Hello how to world

本文拟从三个方面来说明一个 helloworld 可执行文件如何运行：

1. 从 GCC 编译器的角度，来分析一个 helloworld 可执行文件的形成。
2. 从程序调用角度来说明一个 helloworld 可执行文件的运行。
3. 从程序运行时堆栈空间的使用来进一步加深一个 helloworld 的运行机制。

How do GCC do for the helloworld

我们的测试环境为：

```
<onlyforos>[/home/onlyforos/hello]%gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix
--disable-checking --host=i386-redhat-linux --with-system-zlib
--enable-__cxa_atexit
Thread model: posix
gcc version 3.2 20020903 (Red Hat Linux 8.0 3.2-7)
```

同样先看一下我们熟悉的 Helloworld 程序：

```
<onlyforos>[/home/onlyforos/hello]%cat hello.c
#include <stdio.h>
int main()
{
    printf("hello,world\n");
    return 0;
}

<onlyforos>[/home/onlyforos/hello]%gcc -o hello hello.c
<onlyforos>[/home/onlyforos/hello]%hello
hello,world
```

这个 hello 的可执行文件到底是怎么生成的呢？理论上简单地说来，GCC 编译器将 ASCII 码源文件翻译成可执行目标文件时，首先运行 C 预处理器(cpp)，它将 C 的源程序 main.c 翻译成一个 ASCII 码的中间文件 hello.i：

```
cpp [other arguments] hello.c /tmp/hello.i
```

接下来，运行 C 编译器(cc1)，它将 hello.i 翻译成一个 ASCII 汇编语言文件为 hello.s。

```
cc1 [other arguments] –o /tmp/hello.s /tmp/hello.i
```

然后，运行汇编器(as)，它将 hello.s 翻译成一个可重定位目标文件 hello.o：

```
as [] –o /tmp/hello.o /tmp/hello.s
```

最后，运行链接器程序 ld，将 hello.o 与一些必要的系统目标文件组合起来，创建一个可执

行的目标文件 hello：

ld –o hello /tmp/hello.o

那么，实际中 GCC 是怎么处理的呢？GCC 选项-v 会显示编译的所有过程，我们来分析一下：

```
<onlyforos>[/home/onlyforos/hello]%gcc -v -o he hello.c
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix
--disable-checking --host=i386-redhat-linux --with-system-zlib
--enable-__cxa_atexit
Thread model: posix
gcc version 3.2 20020903 (Red Hat Linux 8.0 3.2-7)
 /usr/lib/gcc-lib/i386-redhat-linux/3.2/cc1 -lang-c -v -D__GNUC__=3
 -D__GNUC_MINOR__=2 -D__GNUC_PATCHLEVEL__=0 -D__GXX_ABI_VERSION=102
 -D__ELF__ -Dunix -D__gnu_linux__ -Dlinux -D__ELF__ -D__unix__
 -D__gnu_linux__ -D__linux__ -D__unix -D__linux -Asystem=posix
 -D__NO_INLINE__ -D__STDC_HOSTED__=1 -Acpu=i386 -Amachine=i386 -Di386
 -D__i386 -D__i386__ -D__tune_i386__ hello.c -quiet -dumpbase hello.c
 -version -o /tmp/ccT5vlcX.s
 GNU CPP version 3.2 20020903 (Red Hat Linux 8.0 3.2-7) (cpplib) (i386
 Linux/ELF)GNU C version 3.2 20020903 (Red Hat Linux 8.0 3.2-7)
 (i386-redhat-linux)
 compiled by GNU C version 3.2 20020903 (Red Hat Linux 8.0 3.2-7).
ignoring nonexistent directory
"/usr/i386-redhat-linux/include"
#include "..." search starts here:
#include <...> search starts here:
/usr/local/include
/usr/lib/gcc-lib/i386-redhat-linux/3.2/include
/usr/include
End of search list.
as -V -Qy -o /tmp/ccrwadwa.o /tmp/ccT5vlcX.s
GNU assembler version 2.13.90.0.2 (i386-redhat-linux)
using BFD version 2.13.90.0.2 20020802
/usr/lib/gcc-lib/i386-redhat-linux/3.2/collect2
--eh-frame-hdr -m elf_i386 -dynamic-linker
/lib/ld-linux.so.2 -o he
/usr/lib/gcc-lib/i386-redhat-linux/3.2/../../../crt1.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2/../../../crti.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2/crtbegin.o
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2/../../..
/tmp/ccrwadwa.o -lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/lib/gcc-lib/i386-redhat-linux/3.2/crtend.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2/../../../crtn.o
```

可以看到，编译过程并没有象我们前面所说，首先是 CPP 预处理，而是直接调用 CC1 进行编译了。难道不进行预处理了吗？这肯定是不能的。那么，让我们用 cpp –v 命令执行一下，对比上面输出，就可以发现 cpp0 的处理和浅色部分的处理是一致的。由此可判定，应该是新版本的 GCC 把 CPP 处理功能集成到 CC1 中了。

```
<onlyforos>[/home/onlyforos/hello]%echo 'main(){printf("hello world\n");}' | cpp -v -
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix
--disable-checking --host=i386-redhat-linux --with-system-zlib
```

```
--enable-__cxa_atexit
Thread model: posix
gcc version 3.2 20020903 (Red Hat Linux 8.0 3.2-7)
/usr/lib/gcc-lib/i386-redhat-linux/3.2/cpp0 -lang-c -v -D__ELF__
-Dunix -D__gnu_linux__ -Dlinux -D__ELF__ -D__unix__ -D__gnu_linux__
-D__linux__ -D__unix -D__linux -Asystem=posix -D__NO_INLINE__
-D__STDC_HOSTED__=1 -Acpu=i386 -Amachine=i386 -Di386 -D__i386
-D__i386__ -D__tune_i386__ -
GNU CPP version 3.2 20020903 (Red Hat Linux 8.0 3.2-7) (cpplib) (i386
Linux/ELF)ignoring nonexistent directory
"/usr/i386-redhat-linux/include"
#include "..." search starts here:
#include <...> search starts here:
/usr/local/include
/usr/lib/gcc-lib/i386-redhat-linux/3.2/include
/usr/include
End of search list.
<onlyforos>[/home/onlyforos/hello]%cat cpp.txt
# 1 "<stdin>"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "<stdin>"
main(){printf("hello world\n");}
```

从中我们可以看到 GCC 的版本，预定义的宏(使用 cpp –dM 选项可列出所有预定义的有效宏)，及标准头文件搜索路径等。也可以使用 GCC 的-E(只进行预处理工作)选项来查看预处理后的结果

```
<onlyforos>[/home/onlyforos/hello]%gcc -E -o hello_cpp.txt hello.c
<onlyforos>[/home/onlyforos/hello]%wc -l hello_cpp.txt
929 hello_cpp.txt
```

几乎有 929 行的输出！其中大多数来自 stdio.h 头文件。如果你不用 -o 选项指定输出文件名的话，它就输出到控制台。预编译过程通过完成三个主要任务给了代码很大的灵活性。

1. 把"include"的文件拷贝到要编译的源文件中。
2. 用实际值替代"define"的文本。
3. 在调用宏的地方进行宏替换。

这就使你能够在整个源文件中使用符号常量，而符号常量是在一个地方定义的，如果它的值发生了变化，所有使用符号常量的地方都能自动更新。

完成预处理后，接下来的处理就是 cc1 把代码翻译成汇编语言，此时如有什么语法错误，编译器也会指出后错误退出。接下来 as 把汇编代码转换为目标代码。在 as manual 中找到-V 选项只是针对 VAX 版本(？)的，意味着将该信息保存在内存中而不是磁盘文件中，不过通常 as 都是这么处理的，所以该选项也是多余的。至于-Qy 选项没在文档中找到，不知是什么意思。反正是汇编生成了个目标文件。然后是 collect2，看着该命令后的那些选项都是 ld 的选项，可为什么不是 ld 而是 collect2，collect2 又是什么呢。源文件在(gcc-3.3.2\gcc\collect2.c)，翻开 gcc manual，可以看到 collect2 最终还是要找到真正的 ld 来完成链接工作，那么 collect2 与 ld 的最大区别是什么呢？collect2 会在开始运行时首先调用一些初始化(构造)函数，如果找到构造函数，它会先创建一个包含它们的临时的.c 文件，编译后把它们一起进行链接。进行这样处理的是__main()函数，在 main 函数前调用，定义在标准的 GCC 库中，所以我们看到编译时要包含’-lgcc’。再往下看，我们还发现一些没见过的目标文件，好奇心稍重一些，就会感到奇怪，我们的目标文件是和一堆什么文件一起链接生

成可执行文件的？crt1.o，crti.o，crtbegin.o，crtend.o，crtn.o，这些是些什么呀，为什么会出现在我们的 helloworld 的可执行文件呢。我们可以在/usr/lib/gcc-lib/i386-redhat-linux/3.2 找到这些文件包括那些 GCC 的库文件：

```
<onlyforos>[/usr/lib/gcc-lib/i386-redhat-linux/3.2]%ls
cc1*            crtbegin.o     include/        libgcj.a        libstdc++.a
cc1obj*         crtbeginS.o    jc1*            libgcj.so@      libstdc++.so@
cc1plus*        crtbeginT.o    jvgenmain*      libgcj.spec     libsupc++.a
collect2*       crtend.o       libgcc.a        libobjc.a       specs
cpp0*           crtendS.o      libgcc_eh.a     libobjc.so@     tradcpp0*
```

