**Glibc系统调用实现分析**

注：本分析内容只对函数调用进行跟踪，并不关注代码内部的实现细节，本文以open系统调用为例，跟踪Glibc代码生成open系统调用的汇编代码。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 版本 | 修改内容简介 | 修改日期 | 修改人 |
| 1 | V1.0 | 完成Glibc系统调用分析 | 2016-03-02 | 李小龙 |

[1、 带缓冲的fopen函数 2](#_Toc527317508)

[2、 无缓冲的open函数 8](#_Toc1346997539)

1. **带缓冲的fopen函数**

Glibc中fopen()函数的实现是对open()函数的封装，下述代码跟踪了从fopen()函数是如何进入内核态，或者最后是不是调用了不带缓冲的open()函数。

Include/stdio.h

fopen()函数采用宏定义的方式定义。

122 extern \_IO\_FILE \*\_IO\_new\_fopen (const char\*, const char\*);

123 # define fopen(fname, mode) \_IO\_new\_fopen (fname, mode)

libio/iofopen.c

94 \_IO\_FILE \*

95 \_IO\_new\_fopen (const char \*filename, const char \*mode)

96 {

97 return \_\_fopen\_internal (filename, mode, 1);

98 }

59 \_IO\_FILE \*

60 \_\_fopen\_internal (const char \*filename, const char \*mode, int is32)

61 {

62 struct locked\_FILE

63 {

64 struct \_IO\_FILE\_plus fp;

65 #ifdef \_IO\_MTSAFE\_IO

66 \_IO\_lock\_t lock;

67 #endif

68 struct \_IO\_wide\_data wd;

69 } \*new\_f = (struct locked\_FILE \*) malloc (sizeof (struct locked\_FILE)) ;

71 if (new\_f == NULL)

72 return NULL;

73 #ifdef \_IO\_MTSAFE\_IO

74 new\_f->fp.file.\_lock = &new\_f->lock;

75 #endif

76 #if defined \_LIBC || defined \_GLIBCPP\_USE\_WCHAR\_T

77 \_IO\_no\_init (&new\_f->fp.file, 0, 0, &new\_f->wd, &\_IO\_wfile\_jumps);

78 #else

79 \_IO\_no\_init (&new\_f->fp.file, 1, 0, NULL, NULL);

80 #endif

81 \_IO\_JUMPS (&new\_f->fp) = &\_IO\_file\_jumps;

82 \_IO\_file\_init (&new\_f->fp);

83 #if !\_IO\_UNIFIED\_JUMPTABLES

84 new\_f->fp.vtable = NULL;

85 #endif

86 if (\_IO\_file\_fopen ((\_IO\_FILE \*) new\_f, filename, mode, is32) != NULL)

87 return \_\_fopen\_maybe\_mmap (&new\_f->fp.file);

88

89 \_IO\_un\_link (&new\_f->fp);

90 free (new\_f);

91 return NULL;

}

69 # define \_IO\_new\_file\_fopen \_IO\_file\_fopen

246 \_IO\_FILE \*

247 \_IO\_new\_file\_fopen (\_IO\_FILE \*fp, const char \*filename, const char \*mode,

248 int is32not64)

249 {

250 int oflags = 0, omode;

251 int read\_write;

252 int oprot = 0666;

253 int i;

254 \_IO\_FILE \*result;

255 #ifdef \_LIBC

256 const char \*cs;

257 const char \*last\_recognized;

258 #endif

259

260 if (\_IO\_file\_is\_open (fp))

261 return 0;

262 switch (\*mode)

263 {

264 case 'r':

265 omode = O\_RDONLY;

266 read\_write = \_IO\_NO\_WRITES;

267 break;

268 case 'w':

269 omode = O\_WRONLY;

270 oflags = O\_CREAT|O\_TRUNC;

271 read\_write = \_IO\_NO\_READS;

272 break;

273 case 'a':

274 omode = O\_WRONLY;

275 oflags = O\_CREAT|O\_APPEND;

