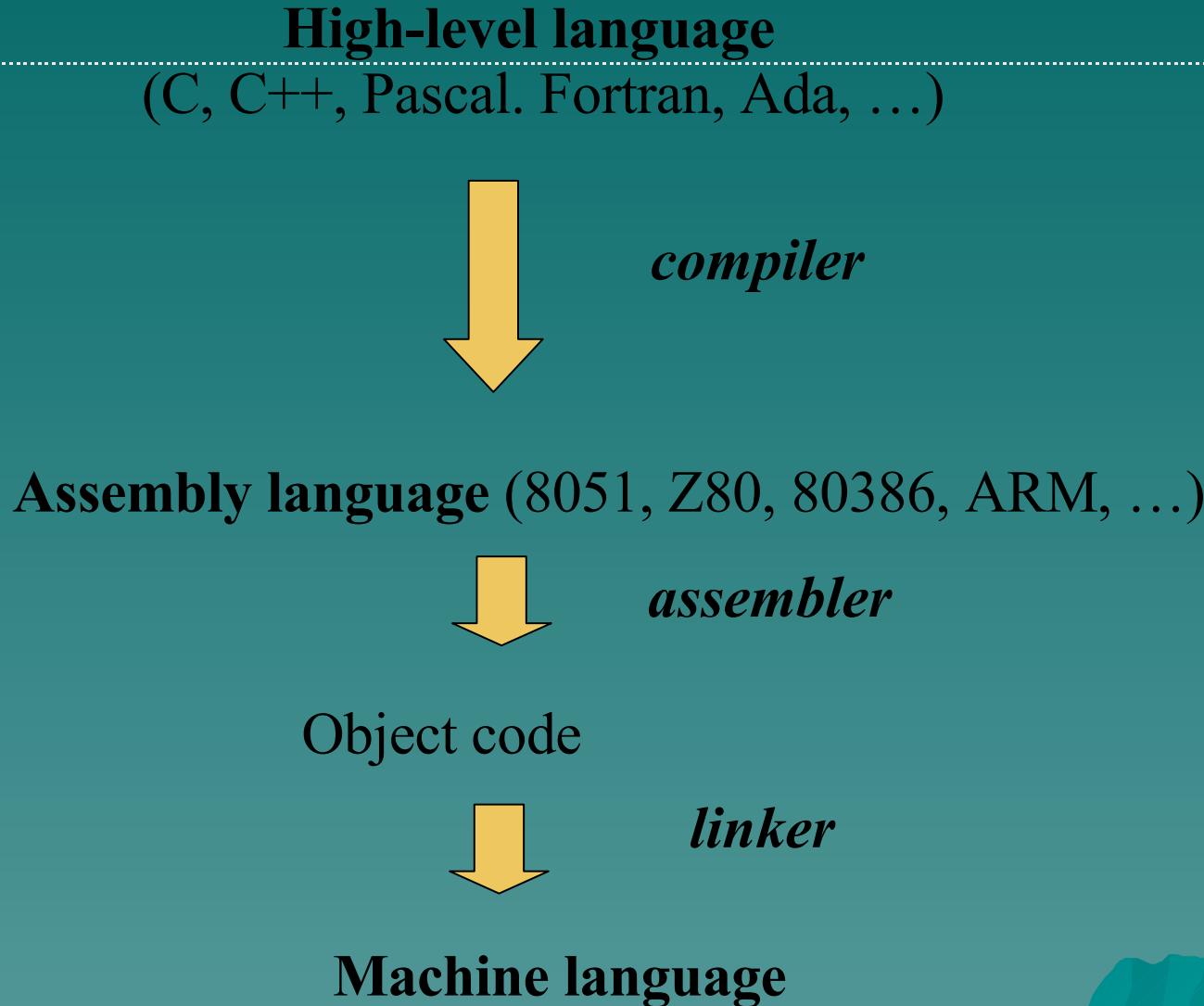
- glibc runtime & utilities

## ◆ ELF editor

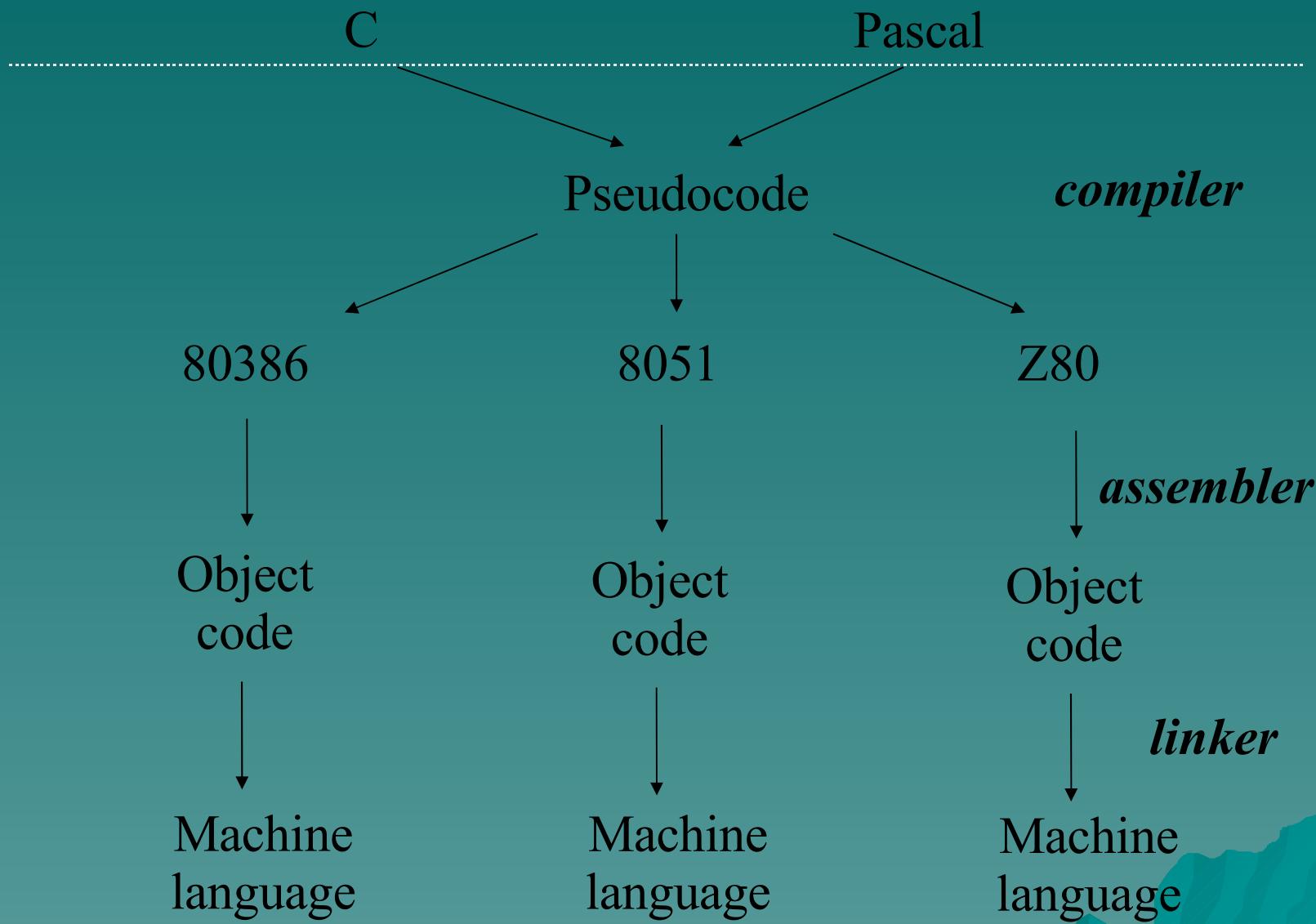
## ◆ ggcov

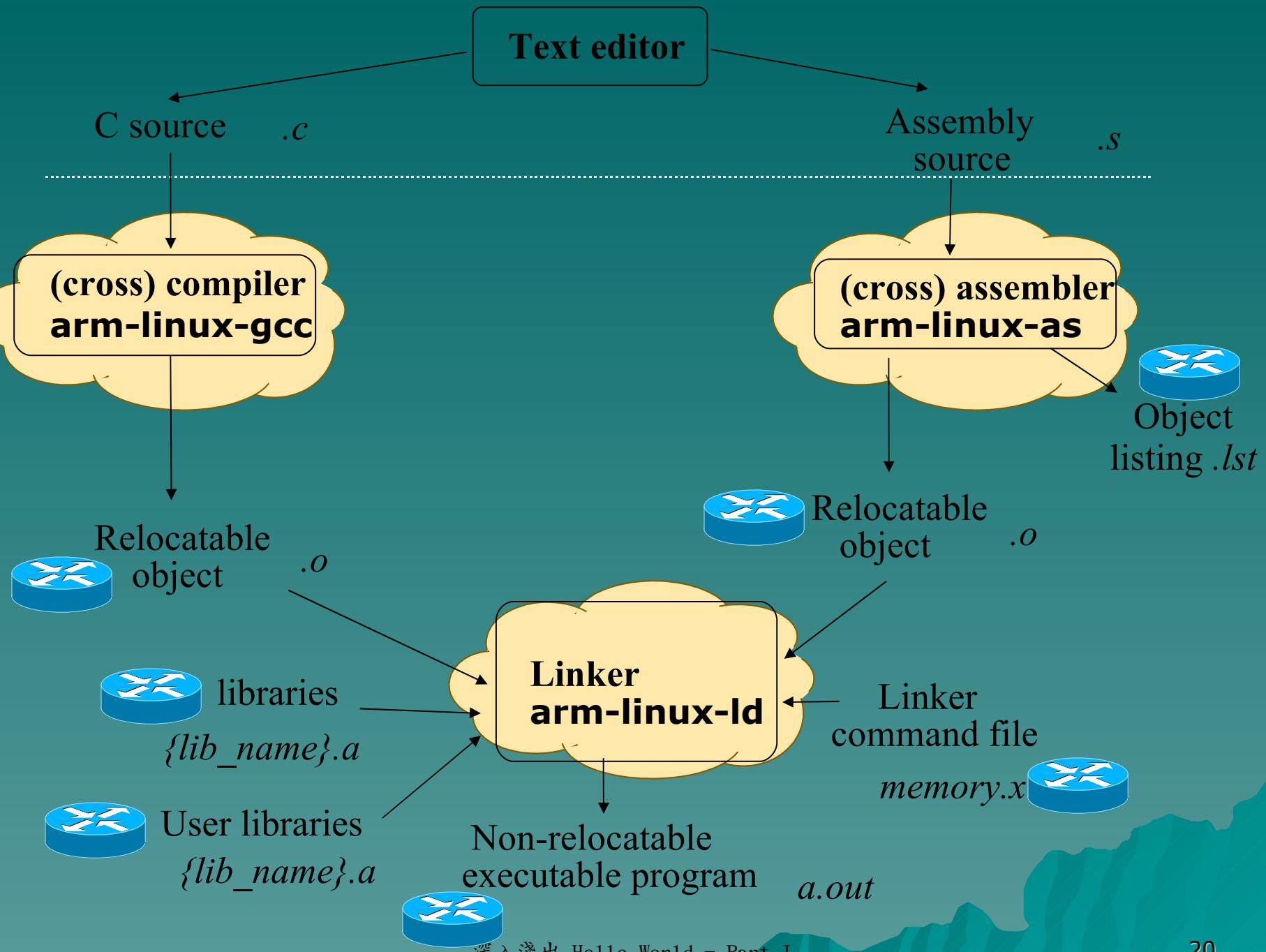


# Level of Software(1)



# Level of Software(2)







Non-relocatable  
executable image

*a.out*

Downloadable  
file converter :  
**arm-linux-objcopy**



Downloadable  
file

# binutils 的常用工具

- ◆ ar
- ◆ strings
- ◆ strip
- ◆ nm
- ◆ size
- ◆ readelf
- ◆ objdump
- ◆ 所有工具的源頭，可顯示 object file 中所有資訊
  - 對 **.text section** 的 dis-assembley

- ◆ 建立 static library
  - Insert
  - Delete
  - List
  - Extract

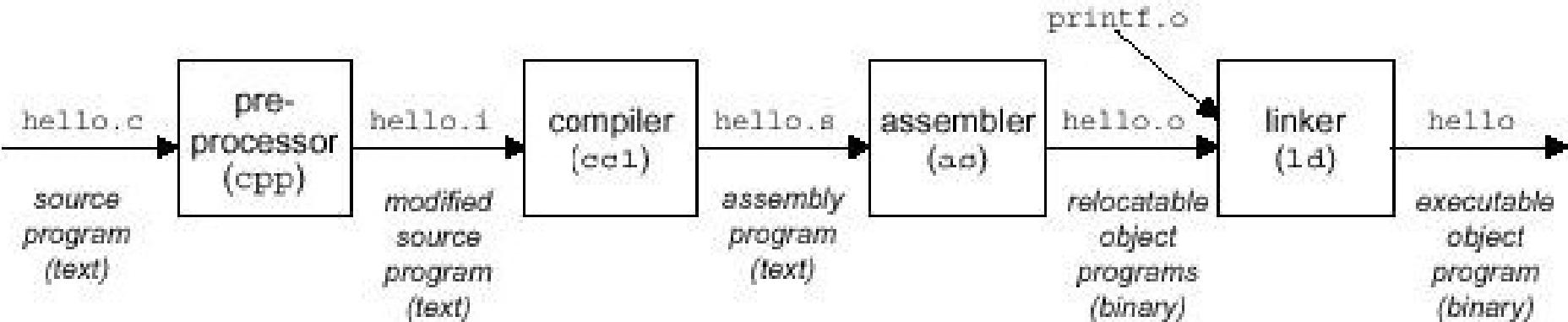
- ◆ 列出 object code 中可見字元與字串

- ◆ 自 object file 中移除 symbol table information

- ◆ 列出 object file 中定義的 symbol

- ◆ 印列自 object file 的完整結構，包含 ELF header 中編碼的資訊

# “gcc -o hello hello.c”的過程

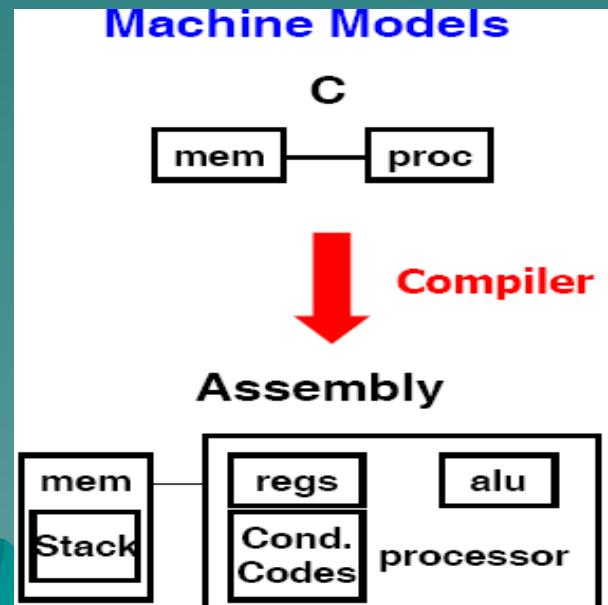


## ◆ 四大階段

- Pre-processing
- Compilation
- Assembly
- Linking

## ◆ GCC 選項 (在…階段結束)

-E  
-C  
-S  
ld



# 開始觀察 (1)

---

```
$ gcc -g -c hello.c
```

```
$ file hello
```

```
hello: ELF 32-bit LSB executable, Intel 80386,  
version 1 (SYSV), for GNU/Linux 2.6.0, dynamically  
linked (uses shared libs), for GNU/Linux 2.6.0, not  
stripped
```

```
$ ldd hello
```

```
linux-gate.so.1 => (0xffffe000)
```

```
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e62000)
```

```
/lib/ld-linux.so.2 (0xb7fad000)
```