那么，这些文件到底是干什么的呀，让我们一一来看看它们的内容。在看这些文件内容之前，先看看我们 Helloworld 目标文件的内容吧。

```
<onlyforos>[/home/onlyforos/hello]%gcc –c hello.c
<onlyforos>[/home/onlyforos/hello]%objdump –d hello.o
hello.o:        file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 e4 f0                and    $0xfffffff0,%esp
   9:   b8 00 00 00 00          mov    $0x0,%eax
   e:   29 c4                   sub    %eax,%esp
  10:   83 ec 0c                sub    $0xc,%esp
  13:   68 00 00 00 00          push   $0x0
  18:   e8 fc ff ff ff          call   19 <main+0x19>
  1d:   83 c4 10                add    $0x10,%esp
  20:   b8 00 00 00 00          mov    $0x0,%eax
  25:   c9                      leave
  26:   c3                      ret
```

看到了文件的内容就是我们写的 main 函数。那么，再看 Helloworld 可执行文件的内容：

```
<onlyforos>[/home/onlyforos/hello]%gcc –o hello hello.c
<onlyforos>[/home/onlyforos/hello]%objdump –d hello

hello:        file format elf32-i386

Disassembly of section .init:

08048230 <_init>:
 8048230:       55                      push   %ebp
 8048231:       89 e5                   mov    %esp,%ebp
 8048233:       83 ec 08                sub    $0x8,%esp
 8048236:       e8 61 00 00 00          call   804829c <call_gmon_start>
 804823b:       90                      nop
 804823c:       e8 bb 00 00 00          call   80482fc <frame_dummy>
 8048241:       e8 0a 01 00 00          call   8048350 <__do_global_ctors_aux>
 8048246:       c9                      leave
 8048247:       c3                      ret
Disassembly of section .plt:

08048248 <.plt>:
 8048248:       ff 35 98 94 04 08       pushl  0x8049498
 804824e:       ff 25 9c 94 04 08       jmp    *0x804949c
```

```
 8048254:      00 00                       add     %al,(%eax)
 8048256:      00 00                       add     %al,(%eax)
 8048258:      ff 25 a0 94 04 08           jmp     *0x80494a0
 804825e:      68 00 00 00 00              push    $0x0
 8048263:      e9 e0 ff ff ff              jmp     8048248 <_init+0x18>
 8048268:      ff 25 a4 94 04 08           jmp     *0x80494a4
 804826e:      68 08 00 00 00              push    $0x8
 8048273:      e9 d0 ff ff ff              jmp     8048248 <_init+0x18>
Disassembly of section .text:

08048278 <_start>:
 8048278:      31 ed                       xor     %ebp,%ebp
 804827a:      5e                          pop     %esi
 804827b:      89 e1                       mov     %esp,%ecx
 804827d:      83 e4 f0                    and     $0xfffffff0,%esp
 8048280:      50                          push    %eax
 8048281:      54                          push    %esp
 8048282:      52                          push    %edx
 8048283:      68 74 83 04 08              push    $0x8048374
 8048288:      68 30 82 04 08              push    $0x8048230
 804828d:      51                          push    %ecx
 804828e:      56                          push    %esi
 804828f:      68 28 83 04 08              push    $0x8048328
 8048294:      e8 bf ff ff ff              call    8048258 <_init+0x28>
 8048299:      f4                          hlt
 804829a:      90                          nop
 804829b:      90                          nop

0804829c <call_gmon_start>:
 804829c:      55                          push    %ebp
 804829d:      89 e5                       mov     %esp,%ebp
 804829f:      53                          push    %ebx
 80482a0:      50                          push    %eax
 80482a1:      e8 00 00 00 00              call    80482a6 <call_gmon_start+0xa>
 80482a6:      5b                          pop     %ebx
 80482a7:      81 c3 ee 11 00 00           add     $0x11ee,%ebx
 80482ad:      8b 83 14 00 00 00           mov     0x14(%ebx),%eax
 80482b3:      85 c0                       test    %eax,%eax
 80482b5:      74 02                       je      80482b9 <call_gmon_start+0x1d>
 80482b7:      ff d0                       call    *%eax
 80482b9:      8b 5d fc                    mov     0xfffffffc(%ebp),%ebx
 80482bc:      c9                          leave
 80482bd:      c3                          ret
 80482be:      90                          nop
 80482bf:      90                          nop

080482c0 <__do_global_dtors_aux>:
 80482c0:      55                          push    %ebp
 80482c1:      89 e5                       mov     %esp,%ebp
 80482c3:      83 ec 08                    sub     $0x8,%esp
 80482c6:      80 3d ac 94 04 08 00        cmpb    $0x0,0x80494ac
 80482cd:      75 29                       jne     80482f8 <__do_global_dtors_aux+0x38>
 80482cf:      a1 b0 93 04 08              mov     0x80493b0,%eax
 80482d4:      8b 10                       mov     (%eax),%edx
 80482d6:      85 d2                       test    %edx,%edx
 80482d8:      74 17                       je      80482f1 <__do_global_dtors_aux+0x31>
 80482da:      89 f6                       mov     %esi,%esi
```

```
 80482dc:    83 c0 04                    add    $0x4,%eax
 80482df:    a3 b0 93 04 08              mov    %eax,0x80493b0
 80482e4:    ff d2                       call   *%edx
 80482e6:    a1 b0 93 04 08              mov    0x80493b0,%eax
 80482eb:    8b 10                       mov    (%eax),%edx
 80482ed:    85 d2                       test   %edx,%edx
 80482ef:    75 eb                       jne    80482dc <__do_global_dtors_aux+0x1c>
 80482f1:    c6 05 ac 94 04 08 01        movb   $0x1,0x80494ac
 80482f8:    c9                          leave
 80482f9:    c3                          ret
 80482fa:    89 f6                       mov    %esi,%esi

080482fc <frame_dummy>:
 80482fc:    55                          push   %ebp
 80482fd:    89 e5                       mov    %esp,%ebp
 80482ff:83 ec 08                        sub    $0x8,%esp
 8048302:    a1 90 94 04 08              mov    0x8049490,%eax
 8048307:    85 c0                       test   %eax,%eax
 8048309:    74 19                       je     8048324 <frame_dummy+0x28>
 804830b:    b8 00 00 00 00              mov    $0x0,%eax
 8048310:    85 c0                       test   %eax,%eax
 8048312:    74 10                       je     8048324 <frame_dummy+0x28>
 8048314:    83 ec 0c                    sub    $0xc,%esp
 8048317:    68 90 94 04 08              push   $0x8049490
 804831c:    e8 df 7c fb f7              call   0 <_init-0x8048230>
 8048321:    83 c4 10                    add    $0x10,%esp
 8048324:    c9                          leave
 8048325:    c3                          ret
 8048326:    90                          nop
 8048327:    90                          nop

08048328 <main>:
 8048328:    55                          push   %ebp
 8048329:    89 e5                       mov    %esp,%ebp
 804832b:    83 ec 08                    sub    $0x8,%esp
 804832e:    83 e4 f0                    and    $0xfffffff0,%esp
 8048331:    b8 00 00 00 00              mov    $0x0,%eax
 8048336:    29 c4                       sub    %eax,%esp
 8048338:    83 ec 0c                    sub    $0xc,%esp
 804833b:    68 98 83 04 08              push   $0x8048398
 8048340:    e8 23 ff ff ff              call   8048268 <_init+0x38>
 8048345:    83 c4 10                    add    $0x10,%esp
 8048348:    b8 00 00 00 00              mov    $0x0,%eax
 804834d:    c9                          leave
 804834e:    c3                          ret
 804834f:    90                          nop

08048350 <__do_global_ctors_aux>:
 8048350:    55                          push   %ebp
 8048351:    89 e5                       mov    %esp,%ebp
 8048353:    53                          push   %ebx
 8048354:    52                          push   %edx
 8048355:    a1 80 94 04 08              mov    0x8049480,%eax
 804835a:    83 f8 ff                    cmp    $0xffffffff,%eax
 804835d:    bb 80 94 04 08              mov    $0x8049480,%ebx
 8048362:    74 0c                       je     8048370 <__do_global_ctors_aux+0x20>
 8048364:    83 eb 04                    sub    $0x4,%ebx
```

```
 8048367:    ff d0                       call   *%eax
 8048369:    8b 03                       mov    (%ebx),%eax
 804836b:    83 f8 ff                    cmp    $0xffffffff,%eax
 804836e:    75 f4                       jne    8048364 <__do_global_ctors_aux+0x14>
 8048370:    58                          pop    %eax
 8048371:    5b                          pop    %ebx
 8048372:    c9                          leave
 8048373:    c3                          ret
Disassembly of section .fini:

08048374 <_fini>:
 8048374:    55                          push   %ebp
 8048375:    89 e5                       mov    %esp,%ebp
 8048377:    53                          push   %ebx
 8048378:    52                          push   %edx
 8048379:    e8 00 00 00 00              call   804837e <_fini+0xa>
 804837e:    5b                          pop    %ebx
 804837f:    81 c3 16 11 00 00           add    $0x1116,%ebx
 8048385:    90                          nop
 8048386:    e8 35 ff ff ff              call   80482c0 <__do_global_dtors_aux>
 804838b:    8b 5d fc                    mov    0xfffffffc(%ebp),%ebx
 804838e:    c9                          leave
 804838f:    c3                          ret
```

