前面写过一篇北极之北之main函数之前，这篇文章其实解决的问题是main函数并不是第一个执行的函数，在main之前，函数的入口点是\_start, \_start会调用glibc里的\_\_libc\_start\_main，main函数只是这个函数的入参。 在\_\_libc\_start\_main中某一步，会执行main函数。这是上面一篇博文获取到的知识。

但是我还是强力推荐。http://www.acsu.buffalo.edu/~charngda/elf.html。

#include <stdio.h>

#include <stdlib.h>

void preinit(int argc, char \*\*argv, char \*\*envp) {

printf("%s\n", \_\_FUNCTION\_\_);

}

void init(int argc, char \*\*argv, char \*\*envp) {

printf("%s\n", \_\_FUNCTION\_\_);

}

void fini() {

printf("%s\n", \_\_FUNCTION\_\_);

}

\_\_attribute\_\_((section(".init\_array"))) typeof(init) \*\_\_init = init;

\_\_attribute\_\_((section(".preinit\_array"))) typeof(preinit) \*\_\_preinit = preinit;

\_\_attribute\_\_((section(".fini\_array"))) typeof(fini) \*\_\_fini = fini;

void \_\_attribute\_\_ ((constructor)) constructor() {

printf("%s\n", \_\_FUNCTION\_\_);

}

void \_\_attribute\_\_ ((constructor)) constructor\_2() {

printf("%s\n", \_\_FUNCTION\_\_);

}

void \_\_attribute\_\_ ((destructor)) destructor() {

printf("%s\n", \_\_FUNCTION\_\_);

}

void \_\_attribute\_\_ ((destructor)) destructor\_2() {

printf("%s\n", \_\_FUNCTION\_\_);

}

void my\_atexit() {

printf("%s\n", \_\_FUNCTION\_\_);

}

void my\_atexit2() {

printf("%s\n", \_\_FUNCTION\_\_);

}

int main() {

atexit(my\_atexit);

atexit(my\_atexit2);

printf("%s\n",\_\_FUNCTION\_\_);

}

我们写了好多的函数，我们定义了preinit\_array,init\_array, fini\_array这些段，我们也定义了多个construct和destructor，还用atexit注册了两个函数，这是广撒英雄帖，把能召唤的兄弟们都聚集齐了。看执行结果：

root@manu:~/code/c/self/initfini# ./test

preinit

init

constructor\_2

constructor

main

my\_atexit2

my\_atexit

destructor

destructor\_2

fini

为什么会这样？问题的答案在libc里面。好在我们有glibc的代码，这也不算啥，我们先分析下main函数之前执行的动作，以及他们的先后顺序。

我的glibc的版本号是：

root@manu:~/code/c/self/initfini# ldd test

linux-gate.so.1 => (0xb773c000)

libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7578000)

/lib/ld-linux.so.2 (0xb773d000)

root@manu:~/code/c/self/initfini# ll /lib/i386-linux-gnu/libc.so.6

lrwxrwxrwx 1 root root 12 10月 6 04:39 /lib/i386-linux-gnu/libc.so.6 -> libc-2.15.so\*

我去网上下载了一份glibc的2.15版本的code,准备工作就绪，我们开始调试和研究。

(gdb) b preinit

Breakpoint 1 at 0x804843a: file test.c, line 5.

(gdb) r

Starting program: /home/manu/code/c/self/initfini/test

Breakpoint 1, preinit (argc=1, argv=0xbffff774, envp=0xbffff77c) at test.c:5

5 printf("%s\n", \_\_FUNCTION\_\_);

(gdb) bt

#0 preinit (argc=1, argv=0xbffff774, envp=0xbffff77c) at test.c:5

#1 0xb7fecfd2 in \_dl\_init (main\_map=0xb7fff918, argc=1, argv=0xbffff774, env=0xbffff77c) at dl-init.c:119

