Linux命令行參數在棧中的分配

Jex — 2012-12-25

《Professional Assembly Language — Richard•Blum》一書第11章Using Command-Line Parameters一節講到，32位Linux環境下ELF 程序被加載之後，分配的虛擬內存起始字節地址為`0x08048000`，結束字節地址為`0xbfffffff`，如下所示：

`0xbfffffff`	Stack Data
……	
`0x08048000`	Program Code and Data

預期Stack Pointer（`%esp`）初始值也應該是`0xbfffffff`，但由於Linux會在程序初始化前，將一些諸如命令行參數及環境變量等信息放到棧上，所以（可能從下往上看更易讀些）：

`0xc0000000`：棧底

0xbfffffc：NULL（0x00000000）
程序名稱字符串值 *
環境變量字符串值 *
命令行參數字符串值 *
ELF Auxiliary Vectors
NULL（結束envp[]）
環境變量字符串地址列表（envp[]）
NULL（結束argv[]）
命令行參數字符串地址列表（argv[]）
%esp：命令行參數個數（dword argc）

* 指多個asciz類型的字符串

ELF Auxiliary Vectors不是程序所要關心的，可以打開/usr/include/elf.h查看struct Elf32_auxv_t的定義。通過設置環境變量：LD_SHOW_AUXV=1，可以在執行程序前輸出AUXV值。

按上圖所示，下面的代碼應該能夠毫無懸念地輸出命令行參數及環境變量：

```
# File:args.s
# Print command line arguments and environment variables
.section .data
  argcs:.asciz "argc=%d\n"
  argvs:.asciz "argv[%d]=%s\n"
  env_header:.asciz "Environment variables:\n"
  envs:.asciz "%s\n"
```

```
.section .text
.global _start
_start:
  movl %esp,%ebp
  pushl (%ebp)
  pushl $argcs
  call printf # Print argc
  addl $8,%esp
  # Print argv[]
  movl $0,%eax
  addl $4,%ebp
argvloop:
  movl (%ebp,%eax,4),%ecx
  jecxz argvloop_end # NULL ends argv[]
  pushl %ecx # String addr
  pushl %eax
  pushl $argvs
  call printf
  # printf ret value override eax,restore from stack
  movl 4(%esp),%eax
  addl $12,%esp
  inc %eax
jmp argvloop
argvloop_end:
  leal 4(%ebp,%eax,4),%ebp # skip argv[] and NULL
  pushl $env_header
  call printf
  addl $4,%esp
envloop:
  movl (%ebp),%ecx
  jecxz end # NULL ends envp[]
  pushl %ecx
  pushl $envs
  call printf
  addl $8,%esp
  addl $4,%ebp
jmp envloop
```

```
end:
  pushl $0
  call exit
```

但要注意編譯方式，因為使用了C的`printf`函數，所以需要鏈接libc：

```
$ as --gstabs args.s -o args.o
$ ld args-libc.o -o args.bin  --dynamic-linker
/lib/ld-linux.so.2 -lc
$ ./args.bin
```

很明顯，用GAS編譯很麻煩，用GCC則方便很多，只需要一條命令就可以了：`gcc -o args.bin args.s`，GCC會一步做好編譯鏈接工作。不過GCC和GAS有一個區別，GAS將`_start`視作程序執行起點，而GCC則將main當作執行起點。如果要使用GCC編譯，則需要將`.global _start`改成`.global main`：

```
# hello.s
# 使用GCC的Hello,world彙編程序示例
.section .data
  msg:.asciz "Hello,world!\n"
.section .text
.global main
main:
  push $msg
  call printf
  push $0
  call exit
```

《Professional Assembly Language》第四章Creating a Simple Program - Assembling using a compiler一節中介紹了這種方式，不過書中並沒有就這種差別再作深層解釋，給讀者留了一個大坑。為什麼說是大坑呢？其實，GCC所指定的main，文中所

說的Beginning of the program， 並不是真正的Entry Point。如果只是Entry Point名稱的區別的話，你會立即查找到`ld`命令有一個`-e`參數可用於指定Entry Point名稱：

-e entry

--entry=entry

Use entry as the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named entry, the linker will try to parse entry as a number, and use that as the entry address (the number will be interpreted in base 10; you may use a leading 0x for base 16, or a leading 0 for base 8).