呵呵，我们就写了个 main 函数，可执行文件却出来这么一大堆从来没见过的东东，哪儿来的啊？

回顾《hello~,the ELF of the world》附录 B 中的 section table 中，此处用到了几个其中比较特殊的 section：.ctors, .dtors, .init, .fini，前二者总会被加载到数据段中，而后二者则总会被加载到文本段中。它们的含义是：

.ctors

该 section 保存着程序的全局的构造函数的指针数组。

.dtors

该 section 保存着程序的全局的析构函数的指针数组。

.fini

该 section 保存着进程终止代码指令。因此，当一个程序正常退出时，系统安排执行这个 section 的中的代码。

.init

该 section 保存着可执行指令，它构成了进程的初始化代码。因此，当一个程序开始运行时，在 main 函数被调用之前，系统安排执行这个 section 的中的代码。

.init 和.fini sections 的存在有着特别的目的。假如一个函数放到.init section，在 main 函数执行前系统就会执行它。同理，假如一个函数放到.fini section，在 main 函数返回后该函数就会执行。该特性被 C++编译器使用，完成全局的构造和析构函数功能。

此时，我们再回到前面的疑问，来看看和我们目标文件进行链接的那几个文件的内容。

1. crtbegin.o

查看目标文件所有的 section：

```
<onlyforos>[/home/onlyforos]%objdump –D /usr/lib/gcc-lib/i386-readhat-linux/3.2/ crtbegin.o
crtbegin.o:         file format elf32-i386

Disassembly of section .text:
```

```
00000000 <__do_global_dtors_aux>:
    0:   55                      push   %ebp
    1:   89 e5                   mov    %esp,%ebp
    3:   83 ec 08                sub    $0x8,%esp
    6:   80 3d 00 00 00 00 00    cmpb   $0x0,0x0
    d:   75 29                   jne    38 <__do_global_dtors_aux+0x38>
    f:   a1 04 00 00 00          mov    0x4,%eax
   14:   8b 10                   mov    (%eax),%edx
   16:   85 d2                   test   %edx,%edx
   18:   74 17                   je     31 <__do_global_dtors_aux+0x31>
   1a:   89 f6                   mov    %esi,%esi
   1c:   83 c0 04                add    $0x4,%eax
   1f:   a3 04 00 00 00          mov    %eax,0x4
   24:   ff d2                   call   *%edx
   26:   a1 04 00 00 00          mov    0x4,%eax
   2b:   8b 10                   mov    (%eax),%edx
   2d:   85 d2                   test   %edx,%edx
   2f:   75 eb                   jne    1c <__do_global_dtors_aux+0x1c>
   31:   c6 05 00 00 00 00 01    movb   $0x1,0x0
   38:   c9                      leave
   39:   c3                      ret
   3a:   89 f6                   mov    %esi,%esi

0000003c <frame_dummy>:
   3c:   55                      push   %ebp
   3d:   89 e5                   mov    %esp,%ebp
   3f:   83 ec 08                sub    $0x8,%esp
   42:   a1 00 00 00 00          mov    0x0,%eax
   47:   85 c0                   test   %eax,%eax
   49:   74 19                   je     64 <frame_dummy+0x28>
   4b:   b8 00 00 00 00          mov    $0x0,%eax
   50:   85 c0                   test   %eax,%eax
   52:   74 10                   je     64 <frame_dummy+0x28>
   54:   83 ec 0c                sub    $0xc,%esp
   57:   68 00 00 00 00          push   $0x0
   5c:   e8 fc ff ff ff          call   5d <frame_dummy+0x21>
   61:   83 c4 10                add    $0x10,%esp
   64:   c9                      leave
   65:   c3                      ret
Disassembly of section .data:

00000000 <__dso_handle>:
    0:   00 00                   add    %al,(%eax)
      ...

00000004 <p.0>:
    4:   04 00                   add    $0x0,%al
      ...
Disassembly of section .ctors:

00000000 <__CTOR_LIST__>:
    0:   ff                      (bad)
    1:   ff                      (bad)
    2:   ff                      (bad)
    3:   ff                      .byte 0xff
Disassembly of section .dtors:
```

```
00000000 <__DTOR_LIST__>:
    0:    ff                      (bad)
    1:    ff                      (bad)
    2:    ff                      (bad)
    3:    ff                      .byte 0xff
Disassembly of section .eh_frame:
Disassembly of section .jcr:
Disassembly of section .fini:

00000000 <.fini>:
    0:    e8 fc ff ff ff          call   1 <.fini+0x1>
Disassembly of section .init:

00000000 <.init>:
    0:    e8 38 00 00 00          call   3d <frame_dummy+0x1>
```

可见，该目标文件包括几个 section: .text;.data;.ctors;.dtors;.fini;.init。首先看到.text section 中有两个函数__do_global_dtors_aux、frame_dummy。__do_global_dtors_aux 函数,该函数遍历__DTOR_LIST__ 列表，调用列表中的每个析构函数。frame_dummy 不知是不是进行与异常函数帧相关的处理，没找到说明，还待查。.data section 中的那 4 个字节也不知是什么作用了。.fini section 中只有一个__do_global_dtors_aux 函数调用，而.init section 中只有一个 frame_dummy 函数的调用。在.cstors 中标号__CTOR_LIST__指向全局构造函数的指针数组头，在.dtors 中标号__DTOR_LIST__指向全局析构函数的指针数组头。该节中存放全局的构造及析构函数指针地址，以 0xFFFFFFFF 开头，以 0x00000000 结束，中间就是指针地址内容。

```
<onlyforos>[/]%objdump –s –j .ctors /usr/lib/gcc-lib/i386-readhat-linux/3.2/ crtbegin.o
crtbegin.o:         file format elf32-i386

Contents of section .ctors:
 0000 ffffffff                                    ....
<onlyforos>[/]%objdump –s –j .dtors /usr/lib/gcc-lib/i386-readhat-linux/3.2/ crtbegin.o
crtbegin.o:         file format elf32-i386

Contents of section .dtors:
 0000 ffffffff                                    ....
```

2. crtend.o

同样分析一下内容：

```
<onlyforos>[/home/onlyforos]%objdump –D /usr/lib/gcc-lib/i386-readhat-linux/3.2/ crtbegin.o

crtend.o:         file format elf32-i386

Disassembly of section .text:

00000000 <__do_global_ctors_aux>:
    0:    55                      push   %ebp
    1:    89 e5                   mov    %esp,%ebp
    3:    53                      push   %ebx
    4:    52                      push   %edx
    5:    a1 fc ff ff ff          mov    0xfffffffc,%eax
    a:    83 f8 ff                cmp    $0xffffffff,%eax
    d:    bb fc ff ff ff          mov    $0xfffffffc,%ebx
   12:    74 0c                   je     20 <__do_global_ctors_aux+0x20>
   14:    83 eb 04                sub    $0x4,%ebx
   17:    ff d0                   call   *%eax
```

```
  19:    8b 03                    mov    (%ebx),%eax
  1b:    83 f8 ff                 cmp    $0xffffffff,%eax
  1e:    75 f4                    jne    14 <__do_global_ctors_aux+0x14>
  20:    58                       pop    %eax
  21:    5b                       pop    %ebx
  22:    c9                       leave
  23:    c3                       ret
Disassembly of section .data:
Disassembly of section .ctors:

00000000 <__CTOR_END__>:
   0:    00 00                    add    %al,(%eax)
     ...
Disassembly of section .dtors:

00000000 <__DTOR_END__>:
   0:    00 00                    add    %al,(%eax)
     ...
Disassembly of section .eh_frame:

00000000 <__FRAME_END__>:
   0:    00 00                    add    %al,(%eax)
     ...
Disassembly of section .jcr:

00000000 <__JCR_END__>:
   0:    00 00                    add    %al,(%eax)
     ...
Disassembly of section .init:

00000000 <.init>:
   0:    e8 fc ff ff ff           call   1 <.init+0x1>
```

可见，这个目标文件中，也包含这些 section：.text;.ctors;.dtors;.eh_frame;.jcr;.init。其中的.text 中有一个函数__do_global_ctors_aux 调用。在.ctors 中标号__CTOR_END__指向全局构造函数的指针数组尾部。在.dtors 中标号__DTOR_END__指向全局析构函数的指针数组尾部。对比上个目标文件内容可知在.ctors 及.dtors section 中只一个空的数组(以 0xffffffff 开头，以 0x00000000 结尾)。.eh_frame 这一节被 gcc 用来为支持它们的语言存储异常处理函数指针。.jcr section 也不知作些什么，没有深入探询。.init 中只是调用了__do_global_ctors_aux 函数。

