

哈爾濱工業大學

計算機系統

大作業

題 目 程序人生-Hello's P2P

專 業 計算學部

學 號 1190200717

班 級 1903008

學 生 梁浩

指 導 教 師 吳銳

計算機科學與技術學院

2021 年 5 月

摘 要

本文阐述了 `hello.c` 从预处理、编译、汇编到链接再到运行中间经过的全过程，在 Linux 系统下结合 `edb`、`objdump` 等工具进行分析。运行阶段，以 `hello` 为例，分析了 `fork`、`execve` 等函数以及程序是如何访问内存等内容。结合课本和网络资料，将理论知识和 `hello` 实例结合起来更加形象深入的学习计算机系统。

关键词：汇编；编译；链接；虚拟内存；动态申请内存

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

第 1 章 概述	- 4 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 5 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 7 -
2.4 本章小结	- 9 -
第 3 章 编译	- 10 -
3.1 编译的概念与作用	- 10 -
3.2 在 UBUNTU 下编译的命令	- 10 -
3.3 HELLO 的编译结果解析	- 11 -
3.4 本章小结	- 22 -
第 4 章 汇编	- 23 -
4.1 汇编的概念与作用	- 23 -
4.2 在 UBUNTU 下汇编的命令	- 23 -
4.3 可重定位目标 ELF 格式	- 23 -
4.4 HELLO.O 的结果解析	- 27 -
4.5 本章小结	- 29 -
第 5 章 链接	- 30 -
5.1 链接的概念与作用	- 30 -
5.2 在 UBUNTU 下链接的命令	- 30 -
5.3 可执行目标文件 HELLO 的格式	- 30 -
5.4 HELLO 的虚拟地址空间	- 32 -
5.5 链接的重定位过程分析	- 34 -
5.6 HELLO 的执行流程	- 35 -
5.7 HELLO 的动态链接分析	- 36 -
5.8 本章小结	- 36 -
第 6 章 HELLO 进程管理	- 37 -
6.1 进程的概念与作用	- 37 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	37 -
6.3 HELLO 的 FORK 进程创建过程.....	37 -
6.4 HELLO 的 EXECVE 过程.....	38 -
6.5 HELLO 的进程执行.....	38 -
6.6 HELLO 的异常与信号处理.....	39 -
6.7 本章小结.....	42 -
第 7 章 HELLO 的存储管理.....	43 -
7.1 HELLO 的存储器地址空间.....	43 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	43 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	44 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	44 -
7.5 三级 CACHE 支持下的物理内存访问.....	46 -
7.6 HELLO 进程 FORK 时的内存映射.....	46 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	47 -
7.8 缺页故障与缺页中断处理.....	47 -
7.9 动态存储分配管理.....	48 -
7.10 本章小结.....	49 -
第 8 章 HELLO 的 IO 管理.....	50 -
8.1 LINUX 的 IO 设备管理方法.....	50 -
8.2 简述 UNIX IO 接口及其函数.....	50 -
8.3 PRINTF 的实现分析.....	51 -
8.4 GETCHAR 的实现分析.....	53 -
8.5 本章小结.....	53 -
结论.....	53 -
附件.....	55 -
参考文献.....	56 -

第 1 章 概述

1.1 Hello 简介

1)P2P 简述:

用户通过编辑器编写代码，完成 `hello.c` 源程序文本的编写；在提前搭建好的 Ubuntu 系统下，调用预处理器（`cpp`）得到预处理后的文本程序 `hello.i`；接着调用编译器（`cc1`）得到汇编文本程序 `hello.s`；然后汇编器（`as`）将 `hello.s` 翻译成机器语言指令，把这些指令打包成可重定位目标程序的格式，得到可重定位目标文件 `hello.o`；最后通过链接器（`ld`）得到可执行目标文件 `hello`。用户在 shell 输入 `./hello` 执行此程序，shell 调用 `fork` 函数为其产生子进程，`hello` 便最终成为了一个进程。

2) O2O 简述:

OS 的进程管理调用 `fork` 函数，产生子进程，调用 `execve` 函数，并进行虚拟内存映射，并为运行的 `hello` 分配时间片来执行取指译码流水线等操作；当 CPU 引用一个被映射的虚拟页时，可执行目标文件 `hello` 中的代码和数据从磁盘复制到物理内存，然后跳转到程序入口点 `_start` 函数的地址，最终调用执行 `hello` 中的 `main` 函数，OS 的储存管理以及 MMU 解决 VA 到 PA 的转换，`cache`、`TLB`、页表等加速访问过程；程序结束时，shell 回收 `hello` 进程，内核从系统中清除它的痕迹。

1.2 环境与工具

硬件工具:

处理器：Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40GHz
已安装的内存(RAM)：8.00GB(7.81GB 可用)

软件工具:

Windows 10 家庭中文版; VirtualBox 6.1; Ubuntu 20.04

开发者与调试工具:

`gcc`, `gdb`, `edb`, `vim`, `readelf`, `objdump` 等

1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名称，文件的作用等。

生成的中间结果文件名	文件的作用
------------	-------

hello.i	hello.c 预处理后得到的文件
hello.s	hello.i 编译后得到的文件
hello.o	hello.s 汇编后的可重定位目标文件
hello-elf.txt	hello.o 的 elf 格式文件
hello	hello.o 经过链接后的可执行目标文件
hello-out-elf.txt	hello 的 elf 格式文件
hello-asm.txt	hello 的反汇编代码文件
hello-o-asm.txt	hello.o 的反汇编代码文件

1.4 本章小结

本章简要介绍了 hello.c 的 P2P 与 O2O，然后列举了本篇文章用到的环境与工具以及生成的中间文件（包括它们的作用）。

（第 1 章 0.5 分）

第 2 章 预处理

2.1 预处理的观念与作用

预处理的观念：

预处理是在程序编译之前进行的处理，可放在程序中任何位置。预处理是 C 语言的一个重要功能，它由预处理程序负责完成。当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分作处理，处理完毕自动进入对源程序的编译。

ISO C 和 ISO C++ 都规定程序由源代码被翻译分为若干有序的阶段，通常前几个阶段由预处理器实现。预处理中会展开以 # 起始的行，试图解释为预处理指令，其中 ISO C/C++ 要求支持的包括 `#if/#ifdef/#ifndef/#else/#elif/#endif`（条件编译）、`#define`（宏定义）、`#include`（源文件包含）、`#line`（行控制）、`#error`（错误指令）、`#pragma`（和实现相关的杂注）以及单独的 `#`（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

预处理的作用：

主要和三个部分有关：宏定义，文件包含，条件编译

- 1、宏定义。预处理程序中的 `#define` 标识符文本，预处理工作也叫做宏展开：将宏名替换为文本（这个文本可以是字符串、可以是代码等）。
- 2、文件包含。预处理程序中的 `#include`，将头文件的内容插入到该命令所在的位置，从而把头文件和当前源文件连接成一个源文件。
- 3、条件编译相关。根据 `#if` 以及 `#endif` 和 `#ifdef` 以及 `#ifndef` 来判断执行编译的条件。