# 開始觀察 (2)

```
$ objdump -x hello  
hello:      file format elf32-i386  
hello  
architecture: i386, flags 0x00000112:  
EXEC_P, HAS_SYMS, D_PAGED  
start address 0x08048870
```

Program Header:

<b>PHDR</b>	off	0x00000034	vaddr	0x08048034	paddr	0x08048034	align	2***2
	filesz	0x000000e0	memsz	0x000000e0	flags	r-x		
<b>INTERP</b>	off	0x00000114	vaddr	0x08048114	paddr	0x08048114	align	2***0
	filesz	0x00000013	memsz	0x00000013	flags	r--		
<b>LOAD</b>	off	0x00000000	vaddr	0x08048000	paddr	0x08048000	align	2***12
	filesz	0x00002120	memsz	0x00002120	flags	r-x		
<b>LOAD</b>	off	0x00002120	vaddr	0x0804b120	paddr	0x0804b120	align	2***12
	filesz	0x000001f4	memsz	0x00001280	flags	rw-		
<b>DYNAMIC</b>	off	0x0000213c	vaddr	0x0804b13c	paddr	0x0804b13c	align	2***2
	filesz	0x000000c8	memsz	0x000000c8	flags	rw-		
<b>NOTE</b>	off	0x00000128	vaddr	0x08048128	paddr	0x08048128	align	2***2
	filesz	0x00000020	memsz	0x00000020	flags	r--		

...

# 開始觀察 (3)

```
$ objdump -d hello  
hello:   file format elf32-i386
```

Disassembly of section .init:

```
08048690 <_init>:  
8048690: 55          push  %ebp  
8048691: 89 e5        mov    %esp,%ebp  
8048693: 83 ec 08      sub    $0x8,%esp  
8048696: e8 f9 01 00 00  call   8048894 <call_gmon_start>  
804869b: e8 4b 02 00 00  call   80488eb <frame_dummy>  
80486a0: e8 1b 18 00 00  call   8049ec0 <__do_global_ctors_aux>  
80486a5: c9            leave  
80486a6: c3            ret
```

Disassembly of section .plt:

```
080486a8 <mkdir@plt-0x10>:  
80486a8: ff 35 14 b2 04 08  pushl 0x804b214  
80486ae: ff 25 18 b2 04 08  jmp    *0x804b218  
80486b4: 00 00          add    %al,(%eax)
```

...

# 開始觀察 (4)

```
$ cat hello.c
int main() { printf("Hello World."); return 0; }
$ gcc -static -o hello hello.c
$ strace ./hello
execve("./hello", ["/./hello"], /* 26 vars */) = 0
uname({sys="Linux", node="venux", ...}) = 0
brk(0)                                = 0x80b7000
brk(0x80b7c90)                          = 0x80b7c90
set_thread_area({entry_number:-1 -> 6, base_addr:0x80b7830, limit:1048575,
    seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
    useable:1}) = 0
brk(0x80d8c90)                          = 0x80d8c90
brk(0x80d9000)                          = 0x80d9000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 9), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7f10000
write(1, "Hello World.", 12Hello World.)      = 12
exit_group(0)                            = ?
Process 14211 detached
```

# 開始觀察 (5)

```
$ cat hello.c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf ("Hello World! via %ox\n", printf);
    return 0;
}
```

- ◆ “printf” 是 Standard C Library 的 function
- ◆ Function-symbol 在 Executable 與 Runtime 有何關聯？

# 開始觀察 (6)

```
$ make  
gcc -g -c hello.c  
gcc -g -o hello hello.o  
$ ./run.sh  
Launch Hello World..  
Hello World! via 8048290
```

- ◆ ./hello
- ◆ readelf -a hello | grep printf

```
08049548 00000207 R_386_JUMP_SLOT 08048290 printf  
2: 08048290 57 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0 (2)  
73: 08048290 57 FUNC GLOBAL DEFAULT UND printf@@GLIBC_2.0
```

- ◆ 心中的疑惑…
  - “printf” 真如我們想像一般簡單？
  - Address 哪來的？
  - Executable 與 Memory Image 的緊密關聯
  - Object file linking 的機制

# 開始觀察 (7)

```
$ gcc -g -S hello.c
```

```
$ cat hello.s
```

```
...
```

```
.section .rodata
```

```
.LC0:
```

```
    .string "Hello World! via %x\n"
```

```
.text
```

```
.globl main
```

```
.type main, @function
```

```
main:
```

```
    leal 4(%esp), %ecx
```

```
    andl $-16, %esp
```

```
    pushl -4(%ecx)
```

```
    pushl %ebp
```

```
    movl %esp, %ebp
```

```
    pushl %ecx
```

```
    subl $20, %esp
```

```
    movl $printf, 4(%esp)
```

```
    movl $.LC0, (%esp)
```

```
    call printf
```

```
    movl $0, %eax
```

```
    addl $20, %esp
```

```
    popl %ecx
```

```
    popl %ebp
```

```
    leal -4(%ecx), %esp
```

```
    ret
```

```
int main(int argc, char **argv)  
{
```

```
    printf ("Hello World! via %x\n", printf);  
    return 0;
```

```
}
```

# 開始觀察 (8)

```
$ objdump -D --disassemble-zeroes hello
```

Disassembly of section .text:

...

08048350 <main>:

8048350:	8d 4c 24 04	
8048354:	83 e4 f0	
8048357:	ff 71 fc	
804835a:	55	
804835b:	89 e5	x86:little-endian
804835d:	51	
804835e:	83 ec 14	
8048361:	c7 44 24 04 90 82 04	
8048368:	08	
8048369:	c7 04 24 3c 84 04 08	
8048370:	e8 1b ff ff ff	
8048375:	b8 00 00 00 00	
804837a:	83 c4 14	
804837d:	59	
804837e:	5d	
804837f:	8d 61 fc	
8048382:	c3	
8048383:	90	

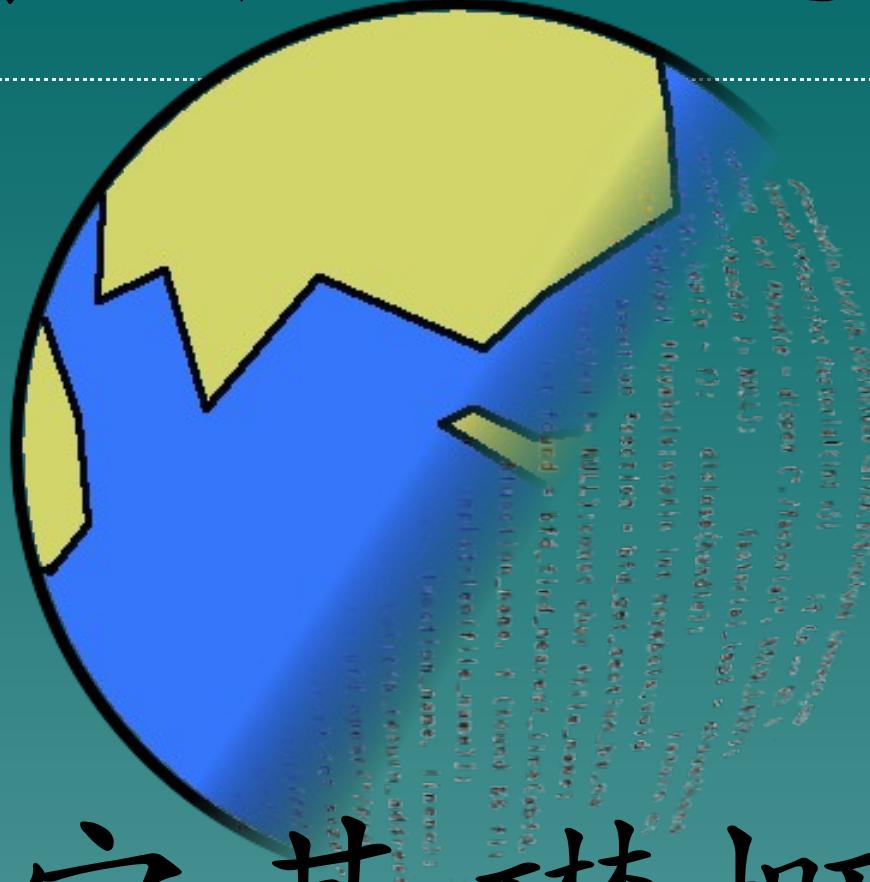
```
$ file hello
```

hello: **ELF 32-bit LSB executable, Intel  
80386**, version 1 (SYSV), for GNU/Linux  
2.6.0, dynamically linked (uses shared  
libs), for GNU/Linux 2.6.0, not stripped

1ea	0x4(%esp),%ecx
and	\$0xffffffff0,%esp
pushl	0xfffffff(%ecx)
push	%ebp
mov	%esp,%ebp
push	%ecx
sub	\$0x14,%esp
movl	\$0x8048290,0x4(%esp)
movl	\$0x804843c,(%esp)
call	8048290 <printf@plt>
mov	\$0x0,%eax
add	\$0x14,%esp
pop	%ecx
pop	%ebp
lea	0xfffffff(%%
ret	
nop	

@plt ?  
**PLT** (Procedure  
Linkage Table)

# 爲了解答上述疑惑…



# 奠定基礎概念

# bss / data / code

```
int a;           // bss
int k = 3;        // bss
int foo(void)    // bss
{
    return (k);
}

int b = 12;       // data
int bar (void)   // data
{
    a = 0;
    return (a + b);
}
```