3. crti.o

```
<onlyforos>[/home/onlyforos]%objdump –d /usr/lib/crti.o
/usr/lib/crti.o:        file format elf32-i386

Disassembly of section .text:

00000000 <call_gmon_start>:
   0:    55                       push   %ebp
   1:    89 e5                    mov    %esp,%ebp
   3:    53                       push   %ebx
   4:    50                       push   %eax
   5:    e8 00 00 00 00           call   a <call_gmon_start+0xa>
   a:    5b                       pop    %ebx
   b:    81 c3 03 00 00 00        add    $0x3,%ebx
  11:    8b 83 00 00 00 00        mov    0x0(%ebx),%eax
```

```
  17:    85 c0                    test   %eax,%eax
  19:    74 02                    je     1d <call_gmon_start+0x1d>
  1b:    ff d0                    call   *%eax
  1d:    8b 5d fc                 mov    0xfffffffc(%ebp),%ebx
  20:    c9                       leave
  21:    c3                       ret
Disassembly of section .init:

00000000 <_init>:
    0:    55                       push   %ebp
    1:    89 e5                    mov    %esp,%ebp
    3:    83 ec 08                 sub    $0x8,%esp
    6:    e8 fc ff ff ff           call   7 <_init+0x7>
    b:    90                       nop
Disassembly of section .fini:

00000000 <_fini>:
    0:    55                       push   %ebp
    1:    89 e5                    mov    %esp,%ebp
    3:    53                       push   %ebx
    4:    52                       push   %edx
    5:    e8 00 00 00 00           call   a <_fini+0xa>
    a:    5b                       pop    %ebx
    b:    81 c3 03 00 00 00        add    $0x3,%ebx
   11:    90                       nop
```

4. crtn.o

```
<onlyforos>[/home/onlyforos]%objdump –d /usr/lib/ crtn.o
/usr/lib/crtn.o:         file format elf32-i386

Disassembly of section .text:
Disassembly of section .init:

00000000 <.init>:
    0:    c9                       leave
    1:    c3                       ret
Disassembly of section .fini:

00000000 <.fini>:
    0:    8b 5d fc                 mov    0xfffffffc(%ebp),%ebx
    3:    c9                       leave
    4:    c3                       ret
```

5. crt1.o

```
<onlyforos>[/home/onlyforos]%objdump –d /usr/lib/ crt1.o
    /usr/lib/crt1.o:         file format elf32-i386

    Disassembly of section .text:

    00000000 <_start>:
        0:31 ed                  xor    %ebp,%ebp
        2:5e                     pop    %esi
        3:89 e1                  mov    %esp,%ecx
        5:83 e4 f0               and    $0xfffffff0,%esp
        8:50                     push   %eax
        9:54                     push   %esp
```

```
 a:52                    push   %edx
 b:68 00 00 00 00        push   $0x0
10:68 00 00 00 00        push   $0x0
15:51                    push   %ecx
16:56                    push   %esi
17:68 00 00 00 00        push   $0x0
1c:e8 fc ff ff ff        call   1d <_start+0x1d>
21:f4                    hlt
22:90                    nop
23:90                    nop
```

后几个没再罗嗦，因为分析也分析不明白。不过，对比 Helloworld 目标文件及可执行文件内容，我们好象可以恍然般的恍然有悟：原来可执行文件中那些除了 main 函数外的代码都来自这些目标文件的啊。

编译产生可重定位文件时，gcc 把每个全局构造函数挂在__CTOR_LIST 上（通过把指向构造函数的指针放到.ctors section 中）。它也把每个全局析构函挂在__DTOR_LIST 上（通过把指向析构函的指针放到.dtors section 中）。连接时，gcc 在所有重定位文件前处理 crtbegin.o, 在所有重定位文件后处理 crtend.o。另外，crti.o 在 crtbegin.o 之前被处理，crtn.o 在 crtend.o 之后被处理。当产生可执行文件时，连接器 ld 分别的连接所有可重定位文件的 ctors 和.dtors section 到__CTOR_LIST__和__DTOR_LIST__列表中。.init section 由所有的可重定位文件中_init 函数组成。.fini 由_fini 函数组成。运行时，系统将在 main 函数之前执行_init 函数，在 main 函数返回后执行_fini 函数。

总之可见，一个程序的编译还是遵循着“ 预处理->编译->汇编->链接 ”这个顺序的。翻过 GCC 处理这一页，我们重点来讲述站在程序代码角度来看 Helloworld 如何运行。

hello how to world

我们一步一步来，首先还是从 ELF 文件开始分析。

```
<onlyforos>[/home/onlyforos/hello]%readelf –a hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048278
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1672 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
```

```
Size of section headers:           40 (bytes)
Number of section headers:         25
Section header string table index: 24
```

很生气，却没啥后果。sigh~。这些代码或内容在其它编辑器中明明很整齐，可是一拷到 WORD 中，就参差不齐，还得一空格一空格地对齐还对不很齐，总这么对齐实在影响热情，郁闷苦恼地都快写不下去，各位高人谁知道咋回事来个信儿。

接着看吧，从头中可以看出，程序的入口地址为：0x8048278。那么这个地址是什么呢？是 main 函数入口吧？。翻看前面 Helloworld 可执行文件内容发现原来这个地址处是_start 标签。这说明整个可执行文件的入口并不是我们熟知的 main 函数，而是_start 标签。

```
Disassembly of section .text:

08048278 <_start>:
 8048278:    31 ed                   xor    %ebp,%ebp
 804827a:    5e                      pop    %esi
 804827b:    89 e1                   mov    %esp,%ecx
 804827d:    83 e4 f0                and    $0xfffffff0,%esp
 8048280:    50                      push   %eax
 8048281:    54                      push   %esp
 8048282:    52                      push   %edx
 8048283:    68 74 83 04 08          push   $0x8048374
 8048288:    68 30 82 04 08          push   $0x8048230
 804828d:    51                      push   %ecx
 804828e:    56                      push   %esi
 804828f:    68 28 83 04 08          push   $0x8048328
 8048294:    e8 bf ff ff ff          call   8048258 <_init+0x28>
 8048299:    f4                      hlt
 804829a:    90                      nop
 804829b:    90                      nop
```

直接往下看看它在做些什么，好象是准备了一堆参数，然后调用了一个函数在 0x8048258，找到该地址看看是什么？

```
8048258: ff 25 a0 94 04 08              jmp    *0x80494a0
```

跳转到 0x80494a0 地址内容处，是什么？

```
<onlyforos>[/home/onlyforos/hello]%objdump -R hello
hello:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080494a8 R_386_GLOB_DAT    __gmon_start__
080494a0 R_386_JUMP_SLOT    __libc_start_main
080494a4 R_386_JUMP_SLOT    printf
```

嗬嗬，原来是在调用__libc_start_main。是什么？glibc-2.3\sysdeps\generic\libc-start.c:

```
int
/* GKM FIXME: GCC: this should get __BP_ prefix by virtue of the
   BPs in the arglist of startup_info.main and startup_info.init. */
BP_SYM (__libc_start_main) (int (*main) (int, char **, char **),
                            int argc, char *__unbounded *__unbounded ubp_av,
                            void (*init) (void), void (*fini) (void),
                            void (*rtld_fini) (void), void *__unbounded stack_end)
{
  char *__unbounded *__unbounded ubp_ev = &ubp_av[argc + 1];
#if __BOUNDED_POINTERS__
  char **argv;
```

```
#else
# define argv ubp_av
#endif

   /* Result of the 'main' function.    */
   int result;

#ifndef SHARED
# ifdef HAVE_AUX_VECTOR
   void *__unbounded *__unbounded auxvec;
# endif

   __libc_multiple_libcs = &_dl_starting_up && !_dl_starting_up;
#endif

   INIT_ARGV_and_ENVIRON;

   /* Store the lowest stack address.    */
   __libc_stack_end = stack_end;

#ifndef SHARED
# ifdef HAVE_AUX_VECTOR
   /* First process the auxiliary vector since we need to find the
        program header to locate an eventually present PT_TLS entry.    */
   for (auxvec = (void *__unbounded *__unbounded) ubp_ev;
           *auxvec != NULL; ++auxvec);
   ++auxvec;
   _dl_aux_init ((ElfW(auxv_t) *) auxvec);
# endif

   /* Initialize the thread library at least a bit since the libgcc
        functions are using thread functions if these are available and
        we need to setup errno.    If there is no thread library and we
        handle TLS the function is defined in the libc to initialized the
        TLS handling.    */
# if !(USE_TLS - 0)
   if (__pthread_initialize_minimal)
# endif
      __pthread_initialize_minimal ();

   /* Some security at this point.    Prevent starting a SUID binary where
        the standard file descriptors are not opened.    We have to do this
        only for statically linked applications since otherwise the dynamic
        loader did the work already.    */
   if (__builtin_expect (__libc_enable_secure, 0))
      __libc_check_standard_fds ();
#endif

   /* Register the destructor of the dynamic linker if there is any.    */
   if (__builtin_expect (rtld_fini != NULL, 1))
      __cxa_atexit ((void (*) (void *)) rtld_fini, NULL, NULL);

   /* Call the initializer of the libc.    This is only needed here if we
        are compiling for the static library in which case we haven't
        run the constructors in `_dl_start_user'.    */
#ifndef SHARED
   __libc_init_first (argc, argv, __environ);
```

```
#endif

  /* Register the destructor of the program, if any.    */
  if (fini)
    __cxa_atexit ((void (*) (void *)) fini, NULL, NULL);

  /* Call the initializer of the program, if any.    */
#ifdef SHARED
  if (__builtin_expect (GL(dl_debug_mask) & DL_DEBUG_IMPCALLS, 0))
    _dl_debug_printf ("\ninitialize program: %s\n\n", argv[0]);
#endif
  if (init)
    (*init) ();

#ifdef SHARED
  if (__builtin_expect (GL(dl_debug_mask) & DL_DEBUG_IMPCALLS, 0))
    _dl_debug_printf ("\ntransferring control: %s\n\n", argv[0]);
#endif

#ifdef HAVE_CANCELBUF
  if (setjmp (THREAD_SELF->cancelbuf) == 0)
#endif
    {
      /* XXX This is where the try/finally handling must be used.    */

      result = main (argc, argv, __environ);
    }
#ifdef HAVE_CANCELBUF
  else
    /* Not much left to do but to exit the thread, not the process.    */
    __exit_thread (0);
#endif

  exit (result);
}
```