2.2 在 Ubuntu 下预处理的命令

命令 1: `gcc -E hello.c -o hello.i`

执行后效果如下图所示：

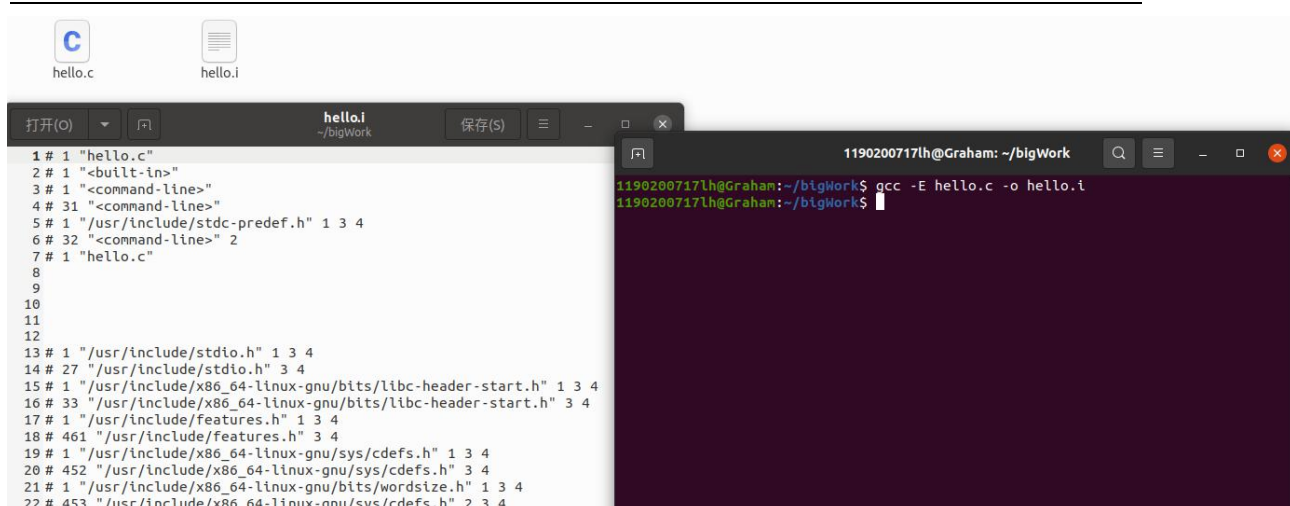


图2.2.1 命令1执行效果

命令 2: `cpp hello.c > hello.i`

执行后效果如图所示:

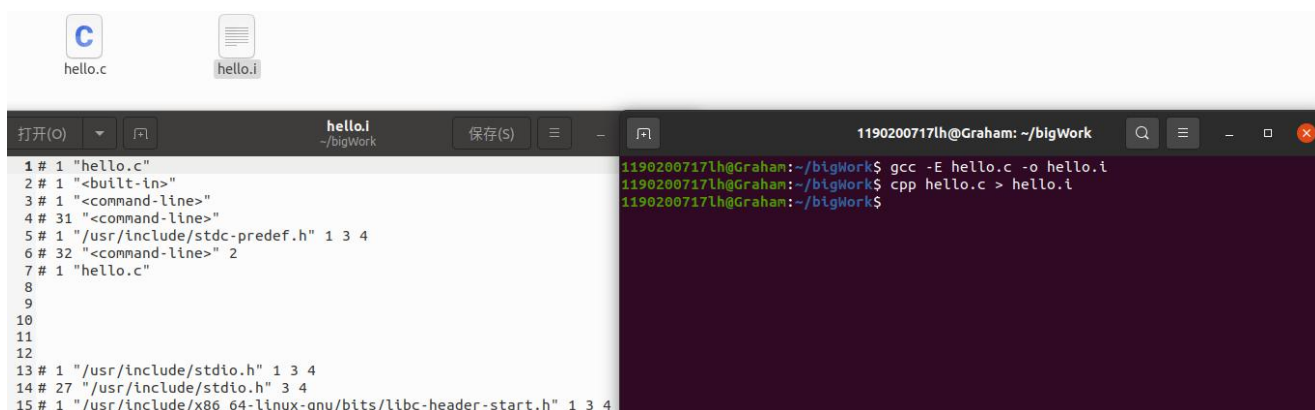


图2.2.2 命令2执行效果

2.3 Hello 的预处理结果解析

hello.i 文件的部分截图:


```
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"
8
9
10
11
12
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
18 # 461 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
20 # 452 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 453 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
24 # 454 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 462 "/usr/include/features.h" 2 3 4
26 # 485 "/usr/include/features.h" 3 4
27 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
28 # 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
29 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
30 # 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
31 # 486 "/usr/include/features.h" 2 3 4
32 # 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
33 # 28 "/usr/include/stdio.h" 2 3 4
34
```

图2.3.1 hello.i 文件部分截图

仔细观察 hello.i 文本文件，可以发现由于经过了预处理，原本 23 行的 hello.c 文件现在变成了 3060 行，这主要是因为预处理过程实现了头文件的展开，宏替换和去注释并作条件编译。

