Link 3.A Object Files

Young W. Lim

2018-01-21 Mon

Outline

Based on

"Self-service Linux: Mastering the Art of Problem Determination",
Mark Wilding
"Computer Architecture: A Programmer's Perspective",
Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

Object Files

1. Relocatable object file
2. Executable object file
3. Types of Executable Object Files

Relocatable Object Files

- Relocatable object file
  - contains binary code and data in a form
  - that can be combined with other relocatable object files
  - at compile time to create an executable object file

Executable Object Files

- Shared object file
  - a special relocatable object file
  - that can be loaded into memory and linke dynamically
  - at either load time or run time

Types of Excutable Object Files

- a.out
- COFF (Common Object File format)
- PE (Portable Executable format)
- ELF (Executable and Linkable Format)

ELF Relocatable Object Files

Linking View

ELF Header
Program Header Table
Section 1
Section 2
Section 3
...
...
...
...
...
Section n
Section Header Table

Possible Section Types

.text
.rodata
.data
.bss
.symtab
.rel.text
.rel.data
.debug
.line
.strtab

- Program Header Table is optional

ELF Files

1. ELF Header
2. Program Header Table
3. Section Header Table
4. Memory Map
5. ELF Relocatble and Executbale Object Files
6. ELF Executable Object Files

ELF Header

- 16-byte sequence :
  the word size and byte ordering
- information that allows a linker
  to parse and interpret the object file
  - the size of ELF header
  - the object file type (relocatble, executable, shared)
  - the machine type

Program Header Table [1]

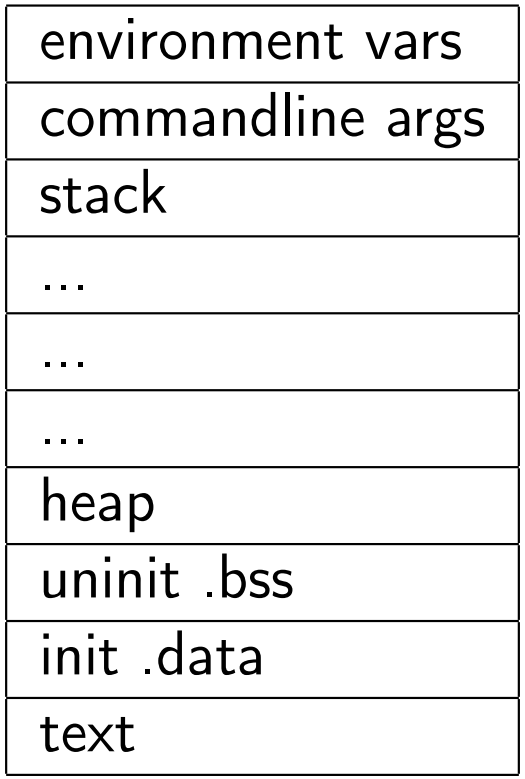
- tells the system how to create a process image
- executable files must have program header table
- relocatable files do not need (optional)
- need for execution

[1] http://www.skyfree.org/linux/references/ELF_Format.pdf

Section Header Table [1]

- identifies all the sections in the file
- every section in the file has an entry
  in the section header table
- an array of structures with an index
- the file offset of the section header table
- the size and number of entries in the section header table
- contains a fixed sized entry for each section in the object file
- relocatable files must have program header table
- executable files do not need (optional)
- need for linking

Memory Map


environment vars
commandline args
stack
...
...
...
heap
uninit .bss
init .data
text

ELF Relocatable and Executable Object Files

Linking View

ELF Header
Program Header Table
Section 1
Section 2
Section 3
...
...
...
...
...
Section n
Section Header Table

- Program Header Table is optional

Execution View

ELF Header
Segment Header Table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header Table

ELF Executable Object Files (1)

Execution View

ELF Header
Segment Header Table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header Table

- RO Memory (Code) Segment
  - ELF Header
  - Segment Header Table
  - .init
  - .text
  - .rodata
- RW Memory (Data) Segment
  - .data
  - .bss
- Non-memory Segment
  - .debug
  - .line
  - .strtab
  - Section Header Table

ELF Executable Object Files (2)