__libc_start_main 代码中注释写得很清楚，参数中 char *__unbounded *__unbounded 中 __unbounded 为宏，定义在 cdefs.h 中，此处为空。整个函数处理流程基本上是注册动态链接器的析构函数，注册程序的析构函数，这些注册的函数将会 exit 中得到调用。然后调用 init()函数，翻翻前面的汇编代码，在此函数中调用 call_gmon_start，frame_dummy，__do_global_ctors_aux 等函数，之后，真正到了调用我们熟知的 main()入口函数!main 函数返回后，调用 exit()真正的退出整个程序。到此时再仔细看看_start 的代码就好理解了：

```
08048278 <_start>:
 8048278:    31 ed                   xor    %ebp,%ebp       //ebp 清 0
 804827a:    5e                      pop    %esi            //argc -> esi
 804827b:    89 e1                   mov    %esp,%ecx       //argv -> ecx
 804827d:    83 e4 f0                and    $0xfffffff0,%esp //esp-16
 8048280:    50                      push   %eax              //保存 eax
 8048281:    54                      push   %esp //__libc_start_main 参数 stack_end
 8048282:    52                      push   %edx        //rtld_fini
 8048283:    68 74 83 04 08          push   $0x8048374    //fini
 8048288:    68 30 82 04 08          push   $0x8048230    //init
 804828d:    51                      push   %ecx      //argv
 804828e:    56                      push   %esi      //argc
 804828f:    68 28 83 04 08          push   $0x8048328 //main
```

```
8048294:    e8 bf ff ff ff          call    8048258 <_init+0x28> //__libc_start_main
8048299:    f4                      hlt
804829a:    90                      nop
804829b:    90                      nop
```

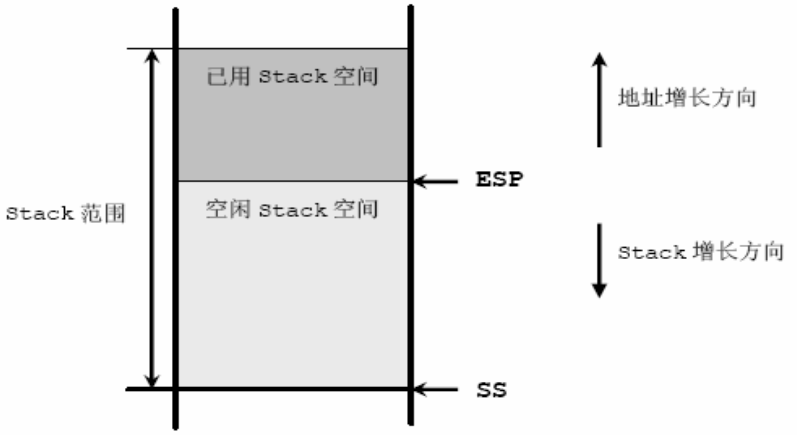
初步和我们的认知接上了轨，接下来充分分析一下 helloworld 运行的程序的栈空间的使用情况，进一步加深对可执行程序运行的认识。

Stack in hello world

文中假设大家已经了解在函数调用时如何使用栈空间。关于栈的使用此处只做简单地介绍，更为详细地说明请见《》，这里重点介绍 helloworld 运行时所有栈空间的内容。关于 AT&T Asm 详细介绍请参见《》，关于 GDB 调试指令，本文也不做更多说明，详细说明请参见 GDB 手册。

Stack

在80386 平台上，Stack 通过两个寄存器来控制，%esp 和%ss，其中%ss 指出Stack 的边界——基地址和Stack Segment 长度，%esp 作为栈指针总是指向栈顶，用来分配空间。



除了上述两个寄存器，与栈操作有关系的寄存器还有%ebp，%ebp 指向Stack 当前的底部，%ebp 始终被用来维护Stack的使用，当一个函数调用结束后，此函数所占用的一切Stack 空间都被释放。所以我们常能看到进入一个函数调用后首先的两条指令就是：

```
push    %ebp
mov     %esp,%ebp
```

作用就是首先将上个栈底(%ebp)压栈，然后将%ebp指向当前的栈底(栈底，指高地址，栈顶指低地址)。底呀顶呀挺迷糊的，反正对于栈有几点需要搞清楚就行了：

1)IA32的栈是用来存放临时数据，而且是LIFO，即后进先出的。栈的增长方向是从高地址向低地址增长，按字节为单位编址。

2) EBP是栈基址的指针，永远指向栈底（高地址），ESP是栈指针，永远指向栈顶（低地址）。

3) 执行push 指令时，%esp 就向下移动，当当%esp 到达%ss 所指向的地址时，Stack 空间就被用尽了，如果继续向其中push 数据，将会引起异常。PUSH一个long型数据时，以字节为单位将数据压入栈，从高到低按字节依次将数据存入ESP-1、ESP-2、ESP-3、ESP-4的地址

单元。

4）执行POP 指令时，%esp 就向上移动。当%esp 所指向的地址等于Stack范围的最大地址时，Stack 完全为空，如果继续进行pop 操作，也会引起异常。POP一个long型数据，过程与PUSH相反，依次将ESP-4、ESP-3、ESP-2、ESP-1从栈内弹出，放入一个32位寄存器。

5）CALL指令用来调用一个函数或过程，此时，下一条指令地址会被压入堆栈，以备返回时能恢复执行下条指令。

6）RET指令用来从一个函数或过程返回，之前CALL保存的下条指令地址会从栈内弹出到EIP寄存器中，程序转到CALL之前下条指令处执行

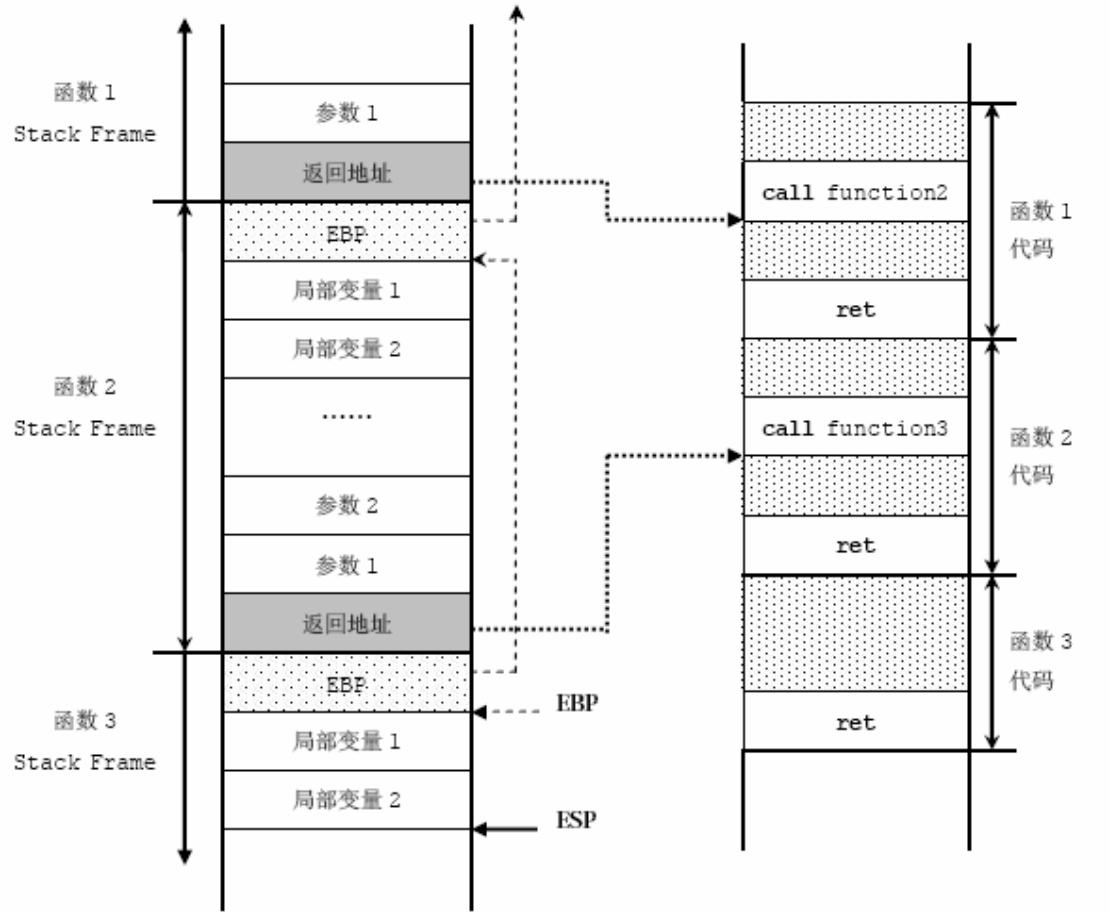
7）ENTER是建立当前函数的栈框架，即相当于以下两条指令：

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

8）LEAVE是释放当前函数或者过程的栈框架，即相当于以下两条指令：

```
movl ebp esp
popl  ebp
```

在函数调用中栈作用如图所示：



从图中，我们可以得知：

- 当一个函数调用发生时，一段Stack 空间将被占用；当从被调用函数返回后，此函数占用的Stack 空间也被释放。这期间被某个函数占用Stack 空间被称作**Stack Frame**。
- 每次进入一个函数，寄存器%ebp 将会被push 到Stack 中，然后%ebp 被修改为指向当前%esp 的值。从而形成一个链接关系。