276 read\_write = \_IO\_NO\_READS|\_IO\_IS\_APPENDING;

277 break;

278 default:

279 \_\_set\_errno (EINVAL);

280 return NULL;

281 }

282 #ifdef \_LIBC

283 last\_recognized = mode;

284 #endif

285 for (i = 1; i < 7; ++i)

286 {

287 switch (\*++mode)

288 {

289 case '\0':

290 break;

291 case '+':

292 omode = O\_RDWR;

293 read\_write &= \_IO\_IS\_APPENDING;

294 #ifdef \_LIBC

295 last\_recognized = mode;

296 #endif

297 continue;

298 case 'x':

299 oflags |= O\_EXCL;

300 #ifdef \_LIBC

301 last\_recognized = mode;

302 #endif

303 continue;

304 case 'b':

305 #ifdef \_LIBC

306 last\_recognized = mode;

307 #endif

308 continue;

309 case 'm':

310 fp->\_flags2 |= \_IO\_FLAGS2\_MMAP;

311 continue;

312 case 'c':

313 fp->\_flags2 |= \_IO\_FLAGS2\_NOTCANCEL;

314 continue;

315 case 'e':

316 #ifdef O\_CLOEXEC

317 oflags |= O\_CLOEXEC;

318 #endif

319 fp->\_flags2 |= \_IO\_FLAGS2\_CLOEXEC;

320 continue;

321 default:

322 /\* Ignore. \*/

323 continue;

324 }

325 break;

326 }

327

328 result = \_IO\_file\_open (fp, filename, omode|oflags, oprot, read\_write,

329 is32not64);

330

331 if (result != NULL)

332 {

333 #ifndef \_\_ASSUME\_O\_CLOEXEC

334 if ((fp->\_flags2 & \_IO\_FLAGS2\_CLOEXEC) != 0 && \_\_have\_o\_cloexec <= 0)

335 {

336 int fd = \_IO\_fileno (fp);

337 if (\_\_have\_o\_cloexec == 0)

338 {

339 int flags = \_\_fcntl (fd, F\_GETFD);

340 \_\_have\_o\_cloexec = (flags & FD\_CLOEXEC) == 0 ? -1 : 1;

341 }

342 if (\_\_have\_o\_cloexec < 0)

343 \_\_fcntl (fd, F\_SETFD, FD\_CLOEXEC);

344 }

345 #endif

346

347 /\* Test whether the mode string specifies the conversion. \*/

348 cs = strstr (last\_recognized + 1, ",ccs=");

349 if (cs != NULL)

350 {

351 /\* Yep. Load the appropriate conversions and set the orientation

352 to wide. \*/

353 struct gconv\_fcts fcts;

354 struct \_IO\_codecvt \*cc;

355 char \*endp = \_\_strchrnul (cs + 5, ',');

356 char \*ccs = malloc (endp - (cs + 5) + 3);

357

358 if (ccs == NULL)

359 {

360 int malloc\_err = errno; /\* Whatever malloc failed with. \*/

361 (void) \_IO\_file\_close\_it (fp);

362 \_\_set\_errno (malloc\_err);

363 return NULL;

364 }

365

366 \*((char \*) \_\_mempcpy (ccs, cs + 5, endp - (cs + 5))) = '\0';

367 strip (ccs, ccs);

368

369 if (\_\_wcsmbs\_named\_conv (&fcts, ccs[2] == '\0'

370 ? upstr (ccs, cs + 5) : ccs) != 0)

371 {

372 /\* Something went wrong, we cannot load the conversion modules.