bss

data

code

a = 0	8192
k = 3 b = 12	4096
... ret leave movl _k,%eax	0

# Loader: Image File → Memory Image

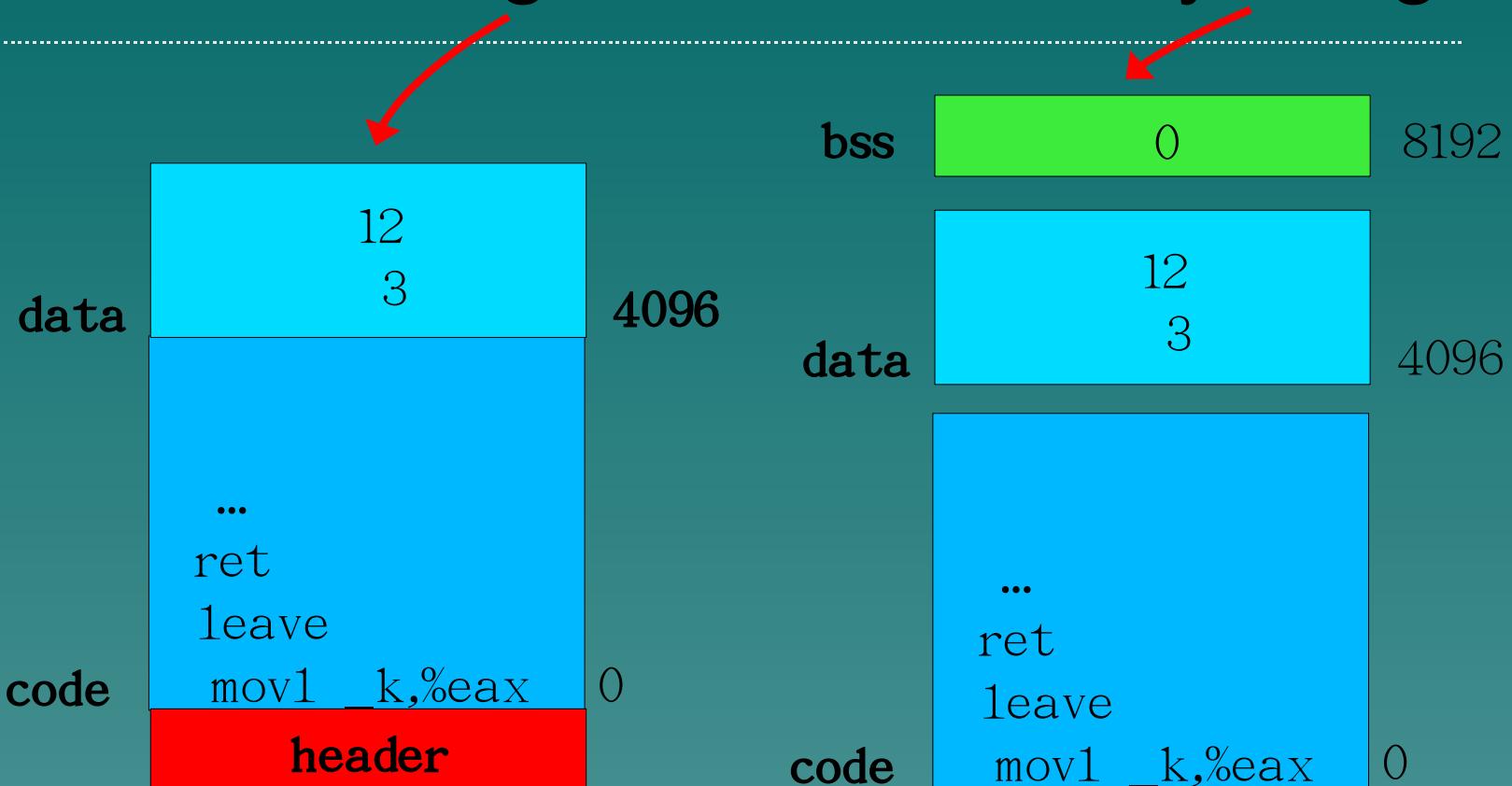
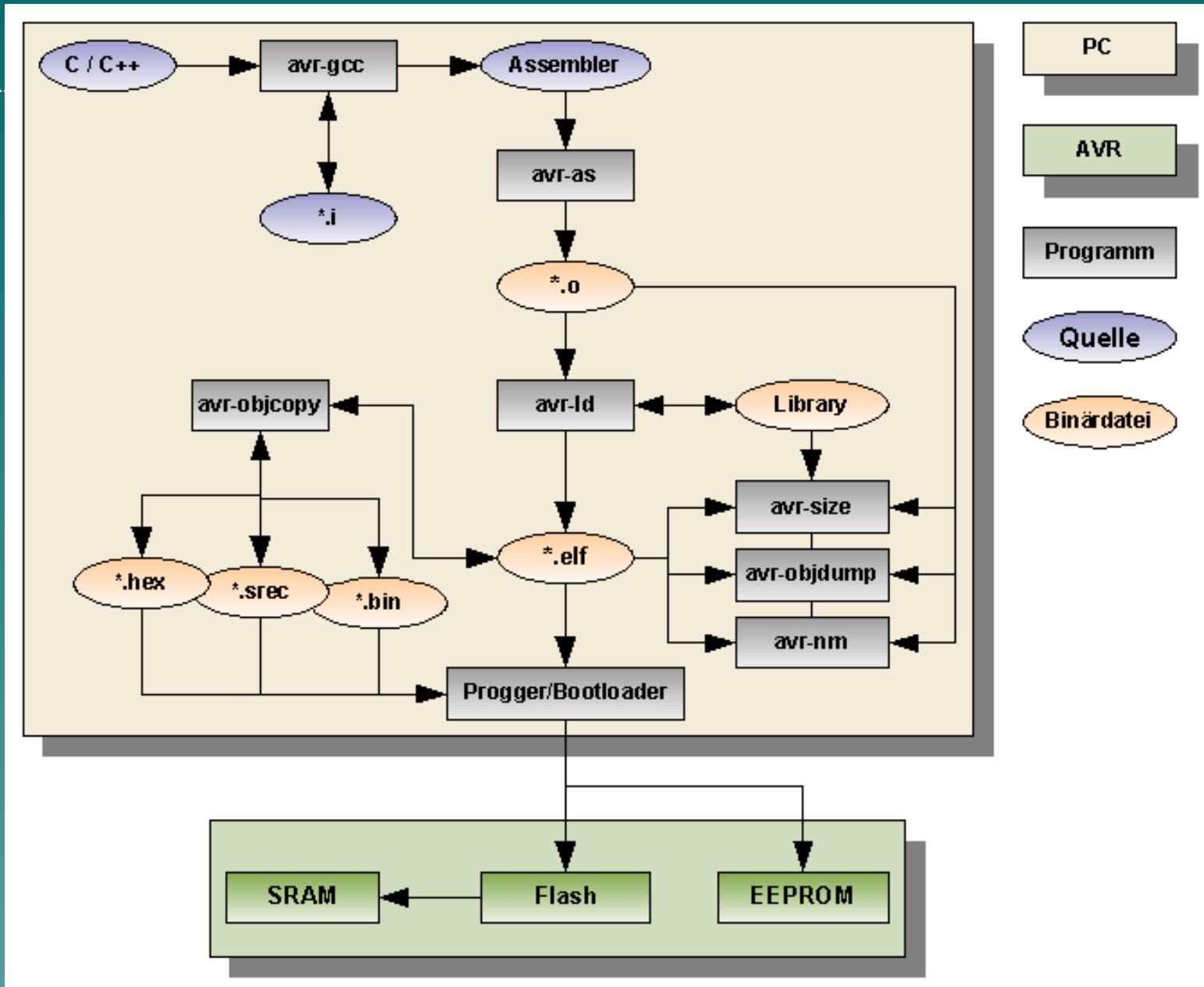
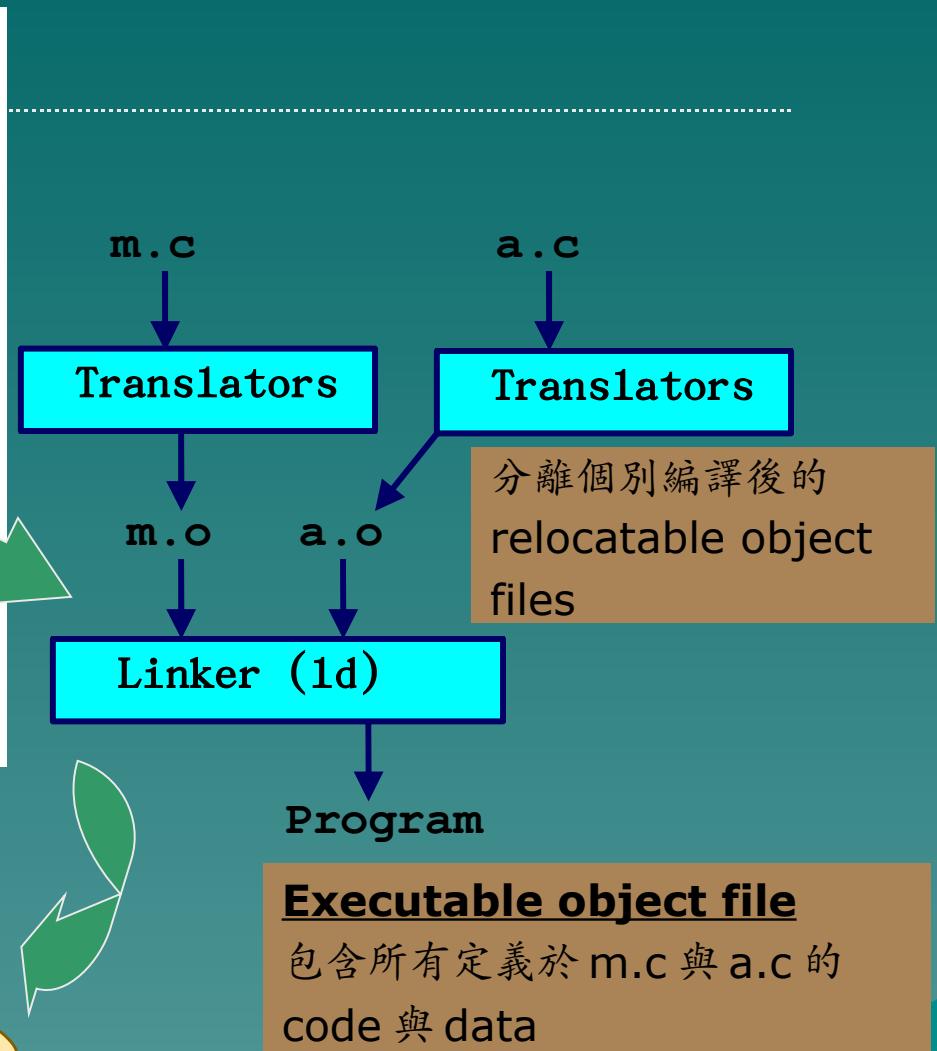
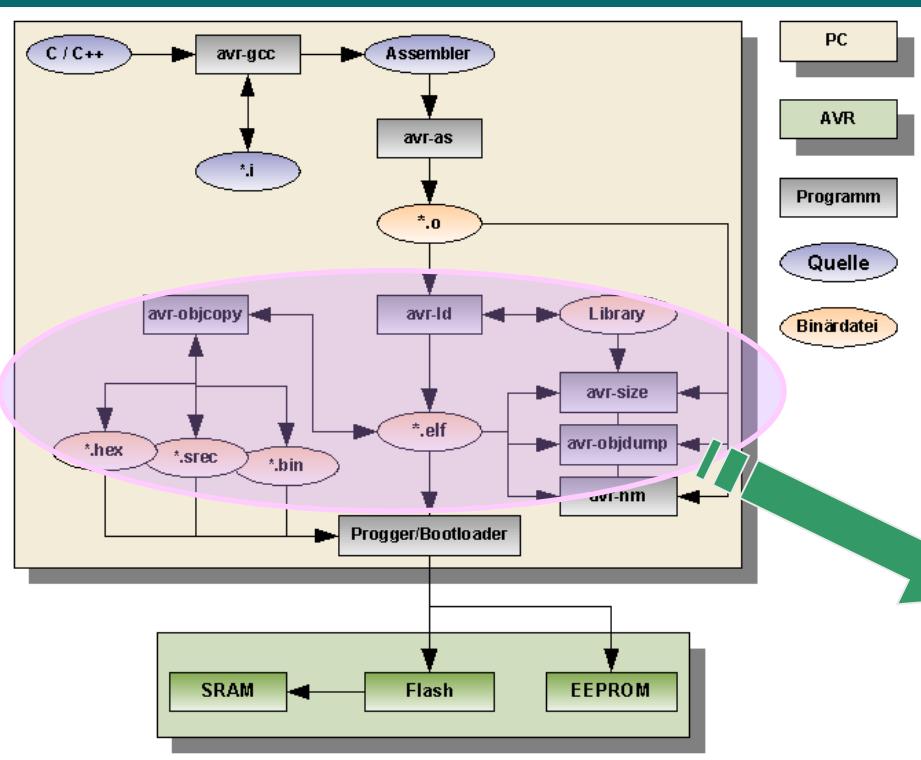


Image file 有個 header，對 Loader 來說具有特別的意義  
Memory image 多了 bss 區段

# GNU Toolchain 運作流程

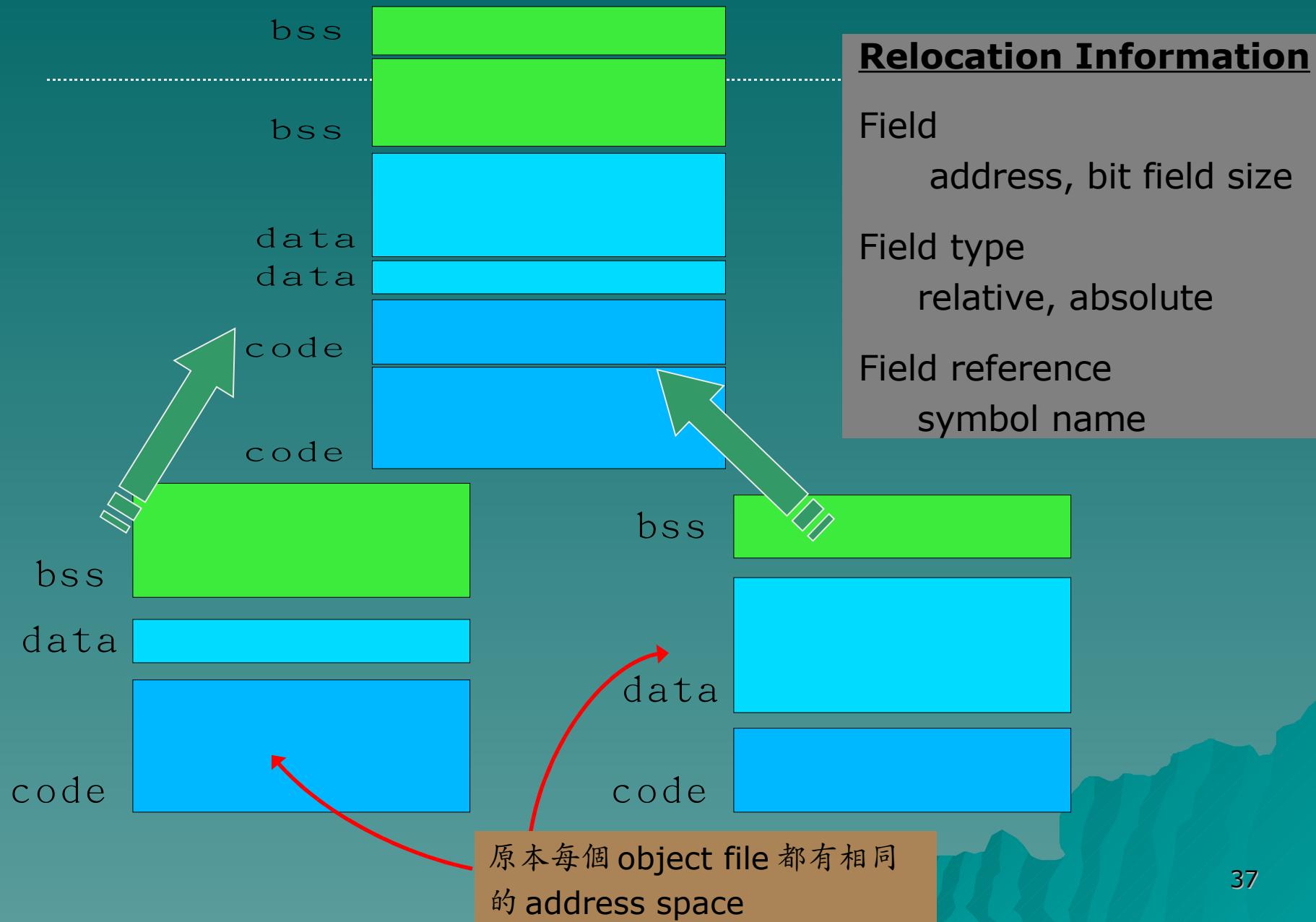


# GCC Linker - ld(1)



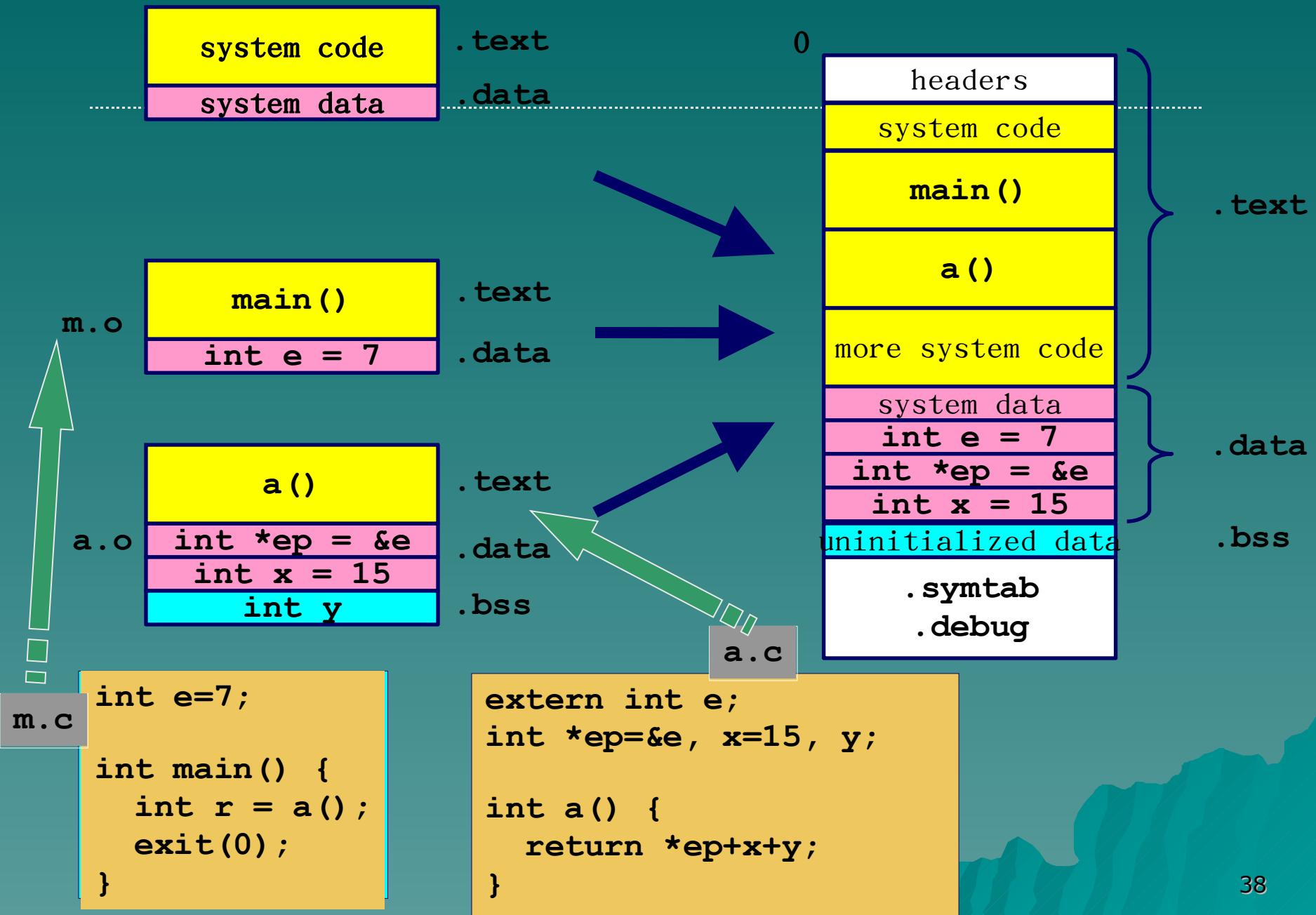
- Merge object files
- Resolve external reference
- Relocate symbols