- 每次当执行一个call 指令后，当前的%eip 内存被压栈，从而保存了函数调用点。当函数执行结束后，ret 指令将当前%esp 所指向的Stack 中的内容pop到%eip 中，使程序可以从函数调用点处继续执行。
- 在执行一条call 指令之前，需要将被调用函数所需参数从后向前依次push 到stack中，在call 指令被执行后，被调用函数将会按照此规则到Stack 中操作这些参数。
- 局部变量所需空间，从当前函数的%ebp 保存点之后的Stack 中进行分配。

stack in hello wrold

我们使用 GDB 调试工具，一步一步按 helloworld 运行顺序来查看堆栈使用情况：

```
[root@Little foros]# gcc -o hello hello.c
[root@Little foros]# gdb -q hello
(gdb) b __libc_start_main
Breakpoint 1 at 0x8048258
(gdb) b main
Breakpoint 2 at 0x804832e
(gdb) r
Starting program: /home/foros/hello
Breakpoint 1 at 0x42015836
Breakpoint 1, 0x42015836 in __libc_start_main () from /lib/i686/libc.so.6
```

首先设置两个断点，一个在函数__libc_start_main 处，一个 main 函数处。然后 run，这时我们看到运行第一个断点在__libc_start_main 函数的 0x42015836 地址处。那么这个地址处是什么内容呢？

```
(gdb) disass __libc_start_main
Dump of assembler code for function __libc_start_main:
0x42015830 <__libc_start_main>:          push   %ebp
0x42015831 <__libc_start_main+1>:        mov    %esp,%ebp
0x42015833 <__libc_start_main+3>:        push   %edi
0x42015834 <__libc_start_main+4>:        push   %esi
0x42015835 <__libc_start_main+5>:        push   %ebx
0x42015836 <__libc_start_main+6>:        call   0x4201575d <__i686.get_pc_thunk.bx>
0x4201583b <__libc_start_main+11>:       add    $0x114a95,%ebx
0x42015841 <__libc_start_main+17>:       sub    $0xc,%esp
0x42015844 <__libc_start_main+20>:       mov    0xc(%ebp),%edx
0x42015847 <__libc_start_main+23>:       mov    0x10(%ebp),%edi
0x4201584a <__libc_start_main+26>:       mov    0x1c(%ebp),%ecx
0x4201584d <__libc_start_main+29>:       mov    0x18(%ebp),%esi
0x42015850 <__libc_start_main+32>:       lea    0x4(%edi,%edx,4),%eax
0x42015854 <__libc_start_main+36>:       mov    0x214(%ebx),%edx
0x4201585a <__libc_start_main+42>:       test   %ecx,%ecx
0x4201585c <__libc_start_main+44>:       mov    %eax,(%edx)
0x4201585e <__libc_start_main+46>:       mov    0x348(%ebx),%eax
0x42015864 <__libc_start_main+52>:       mov    0x20(%ebp),%edx
0x42015867 <__libc_start_main+55>:       mov    %edx,(%eax)
0x42015869 <__libc_start_main+57>:       je     0x42015883 <__libc_start_main+83>
---Type <return> to continue, or q <return> to quit---q
Quit
```

可以看到 0x42015836 之前，__libc_start_main 先将寄存器 ebp,edi,esi,ebx 压栈保存。查看一下此时 esp 的值：

```
(gdb) print $esp
$1 = (void *) 0xbffff9cc
```

```
(gdb) x/13x 0xbffff9cc
0xbffff9cc:     0x400124b8      0x00000001      0x08048278      0x00000000
0xbffff9dc:     0x08048299      0x08048328      0x00000001      0xbffffa04
0xbffff9ec:     0x08048230      0x08048374      0x4000a950      0xbffff9fc
0xbffff9fc:     0x4001020c
```

这时，我们可以看到进入__libc_start_main 函数后栈空间的使用情况。来分析一下，根据上面的代码看起来，栈指针处的第 1 个内容就应该是代码中最后一个压栈的内容，即%ebx(0x400124b8)，%esi(0x00000001)，%edi(0x0848278)，%ebp(0x00000000)。%ebp 内容为 0，也说明这是第一个函数入口。那么随后的 0x8048299 又是什么呢？根据前面简述可知，一个函数的栈中内容为%ebp 的上面就应该是返回地址，即为调用函数处的下一条指令地址。那么，__libc_start_main 是在_start 中调用的，若按这样分析，这个返回地址应该在_start 中。我们来看看_start 代码：

```
(gdb) disass _start
Dump of assembler code for function _start:
0x8048278 <_start>:     xor    %ebp,%ebp
0x804827a <_start+2>:   pop    %esi
0x804827b <_start+3>:   mov    %esp,%ecx
0x804827d <_start+5>:   and    $0xfffffff0,%esp
0x8048280 <_start+8>:   push   %eax
0x8048281 <_start+9>:   push   %esp
0x8048282 <_start+10>:  push   %edx
0x8048283 <_start+11>:  push   $0x8048374
0x8048288 <_start+16>:  push   $0x8048230
0x804828d <_start+21>:  push   %ecx
0x804828e <_start+22>:  push   %esi
0x804828f <_start+23>:  push   $0x8048328
0x8048294 <_start+28>:  call   0x8048258 <__libc_start_main>
0x8048299 <_start+33>:  hlt
0x804829a <_start+34>:  nop
0x804829b <_start+35>:  nop
End of assembler dump.
```

哈哈，果然是调用_libc_start_main 处的下一条指令地址。那么，栈中接下来的那几个地址是什么呢？看到_start 的内容中的 8 个 push，显而易见，对号入座吧。依次为 7 个__libc_start_main 的参数及保存%eax 的内容。具体参数的对应关系，在上面介绍_start 函数含义处的蓝色注释中已经说明了。继续运行程序，接着往下看。

```
(gdb) c
Continuing.
Breakpoint 2, 0x0804832e in main ()
(gdb) print $esp
$2 = (void *) 0xbffff9b0
(gdb) x/20x 0xbffff9b0
0xbffff9b0:     0x4200aec8      0x4212a2d0      0xbffff9d8      0x420158d4
0xbffff9c0:     0x00000001      0xbffffa04      0xbffffa0c      0x400124b8
0xbffff9d0:     0x00000001      0x08048278      0x00000000      0x08048299
0xbffff9e0:     0x08048328      0x00000001      0xbffffa04      0x08048230
0xbffff9f0:     0x08048374      0x4000a950      0xbffff9fc      0x4001020c
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:       push   %ebp
0x8048329 <main+1>:     mov    %esp,%ebp
0x804832b <main+3>:     sub    $0x8,%esp
0x804832e <main+6>:     and    $0xfffffff0,%esp
0x8048331 <main+9>:     mov    $0x0,%eax
```

```
0x8048336 <main+14>:    sub    %eax,%esp
0x8048338 <main+16>:    sub    $0xc,%esp
0x804833b <main+19>:    push   $0x8048398
0x8048340 <main+24>:    call   0x8048268 <printf>
0x8048345 <main+29>:    add    $0x10,%esp
0x8048348 <main+32>:    mov    $0x0,%eax
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
0x804834f <main+39>:    nop
End of assembler dump.
```

看到进入 main 函数后的处理为将%ebp 压栈，将%esp 保存在%ebp 中，然后将%esp 下移 8 个字节。此时%esp 的值为 0xbffff9b0。所以此地址开始的 8 个字节应该就是 sub $0x8,%esp 处理的结果，此时这 8 个字节内容应该没有意义。而紧接着的 0xbffff9d8 就应该是%ebp 的值，也即栈桢。%ebp 上面的 0x420158d4 就应该是返回地址了。__libc_start_main 中调用的 main，那么返回地址也应该是该函数调用 main 的下一指令的地址。查了一下，正是如此。然后就应该是 main 函数的参数了。看__libc_start_main，main 函数有三个参数，分别是 argc,argv, __environ。对应栈中的值为 0x00000001，0xbffffa04，0xbffffa0c。argc=1 很好理解，argv 及__environ 的内容是什么？

```
(gdb) disass 0x420158d4
Dump of assembler code for function __libc_start_main:
… …
0x420158bb <__libc_start_main+139>:     mov    %edi,0x4(%esp,1)
0x420158bf <__libc_start_main+143>:     mov    0xc(%ebp),%eax
0x420158c2 <__libc_start_main+146>:     mov    0x214(%ebx),%ecx
0x420158c8 <__libc_start_main+152>:     mov    %eax,(%esp,1)
0x420158cb <__libc_start_main+155>:     mov    (%ecx),%eax
0x420158cd <__libc_start_main+157>:     mov    %eax,0x8(%esp,1)
0x420158d1 <__libc_start_main+161>:     call   *0x8(%ebp)
0x420158d4 <__libc_start_main+164>:     mov    %eax,(%esp,1)
0x420158d7 <__libc_start_main+167>:     call   0x4202b0f0 <exit>
… …
End of assembler dump.
```