373 This means we cannot proceed since the user explicitly asked

374 for these. \*/

375 (void) \_IO\_file\_close\_it (fp);

376 free (ccs);

377 \_\_set\_errno (EINVAL);

378 return NULL;

379 }

380

381 free (ccs);

382

383 assert (fcts.towc\_nsteps == 1);

384 assert (fcts.tomb\_nsteps == 1);

385

386 fp->\_wide\_data->\_IO\_read\_ptr = fp->\_wide\_data->\_IO\_read\_end;

387 fp->\_wide\_data->\_IO\_write\_ptr = fp->\_wide\_data->\_IO\_write\_base;

388

389 /\* Clear the state. We start all over again. \*/

390 memset (&fp->\_wide\_data->\_IO\_state, '\0', sizeof (\_\_mbstate\_t));

391 memset (&fp->\_wide\_data->\_IO\_last\_state, '\0', sizeof (\_\_mbstate\_t));

392

393 cc = fp->\_codecvt = &fp->\_wide\_data->\_codecvt;

394

395 /\* The functions are always the same. \*/

396 \*cc = \_\_libio\_codecvt;

397

398 cc->\_\_cd\_in.\_\_cd.\_\_nsteps = fcts.towc\_nsteps;

399 cc->\_\_cd\_in.\_\_cd.\_\_steps = fcts.towc;

400

401 cc->\_\_cd\_in.\_\_cd.\_\_data[0].\_\_invocation\_counter = 0;

402 cc->\_\_cd\_in.\_\_cd.\_\_data[0].\_\_internal\_use = 1;

403 cc->\_\_cd\_in.\_\_cd.\_\_data[0].\_\_flags = \_\_GCONV\_IS\_LAST;

404 cc->\_\_cd\_in.\_\_cd.\_\_data[0].\_\_statep = &result->\_wide\_data->\_IO\_state;

405

406 cc->\_\_cd\_out.\_\_cd.\_\_nsteps = fcts.tomb\_nsteps;

407 cc->\_\_cd\_out.\_\_cd.\_\_steps = fcts.tomb;

408

409 cc->\_\_cd\_out.\_\_cd.\_\_data[0].\_\_invocation\_counter = 0;

410 cc->\_\_cd\_out.\_\_cd.\_\_data[0].\_\_internal\_use = 1;

411 cc->\_\_cd\_out.\_\_cd.\_\_data[0].\_\_flags

412 = \_\_GCONV\_IS\_LAST | \_\_GCONV\_TRANSLIT;

413 cc->\_\_cd\_out.\_\_cd.\_\_data[0].\_\_statep =

414 &result->\_wide\_data->\_IO\_state;

415

416 /\* From now on use the wide character callback functions. \*/

417 \_IO\_JUMPS\_FILE\_plus (fp) = fp->\_wide\_data->\_wide\_vtable;

418

419 /\* Set the mode now. \*/

420 result->\_mode = 1;

421 }

422 }

423

424 return result;

425 }

211 \_IO\_FILE \*

212 \_IO\_file\_open (\_IO\_FILE \*fp, const char \*filename, int posix\_mode, int prot,

213 int read\_write, int is32not64)

214 {

215 int fdesc;

216 #ifdef \_LIBC

217 if (\_\_glibc\_unlikely (fp->\_flags2 & \_IO\_FLAGS2\_NOTCANCEL))

218 fdesc = open\_not\_cancel (filename,

219 posix\_mode | (is32not64 ? 0 : O\_LARGEFILE), prot);

220 else

221 fdesc = open (filename, posix\_mode | (is32not64 ? 0 : O\_LARGEFILE), prot);

222 #else

223 fdesc = open (filename, posix\_mode, prot);

224 #endif

225 if (fdesc < 0)

226 return NULL;

227 fp->\_fileno = fdesc;

228 \_IO\_mask\_flags (fp, read\_write,\_IO\_NO\_READS+\_IO\_NO\_WRITES+\_IO\_IS\_APPENDING);

229 /\* For append mode, send the file offset to the end of the file. Don't

230 update the offset cache though, since the file handle is not active. \*/

231 if ((read\_write & (\_IO\_IS\_APPENDING | \_IO\_NO\_READS))

232 == (\_IO\_IS\_APPENDING | \_IO\_NO\_READS))

233 {

234 \_IO\_off64\_t new\_pos = \_IO\_SYSSEEK (fp, 0, \_IO\_seek\_end);

235 if (new\_pos == \_IO\_pos\_BAD && errno != ESPIPE)