事實上main並不是Entry Point名稱！使用GCC編譯，GCC會自己添加上libc的`_start`作為entry point，然後再在執行時調用main。將上面的代碼編譯一下：`gcc -o hello.bin hello.s -gstabs`（參數`-gstabs`是為了生成可用於GDB調試的信息）。可以通過`readelf`命令查看ELF Header：

```
$ readelf -h hello.bin|grep Entry
  Entry point address:               0x8048360
```

地址`0x8048360`才是它的入口地址。通過GDB來驗證一下：

```
$ gdb -q hello.bin
Reading symbols from ./hello.bin...done.
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x08048360 <+0>:     xor    %ebp,%ebp
   0x08048362 <+2>:     pop    %esi
   0x08048363 <+3>:     mov    %esp,%ecx
   0x08048365 <+5>:     and    $0xfffffff0,%esp
   0x08048368 <+8>:     push   %eax
   0x08048369 <+9>:     push   %esp
```

```
   0x0804836a <+10>:    push   %edx
   0x0804836b <+11>:    push   $0x80484a0
   0x08048370 <+16>:    push   $0x8048430
   0x08048375 <+21>:    push   %ecx
   0x08048376 <+22>:    push   %esi
   0x08048377 <+23>:    push   $0x8048414
   0x0804837c <+28>:    call   0x8048350 <__libc_start_main@plt>
(gdb) disassemble main
Dump of assembler code for function main:
   $0x8048414 <+0>:     push   $0x804a018
   0x08048419 <+5>:     call   0x8048320 <printf@plt>
   0x0804841e <+10>:    push   $0x0
   0x08048420 <+12>:    call   0x8048340 <exit@plt>
(gdb)
```

可以看到，`main`部分確實是我們寫的代碼，但程序執行，卻是從`_start`開始的。GCC加載的libc中的_start代碼，會將main的地址作為參數（`push $0x8048414`）傳給`__libc_start_main`，`__libc_start_main`執行作了很多準備操作後再去調用`main`。這會帶來什麼問題呢？事實上，如果我們將上面的`args.s`的_start改成main，然後用GCC編譯的話，就會取不到正確的命令行參數。看到main這個名稱，我們一定會聯想到C語言的main函數，事實上這兩者幾乎是等價的，可以通過查看`main.c`生成的彙編代碼驗證這點：

```
#include <stdio.h>
int main(int argc,char *argv[],char *envp[]) {
  printf("Hello,world!\n");
}
```

執行`gcc -S main.c`以生成`main.s`文件：

```
.file "main.c"
  .section  .rodata
.LC0:
```

```
    .string "Hello,world!"
    .text
    .globl  main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
# 省略部分內容
```

可以看到，C中的main函數也是被轉換成Assembly中的`.global main`，那麼看一下C中main函數的聲明：`int main(int argc,char *argv[],char *envp[]);`，就會明白，最終在調用main時，棧中已經變成這樣了：

`%esp+12`：環境變量字符串數組的指針（*envp[]）

`%esp+8`：命令行參數字符串數組的指針（*argv[]）

`%esp+4`：命令行參數個數（argc）

`%esp`：main返回地址

* 注意，棧頂第一個元素不是argc而是main返回地址。

因為main也是通過call指令調用，而call指令會將返回地址push進棧！

再通過GDB驗證一下：

* 注意是從彙編代碼編譯，而不是直接從C代碼編譯。因為從C代碼編譯，生成的調試信息中main標籤的地址，是main函數第一行代碼的地址，通過gdb調試時，`break main`將在main的第一行代碼處停止，而在這之前，已經運行