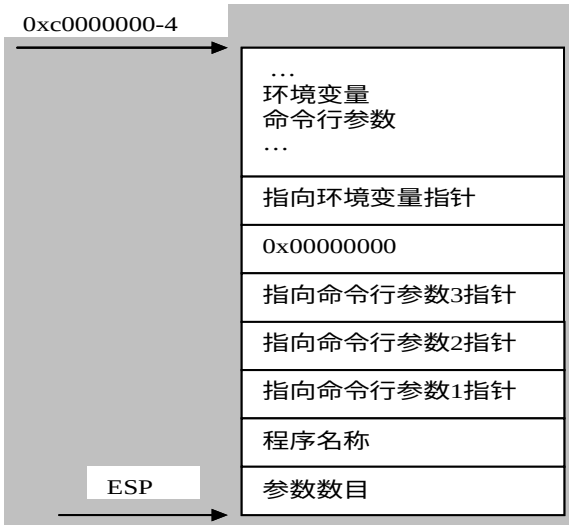
查一下 0xbffffa04，0xbffffa0c 处的内容，分别是 0xbffffb09,0xbffffb1b。

```
(gdb) x/20x 0xbffff9b0
0xbffff9b0:     0x4200aec8      0x4212a2d0      0xbffff9d8      0x420158d4
0xbffff9c0:     0x00000001      0xbffffa04      0xbffffa0c      0x400124b8
0xbffff9d0:     0x00000001      0x08048278      0x00000000      0x08048299
0xbffff9e0:     0x08048328      0x00000001      0xbffffa04      0x08048230
0xbffff9f0:     0x08048374      0x4000a950      0xbffff9fc      0x4001020c
(gdb)
0xbffffa00:     0x00000001      0xbffffb09      0x00000000      0xbffffb1b
0xbffffa10:     0xbffffb2d      0xbffffb43      0xbffffb4e      0xbffffb5e
0xbffffa20:     0xbffffb6c      0xbffffba0      0xbffffbb2      0xbffffbbc
0xbffffa30:     0xbffffd7f      0xbffffda9      0xbffffddf      0xbffffe40
0xbffffa40:     0xbffffe5a      0xbffffe66      0xbffffe76      0xbffffe8b
```

看看这两个地址处是什么内容：

```
(gdb) x/s 0xbffffb09
0xbffffb09:     "/home/foros/hello"
(gdb)
0xbffffb1b:     "SSH_AGENT_PID=870"
```

可见就是程序名称及环境参数的开始。这似乎又开始和我们认识中的程序运行时栈空间布置吻合了：



很明显，我们的可执行文件不需要命令行参数，所以在进入_start 入口后，在栈空间压入__libc_start_main 参数之前的栈空间布置就应该依次是 :参数数目，程序名称，0x00000000，指向环境变量的指针，环境变量。

```
0xbffffa00:     0x00000001      0xbffffb09      0x00000000      0xbffffb1b
0xbffffa10:     0xbffffb2d      0xbffffb43      0xbffffb4e      0xbffffb5e
```