- ELF Header
  - describes the overall format of the file
  - includes the program's entry point
    the address of the 1st instruction to execute
    when the program runs
- Segment Header Table
  - maps contiguous file sections
  - to run-time memory segments
- .init
  - defines a small function `_init`
  - called by the program's initialization cdoe
- .text, .rodata, data
  - relocated to the final run-time memory addresses
- no .rel.text neither .rel.data

ELF Sections

1. ELF Section Types Overview
2. .text, .rodata
3. .data, .bss
4. .symtab
5. .rel.text
6. .rel.data
7. .debug
8. .line
9. .strtab

ELF Section Types Overview

.text	the machine code
.rodata	such as the format strings of printf
.data	initialized global c variables
.bss	uninitialized global c variables
.symtab	about functions and global variables
.rel.text	a list of locations in the .text to be modified
.rel.data	global variables referenced or defined
.debug	local, global variables, typedefs, c source files
.line	line number information
.strtab	a string table for .symtab and .debug

.text, .rodata

- `.text`
  - the machine code
- `.rodata`
  - read only data
  - string constants
  - jump tables for swtich statements

.data, .bss

- `.data`
  - initialized *global* variables
  - initialized *static* variables
  - no local variables (on the stack, at the run time)
  - *occupy* actual space in the object file
- `.bss`
  - uninitialized *global* variables
  - uninitialized *static* variables
  - no local variables (on the stack, at the run time)
  - no actual space in the object file
  - just a place holder for space efficiency

.symtab

- `.symtab`
  - symbol table information about *functions* and *global variables*
  - regardless of -g compile switch
  - every *relocatable* object file has a symbol table in .symtab
  - the symbol table in `.symtab` no entries for *local* variables
  - the symbol table inside a compiler does have entries for *local* variables

.rel.text

- `.rel.text`
  - locations in the text section
    will be changed when linker combines object files
  - instructions that need to be changed:
    - calls an *external function*
    - references a *global variable*
  - instructions that calls *local functions* :
    no need to be changed
  - executable object files do not need
    relocation information
    is usually omitted
    without an explicit instruction to include it

.rel.data

- `.rel.data`
  - relocation information for any *global variables* that are referenced or defined by the module
  - any initialized global variable will need to be modified whose initial value is
    - the *address* of a global variable
    - *externally* defined function

.debug

- .debug
  - a debugging symbole table entries
    - local variables
    - typedefs
    - global variables
  - only present when compiled with the -g option

.line

- `.line`
  - a mapping between line numbers and machine code instructions in the `.text`
  - only present when compiled with the -g option

.strtab

- `.strtab`
  - a string table for the symbol tables
    in the `.symtab` and `.debug` sections
  - and for the section names in the section headers
  - a string table is a sequence of
    null-terminated character string

Example Program Source Codes

1. c source code
2. main source and assembly codes
3. swap source and assembly codes
4. swap assembly codes without optimization P

c source code

```
// main.c ---------------------

void swap();

int buf[2]  = {1, 2};

int main()
{
  swap();

  return 0;
}
```

```
// swap.c ---------------------

extern int buf[];

int *p0 = &buf[0];
int *p1;

void swap()
{
  int tmp;

  p1 = &buf[1];

  tmp = *p0;
  *p0 = *p1;
  *p1 = tmp;

}
```

main source and assembly codes

```
// main.c ---------------------

void swap();

int buf[2]  = {1, 2};

int main()
{

  swap();

  return 0;
}
```

```
00000000 <main>:
   0: lea    0x4(%esp),%ecx
   4: and    $0xfffffff0,%esp
   7: pushl  -0x4(%ecx)
   a: push   %ebp
   b: mov    %esp,%ebp
   d: push   %ecx
   e: sub    $0x4,%esp
  11: call   12 <main+0x12>
  16: add    $0x4,%esp
  19: xor    %eax,%eax
  1b: pop    %ecx
  1c: pop    %ebp
  1d: lea    -0x4(%ecx),%esp
  20: ret
```

swap source and assembly codes

```
// swap.c --------------------

extern int buf[];

int *p0 = &buf[0];
int *p1;

void swap()
{
  int tmp;

  p1 = &buf[1];

  tmp = *p0;
  *p0 = *p1;
  *p1 = tmp;

}
```

```
00000000 <swap>:
   0: mov    0x0,%eax
   5: mov    0x4,%ecx
   b: movl   $0x4,0x0
  12:
  15: mov    (%eax),%edx
  17: mov    %ecx,(%eax)
  19: mov    %edx,0x4
  1f: ret
```

```
     +----------+
  p0 |  buf[0]  |
     +----------+
  p1 |  buf[1]  |
     +----------+
```