236 {

237 close\_not\_cancel (fdesc);

238 return NULL;

239 }

240 }

241 \_IO\_link\_in ((struct \_IO\_FILE\_plus \*) fp);

242 return fp;

243 }

从上述代码中可以看出fopen()函数的实现依赖open()函数。

1. **无缓冲的open函数**

glibc的代码在涉及系统调用这块时相当难懂，因为里面涉及了大量的宏定义，而且有些代码是在编译过程中动态生成的，所以不容易找到。下面部分将叙述glibc2.22中open系统调用函数的实现。

注：网上有很多分析open系统调用的帖子，但是大多数都没有详细介绍glibc源代码是怎么实现open系统调用的，并且部分帖子的叙述是完全错误的。

由于我们要采用调试功能跟踪，自带的glibc库编译时应该没有用-g选项，所以用-g选项编译一个库，把编译过程重定向到一个文件（以备后用）。

下载地址：<https://www.gnu.org/software/libc/index.html>

文档：所有从网上找来的说明当然没有官方文档直接，

这里重点引用一篇：<https://sourceware.org/glibc/wiki/SyscallWrappers>

编译过程：要注意的是glibc编译不能在，源码目录进行，并且应当设置一个新的安装路径，以免和已有库 冲突，造成系统出现问题。

1、解压。

2、创建一个新目录，mkdir glibc\_bulid && cd glibc\_build

3、config，../glib-version/configure --perfix=/usr/glib #指定一个不同的安装路径

4、编译 make > makelog.log #保存编译输出，后面要用到

5、安装，sudo make install

1、open函数调用接口的定义。在glibc中大量使用了函数或者变量别名的方法，使glibc对外提供的接口是唯一的，例如用户只需要知道open()函数，但glibc库中的实现函数可能是open()也可能是open64()。具体实现我也不太清楚，应该是通过宏定义和函数weak别名现的。

io/fcntl.h

178 #ifndef \_\_USE\_FILE\_OFFSET64

179 extern int open (const char \*\_\_file, int \_\_oflag, ...) \_\_nonnull ((1));

180 #else

181 # ifdef \_\_REDIRECT

182 extern int \_\_REDIRECT (open, (const char \*\_\_file, int \_\_oflag, ...), open64)//在汇编

183 \_\_nonnull ((1));

184 # else

185 # define open open64

186 # endif

187 #endif

2、为了进行验证，我们写一个简单的程序进行验证。

test.c/

1 #include <fcntl.h>

2

3 int main(){

4 int fd;

5 fd=open("t.c",O\_RDONLY,0);

6 return 0;

7 }

采用-g编译:gcc -g test.c -o test

1. 采用gdb进行跟踪。

$ gdb test

(gdb) break main

Breakpoint 1 at 0x400e06: file t.c, line 5.

(gdb) list

1 #include <fcntl.h>

2

3 int main(){

4 int fd;

5 fd=open("t.c",O\_RDONLY,0);

6 return 0;

7 }

(gdb) break open

Breakpoint 2 at 0x436450

(gdb) r

Starting program: /home/fly/t

Breakpoint 1, main () at t.c:5

5 fd=open("t.c",O\_RDONLY,0);

(gdb) disassemble

Dump of assembler code for function main:

0x0000000000400dfe <+0>: push %rbp

0x0000000000400dff <+1>: mov %rsp,%rbp

0x0000000000400e02 <+4>: sub $0x10,%rsp

=> 0x0000000000400e06 <+8>: mov $0x0,%edx

0x0000000000400e0b <+13>: mov $0x0,%esi

0x0000000000400e10 <+18>: mov $0x497504,%edi

0x0000000000400e15 <+23>: mov $0x0,%eax

0x0000000000400e1a <+28>: callq 0x436450 <open64>

0x0000000000400e1f <+33>: mov %eax,-0x4(%rbp)

0x0000000000400e22 <+36>: mov $0x0,%eax

0x0000000000400e27 <+41>: leaveq

0x0000000000400e28 <+42>: retq

End of assembler dump.