# GCC Linker - Id<sub>(2)</sub>

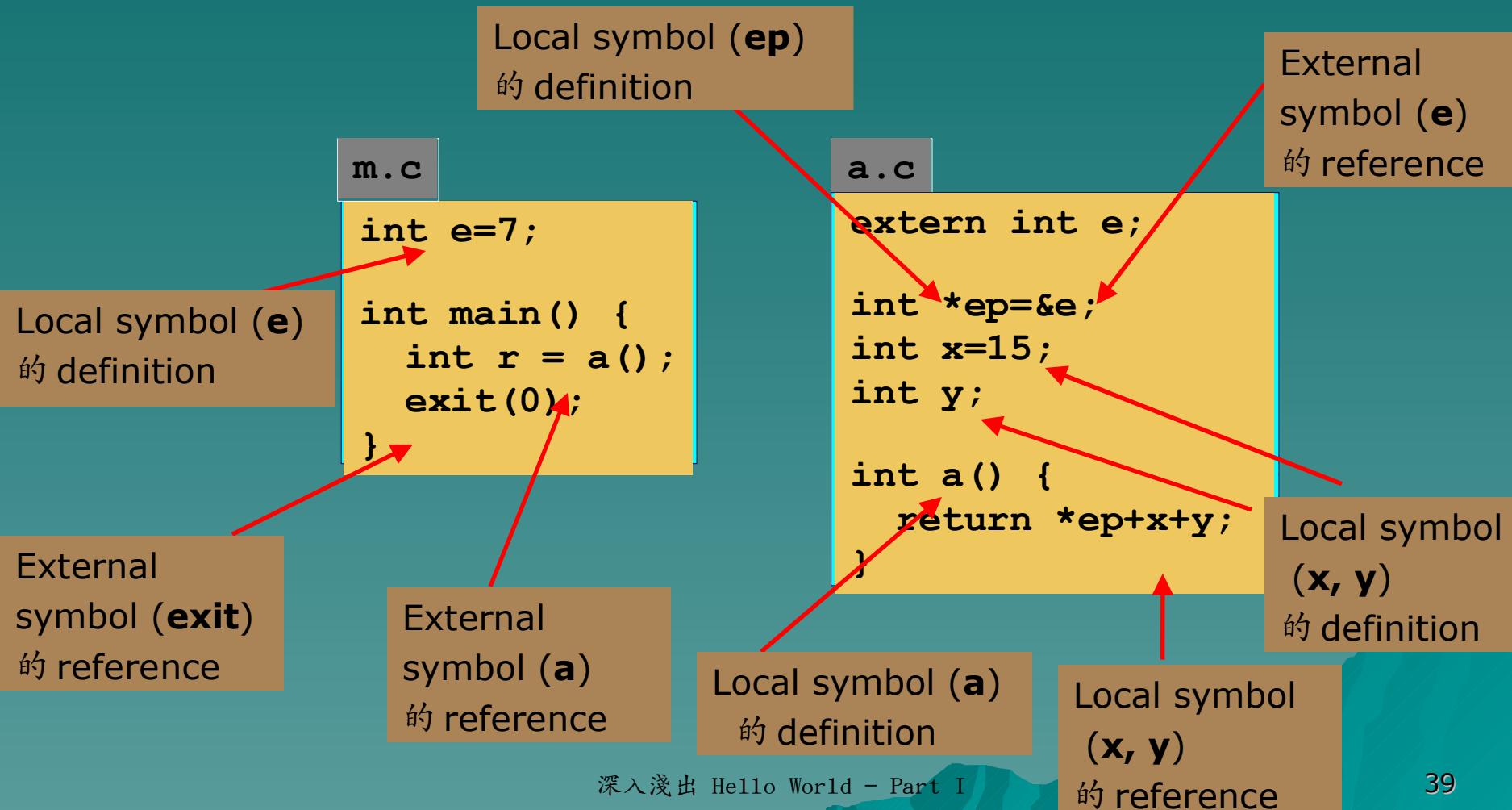


## Relocatable Object Files

## Executable Object File



- ◆ 每個 symbol 都賦予一個特定值，一般來說就是 memory address
- ◆ Code → symbol definitions / reference
- ◆ Reference → local / external



# GCC Linker - Id<sub>(3)</sub>

m.c

```
int e=7;

int main() {
    int r = a();
    exit(0);
}
```

Relocation Info

Disassembly of section .text:

```
00000000 <main>: 00000000 <main>:
 0: 55                      pushl %ebp
 1: 89 e5                   movl %esp,%ebp
 3: e8 fc ff ff ff         call 4 <main+0x4>
 4: R_386_PC32   a          pushl $0x0
                           call b <main+0xb>
 5: R_386_PC32   exit       nop
```

Disassembly of section .data:

```
00000000 <e>:
 0: 07 00 00 00
```

# GCC Linker - ld(4)

a.c

```
extern int e;  
  
int *ep=&e;  
int x=15;  
int y;  
  
int a() {  
    return *ep+x+y;  
}
```

Disassembly of section .text:

00000000 <a>:

0:	55	pushl %ebp
1:	8b 15 00 00 00	movl 0x0,%edx
6:	00	
7:	a1 00 00 00 00	3: R_386_32 ep movl 0x0,%eax
c:	89 e5	
e:	03 02	8: R_386_32 x movl %esp,%ebp
10:	89 ec	addl (%edx),%eax
12:	03 05 00 00 00	movl %ebp,%esp
17:	00	addl 0x0,%eax
18:	5d	14: R_386_32 y popl %ebp
19:	c3	ret

Relocation Info

Disassembly of section .data:

```
00000000 <ep>:  
0: 00 00 00 00  
      0: R_386_32 e  
00000004 <x>:  
4: 0f 00 00 00
```

```
08048530 <main>:  
    8048530: 55          pushl  %ebp  
    8048531: 89 e5        movl   %esp,%ebp  
    8048533: e8 08 00 00 00 call   8048540 <a>  
    8048538: 6a 00        pushl  $0x0  
    804853a: e8 35 ff ff ff call   8048474 <_init+0x94>  
    804853f: 90          nop
```

```
int e=7;
```

```
int main()  
{  
    int r = a();  
    exit(0);  
}
```

```
08048540 <a>:  
    8048540: 55          pushl  %ebp  
    8048541: 8b 15 1c a0 04 movl   0x804a01c,%edx  
    8048546: 08          addl   (%edx),%eax  
    8048547: a1 20 a0 04 08 movl   0x804a020,%eax  
    804854c: 89 e5        movl   %esp,%ebp  
    804854e: 03 02        addl   (%ebp),%eax  
    8048550: 89 ec        movl   %ebp,%esp  
    8048552: 03 05 d0 a3 04 addl   0x804a3d0,%eax  
    8048557: 08          popl   %ebp  
                           ret
```

Executable After Relocation and External Reference Resolution

Disassembly of section .data:

```
0804a018 <e>:  
 804a018: 07 00 00 00
```

```
0804a01c <ep>:  
 804a01c: 18 a0 04 08
```

```
0804a020 <x>:  
 804a020: 0f 00 00 00
```

```
extern int e;
```

```
int *ep=&e;  
int x=15;  
int y;
```

```
int a() {  
    return *ep+x+y;  
}
```

# GCC Linker - ld(5)

- ◆ Linker script
  - 描述哪些sections會出現於最終程式中
  - Compiler/Linker會有預設的linker script，讓一般的應用程式能正常完成build process
- ◆ GCC移植於多種硬體平台，並沒有定義memory model，相反地，將這些工作交付給Linker script

```
$ dpkg -L binutils | grep ldscripts
/usr/lib/ldscripts
/usr/lib/ldscripts/elf_i386.x
/usr/lib/ldscripts/elf_i386.xbn
/usr/lib/ldscripts/elf_i386.xc
/usr/lib/ldscripts/elf_i386.xd
...
```



```
$ ld --verbose
GNU ld version 2.17 Debian GNU/Linux
Supported emulations:
  elf_i386
  i386linux
  elf_x86_64
using internal linker script:
/* Script for -z combreloc: combine and sort reloc
   sections */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
```

# GCC Linker - ld(6)

## ◆ 重要的幾個sections :

**.text** code and constants

**.data**

initialized/un-initialized data

## ◆ 其他 :

**.init / .fini** to/exit entry

**.ctor / .dtor** C/C++ constructor/destructor

**.rodata** ROM variables

**.common** shared overlayed data sections

**.eeprom** EEPROMable sections

**.install** startup code

**.vector** interrupt vector table

**.debug** debugging information

**.comment** documenting comments

# GCC Linker - ld<sub>(7)</sub>

- *gcc linker script* ( 簡化自 /usr/lib/ldscripts/elf\_i386.x )

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-i386","elf32-i386","elf32-i386")
OUTPUT_ARCH(i386)
...
SECTIONS
{
...
.text      :
{
    *(.text .stub .text.* .gnu.linkonce.t.*)
    KEEP (*(.text.*personality*))
}
}

.data      :
{
    *(.data .data.* .gnu.linkonce.d.*)
    KEEP (*(.gnu.linkonce.d.*personality*))
    SORT(CONSTRUCTORS)
}

.bss      :
{
    *(.dynbss)
    *(.bss .bss.* .gnu.linkonce.b.*)
    *(COMMON)
}
```

**.text**  
code and constants

Stored in  
memory block data

**.data**  
Initialized variables

**.bss**  
Un-initialized variables

# GCC Linker - ld<sub>(8)</sub>

- \$ objdump /usr/lib/libc.a -xd > libc.txt

```
printf.o:    file format elf32-i386  
rw-r--r-- 0/0 868 Jun 24 03:09 2006 printf.o  
architecture: i386, flags 0x00000011:  
HAS_RELOC, HAS_SYMS  
start address 0x00000000
```

(x) = --all-headers

顯示所有 **header** 資訊，包含  
**symbol table** 與 **relocation entries**

Sections:

Idx	Name	Size	VMA	LMA	File off	Align
0	.text	<b>00000026</b>	00000000	00000000	<b>00000034</b>	2**2
1	.data	00000000	00000000	00000000	0000005c	2**2
2	.bss	00000000	00000000	00000000	0000005c	2**2
3	.comment	00000040	00000000	00000000		
					CONTENTS, READONLY	
...						

.text → 0x26 = 38 bytes

SYMBOL TABLE:  
00000000 I d .text 00000000 .text  
00000000 I d .data 00000000 .data  
00000000 I d .bss 00000000 .bss  
00000000 I d .comment 00000000 .comment  
00000000 I d .note.GNU-stack 00000000  
.note.GNU-stack  
00000000 g F .text 00000026 \_\_printf  
00000000 \*UND\* 00000000 stdout  
00000000 \*UND\* 00000000 vfprintf  
00000000 g F .text 00000026 printf  
00000000 g F .text 00000026 \_IO\_printf

# GCC Linker - ld<sub>(9)</sub>

- \$ objdump /usr/lib/libc.a -xd > libc.txt

## SYMBOL TABLE:

00000000	I	d .text	00000000 .text
00000000	I	d .data	00000000 .data
00000000	I	d .bss	00000000 .bss
00000000	I	d .comment	00000000 .comment
00000000	I	d .note.GNU-stack	00000000 .note.GNU-stack
00000000	g	F .text	00000026 __printf
00000000		*UND*	00000000 stdout
00000000		*UND*	00000000 <b>vfprintf</b>
00000000	g	F .text	00000026 printf
00000000	g	F .text	00000026 _IO_printf



**vfprintf → vfprintf.o**

Symbol table 顯示：printf 依賴更大的 module – vfprintf

# GCC Linker - ld(10)

Disassembly of section .text:

00000000 <\_IO\_printf>:

```
0: 55                      push   %ebp
1: 89 e5                   mov    %esp,%ebp
3: 83 ec 10                sub    $0x10,%esp
6: 8d 45 0c                lea    0xc(%ebp),%eax
9: 89 45 fc                mov    %eax,0xfffffff(%ebp)
c: 89 44 24 08             mov    %eax,0x8(%esp)
10: 8b 45 08                mov    0x8(%ebp),%eax
13: 89 44 24 04             mov    %eax,0x4(%esp)
17: al 00 00 00 00           mov    0x0,%eax
18: R_386_32    stdout
lc: 89 04 24                mov    %eax,(%esp)
1f: e8 fc ff ff ff           call   20 <_IO_printf+0x20>
20: R_386_PC32  vfprintf
24: c9                      leave
25: c3                      ret
```

注意 i386 的 call 指令

事實上，vfprintf 才是 printf 的實做部份

在 GNU C Library 中，“vf”開頭的 symbol 即這一類的表現形式

# 好朋友：ELF & DWARF



## ◆ **elf**

- 小精靈、小妖精
- 小淘氣



## ◆ **dwarf**

- 矮子、侏儒
- 矮小的動植物
- (神話中) 有魔法的小矮人

# Executable File vs. Image File

---

- ◆ Linked program → 多個 "sections"
- ◆ Linker彙整object modules為特定的 Executable File Format
  - a.out (Assembly OUTput)
    - ◆ Unix format
  - Mach-O (Mach Object)
    - ◆ 為 Mac OS X 所採用
  - ELF (Executable and Linkable Format)
    - ◆ 包含 "DWARF"

# Linux Kernel v2.6.17.3-ubuntu Configuration

## Linux Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus --->.

Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ] excluded <M> module < >

↑(-)

Loadable module support --->

Block layer --->

Processor type and features

Power management options (ACP)

Bus options (PCI, PCMCIA, EIS)

**Executable file formats --->**

Networking --->

Device Drivers --->

File systems --->

Instrumentation Support --->

↓(+)

**<Select>**

< Exit >

## Linux Kernel v2.6.17.3-ubuntu Configuration

### Executable file formats

Arrow keys navigate the menu. <Enter> selects subm

Highlighted letters are hotkeys. Pressing <Y> incl

<M> modularizes features. Press <Esc><Esc> to exit

for Search. Legend: [\*] built-in [ ] excluded <M>

**[\*] Kernel support for ELF binaries**

<M> Kernel support for a.out and ECOFF binaries

<M> Kernel support for MISC binaries

**<Select>**

< Exit >

< Help >

# ELF(1)

- ◆ ELF (Executable and Linkable Format)

- 最初由UNIX System Laboratories發展，為AT&T System V Unix所使用，稍後成為BSD家族與GNU/Linux上object file的標準二進位格式

- ◆ COFF (Common Object File Format)

- System V Release 3使用的二進位格式

- ◆ DWARF-1/2 (Debug Information Format)

- 通常搭配ELF或COFF等格式

◆ 只要符合 DWARF 規範的 object file ，即可使用 **GNU gdb** 一類 source-level debugger

◆ 格式上， **Machine-Independent**

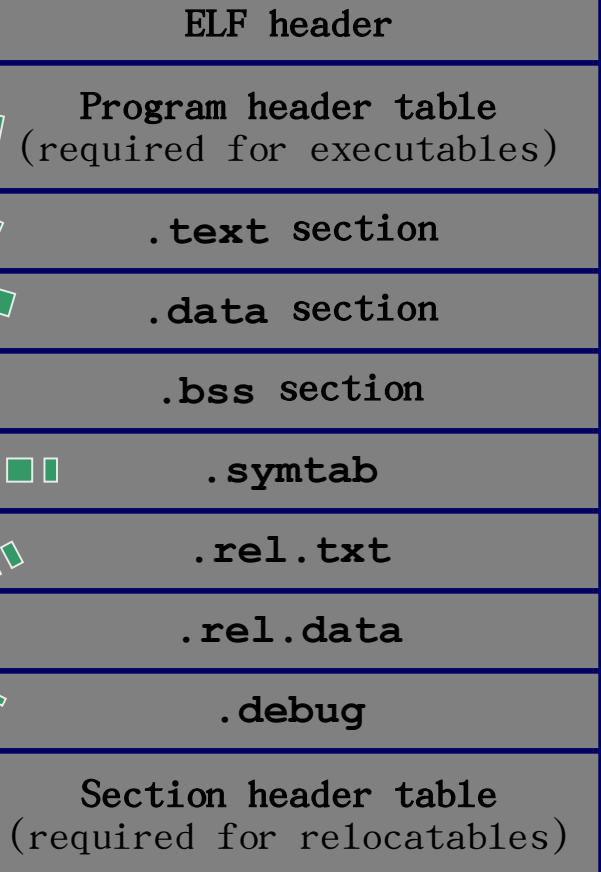
# ELF(2)

- ◆ Page size
- ◆ Virtual address
- memory segment  
(sections)
- ◆ Segment size
- ◆ Magic number
- ◆ type (.o / .so / exec)
- ◆ Machine
- ◆ byte order

- ◆ Initialized (static) data
- ◆ Un-initialized (static) data
- ◆ Block started by symbol
- ◆ **Has section header but occupies no space**

- ◆ Symbol table
- ◆ Procedure and static variable names
- ◆ Section name
- ◆ Relocation info for .text section
- ◆ Addresses of instructions that need to be modified in the executable
- instructions for modifying.

- ◆ Relocation info for .data section
- ◆ Address pointer data will need to be modified in the merged executable
- ◆ Info for symbolic debugging



# ELF(3)

- ◆ Page size
- ◆ Virtual address
- memory segment  
(sections)
- ◆ Segment size

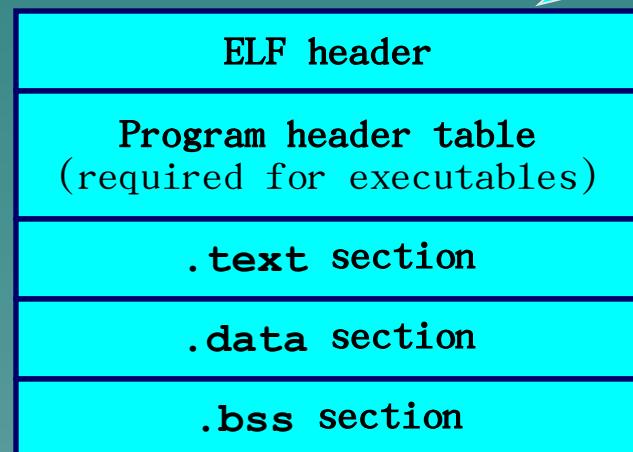
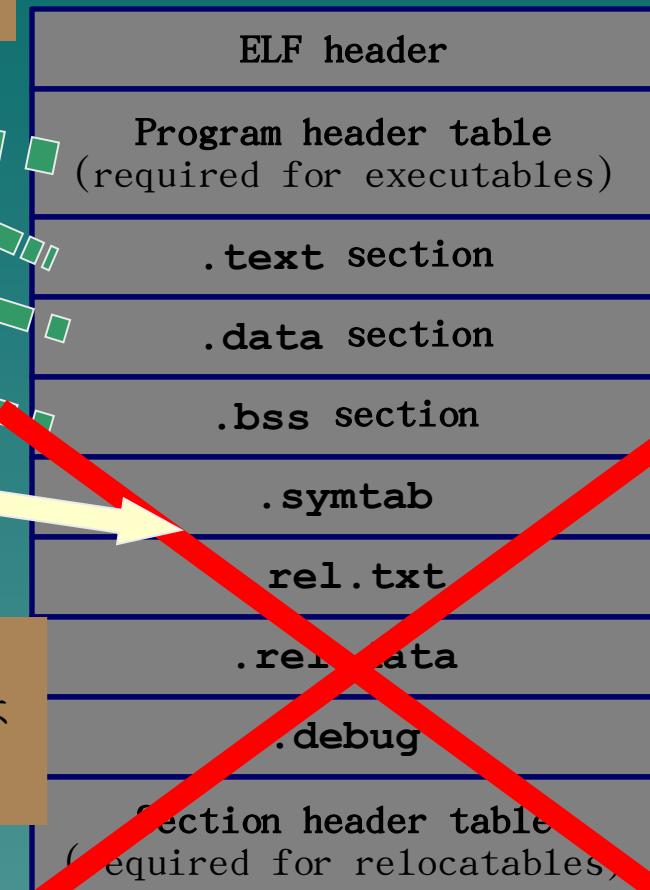
- ◆ Magic number
- ◆ type (.o / .so / exec)
- ◆ Machine
- ◆ byte order

- ◆ Initialized (static) data
- ◆ Un-initialized (static) data
- ◆ Block started by symbol
- ◆ **Has section header but occupies no space**

code

注意：.dynsym 還保留

Runtime 只需要左邊欄位  
可透過 “strip” 指令去除不  
需要的 section



```
$ readelf -s hello
```

Symbol table '**.dynsym**' contains 5 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	399	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.0 (2)
2:	00000000	415	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.0 (2)
3:	08048438	4	OBJECT	GLOBAL	DEFAULT	14	_IO_stdin_used
4:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__

Symbol table '**.syms**' contains 81 entries:

```
$ cp -f hello hello.strip  
$ strip -s hello.strip
```

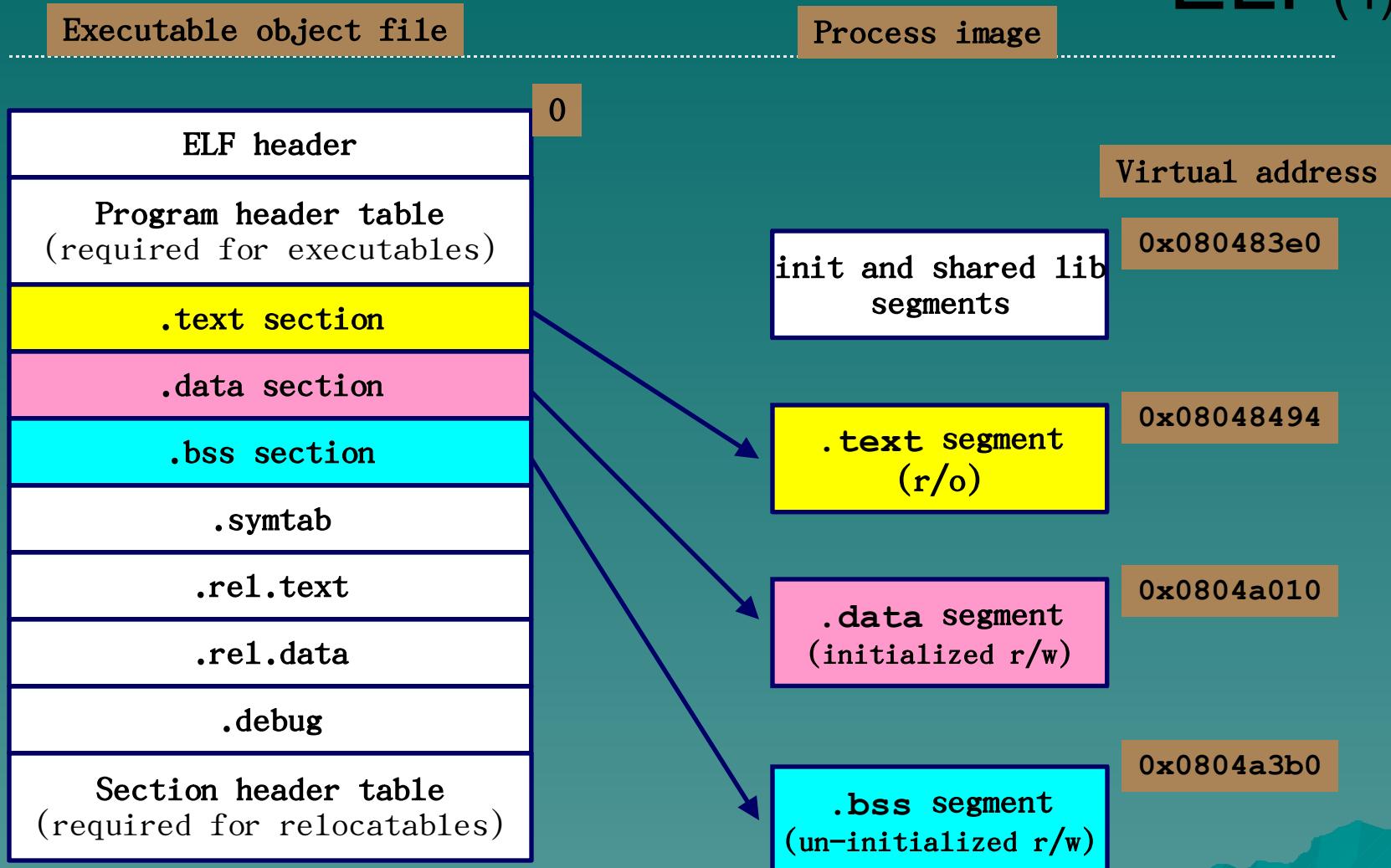
- ◆ **-s|--syms|--symbols**
  - Displays the entries in symbol table section of the file, if it has one.

```
$ readelf -s hello.strip
```

Symbol table '**.dynsym**' contains 5 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	399	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.0 (2)
2:	00000000	415	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.0 (2)
3:	08048438	4	OBJECT	GLOBAL	DEFAULT	14	_IO_stdin_used
4:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__

# ELF(4)



## Loading ELF Binaries...

<.tex> @00000870 xor ebp,ebp

entrypoint+0

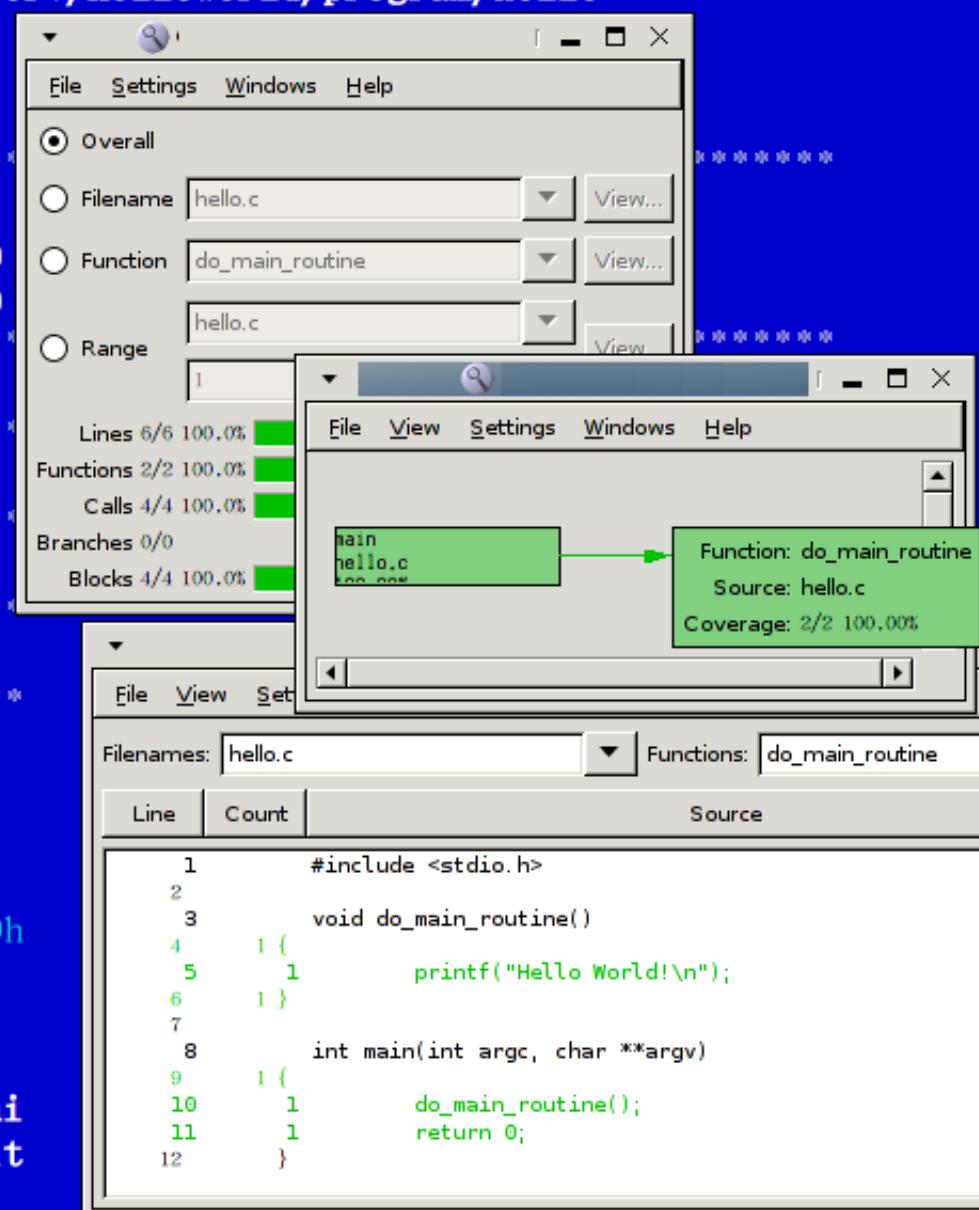
8048870 !