swap assembly codes witout optimization

- -O2

```
00000000 <swap>:
   0: push   %ebp
   1: mov    %esp,%ebp
   3: sub    $0x10,%esp
   6: movl   $0x4,0x0
   d:
  10: mov    0x0,%eax
  15: mov    (%eax),%eax
  17: mov    %eax,-0x4(%ebp)
  1a: mov    0x0,%eax
  1f: mov    0x0,%edx
  25: mov    (%edx),%edx
  27: mov    %edx,(%eax)
  29: mov    0x0,%eax
  2e: mov    -0x4(%ebp),%edx
  31: mov    %edx,(%eax)
  33: nop
  34: leave
  35: ret
```

- -O3

```
00000000 <swap>:
   0: mov    0x0,%eax
   5: mov    0x4,%ecx
   b: movl   $0x4,0x0
  12:
  15: mov    (%eax),%edx
  17: mov    %ecx,(%eax)
  19: mov    %edx,0x4
  1f: ret
```

```
      +----------+
   p0 |  buf[0]  |
      +----------+
   p1 |  buf[1]  |
      +----------+
```

Object File Comparsion

1. relocatable and executable main's
2. relocatable and executable swap's

relocatable and executable main's

- relocatable main

```
00000000 <main>:
   0: lea    0x4(%esp),%ecx
   4: and    $0xfffffff0,%esp
   7: pushl  -0x4(%ecx)
   a: push   %ebp
   b: mov    %esp,%ebp
   d: push   %ecx
   e: sub    $0x4,%esp
  11: call   12 <main+0x12>
  16: add    $0x4,%esp
  19: xor    %eax,%eax
  1b: pop    %ecx
  1c: pop    %ebp
  1d: lea    -0x4(%ecx),%esp
  20: ret
```

- executable main

```
080482e0 <main>:
 80482e0: lea    0x4(%esp),%ecx
 80482e4: and    $0xfffffff0,%esp
 80482e7: pushl  -0x4(%ecx)
 80482ea: push   %ebp
 80482eb: mov    %esp,%ebp
 80482ed: push   %ecx
 80482ee: sub    $0x4,%esp
 80482f1: call   8048400 <swap>
 80482f6: add    $0x4,%esp
 80482f9: xor    %eax,%eax
 80482fb: pop    %ecx
 80482fc: pop    %ebp
 80482fd: lea    -0x4(%ecx),%esp
 8048300: ret
```

relocatable and executable swap's

- relocatable swap

```
00000000 <swap>:
   0: mov    0x0,%eax
   5: mov    0x4,%ecx
   b: movl   $0x4,0x0
  12:
  15: mov    (%eax),%edx
  17: mov    %ecx,(%eax)
  19: mov    %edx,0x4
  1f: ret
```

- executable swap

```
08048400 <swap>:
 8048400: mov    0x804a020,%eax
 8048405: mov    0x804a01c,%ecx
 804840b: movl   $0x804a01c,0x804a028
 8048412:
 8048415: mov    (%eax),%edx
 8048417: mov    %ecx,(%eax)
 8048419: mov    %edx,0x804a01c
 804841f: ret
```

Assembly code analysis 1. main

1. 16-byte alignment
2. Function Prologue
3. Function Call
4. Function Epilogue

(1) 16-byte alignment

- `0: lea 0x4(%esp),%ecx`
  - %ecx= %esp+4, %ecx-4=%esp
  - %ecx = init_%esp + 4
  - (%ecx) = normally, the 1st argument of a function
- `4: and $0xfffffff0,%esp`
  - %esp = %esp & 0xFFFFFFF0
  - make zero the least 4-bits of %esp
  - new %esp : the 16-byte alignment of %esp
- `7: pushl -0x4(%ecx)`
  - push [%ecx-4]..... (%esp -= 4)
  - push [init_%esp], data where init_%esp points to
  - push once more the return address
    previously, stored at the unaligned init_%esp
    now, also stored at the 16-byte aligned new %esp