(gdb)

查看红色的哪行可以发现，我们调用的open函数，在汇编代码中成了open64。

再断点运行到open函数查看汇编代码：

(gdb) c

Continuing.

Breakpoint 2, 0x0000000000436450 in open64 ()

(gdb) disassemble

Dump of assembler code for function open64:

=> 0x0000000000436450 <+0>: cmpl $0x0,0x28f8a5(%rip) # 0x6c5cfc <\_\_libc\_multiple\_threads>

0x0000000000436457 <+7>: jne 0x43646d <open64+29>

0x0000000000436459 <+0>: mov $0x2,%eax

0x000000000043645e <+5>: syscall

0x0000000000436460 <+7>: cmp $0xfffffffffffff001,%rax

0x0000000000436466 <+13>: jae 0x43ab10 <\_\_syscall\_error>

0x000000000043646c <+19>: retq

0x000000000043646d <+29>: sub $0x8,%rsp

0x0000000000436471 <+33>: callq 0x4393b0 <\_\_libc\_enable\_asynccancel>

0x0000000000436476 <+38>: mov %rax,(%rsp)

0x000000000043647a <+42>: mov $0x2,%eax

0x000000000043647f <+47>: syscall

0x0000000000436481 <+49>: mov (%rsp),%rdi

0x0000000000436485 <+53>: mov %rax,%rdx

0x0000000000436488 <+56>: callq 0x439410 <\_\_libc\_disable\_asynccancel>

0x000000000043648d <+61>: mov %rdx,%rax

0x0000000000436490 <+64>: add $0x8,%rsp

0x0000000000436494 <+68>: cmp $0xfffffffffffff001,%rax

0x000000000043649a <+74>: jae 0x43ab10 <\_\_syscall\_error>

0x00000000004364a0 <+80>: retq

End of assembler dump.

1. 接下来我们分析glibc的源代码，查看其是怎么实现open()系统调用的。

分析代码当然有代码浏览工具最好了，可惜在linux上sourceinsight不容易使用，不过vim的插件完全可以满足我们的需求了。

我只安装了下面几个插件：cscope，ctag，taglist。关于他们的安装和使用方法，网上一搜一大堆，在此不做介绍。其实如果习惯了vim的操作，这些工具的使用真的很简单，虽然vim入门比较难，但是一旦熟练了之后，就能成倍提高工作效率，而且希望即使是在office和浏览器中都拜托鼠标操作。

一切准备就绪之后，打开vim在vim中执行：

*:cs find g open*

可以看到下面一些定义。

Cscope tag: open

# 行 文件名 / 上下文 / 行

1 471 intl/loadmsgcat.c <<open>>

#define open(name, flags) open\_not\_cancel\_2 (name, flags)

2 41 io/bits/fcntl2.h <<open>>

open (const char \*\_\_path, int \_\_oflag, ...)

3 179 io/fcntl.h <<open>>

extern int open (const char \*\_...\_\_oflag, ...) \_\_nonnull ((1));

4 185 io/fcntl.h <<open>>

#define open open64

5 60 libio/fileops.c <<open>>

#define open(Name, Flags, Prot) \_\_open (Name, Flags, Prot)

6 53 libio/oldfileops.c <<open>>

#define open(Name, Flags, Prot) \_\_open (Name, Flags, Prot)

7 146 /usr/include/fcntl.h <<open>>

extern int open (const char \*\_...\_\_oflag, ...) \_\_nonnull ((1));

8 152 /usr/include/fcntl.h <<open>>

#define open open64

键入数字和回车(empty cancels)：

我们可以看出3和7是函数声明，其中3是在源代码中，7是在系统头文件中（因为没有使用cscope -k选项），其它东西显然都不是函数的实现，那么奇了怪了，open或者open函数的实现究竟在哪里。

在经历了千辛万苦后终于又回到了原点，发现自己真的忽略了一个问题：官方说明永远是我们的第一手资料。在上面提到的文档中就有对glibc实现的系统调用函数有较为详细的介绍，文中提到glibc实现系统调用主要有三种方式： assembly, macro, and bespoke.我们在此只看第一种方式，因为open,write,read,close等大多数系统调用都是采用第一种方式实现的。下面贴上原文介绍assembly方式的一段话：

Simple kernel system calls in glibc are translated from a list of names into an assembly wrapper that is then compiled.