```
..... ! ; ****
..... ! ; section 12 <.tex>
..... ! ; virtual address 08048870
..... ! ; file offset 00000870
..... ! ; ****
..... !
..... ! ; ****
..... ! ; function _start (global)
..... !
..... ! ; executable entry point
..... !
..... ! ; ****
```

entrypoint:

8048872 !	xor	ebp, ebp
8048873 !	pop	esi
8048875 !	mov	ecx, esp
8048877 !	and	esp, 0fffffff0h
8048878 !	push	eax
8048879 !	push	esp
804887a !	push	edx
804887b !	push	_libc_csu_fini
8048880 !	push	_libc_csu_init
8048885 !	push	ecx
8048886 !	push	esi
8048887 !	push	main

8048870/@00000870



```
[x]----- /home/jserv/HelloWorld/helloworld/samples/00-pureC/hello
* ELF section headers at offset 000007e4
[+] section 0:
[-] section 1: .interp
  name string index
  type
  flags
  address
  offset
  size
  link
  info
  alignment
  entsize
[+] section 2: .note.ABI-tag
[+] section 3: .hash
[+] section 4: .dynsym
[+] section 5: .dynstr
[+] section 6: .gnu.version
[+] section 7: .gnu.version_r
[+] section 8: .rel.dyn
[+] section 9: .rel.plt
```

0000000b	
00000001	(progbits)
00000002	details
08048114	
00000114	
00000013	
00000000	
00000000	
00000001	
00000000	

.interp →  
elf\_interpreter

\$ **/lib/ld-linux.so.2**

Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]

You have invoked `ld.so', the helper program for shared library executables. This program usually lives in the file `/lib/ld.so', and special directives in executable files using ELF shared libraries tell the system's program loader to load the helper program from this file. This helper program loads the shared libraries needed by the program executable, prepares the program to run, and runs it.

\$ objdump -s -j .interp hello

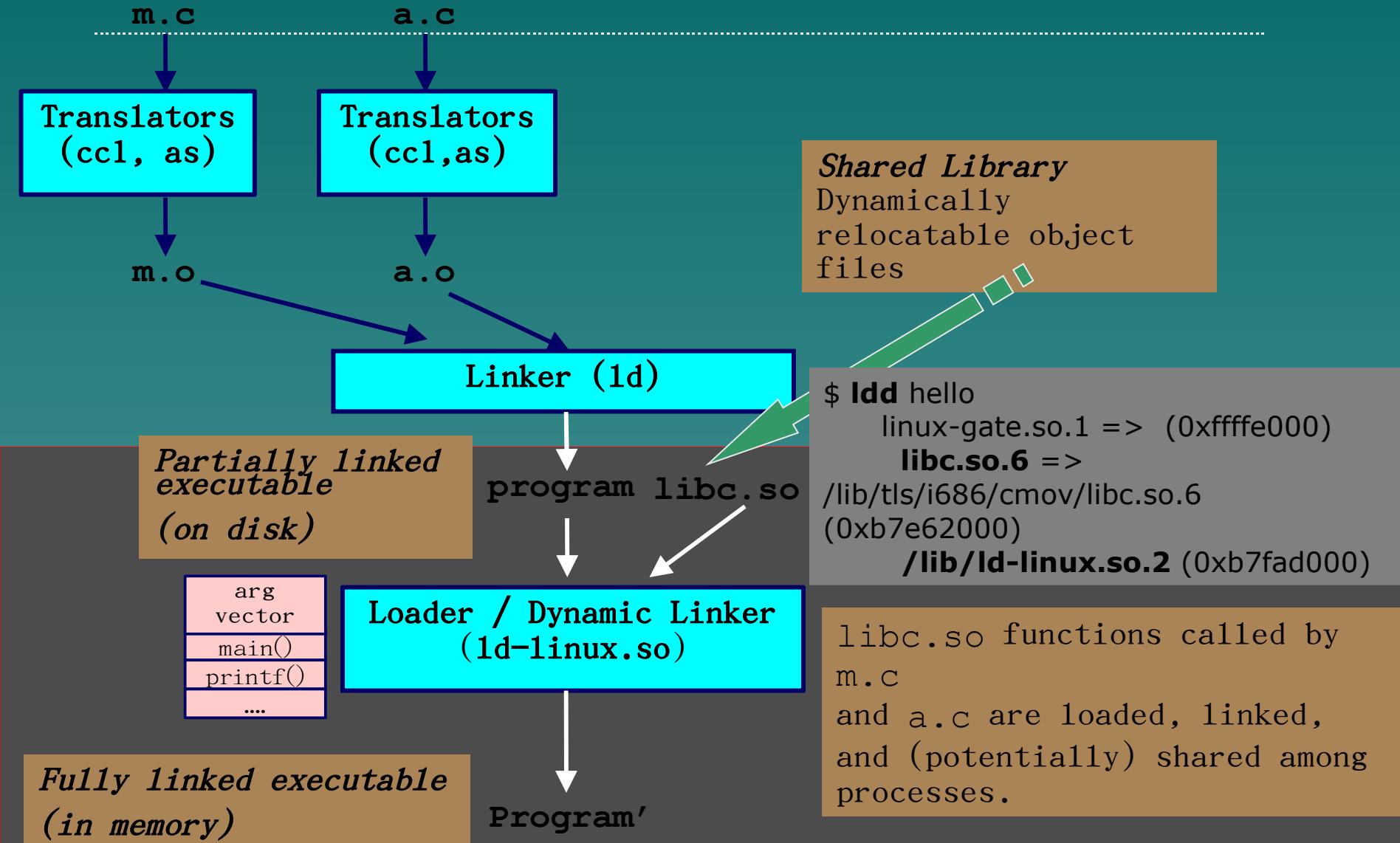
hello: file format elf32-i386

Contents of section .interp:

8048114 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so

8048124 2e3200

# Dynamically Linked Shared Libraries



```
$ /lib/ld-linux.so.2
```

Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-  
PROGRAM...]

```
$ file /lib/ld-linux.so.2
```

/lib/ld-linux.so.2: symbolic link to `ld-2.4.so'

```
$ file /lib/ld-2.4.so
```

/lib/ld-2.4.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), stripped

```
$ objdump -f /lib/ld-2.4.so
```

/lib/ld-2.4.so: file format elf32-i386

architecture: i386, flags 0x00000150:

HAS\_SYMS, DYNAMIC, D\_PAGED

start address 0x00000840

*glibc*

sysdeps/generic/dl-  
sysdep.c  
elf/rtld.c

```
$ LD_DEBUG=help /lib/ld-2.4.so
```

Valid options for the LD\_DEBUG environment variable are:

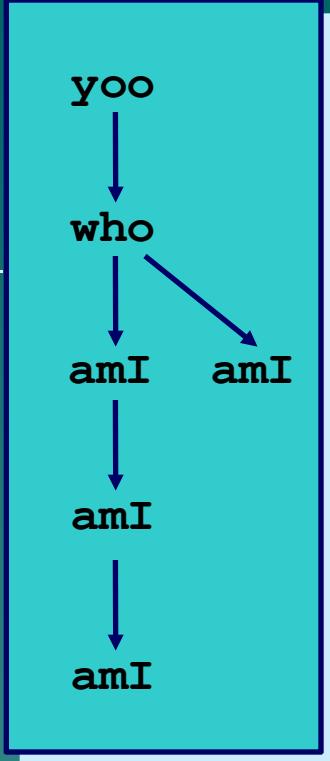
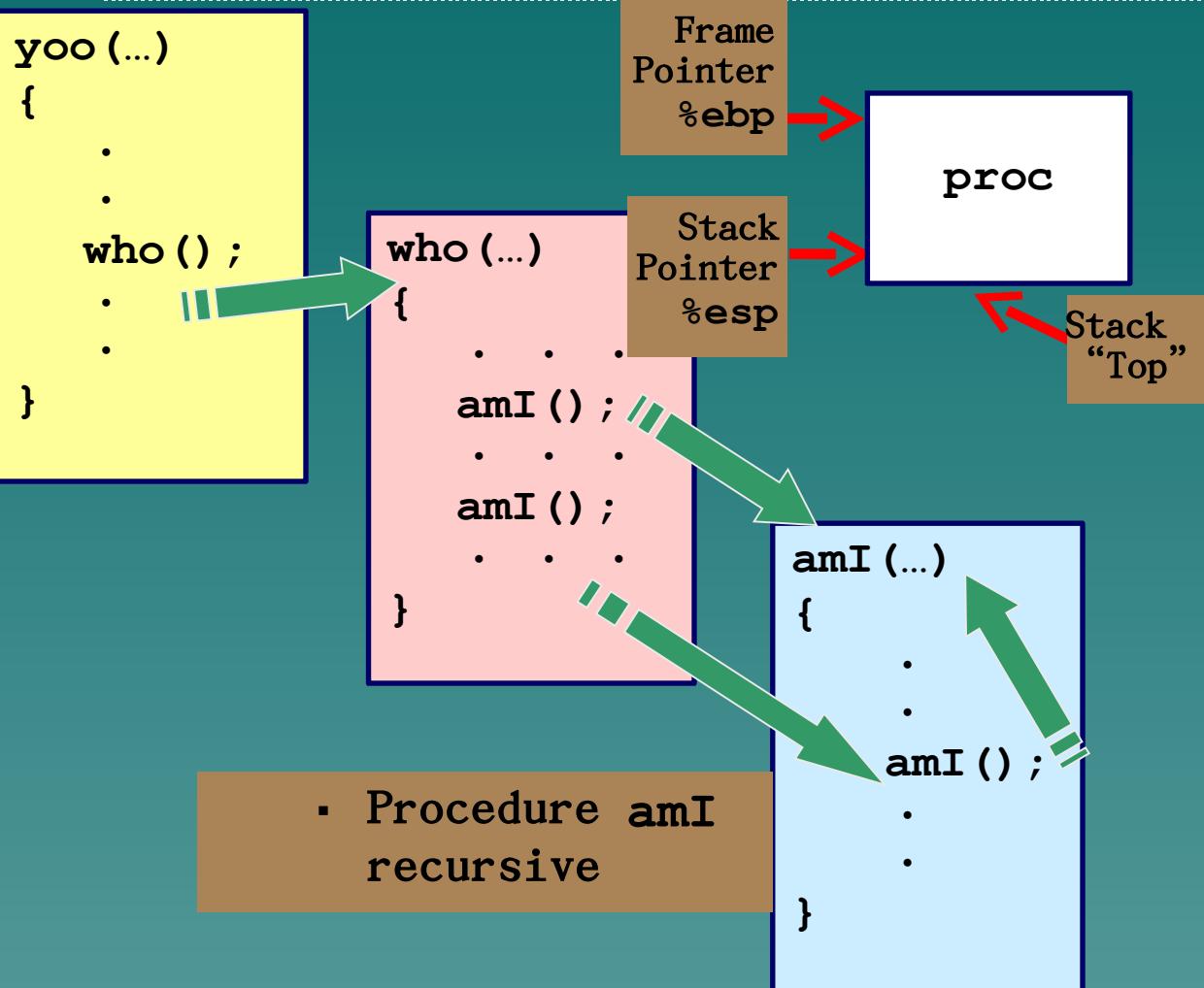
- libs display library search paths
- reloc display relocation processing
- files display progress for input file
- symbols display symbol table processing
- bindings display information about symbol binding
- versions display version dependencies
- all all previous options combined
- statistics display relocation statistics
- unused determined unused DSOs
- help display this help message and exit

*Hint*

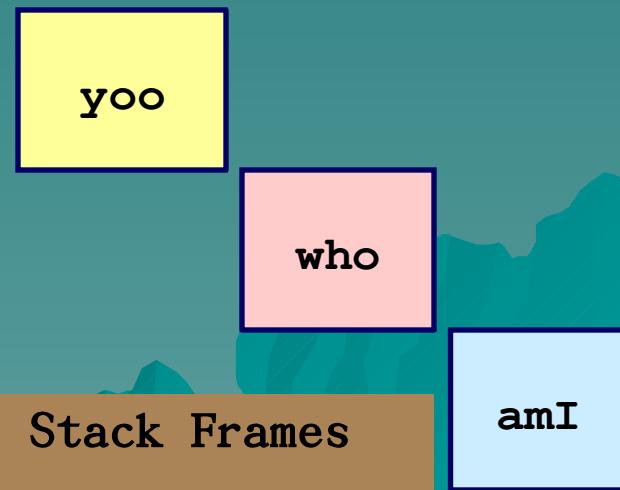
試試 LD\_DEBUG=XXX ./hello

*Hint*

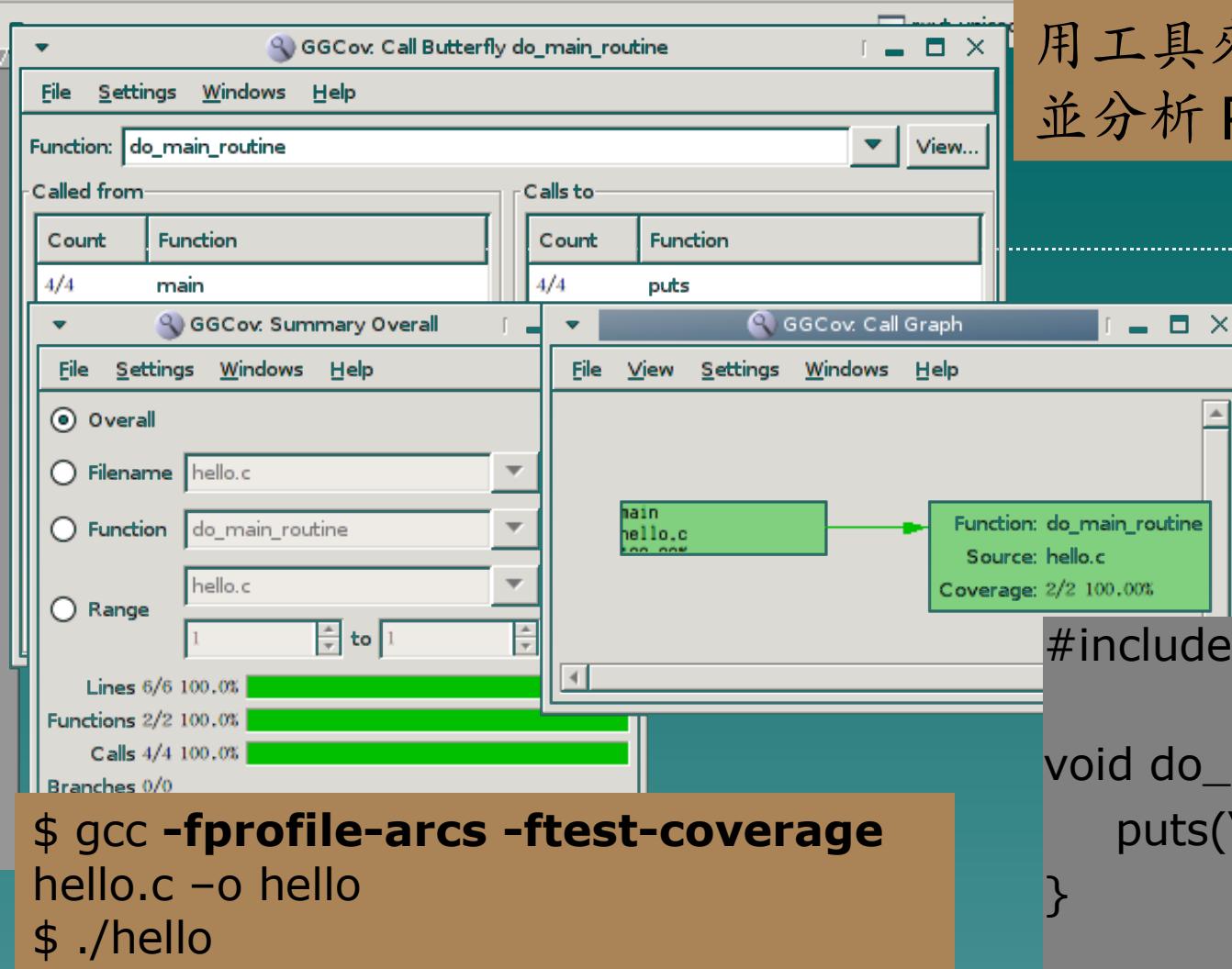
LD\_TRACE\_PRELINKING=1 ./hello



Call Graph



# 用工具來建構 Call Graph 並分析 Runtime 數據



- ◆ **gcov** is a **test coverage** program, which helps discover where your optimization efforts will best affect your code. Using gcov one can find out some basic performance statistics on a per source file level such as:
  - how often each line of code executes
  - what lines of code are actually executed