過了`push %ebp`、`sub $0x64,%esp`（在棧上分配局部變量空間）這些Function Prologue指令了，所以此時棧的結構會受main函數的局部變量大小及其它因素的影響。

```
$ gcc hello.s -o hello.bin -gstabs
$ gdb -q hello.bin
Reading symbols from ./hello.bin...done.
(gdb) break main
Breakpoint 1 at 0x8048414: file hello.s, line 8.
(gdb) run 1 2 3
Starting program: ./hello.bin 1 2 3

Breakpoint 1, main () at hello.s:8
8           push $hello
(gdb) x/4wx $esp # 依次為：返回地址，argc,*argv[],*envp[]
0xbffff17c:     0x001704d3      0x00000004      0xbffff214
0xbffff228
(gdb) x/4wx 0xbffff214  # View argv[]，4個命令行參數
0xbffff214:     0xbffff435      0xbffff45a      0xbffff45c
0xbffff45e
(gdb) x/s 0xbffff45a    # 第二個命令行參數，第一個是程序名稱
0xbffff45a:      "1"
(gdb) x/2wx 0xbffff228 # View envp[]
0xbffff228:     0xbffff460      0xbffff475
(gdb) x/s 0xbffff460  # 第一個環境變量值
0xbffff460:      "LC_PAPER=zh_CN.UTF-8"
查看一下main函數返回地址0x001704d3前後的代碼
(gdb) disassemble 0x001704d3-16,+20
Dump of assembler code from 0x1704c3 to 0x1704d3:
   0x001704c3 <__libc_start_main+227>:  add    $0x89,%al
   0x001704c5 <__libc_start_main+229>:  inc    %esp
   0x001704c6 <__libc_start_main+230>:  and    $0x8,%al
   0x001704c8 <__libc_start_main+232>:  mov    0x74(%esp),%eax
   0x001704cc <__libc_start_main+236>:  mov    %eax,(%esp)
   這一行調用main，main的地址放在0x70(%esp)中
   0x001704cf <__libc_start_main+239>:  call   *0x70(%esp)
   0x001704d3 <__libc_start_main+243>:  mov    %eax,(%esp)
   0x001704d6 <__libc_start_main+246>:  call   0x189fb0
<__GI_exit>
```

查看 0x70(%esp) 單元的值

```
(gdb) x/1wx $esp+4+0x70    # call指令push了返回地址，所以到main執行時，這裡要再加4
0xbffff1f0:     0x8048414   # 正是main的地址
```

所以上面獲取命令行參數的彙編代碼，若想用GCC編譯還能正確運行，則需要改成這樣：

```
.section .data
  argcs:.asciz "argc=%d\n"
  argvs:.asciz "argv[%d]=%s\n"
  env_header:.asciz "Current environment variables:\n"
  envs:.asciz "%s\n"
.section .text
.global main
main:
  movl %esp,%ebp
  pushl 4(%ebp) # Skip ret addr
  pushl $argcs
  call printf # Print argc
  addl $8,%esp
  # Print argv[]
  movl $0,%eax
  movl 8(%esp),%ebp # Skip argc and ret addr
argvloop:
  movl (%ebp,%eax,4),%ecx
  jecxz argvloop_end # NULL ends argv[]
  pushl %ecx # String addr
  pushl %eax
  pushl $argvs
  call printf
  # prinf ret value override eax,restore from stack
  movl 4(%esp),%eax
  addl $12,%esp
  inc %eax
jmp argvloop
```

```
argvloop_end:
  movl 12(%esp),%ebp # Skip argc,ret,argv
  pushl $env_header
  call printf
  addl $4,%esp
envloop:
  movl (%ebp),%ecx
  jecxz end # NULL ends envp[]
  pushl %ecx
  pushl $envs
  call printf
  addl $8,%esp
  addl $4,%ebp
jmp envloop
end:
  pushl $0
  call exit
```

故事還沒有結束，事實上，GCC也可以通過參數，指定不加載libc的startfiles，這樣就和GAS編譯效果一樣了：

```
# hello.s 中仍然聲明 .global _start
$ gcc hello.s -o hello.bin -gstabs -nostartfiles
```

-nostartfiles

Do not use the standard system startup files when linking.

The standard system libraries are used normally,unless -nostdlib or -nodefaultlibs is used.

參考資料

- How main() is executed on Linux
- Startup state of a Linux/i386 ELF binary
- Hackers Hut：ELF
- About ELF Auxiliary Vectors
- 不使用libc，輸出命令行參數及環境變量的彙編代碼：
- https://github.com/JexCheng/anthology/blob/master/asm/args.s