In a build directory disassemble the write syscall and you'll see the syscall-template.S wrapper:

大概意思是：简单的系统调用是通过许多宏定义名字翻译成一个嵌入式汇编而实现的，反汇编系统调用（write）代码时，可以看到里面有syscall-template.S这个文件名称，从名字我们猜测这是一个实现系统调用的模板。

进入刚才编译glibc的目录glibc\_build。指向下面命令反汇编：

*$ objdump -ldr ./io/open.o*

Disassembly of section .text:

0000000000000000 <\_\_libc\_open>:

\_\_libc\_open():

/home/fly/linux\_kernel/glibc-2.22/io/../sysdeps/unix/syscall-template.S:84

0: 83 3d 00 00 00 00 00 cmpl $0x0,0x0(%rip) # 7 <\_\_libc\_open+0x7>

2: R\_X86\_64\_PC32 \_\_libc\_multiple\_threads-0x5

7: 75 14 jne 1d <\_\_open\_nocancel+0x14>

0000000000000009 <\_\_open\_nocancel>:

9: b8 02 00 00 00 mov $0x2,%eax

e: 0f 05 syscall

10: 48 3d 01 f0 ff ff cmp $0xfffffffffffff001,%rax

16: 0f 83 00 00 00 00 jae 1c <\_\_open\_nocancel+0x13>

18: R\_X86\_64\_PC32 \_\_syscall\_error-0x4

1c: c3 retq

1d: 48 83 ec 08 sub $0x8,%rsp

21: e8 00 00 00 00 callq 26 <\_\_open\_nocancel+0x1d>

22: R\_X86\_64\_PC32 \_\_libc\_enable\_asynccancel-0x4

26: 48 89 04 24 mov %rax,(%rsp)

2a: b8 02 00 00 00 mov $0x2,%eax

2f: 0f 05 syscall

31: 48 8b 3c 24 mov (%rsp),%rdi

35: 48 89 c2 mov %rax,%rdx

38: e8 00 00 00 00 callq 3d <\_\_open\_nocancel+0x34>

39: R\_X86\_64\_PC32 \_\_libc\_disable\_asynccancel-0x4

3d: 48 89 d0 mov %rdx,%rax

40: 48 83 c4 08 add $0x8,%rsp

44: 48 3d 01 f0 ff ff cmp $0xfffffffffffff001,%rax

4a: 0f 83 00 00 00 00 jae 50 <\_\_open\_nocancel+0x47>

4c: R\_X86\_64\_PC32 \_\_syscall\_error-0x4

/home/fly/linux\_kernel/glibc-2.22/io/../sysdeps/unix/syscall-template.S:85

50: c3 retq

可以在上述输出中看到：syscall-template.S后面跟着行号，并且汇编码的内部符号是\_\_libc\_open。

在glibc源码中搜索syscall-template.S文件，在sysdeps/unix/目录下，打开查看上述出现的行。

sysdeps/unix/syscall-template.S

81 /\* This is a "normal" system call stub: if there is an error,

82 it returns -1 and sets errno. \*/

83

84 T\_PSEUDO (SYSCALL\_SYMBOL, SYSCALL\_NAME, SYSCALL\_NARGS)

85 ret

86 T\_PSEUDO\_END (SYSCALL\_SYMBOL)

T\_PSEUDO和T\_PSEUDO\_END 是宏定义：

53 #define T\_PSEUDO(SYMBOL, NAME, N) PSEUDO (SYMBOL, NAME, N)

56 #define T\_PSEUDO\_END(SYMBOL) PSEUDO\_END (SYMBOL)