应该看到，0xbffffa0c 开始，到栈底 0xC0000000-4，存储的应该是各个环境变量的指针，然后就是各个环境变量。让我们再跟踪一下，看看结果：

```
(gdb)
0xbffffb2d:     "HOSTNAME=Little.knife"
(gdb)
0xbffffb43:     "TERM=xterm"
(gdb)
0xbffffb4e:     "SHELL=/bin/bash"
(gdb)
0xbffffb5e:     "HISTSIZE=1000"
(gdb)
0xbffffb6c:     "GTK_RC_FILES=/etc/gtk/gtkrc:/root/.gtkrc-1.2-gnome2"
(gdb)
0xbffffba0:     "WINDOWID=23068815"
(gdb)
0xbffffbb2:     "USER=root"
(gdb)
0xbffffbbc:
"LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01:or
=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:*.
bat=00;32:*.sh=00;32:*.csh=00"...
(gdb)
0xbffffc84:
";32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00
;31:*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.rpm=00;31:*.cpio=00;31:*.jpg=00;35:*.gif
=00;35:*.bmp=00;3"...
(gdb)
0xbffffd4c:     "5:*.xbm=00;35:*.xpm=00;35:*.png=00;35:*.tif=00;35:"
(gdb)
0xbffffd7f:     "SSH_AUTH_SOCK=/tmp/ssh-XXiEYmfT/agent.821"
(gdb)
0xbffffda9:     "SESSION_MANAGER=local/Little.knife:/tmp/.ICE-unix/821"
(gdb)
0xbffffddf:
```

```
"PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/home/foros/bin"
(gdb)
0xbffffe40:  "MAIL=/var/spool/mail/root"
(gdb)
0xbffffe5a:  "_=/bin/bash"
(gdb)
0xbffffe66:  "PWD=/home/foros"
(gdb)
0xbffffe76:  "INPUTRC=/etc/inputrc"
(gdb)
0xbffffe8b:  "XMODIFIERS=@im=Chinput"
(gdb)
0xbffffea2:  "LANG=zh_CN.GB18030"
(gdb)
0xbffffeb5:  "GDMSESSION=Default"
(gdb)
0xbffffec8:  "SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass"
(gdb)
0xbffffefb:  "SHLVL=2"
(gdb)
0xbfffff03:  "HOME=/home/foros"
(gdb)
0xbfffff14:  "LANGUAGE=zh_CN.GB18030:zh_CN.GB2312:zh_CN"
(gdb)
0xbfffff3e:  "GNOME_DESKTOP_SESSION_ID=Default"
(gdb)
0xbfffff5f:  "LOGNAME=root"
(gdb)
0xbfffff6c:  "LESSOPEN=|/usr/bin/lesspipe.sh %s"
(gdb)
0xbfffff8e:  "DISPLAY=:0"
(gdb)
0xbfffff99:  "G_BROKEN_FILENAMES=1"
(gdb)
0xbfffffae:  "XAUTHORITY=/home/foros/.Xauthority"
(gdb)
0xbfffffd1:  "COLORTERM=gnome-terminal"
(gdb)
0xbfffffea:  "/home/foros/hello"
(gdb)
0xbffffffc:  ""
(gdb)
0xbffffffd:  ""
(gdb)
0xbffffffe:  ""
(gdb)
0xbfffffff:  ""
(gdb)
0xc0000000:  <Address 0xc0000000 out of bounds>
(gdb)
(gdb) x/20x 0xbffff9b0
0xbffff9b0:  0x4200aec8  0x4212a2d0  0xbffff9d8  0x420158d4
0xbffff9c0:  0x00000001  0xbffffa04  0xbffffa0c  0x400124b8
0xbffff9d0:  0x00000001  0x08048278  0x00000000  0x08048299
0xbffff9e0:  0x08048328  0x00000001  0xbffffa04  0x08048230
0xbffff9f0:  0x08048374  0x4000a950  0xbffff9fc  0x4001020c
(gdb)
```

```
0xbffffa00:  0x00000001  0xbffffb09  0x00000000  0xbffffb1b
0xbffffa10:  0xbffffb2d  0xbffffb43  0xbffffb4e  0xbffffb5e
0xbffffa20:  0xbffffb6c  0xbffffba0  0xbffffbb2  0xbffffbbc
0xbffffa30:  0xbffffd7f  0xbffffda9  0xbffffddf  0xbffffe40
0xbffffa40:  0xbffffe5a  0xbffffe66  0xbffffe76  0xbffffe8b
(gdb)
0xbffffa50:  0xbffffea2  0xbffffeb5  0xbffffec8  0xbffffefb
0xbffffa60:  0xbfffff03  0xbfffff14  0xbfffff3e  0xbfffff5f
0xbffffa70:  0xbfffff6c  0xbfffff8e  0xbfffff99  0xbfffffae
0xbffffa80:  0xbfffffd1  0x00000000  0x00000010  0x07e9fbbf
0xbffffa90:  0x00000006  0x00001000  0x00000011  0x00000064
(gdb)
0xbffffaa0:  0x00000003  0x08048034  0x00000004  0x00000020
0xbffffab0:  0x00000005  0x00000006  0x00000007  0x40000000
0xbffffac0:  0x00000008  0x00000000  0x00000009  0x08048278
0xbffffad0:  0x0000000b  0x00000000  0x0000000c  0x00000000
0xbffffae0:  0x0000000d  0x000001f6  0x0000000e  0x000001f6
(gdb)
0xbffffaf0:  0x0000000f  0xbffffb04  0x00000000  0x00000000
0xbffffb00:  0x00000000  0x36383669  0x6f682f00  0x662f656d
0xbffffb10:  0x736f726f  0x6c65682f  0x53006f6c  0x415f4853
0xbffffb20:  0x544e4547  0x4449505f  0x3037383d  0x534f4800
0xbffffb30:  0x4d414e54  0x694c3d45  0x656c7474  0x696e6b2e
(gdb)
0xbffffb40:  0x54006566  0x3d4d5245  0x72657478  0x4853006d
0xbffffb50:  0x3d4c4c45  0x6e69622f  0x7361622f  0x49480068
0xbffffb60:  0x49535453  0x313d455a  0x00303030  0x5f4b5447
0xbffffb70:  0x465f4352  0x53454c49  0x74652f3d  0x74672f63
0xbffffb80:  0x74672f6b  0x3a63726b  0x6f6f722f  0x672e2f74
(gdb)
0xbffffb90:  0x63726b74  0x322e312d  0x6f6e672d  0x0032656d
0xbffffba0:  0x444e4957  0x4449574f  0x3033323d  0x31383836
0xbffffbb0:  0x53550035  0x723d5245  0x00746f6f  0x435f534c
0xbffffbc0:  0x524f4c4f  0x6f6e3d53  0x3a30303d  0x303d6966
0xbffffbd0:  0x69643a30  0x3b30303d  0x6c3a3433  0x30303d6e
(gdb)
0xbffffbe0:  0x3a36333b  0x343d6970  0x33333b30  0x3d6f733a
0xbffffbf0:  0x333b3030  0x64623a35  0x3b30343d  0x303b3333
0xbffffc00:  0x64633a31  0x3b30343d  0x303b3333  0x726f3a31
0xbffffc10:  0x3b31303d  0x333b3530  0x31343b37  0x3d696d3a
0xbffffc20:  0x303b3130  0x37333b35  0x3a31343b  0x303d7865
(gdb)
0xbffffc30:  0x32333b30  0x632e2a3a  0x303d646d  0x32333b30
0xbffffc40:  0x652e2a3a  0x303d6578  0x32333b30  0x632e2a3a
0xbffffc50:  0x303d6d6f  0x32333b30  0x622e2a3a  0x303d6d74
0xbffffc60:  0x32333b30  0x622e2a3a  0x303d7461  0x32333b30
0xbffffc70:  0x732e2a3a  0x30303d68  0x3a32333b  0x73632e2a
(gdb)
0xbffffc80:  0x30303d68  0x3a32333b  0x61742e2a  0x30303d72
0xbffffc90:  0x3a31333b  0x67742e2a  0x30303d7a  0x3a31333b
0xbffffca0:  0x72612e2a  0x30303d6a  0x3a31333b  0x61742e2a
0xbffffcb0:  0x30303d7a  0x3a31333b  0x7a6c2e2a  0x30303d68
0xbffffcc0:  0x3a31333b  0x697a2e2a  0x30303d70  0x3a31333b
(gdb)
0xbffffcd0:  0x3d7a2e2a  0x333b3030  0x2e2a3a31  0x30303d5a
0xbffffce0:  0x3a31333b  0x7a672e2a  0x3b30303d  0x2a3a3133
0xbffffcf0:  0x327a622e  0x3b30303d  0x2a3a3133  0x3d7a622e
```

```
0xbffffd00:  0x333b3030  0x2e2a3a31  0x303d7a74  0x31333b30
0xbffffd10:  0x722e2a3a  0x303d6d70  0x31333b30  0x632e2a3a
(gdb)
0xbffffd20:  0x3d6f6970  0x333b3030  0x2e2a3a31  0x3d67706a
0xbffffd30:  0x333b3030  0x2e2a3a35  0x3d666967  0x333b3030
0xbffffd40:  0x2e2a3a35  0x3d706d62  0x333b3030  0x2e2a3a35
0xbffffd50:  0x3d6d6278  0x333b3030  0x2e2a3a35  0x3d6d7078
0xbffffd60:  0x333b3030  0x2e2a3a35  0x3d676e70  0x333b3030
(gdb)
0xbffffd70:  0x2e2a3a35  0x3d666974  0x333b3030  0x53003a35
0xbffffd80:  0x415f4853  0x5f485455  0x4b434f53  0x6d742f3d
0xbffffd90:  0x73732f70  0x58582d68  0x6d594569  0x612f5466
0xbffffda0:  0x746e6567  0x3132382e  0x53455300  0x4e4f4953
0xbffffdb0:  0x4e414d5f  0x52454741  0x636f6c3d  0x4c2f6c61
(gdb)
0xbffffdc0:  0x6c747469  0x6e6b2e65  0x3a656669  0x706d742f
0xbffffdd0:  0x43492e2f  0x6e752d45  0x382f7869  0x50003132
0xbffffde0:  0x3d485441  0x7273752f  0x636f6c2f  0x732f6c61
0xbffffdf0:  0x3a6e6962  0x7273752f  0x636f6c2f  0x622f6c61
0xbffffe00:  0x2f3a6e69  0x6e696273  0x69622f3a  0x752f3a6e
(gdb)
0xbffffe10:  0x732f7273  0x3a6e6962  0x7273752f  0x6e69622f
0xbffffe20:  0x73752f3a  0x31582f72  0x2f365231  0x3a6e6962
0xbffffe30:  0x6d6f682f  0x6f662f65  0x2f736f72  0x006e6962
0xbffffe40:  0x4c49414d  0x61762f3d  0x70732f72  0x2f6c6f6f
0xbffffe50:  0x6c69616d  0x6f6f722f  0x3d5f0074  0x6e69622f
(gdb)
0xbffffe60:  0x7361622f  0x57500068  0x682f3d44  0x2f656d6f
0xbffffe70:  0x6f726f66  0x4e490073  0x52545550  0x652f3d43
0xbffffe80:  0x692f6374  0x7475706e  0x58006372  0x49444f4d
0xbffffe90:  0x52454946  0x69403d53  0x68433d6d  0x75706e69
0xbffffea0:  0x414c0074  0x7a3d474e  0x4e435f68  0x3142472e
(gdb)
0xbffffeb0:  0x30333038  0x4d444700  0x53534553  0x3d4e4f49
0xbffffec0:  0x61666544  0x00746c75  0x5f485353  0x504b5341
0xbffffed0:  0x3d535341  0x7273752f  0x62696c2f  0x63657865
0xbffffee0:  0x65706f2f  0x6873736e  0x6f6e672f  0x732d656d
0xbffffef0:  0x612d6873  0x61706b73  0x53007373  0x4c564c48
(gdb)
0xbfffff00:  0x4800323d  0x3d454d4f  0x6d6f682f  0x6f662f65
0xbfffff10:  0x00736f72  0x474e414c  0x45474155  0x5f687a3d
0xbfffff20:  0x472e4e43  0x30383142  0x7a3a3033  0x4e435f68
0xbfffff30:  0x3242472e  0x3a323133  0x435f687a  0x4e47004e
0xbfffff40:  0x5f454d4f  0x4b534544  0x5f504f54  0x53534553
(gdb)
0xbfffff50:  0x5f4e4f49  0x443d4449  0x75616665  0x4c00746c
0xbfffff60:  0x414e474f  0x723d454d  0x00746f6f  0x5353454c
0xbfffff70:  0x4e45504f  0x752f7c3d  0x622f7273  0x6c2f6e69
0xbfffff80:  0x70737365  0x2e657069  0x25206873  0x49440073
0xbfffff90:  0x414c5053  0x303a3d59  0x425f4700  0x454b4f52
(gdb)
0xbfffffa0:  0x49465f4e  0x414e454c  0x3d53454d  0x41580031
0xbfffffb0:  0x4f485455  0x59544952  0x6f682f3d  0x662f656d
0xbfffffc0:  0x736f726f  0x61582e2f  0x6f687475  0x79746972
0xbfffffd0:  0x4c4f4300  0x4554524f  0x673d4d52  0x656d6f6e
0xbfffffe0:  0x7265742d  0x616e696d  0x682f006c  0x2f656d6f
(gdb)
```

```
0xbffffff0:     0x6f726f66      0x65682f73      0x006f6c6c      0x00000000
0xc0000000:     Cannot access memory at address 0xc0000000
(gdb)
```

大致布局和我们理论上是一致的。可是也出现了几个问题。

从上可见从 0xbffffa0c 开始到 0xbffffa84 存放的是环境变量的指针。0xbffffa84 处存放的内容为 0x00000000，不知仅是个随机值还也是一个结束标志。从这些指针指的内容来看，最后一个指针即 0xbffffa80 处的 0xbfffffd1，该地址处内容为"COLORTERM=gnome-terminal"，接下来从 0xbffffea 开始又存放了一次程序名称。接着就是 0xbfffffc 到 0xC0000000 的 4 个字节的应该是保留空间，这点从内核代码中可以看出，布置堆栈时从栈底 0xC0000000 开始首先保留一个 char 指针的空间，即 4 个字节，然后布置 envp[],argv[]，及 argc。遗留问题为：

1. 看来并不是从 0xbfffffc 开始就是环境量字符串，而是首先保存了程序名称字符串，然后才是环境变量。这点好象看到的资料没有提及或没有说的这么详细的，需要从代码中得到验证
2. 看来并不是环境变量指针之后紧跟着环境变量字符串，从 0xbffffa84 到 0xbffffb09(程序名称)之间的黄色空间的内容是什么，里面看到有程序头表的地址及_start 的地址，不知是随机值还是另有用途？若是出于堆栈对齐等考虑而空出的随机值，那这段空间应该可以用来做点事情哈。
3. 关于程序名称在栈中出现两次，一次是在从 0xbffffb09 开始，一次是从 0xbffffea 开始。如果是 ASCII 码，这两个地方值应该相同。可是为什么，在栈空间中看到却不仅不象 ASCII 码，而且两处的十六进制数也不相同，嗯？

Appendix

Reference