#2 0xb7fdf20f in \_dl\_start\_user () from /lib/ld-linux.so.2

\_dl\_start\_user调用了\_dl\_init.我们看下\_dl\_init的函数定义：

void

internal\_function

\_dl\_init (struct link\_map \*main\_map, int argc, char \*\*argv, char \*\*env)

{

ElfW(Dyn) \*preinit\_array = main\_map->l\_info[DT\_PREINIT\_ARRAY];

ElfW(Dyn) \*preinit\_array\_size = main\_map->l\_info[DT\_PREINIT\_ARRAYSZ];

unsigned int i;

if (\_\_builtin\_expect (GL(dl\_initfirst) != NULL, 0))

{

call\_init (GL(dl\_initfirst), argc, argv, env);

GL(dl\_initfirst) = NULL;

}

/\* Don't do anything if there is no preinit array. \*/

if (\_\_builtin\_expect (preinit\_array != NULL, 0)

&& preinit\_array\_size != NULL

&& (i = preinit\_array\_size->d\_un.d\_val / sizeof (ElfW(Addr))) > 0)

{

ElfW(Addr) \*addrs;

unsigned int cnt;

if (\_\_builtin\_expect (GLRO(dl\_debug\_mask) & DL\_DEBUG\_IMPCALLS, 0))

\_dl\_debug\_printf ("\ncalling preinit: %s\n\n",

main\_map->l\_name[0]

? main\_map->l\_name : rtld\_progname);

addrs = (ElfW(Addr) \*) (preinit\_array->d\_un.d\_ptr + main\_map->l\_addr);

for (cnt = 0; cnt < i; ++cnt)

((init\_t) addrs[cnt]) (argc, argv, env);

}

/\* Stupid users forced the ELF specification to be changed. It now

says that the dynamic loader is responsible for determining the

order in which the constructors have to run. The constructors

for all dependencies of an object must run before the constructor

for the object itself. Circular dependencies are left unspecified.

This is highly questionable since it puts the burden on the dynamic

loader which has to find the dependencies at runtime instead of

letting the user do it right. Stupidity \*/

i = main\_map->l\_searchlist.r\_nlist;

while (i-- > 0)

call\_init (main\_map->l\_initfini[i], argc, argv, env);

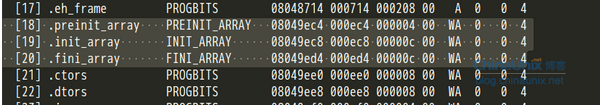
#ifndef HAVE\_INLINED\_SYSCALLS

/\* Finished starting up. \*/

INTUSE(\_dl\_starting\_up) = 0;

#endif

}

 下面我调试了下，

注意DT\_PREINIT\_ARRAY = 32 ， DT\_PREINIT\_ARRAYSZ=33，这是定义在头文件的宏。

(gdb) p main\_map->l\_info[32]

$5 = (Elf32\_Dyn \*) 0x8049f0c

$7 = {d\_tag = 0x20, d\_un = {d\_val = 0x8049ec4, d\_ptr = 0x8049ec4}}

(gdb) p/x \*(main\_map->l\_info[33])

$8 = {d\_tag = 0x21, d\_un = {d\_val = 0x4, d\_ptr = 0x4}}

(gdb) x/4x 0x08049ec4

0x8049ec4 <\_\_preinit>: 0x08048434 0x08048448 0x08048484 0x08048470

我们可以看到这段代码就是查找preinit段的信息，并执行对应的函数。0x8049ec4和我们调用readelf -S获取的preinit的值是一样的。 最后我们得到了这个段对应的function的地址，也就是代码做的事情。

addrs = (ElfW(Addr) \*) (preinit\_array->d\_un.d\_ptr + main\_map->l\_addr);

for (cnt = 0; cnt < i; ++cnt)

((init\_t) addrs[cnt]) (argc, argv, env);

通过计算我们得到0x8049ec4存储的就是function的地址0x08048434。

08048434 ::