```
#include <stdio.h>

void do_myroutine(void) {
    puts("Hello World!\n");
}

int main(int argc, char *argv[])
{
    do_myroutine();
    return;
}
```

File View Settings Windows Help

Filenames: libcml/expr.c

Functions: \_expr\_add\_dependant

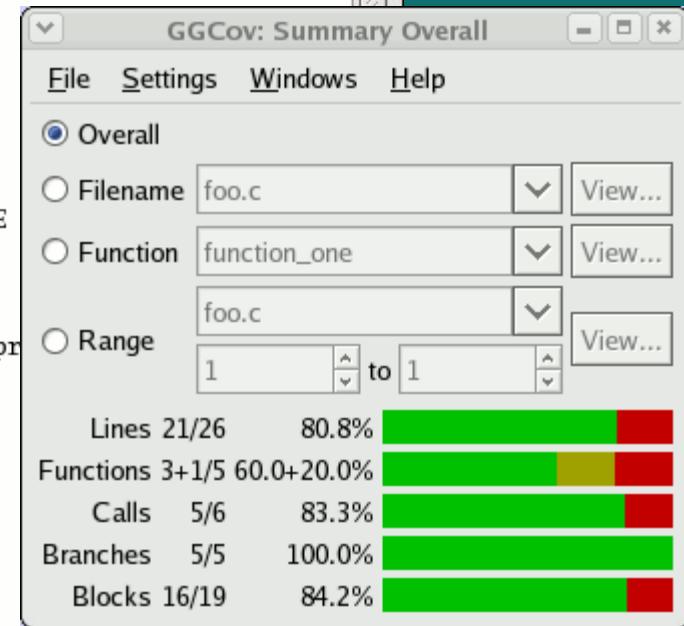
Line Count

Source

```

55     static void
36         expr_loop_init(expr_loop_context_t *lc, const char *fn)
37     {
38         lc->node = 0;
39         lc->nexprs = 0;
40         lc->func = fn;
41     }
42
43     /*
44      * Pushes an expression into the loop context, returns TRUE
45      * that would create a loop.
46      */
47     static gboolean
48     expr_loop_push(expr_loop_context_t *lc, const cml_expr *expr)
49     {
50         int i;
51
52         for (i = 0 ; i < lc->nexprs ; i++)
53         {
54             if (lc->exprs[i] == expr)
55             {
56                 #####
57                 if (lc->node == 0)
58                     cml_errorl(0,
59                                 "INTERNAL ERROR: expression loop in %s",
60                                 lc->func);
61                 #####
62                 cml_errorl(&lc->node->location,
63                             "INTERNAL ERROR: expression loop expanding \"%s\" in %s",
64                             lc->node->name,
65                             lc->func);
66             }
67         }
68         assert(lc->nexprs < LOOP_CHECK_MAX);
69         lc->exprs[lc->nexprs++] = expr;
70         if (lc->node == 0 &&
71             expr->type == E_SYMBOL &&
72             expr->symbol->streetype == MN_DERIVED)

```



◆ GGCov  
<http://ggcov.sf.net/>  
 •Graphical gcov

hello.c

`-fprofile-arcs -ftest-coverage`

gcc

編譯過程結束後

object code  
(修改過)

hello.bb

hello.bbg

植入一些特定的 code

- 找出 arcs 與 blocks execution times

`./hello`

C runtime (glibc) 與執行  
檔連結，提供所需函式，  
並呼叫稍早註冊的函式

較新的版本整合為  
**hello.gcno**

...

...

executable

與 profiling 有關的 function  
entry 加入 “.ctors” section

program exit

gcov 透過  
main wrapper  
function

邁入新紀元…



# Orz 2.0

programming



編 程 認 可



# Orz Programming

---

## ◆ 1.0

- 使用軟體就會遇到挫折
- 信任Compiler與Linker的黑盒子，並且對Runtime做了許多假設，而且大部分符合預期

## ◆ 2.0

- 到處都是陷阱（呼應蘇格拉底名言）
- 求證、求證，再求證
- 「媽媽，這裡好危險，我要回家」

# 00-pureC

```
$ cat hello-gmon.c  
#include <stdio.h>
```

```
int __gmon_start__()  
{ printf("Hello World!\n"); }  
  
int main()  
{  
    return 0;  
}
```

```
$ gcc -o hello-gmon hello-gmon.c  
$ ./hello-gmon  
Hello World!
```

GCC 參數 **-p** 表示加入 **profiling**，  
實做透過 **\_\_gmon\_start\_\_** 的註冊動作

```
$ gcc -p -o hello-gmon hello-gmon.c  
/tmp/cciNIhtV.o: In function `__gmon_start__':  
hello-gmon.c:(.text+0x0): multiple definition of  
`__gmon_start__'  
/usr/lib/gcc/i486-linux-  
gnu/4.1.2/../../../../lib/crti.o:/build/builddd/glibc-2.4/build-  
tree/glibc-2.4/csu/gmon-start.c:61: first defined here  
/usr/bin/ld: Warning: size of symbol `__gmon_start__'  
changed from 61 in /usr/lib/gcc/i486-linux-  
gnu/4.1.2/../../../../lib/crti.o to 25 in /tmp/cciNIhtV.o  
collect2: ld returned 1 exit status
```

```
$ readelf -s hello-gmon | grep -A7 dysym  
Symbol table '.dysym' contains 5 entries:  
Num: Value Size Type Bind Vis Ndx Name  
0: 00000000 0 NOTYPE LOCAL DEFAULT UND  
1: 00000000 399 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.0 (2)  
2: 00000000 415 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.0 (2)  
3: 08048418 4 OBJECT GLOBAL DEFAULT 14 IO stdin used  
4: 08048334 20 FUNC GLOBAL DEFAULT 12 __gmon_start__
```

# 01-preload

```
$ cat hello.c
#include <stdio.h>

char message[128] =
    "Hello World!\n";
int main(int argc, char **argv)
{
    puts(message);
    return 0;
}
gcc -o hack.so -shared hack.c -ldl
LD_PRELOAD=./hack.so ./hello
```

```
$ LD_PRELOAD=./hack.so
LD_TRACE_LOADED_OBJECTS=1 ./hello
linux-gate.so.1 => (0xffffe000)
./hack.so (0xb7f21000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6
(0xb7dd8000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2
(0xb7dd3000)
/lib/ld-linux.so.2 (0xb7f25000)
```

```
$ cat hack.c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int puts (char * s)
{
    char *t = s;
    while (*s) {
        if (*s == 'H')
            *s = 'K';
        else
            *s = toupper( *s );
        s++;
    }
    return (int)
        write(0, t, strlen(t));
}
```

# 02-avoid-preload(1)

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

#define FORKED_ENV "HELLO_BEEN_FORKED"
static int been_forked = 0;

static char message[128] = "Hello World!";

int main(int argc, char **argv)
{
    int pid = fork();
    if (getenv(FORKED_ENV)) {
        been_forked = 1;
    }
}
```

LD\_PRELOAD=./hack.so ./hello

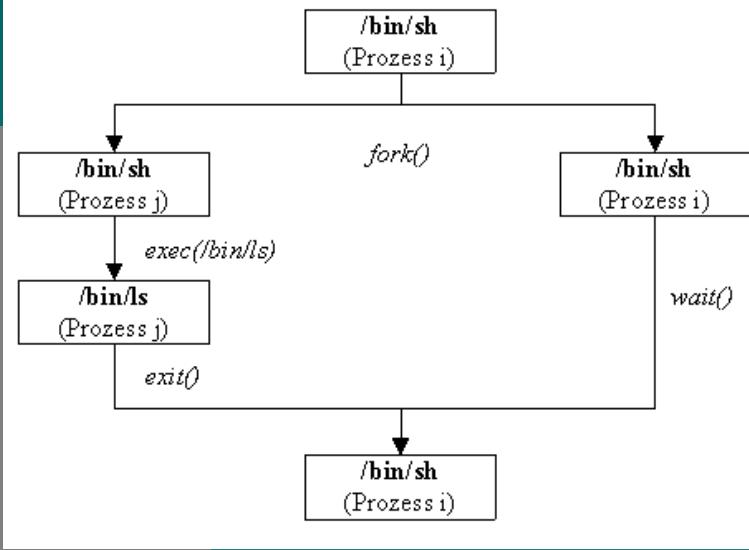


深入浅出

```
/* in child process */
if (! pid) {
    if (! been_forked) {
        setenv(FORKED_ENV, "1", 1);
        execvp(argv[0], argv);
    }
}

/* in parent process */
if (pid) {
    puts(message);
    waitpid (pid, NULL, 0);
}

return 0;
```



# 02-avoid-preload(2)

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

#define FORKED_ENV "HELLO_BEEN_FORKED"
static int been_forked = 0;
```

```
void __attribute__((constructor)) unset_Id_preload()
{
```

```
    printf("Performing %s...\n", __FUNCTION__);
    unsetenv("LD_PRELOAD");
    if (getenv(FORKED_ENV)) {
        been_forked = 1;
    }
}
```

```
static char message[128] = "Hello World!";
```

```
int main(int argc, char **argv)
{
```

```
    int pid = fork();
```

LD\_PRELOAD=./hack.so ./hello

.ctor section

GCC Extension:  
constructor attribute

```
/* in child process */
if (!pid) {
    if (!been_forked) {
        setenv(FORKED_ENV, "1", 1);
        execvp(argv[0], argv);
    }
}

/* in parent process */
if (pid) {
    puts(message);
    waitpid (pid, NULL, 0);
}

return 0;
```

# 03-dynamic loading(1)

```
$ cat hello.c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int main(int argc, char **argv)
{
    void *handle;
    void (*func)(void);
    const char *error;

    /* shared object */
    handle = dlopen("./shared.so",
                    RTLD_NOW);
    if (! handle) {
        fputs(dlerror(), stderr);
        exit(1);
    }
    func = dlsym(handle, "hello");
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr); exit(1);
    }
    (*func)();
    dlclose(handle);
}
```



```
$cat share.c
#include <stdio.h>
#include <stdlib.h>

/* GCC __attribute__ */
void __attribute__((constructor)) Hello_init()
{
    printf("_init invoked!\n");
}

void __attribute__((destructor)) Hello_fini()
{
    printf("_fini invoked!\n"); 可替代為 fini
}

/* Customized routines here */
void hello()
{
    printf("Hello World!\n");
}
```

# 03-dynamic loading(2)

```
$ cat Makefile  
LIBS= -ldl  
SRCS=main.c  
OBJECTS=$(SRCS:.c=.o)  
CFLAGS=-g -rdynamic -fPIC  
CC=gcc
```



## PIC → Position-Independent Code

- shared libraries → 可在任何位址載入與執行，而不需要 Linker 在執行時期介入
- 然而，PIC 會帶來效能衝擊

```
LD_DEBUG=libs ./hello
```

```
all: hello shared.so  
  
hello: $(OBJECTS)  
       $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LIBS)  
  
# Shared Objects here  
shared.so: shared.o  
       gcc -shared shared.o -o shared.so -g
```

```
7043:  
7043:      calling init: ./shared.so  
7043:  
_init invoked!  
Hello World!  
Execution Finished.ok  
7043:  
7043:      calling fini: ./shared.so [0]  
7043:  
_fini invoked!  
7043:
```

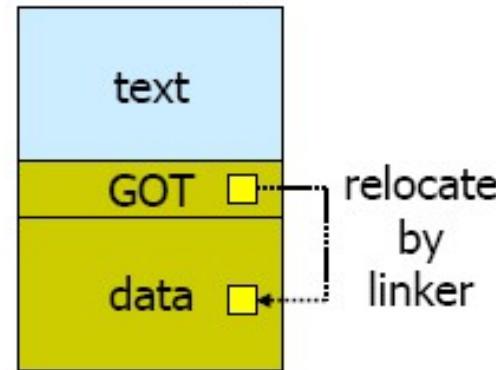
```
7043:      find library=libdl.so.2 [0]; searching  
7043:      search cache=/etc/ld.so.cache  
7043:          trying file=/lib/tls/i686/cmov/libdl.so.2  
7043:  
7043:      find library=libc.so.6 [0]; searching  
7043:      search cache=/etc/ld.so.cache  
7043:          trying file=/lib/tls/i686/cmov/libc.so.6  
7043:  
7043:      calling init: /lib/tls/i686/cmov/libc.so.6  
7043:  
7043:      calling init: /lib/tls/i686/cmov/libdl.so.2  
7043:  
7043:  
7043:      initialize program: ./hello  
7043:  
7043:      transferring control: ./hello  
7043:  
I am about to load Hello World module..ok
```

# 03-dynamic loading(3)

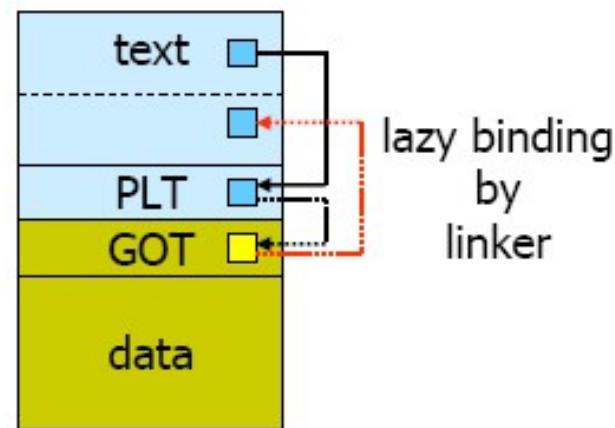
```
// global data reference  
  
    call L1  
L1: popl %ebx  
    addl $GOTENTRY, %ebx  
    movl (%ebx), %eax  
    movl (%eax), %eax
```