在sysdeps/unix/x86\_64/sysdeps.h下找到一个PSEUDO宏定义，来看看是不是。

sysdeps/unix/x86\_64/sysdeps.h

27 /\* The code to disable cancellation depends on the fact that the called

28 functions are special. They don't modify registers other than %rax

29 and %r11 if they return. Therefore we don't have to preserve other

30 registers around these calls. \*/

31 # undef PSEUDO

32 # define PSEUDO(name, syscall\_name, args) \

33 .text; \

34 ENTRY (name) \

35 SINGLE\_THREAD\_P; \

36 jne L(pseudo\_cancel); \

37 .type \_\_##syscall\_name##\_nocancel,@function; \

38 .globl \_\_##syscall\_name##\_nocancel; \

39 \_\_##syscall\_name##\_nocancel: \

40 DO\_CALL (syscall\_name, args); \

41 cmpq $-4095, %rax; \

42 jae SYSCALL\_ERROR\_LABEL; \

43 ret; \

44 .size \_\_##syscall\_name##\_nocancel,.-\_\_##syscall\_name##\_nocancel; \

45 L(pseudo\_cancel): \

46 /\* We always have to align the stack before calling a function. \*/ \

47 subq $8, %rsp; cfi\_adjust\_cfa\_offset (8); \

48 CENABLE \

49 /\* The return value from CENABLE is argument for CDISABLE. \*/ \

50 movq %rax, (%rsp); \

51 DO\_CALL (syscall\_name, args); \

52 movq (%rsp), %rdi; \

53 /\* Save %rax since it's the error code from the syscall. \*/ \

54 movq %rax, %rdx; \

55 CDISABLE

56 movq %rdx, %rax; \

57 addq $8,%rsp; cfi\_adjust\_cfa\_offset (-8); \

58 cmpq $-4095, %rax; \

59 jae SYSCALL\_ERROR\_LABEL

\

看着和反汇编后的代码很像，接着找到DO\_CALL的宏定义。

sysdeps/unix/sysv/linux/x86\_64/sysdep.h

176 # define DO\_CALL(syscall\_name, args) \

177 DOARGS\_##args \

178 movl $SYS\_ify (syscall\_name), %eax; \

179 syscall;