(2) Function Prologue

- `a: push %ebp`
  - push %ebp.....(%esp -= 4)
  - push init_%ebp (init_%ebp address)
  - after the aligned %esp, %ebp is stored on the stack
- `b: mov %esp,%ebp`
  - %ebp= %esp
  - new %ebp = %esp (aligned %esp)
  - the base of the new stack frame is the aligned %esp
- `d: push %ecx`
  - push %ecx.....(%esp -= 4)
  - push init_%esp+4 (address init_%esp +4)
  - %ecx points to the unaligned stack top +4
- `e: sub $0x4,%esp`
  - %esp -= 4
  - alloc local var's.....(%esp -= 4)
  - enlarge %esp by 4

(3) Function Call

- `11: call 12 <main+0x12>`
- `16: add $0x4,%esp`
  - %esp += 4
  - dealloc local variabls .....(%esp += 4)
  - shrink %esp by 4
- `19: xor %eax,%eax`
  - %eax= 0
  - return value %eax= 0
  - return zero

(4) Function Epilogue

- `1b: pop %ecx`
  - pop %ecx.....(%esp += 4)
  - pop init_%esp+4 (address init_%esp+4)
  - now %ecx points to the unaligned stack top +4
- `1c: pop %ebp`
  - pop %ebp.....(%esp += 4)
  - pop init_%ebp (init_%ebp address)
  - restore init_%ebp
- `1d: lea -0x4(%ecx),%esp`
  - %esp= %ecx-4 = (init_%esp+4)-4
  - %esp= init_%esp (int_%esp unaligned address)
  - restore init_%esp
- `20: ret`

Assembly code analysis 2. swap

1. swap assembly code analysis
2. swap assembly code analysis
3. swap assembly code analysis
4. swap assembly code analysis
5. swap assembly code analysis

swap assembly code analysis (1)

- readelf -s p
  - 53: 0804a020 4 OBJECT GLOBAL DEFAULT 25 p0
  - 64: 0804a018 8 OBJECT GLOBAL DEFAULT 25 buf
  - 67: 0804a028 4 OBJECT GLOBAL DEFAULT 26 p1

0x804a028	p1
0x804a024	
0x804a020	p0

0x804a01c	buf[1]
0x804a018	buf[0]

swap assembly code analysis (2)

- initialized global variable is in .data
- readelf -x .data p
  - 0x0804a010 00000000 00000000 01000000 02000000
  - 0x0804a020 18a00408

0x804a018	0x00000001
0x804a01c	0x00000002
0x804a020	0x0804a018

0x804a028	p1
0x804a024	
0x804a020	p0 = 0x08094a018

0x804a01c	buf[1] = 2
0x804a018	buf[0] = 1

swap assembly code analysis (3)

- `0: mov 0x0,%eax` ..... 0x804a020 &p0
  - mov M[0x804a020], %eax
  - %eax= M[0x804a020]
  - %eax= p0 (address)
- `5: mov 0x4,%ecx` ..... 0x804a01c buf+1
  - mov M[0x804a01c], %ecx
  - %ecx= M[0x804a01c]
  - %ecx= buf[1] (integer)

- `b: movl $0x4,0x0` ..... 0x804a01c buf+1, 0x804a028 &p1
  - movl $0x804a01c, M[0x804a028]
  - M[0x804a028] = 0x804a01c
  - p1 = buf+1

swap assembly code analysis (4)

- %eax = p0 = 0x0804a018 (address)
- %ecx = buf[1] = 2 (integer)
- p1 = buf + 1 = 0x804a01c (address)

0x804a028	p1 = 0x0804a01c
0x804a024	
0x804a020	p0 = 0x0804a018

0x804a01c	buf[1] = 2
0x804a018	buf[0] = 1

swap assembly code analysis (5)

- swap((%eax), {0x4})
- 15: `mov (%eax),%edx`
  - mov M[M[0x804a020]], %edx
  - %edx= M[M[0x804a020]] = M[0x804a018] = 1
  - tmp = *p0;
- 17: `mov %ecx,(%eax)`
  - mov M[0x804a01c], (%eax)
  - M[M[0x804a020]] = M[0x804a018] = M[0x804a01c] = 2
  - *p0 = buf[1];
- 19: `mov %edx,0x4` ..... `0x804a01c` buf+1
  - mov %edx, M[0x804a01c]
  - M[0x804a01c]= %edx
  - buf[1] = tmp
- 1f: `ret`