8048434: 55 push %ebp

8048435: 89 e5 mov %esp,%ebp

8048437: 83 ec 18 sub $0x18,%esp

804843a: c7 04 24 86 86 04 08 movl $0x8048686,(%esp)

8048441: e8 0a ff ff ff call 8048350

8048446: c9 leave

8048447: c3 ret

OK ，我们preinit段对应的代码就分析结束了，值得一提的是，此时,我们的入口点\_start还没有执行，我可以证明下：

(gdb) b \_start

Breakpoint 2 at 0x8048380

(gdb) b init

Breakpoint 3 at 0x804844e: init. (5 locations)

(gdb) b \_\_libc\_start\_main

Breakpoint 4 at 0xb7e323e0: file libc-start.c, line 96.

(gdb) c

Continuing.

preinit

Breakpoint 2, 0x08048380 in \_start (）

(gdb) c

Continuing.

Breakpoint 4, \_\_libc\_start\_main (main=0x80484e8

, argc=1, ubp\_av=0xbffff774, init=0x8048520 <\_\_libc\_csu\_init>,

fini=0x8048590 <\_\_libc\_csu\_fini>, rtld\_fini=0xb7fed270 <\_dl\_fini>, stack\_end=0xbffff76c) at libc-start.c:96

96 libc-start.c: 没有那个文件或目录.

(gdb) c

Continuing.

Breakpoint 3, init (argc=1, argv=0xbffff774, envp=0xbffff77c) at test.c:9

9 printf("%s\n", \_\_FUNCTION\_\_);

(gdb) bt

#0 init (argc=1, argv=0xbffff774, envp=0xbffff77c) at test.c:9

#1 0x08048572 in \_\_libc\_csu\_init ()

#2 0xb7e3246a in \_\_libc\_start\_main (main=0x80484e8

, argc=1, ubp\_av=0xbffff774, init=0x8048520 <\_\_libc\_csu\_init>,

fini=0x8048590 <\_\_libc\_csu\_fini>, rtld\_fini=0xb7fed270 <\_dl\_fini>, stack\_end=0xbffff76c) at libc-start.c:185

#3 0x080483a1 in \_start ()

\_start->\_\_libc\_start\_main->\_\_libc\_csu\_init->init，脉络是这样的，OK ，我们先看下\_\_libc\_csu\_init.

void

\_\_libc\_csu\_init (int argc, char \*\*argv, char \*\*envp)

{

/\* For dynamically linked executables the preinit array is executed by

the dynamic linker (before initializing any shared object). \*/

#ifndef LIBC\_NONSHARED

/\* For static executables, preinit happens right before init. \*/

{

const size\_t size = \_\_preinit\_array\_end - \_\_preinit\_array\_start;

size\_t i;

for (i = 0; i < size; i++)

(\*\_\_preinit\_array\_start [i]) (argc, argv, envp);

}

#endif

\_init ();

const size\_t size = \_\_init\_array\_end - \_\_init\_array\_start;

for (size\_t i = 0; i < size; i++)

(\*\_\_init\_array\_start [i]) (argc, argv, envp);

}

\_init中会执行construct的代码， \_\_init\_array\_start[i] 处执行init\_array段的函数，所以我们可以看出执行顺序如下：

Function pointers in .preinit\_array section ，before \_start

Functions marked as \_\_attribute\_\_ ((constructor)), via \_init

Function pointers in .init\_array section

\_init函数的是定义在 /sysdeps/unix/sysv/linux/init-first.c，最后的最后，执行了\_libc\_global\_ctors ，这就是我们说的marked as \_\_attribute\_\_ ((constructor)) 的函数。

OK ，main函数之前的事情都了了，本来把退出一起写了，但是文章太长了，而且太晚了就写这些吧。

还是有很多东西不懂，没办法，一口吃不成胖子，心急吃不了热豆腐，还是得慢慢来。

参考文献

1 www.acsu.buffalo.edu/~charngda/elf.html/

2 程序员的自我修养