34 #define SYS\_ify(syscall\_name) \_\_NR\_##syscall\_name

看到上面这句是不是感觉豁然开朗，\_\_NR##syscall\_name不正是内核中的系统调用号的宏定义方法么。

例如：open系统调用

#define \_\_NR\_open 2

1. 系统调用涉及到的文件我们已经知道了，从前文我们知道，open系统调用是在glibc编译过程中动态生成的，我们来看看大体过程。

查看我们保存的glibc的编译输出，其中有很多下面类似的输出：

makelog.log

(echo '#define SYSCALL\_NAME open'; \

echo '#define SYSCALL\_NARGS 3'; \

echo '#define SYSCALL\_SYMBOL \_\_libc\_open'; \

echo '#define SYSCALL\_CANCELLABLE 1'; \

echo '#define SYSCALL\_NOERRNO 0'; \

echo '#define SYSCALL\_ERRVAL 0'; \

echo '#include <syscall-template.S>'; \

echo 'weak\_alias (\_\_libc\_open, \_\_open)'; \

echo 'hidden\_weak (\_\_open)'; \

echo 'weak\_alias (\_\_libc\_open, open)'; \

echo 'hidden\_weak (open)'; \

echo 'weak\_alias (\_\_libc\_open, \_\_open64)'; \

echo 'hidden\_weak (\_\_open64)'; \

echo 'weak\_alias (\_\_libc\_open, open64)'; \

echo 'hidden\_weak (open64)'; \

) | gcc -c -U\_FORTIFY\_SOURCE -I../include -I/home/fly/linux\_kernel/makeglibc/io -I/home/fly/linux\_kernel/makeglibc -I../sysdeps/unix/sysv/linux/x86\_64/64 -I../sysdeps/unix/sysv/linux/x86\_64 -I../sysdeps/unix/sysv/linux/x86 -I../sysdeps/unix/sysv/linux/wordsize-64 -I../sysdeps/x86\_64/nptl -I../sysdeps/unix/sysv/linux/include -I../sysdeps/unix/sysv/linux -I../sysdeps/nptl -I../sysdeps/pthread -I../sysdeps/gnu -I../sysdeps/unix/inet -I../sysdeps/unix/sysv -I../sysdeps/unix/x86\_64 -I../sysdeps/unix -I../sysdeps/posix -I../sysdeps/x86\_64/64 -I../sysdeps/x86\_64/fpu/multiarch -I../sysdeps/x86\_64/fpu -I../sysdeps/x86/fpu/include -I../sysdeps/x86/fpu -I../sysdeps/x86\_64/multiarch -I../sysdeps/x86\_64 -I../sysdeps/x86 -I../sysdeps/ieee754/ldbl-96 -I../sysdeps/ieee754/dbl-64/wordsize-64 -I../sysdeps/ieee754/dbl-64 -I../sysdeps/ieee754/flt-32 -I../sysdeps/wordsize-64 -I../sysdeps/ieee754 -I../sysdeps/generic -I.. -I../libio -I. -D\_LIBC\_REENTRANT -include /home/fly/linux\_kernel/makeglibc/libc-modules.h -DMODULE\_NAME=libc -include ../include/libc-symbols.h -DASSEMBLER -g -Werror=undef -Wa,--noexecstack -o /home/fly/linux\_kernel/makeglibc/io/open.o -x assembler-with-cpp - -MD -MP -MF /home/fly/linux\_kernel/makeglibc/io/open.o.dt -MT /home/fly/linux\_kernel/makeglibc/io/open.o

看到上述输出，我们可以知道，在编译glibc的过程中，相当于建立了每个系统调用的临时文件，同过管道命令将其传给gcc进行编译生成对应的系统调用函数的二进制目标文件。这么说来也是，所有系统调用的实现无非就是传递系统调用号和参数，采用这种方式提高了代码重用，不必为每一个系统调用写一个实现。

阅读参考文档了解到，在sysdeps/unix/Makefile文件中实现为每个系统调用生成嵌入式汇编代码。

sysdeps/unix/Makefil

81 $(common-objpfx)sysd-syscalls: $(..)sysdeps/unix/make-syscalls.sh \

82 $(wildcard $(+sysdep\_dirs:%=%/syscalls.list)) \

83 $(common-objpfx)libc-modules.stmp

84 for dir in $(+sysdep\_dirs); do \

85 test -f $$dir/syscalls.list && \

86 { sysdirs='$(sysdirs)' \

87 asm\_CPP='$(COMPILE.S) -E -x assembler-with-cpp' \

88 $(SHELL) $(dir $<)$(notdir $<) $$dir || exit 1; }; \

89 test $$dir = $(..)sysdeps/unix && break; \

90 done > $@T

91 mv -f $@T $@

92 endif

上述makefile脚本具体怎么做的我没有分析，不过从主要和make-syscall.sh和syscalls.list文件有关，查看syscalls.list文件，其中的内容应该和下面这些宏定义有关：

sysdeps/unix/syscall-template.S

27 SYSCALL\_NAME syscall name

28 SYSCALL\_NARGS number of arguments this call takes

29 SYSCALL\_SYMBOL primary symbol name

30 SYSCALL\_CANCELLABLE 1 if the call is a cancelation point

31 SYSCALL\_NOERRNO 1 to define a no-errno version (see below)

32 SYSCALL\_ERRVAL 1 to define an error-value version (see below)

下面两个应该是设置变量别名，和gcc的\_\_attibute\_\_属性有关，大概意思是用在嵌入式汇编中，导出函数在外部调用的接口。

weak\_alias (\_\_libc\_open, \_\_open);

hidden\_weak (\_\_open);

参考：

1. <https://sourceware.org/glibc/wiki/SyscallWrappers>
2. <http://blog.csdn.net/astrotycoon/article/details/8008629>
3. <http://12000.org/my_notes/system_calls_in_linux/system_call_in_linux.htm>