```
// global procedure reference  
  
    call $PLTENTRY  
  
-----  
PLT[0]:  pushl (GOT[1])      // special id  
          jmp  *(GOT[2])      // dyn linker  
  
...  
PLTENTRY: jmp   *($GOTENTRY)  
PLTLAZY:  pushl $func_id  
          jmp   PLT[0]  
  
-----  
GOT[1]  : (special id for dyn linker)  
GOT[2]  : (entry point in dynamic linker)  
  
GOTENTRY: $PLTLAZY           // changed by  
          real entry point      // lazy binding
```

Shared library



Shared library



實做方式：

- GOT (global offset table)：位於 data segment
- PLT (procedure linkage table)：位於 code segment

# 04-PIE (Position-Independent Execution)-1

```
gcc -c hello.c  
gcc -o hello hello.o
```

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

```
gcc -c hello.c -fPIE  
gcc -o hello-pie -pie hello.o
```

objdump -d hello | grep main -A3

```
08048360 <main>:  
8048360: 8d 4c 24 04 1ea 0x4(%esp),%ecx  
8048364: 83 e4 f0 and $0xffffffff,%esp  
8048367: ff 71 fc pushl 0xfffffff(%ecx)
```

```
00000060 <main>:  
60e: 8d 4c 24 04 1ea 0x4(%esp),%ecx  
612: 83 e4 f0 and $0xffffffff,%esp  
615: ff 71 fc pushl 0xfffffff(%ecx)
```

Position-Independent Execution

# 04-PIE (Position-Independent Execution)-2

```
$ cat say.c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <assert.h>
typedef void (*hello_t)(void);

int main(void)
{
    hello_t hello;
    void *handle = dlopen (
        "./hello-impl",
        RTLD_LAZY);
    assert(handle);
    hello = dlsym(handle, "hello");
    hello();
    dlclose(handle);
    return 0;
}
```

```
$ cat hello.c
#include <stdio.h>
void hello() {
    printf("Hello World!\n");
}

int main(void) {
    hello();
    return 0;
}
```

build\_say:

```
gcc -c hello.c -fPIE -o hello-impl.o
gcc -o hello-impl -pie -rdynamic hello-impl.o
gcc -g -o say say.c -ldl
```

# 05-printf-vs-puts(1)

- ◆ C語言真的是WYSIWYG (What You See Is What You Get) 嗎？

```
$ cat Makefile
all: PrepareSourceCode Normal NoBuiltIns
```

PrepareSourceCode:

```
    echo "int main() { printf(\"Hello World\\\"\\\"\\\""); return 0; }" > hello.c
```

Normal:

```
gcc -o hello hello.c
readelf -a hello | egrep 'printf|puts'
```

NoBuiltIns:

```
gcc -o hello_opt_nobuiltins -fno-builtin hello.c
readelf -a hello_opt_nobuiltins | egrep 'printf|puts'
```

# 05-printf-vs-puts(2)

```
int main() { printf("Hello World\n"); return 0; }
```

```
$ make
echo "int main() { printf(\"Hello World\"); return 0; }" > hello.c
gcc -o hello hello.c
hello.c: In function 'main':
hello.c:1: warning: incompatible implicit declaration of built-in function
'printf'
```

```
readelf -a hello | egrep 'printf|puts'
08049528 00000107 R_386_JUMP_SLOT 00000000 puts
    1: 00000000    399 FUNC    GLOBAL DEFAULT UND puts@GLIBC_2.0 (2)
    64: 00000000    399 FUNC    GLOBAL DEFAULT UND puts@@GLIBC_2.0
gcc -o hello_opt_nobuiltins -fno-builtin hello.c
```

```
readelf -a hello_opt_nobuiltins | egrep 'printf|puts'
08049538 00000207 R_386_JUMP_SLOT 00000000 printf
    2: 00000000     57 FUNC    GLOBAL DEFAULT UND printf@GLIBC_2.0 (2)
    71: 00000000     57 FUNC    GLOBAL DEFAULT UND printf@@GLIBC_2.0
```

-fno-builtin

## Hint

如果將”\n”自字串中移除，又會如何？

# 06-backtrace

GCC builtin function

```
void hello () {
```

```
    printf("Hello World!\n");
    show_debug_info(
        __builtin_return_address(0) - 1);
}
```

```
void __attribute__((constructor))
init_bfd_stuff () {
```

```
    abfd = bfd_openr(
        "/proc/self/exe", NULL);
    assert(abfd != NULL);
    bfd_check_format(abfd, bfd_object);
```

```
    int size =
        bfd_get_symtab_upper_bound(abfd);
    assert(size > 0);
    symbols = malloc(size);
    assert(symbols != NULL);
    nsymbols = bfd_canonicalize_symtab(
        abfd, symbols);
}
```

```
int main () {
    hello();
    return 0;
}
```

Binary File Descriptor  
libbfd 為 binutils-dev 所提供

```
#include <assert.h>
#include <bfd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libgen.h>
static bfd *abfd;
static asymbol **symbols;
static int nsymbols;

void show_debug_info (void *address) {
    asection *section =
        bfd_get_section_by_name(
            abfd, ".debug_info");
    assert(section != NULL);

    const char *file_name;
    const char *function_name;
    int lineno;
    int found = bfd_find_nearest_line(
        abfd, section, symbols, (long)address,
        &file_name, &function_name, &lineno);
    if (found && file_name != NULL &&
        function_name != NULL) {
        char tmp[strlen(file_name)];
        strcpy(tmp, file_name);
        printf("%s:%s:%d\n",
            basename(tmp),
            function_name, lineno);
    }
}
```

```
$ ls -l /proc/self/exe
```

```
1rwxrwxrwx 1 jserv jserv 0 Jul 10 16:02 /proc/self/exe -> /bin/ls
```

```
$ realpath /proc/self/exe
```

```
/usr/bin.realpath
```

# ELF

```
$ gcc -o hello hello.c -fbfd
```

```
$ ./hello
```

```
Hello World!
```

```
$ gcc -o hello_g hello.c -g -fbfd
```

```
$ ./hello_g
```

```
Hello World!
```

```
hello.c:main:54
```

透過 BFD (Binary File Descriptor) Library，  
得到 Runtime 擷取資訊  
並改變流程的新途徑

0	ELF header
	Program header table (required for executables)
	.text section
	.data section
	.bss section
	.symtab
	.rel.txt
	.rel.data
	.debug
	Section header table (required for relocatables)

```
$ ltrace ./hello > /dev/null
__libc_start_main(0x8048925, 1,
0xbfe27244, 0x8048951,
0x804894c <unfinished ...>
bfd_openr("/proc/self/exe", NULL)
= 0x804a008
bfd_check_format(0x804a008, 1)
= 1
malloc(388)
= 0x804b0d0
puts("Hello World!")
= 13
bfd_get_section_by_name(0x804a0
08, 0x8048a14, 12, 0xbfe27164,
0xb7d67f0b) = 0x8053500
+++ exited (status 0) +++
```

```
$ ltrace ./hello_g > /dev/null
__libc_start_main(0x8048925, 1,
0xbfa04e24, 0x8048951,
0x804894c <unfinished ...>
bfd_openr("/proc/self/exe", NULL)
= 0x804a008
bfd_check_format(0x804a008, 1)
= 1
malloc(396)
= 0x804b0d0
puts("Hello World!")
= 13
bfd_get_section_by_name(0x804a0
08, 0x8048a14, 12, 0xbfa04d44,
0xb7e1af0b) = 0x8053500
strcpy(0xbfa04cd0,
"/home/jserv/HelloWorld/helloworld".
..) = 0xbfa04cd0
__xpg_basename(0xbfa04cd0,
0x8063894, 0xbfa04d44,
0xbfa04d40, 4) = 0xbfa04d07
printf("%s:%s:%d\n", "hello.c",
"main", 54) = 16
+++ exited (status 0) +++
```

# 07-SegFault(1)

```
$ ./hello  
Floating point exception
```

```
$ cat hello.c  
#include <stdio.h>  
  
int magic_num()  
{  
    return (0 / 0);  
}  
  
void hello()  
{  
    printf("Hello World! A special num: %d\n",  
        magic_num());  
}  
  
int main(int argc, char **argv)  
{  
    hello();  
    return 0;  
}
```

```
$ gdb ./hello  
(gdb) run  
Starting program: ./hello
```

**Program received signal SIGFPE, Arithmetic exception.**

**0x08048365 in magic\_num () at hello.c:5**  
**5           return (0 / 0);**  
(gdb) bt  
#0 0x08048365 in magic\_num () at hello.c:5  
#1 0x0804837b in hello () at hello.c:10  
#2 0x080483a3 in main () at hello.c:15

```
$ make  
gcc -o hello -g hello.c  
hello.c: In function 'magic_num':  
hello.c:5: warning: division by zero
```

# 07-SegFault(2)

```
$ ./hello
```

```
Floating point exception
```

```
$ cat hello.c
#include <stdio.h>

int magic_num()
{
    return (0 / 0);
}

void hello()
{
    printf("Hello World! A special num: %d\n",
        magic_num());
}

int main(int argc, char **argv)
{
    hello();
    return 0;
}
```

```
$ ./hello-backtrace
./hello-backtrace[0x8048663]
[0xffffe420]
./hello-backtrace(hello+0xb)[0x80486b0]
./hello-backtrace(main+0x69)[0x804872b]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xd)
]
./hello-backtrace[0x80485c5]
Floating point exception
```

```
$ cat hello-backtrace.c
#include <stdio.h>
#include <stdlib.h>
#include <execinfo.h>
#include <signal.h>
static void stacktrace(int signal) {
    void *trace[128];
    int n = backtrace(trace,
        sizeof(trace) / sizeof(trace[0]));
    backtrace_symbols_fd(trace, n, 1);
}

int magic_num() { }
void hello() { }

int main(int argc, char **argv) {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = stacktrace;
    sa.sa_flags = SA_ONESHOT;
    sigaction(SIGFPE, &sa, NULL);
    hello();
    return 0;
}
```

# 07-SegFault(3)

```
$ ./hello
```

Floating point exception

```
$ cat hello.c
#include <stdio.h>

int magic_num()
{
    return (0 / 0);
}

void hello()
{
    printf("Hello World! A special number is %d\n", magic_num());
}

int main(int argc, char **argv)
{
    hello();
    return 0;
}
```

```
$ SEGFAULT_SIGNALS=all
LD_PRELOAD=/lib/libSegFault.so ./hello
*** Floating point exception
Register dump:

EAX: 00000000  EBX: b7f14ff4  ECX: bff18a70  EDX: 00000000
ESI: b7f4cce0  EDI: 00000000  EBP: bff18a38  ESP: bff18a34

EIP: 08048365  EFLAGS: 00210282

CS: 0073  DS: 007b  ES: 007b  FS: 0000  GS: 0033  SS:
007b

Trap: 00000000  Error: 00000000  OldMask: 00000000
ESP/signal: bff18a34  CR2: 00000000

Backtrace:
/lib/libSegFault.so[0xb7f2e1e9]
[0xfffffe420]
./hello[0x804837b]
./hello[0x80483a3]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xd8)[0xb7df88b8]
./hello[0x80482d1]
```

Memory map:  
...

**SEGFAULT\_SIGNALS=all**

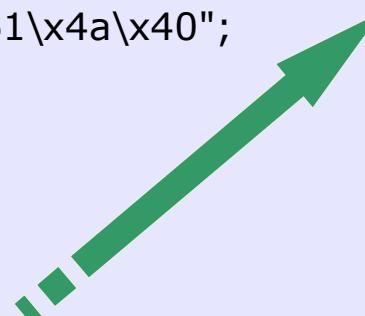
LD\_PRELOAD=/lib/libSegFault.so

# 08-shellcode

```
$ cat hello.c
```

```
unsigned char shellcode[] =  
"\xeb\x0e\x90\x5e\x31\xc9\xb1\x56\x80\x36\x40\x46\xe2\xfa\xeb\x05\xe8\xee\xff"  
"\xff\xff\x15\xc9\xa5\x17\x45\x13\x8a\x40\x40\x40\x40\x1b\xc1\x83\xb5\xbf\xbf"  
"\xb5\xcd\xd3\x08\x40\x40\xc3\xad\x4c\xc9\x91\xc3\x4a\xb0\xff\x41\x40\x40"  
"\x40\xf8\x44\x40\x40\x40\xfa\x4e\x40\x40\x40\x13\xc9\xbb\x8d\xc0\x1b\xc9\xb8"  
"\x13\xfb\x45\x40\x40\x40\x8d\xc0\x1b\xcd\x25\xb4\x1b\x1e\x1f\x89\x83\x08\x25"  
"\x2c\x2c\x2f\x60\x37\x2f\x32\x2c\x24\x61\x4a\x40";
```

```
int main(void)  
{  
    ((void (*)())shellcode)();  
    return 0;  
}
```



```
$ strace ./hello > /dev/null  
...  
write(1, "Hello world!\n\0", 14)           = 14  
...
```

**Attack : Bufferoverflow + shellcode**

# 參考資料

- ◆ **GNU Binary Utilities, Free Software Foundation**  
[http://www.gnu.org/software/binutils/manual/html\\_chapter/binutils.html](http://www.gnu.org/software/binutils/manual/html_chapter/binutils.html)
- ◆ **ELF/DWARF, Free Standards Group – Reference Specifications**  
<http://www.freestandards.org/spec/refspecs/>
- ◆ **The GCC Project, Free Software Foundation**  
<http://gcc.gnu.org/>
- ◆ **O'Reilly Understanding the Linux Kernel**  
<http://www.oreilly.com/catalog/linuxkernel/>

# What's Next?

---

- ◆ Linux Memory Model
- ◆ Linux system call & Hello World