利用 ctrl+f 查询，可以发现 stdio.h 文件的展开开始于 13 行，结束于 728 行，unistd.h 文件的展开开始于 730 行，结束于 1966 行，stdlib.h 文件的展开开始于 1969 行，结束于 3041 行。可以根据给出的路径找到这三个文件的实际位置，如下图所示：

```

11902007171h@Graham:~$ cd /usr/include/
11902007171h@Graham:/usr/include$ ls
aio.h          fenv.h          locale.h        pthread.h       syscall.h
aliases.h      finclude        malloc.h        pty.h          sysexits.h
alloca.h       fmtmsg.h        math.h          pwd.h          syslog.h
argp.h         fnmatch.h       mcheck.h        python3.8      tar.h
argz.h         fpu_control.h   memory.h        rdma           termio.h
ar.h           fstab.h         misc            re_comp.h      termios.h
arpa           fts.h           mntent.h        regex.h         tgmath.h
asm            ftw.h           monetary.h      regexp.h        thread_db.h
asm-generic    gcalc-2         mqueue.h        reglib          threads.h
assert.h       gci-2           mtd             resolv.h        time.h
bits           gconv.h         net             rpc             ttyent.h
boost          getopt.h        netash          rpcsvc          uchar.h
byteswap.h     GL             netatalk        sched.h         ucontext.h
c++            glob.h          netax25         scsi            ulimit.h
capstone       gnu            netdb.h         search.h        unistd.h
complex.h      gnumake.h       neteconet       semaphore.h     utime.h
cpio.h         gnu-versions.h netinet          setjmp.h        utmp.h
crypt.h        graphviz        netipx          sgtty.h         utmpx.h
ctype.h        grp.h           netlucv         shadow.h        valgrind
dirent.h       gshadow.h       netpacket       signal.h        values.h
dlfcn.h        iconv.h         netrom          sound           video
drm            ifaddrs.h       netrose         spawn.h         vulkan
EGL            inttypes.h      nfs             stab.h          wait.h
elf.h          iproute2        nl_types.h      stdc-predef.h  wchar.h
endian.h       KHR             nss.h           stdint.h        wctype.h
envz.h         langinfo.h      obstack.h       stdio_ext.h     wordexp.h
err.h          lastlog.h       openvpn         stdio.h         X11
errno.h        libgen.h        paths.h         stdlib.h        x86_64-linux-gnu
error.h        libintl.h       poll.h          string.h         xcb
execinfo.h     limits.h        printf.h        strings.h        xen
fcntl.h        link.h          proc_service.h  sudo_plugin.h   xorg
features.h     linux           protocols       sys

```

图2.3.2 stdio.h、unistd.h、stdlib.h文件所在地

2.4 本章小结

本章主要介绍了预处理的概念和功能以及 Ubuntu 下预处理的两条指令，将 hello.c 预处理得到了 hello.i 文本，并分析其中的过程。因此，预处理过程主要是负责：头文件的展开、宏替换、去掉注释、条件编译。

（第2章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译的概念：

编译就是将源语言经过词法分析、语法分析、语义分析以及经过一系列优化后生成汇编代码的过程，它是将高级语言程序转化为机器可直接识别处理执行的机器码的中间步骤。

编译的作用：

它包括以下几个部分。

- 1、词法分析。对输入的字符串进行分析和分割，形成所使用的源程序语言所允许的记号，同时标注不规范记号，产生错误提示信息。
- 2、语法分析。分析词法分析得到的记号序列，并按一定规则识别并生成中间表示形式，以及符号表。同时将不符合语法规则的记号识别出其位置并产生错误提示语句。
- 3、语义分析。即静态语法检查，分析语法分析过程中产生的中间表示形式和符号表，以检查源程序的语义是否与源语言的静态语义属性相符合。
- 4、代码优化。将中间表示形式进行分析并转换为功能等价但是运行时间更短或占用资源更少的等价中间代码。

3.2 在 Ubuntu 下编译的命令

编译的命令：`gcc -S hello.i -o hello.s`

执行结果如下图所示：

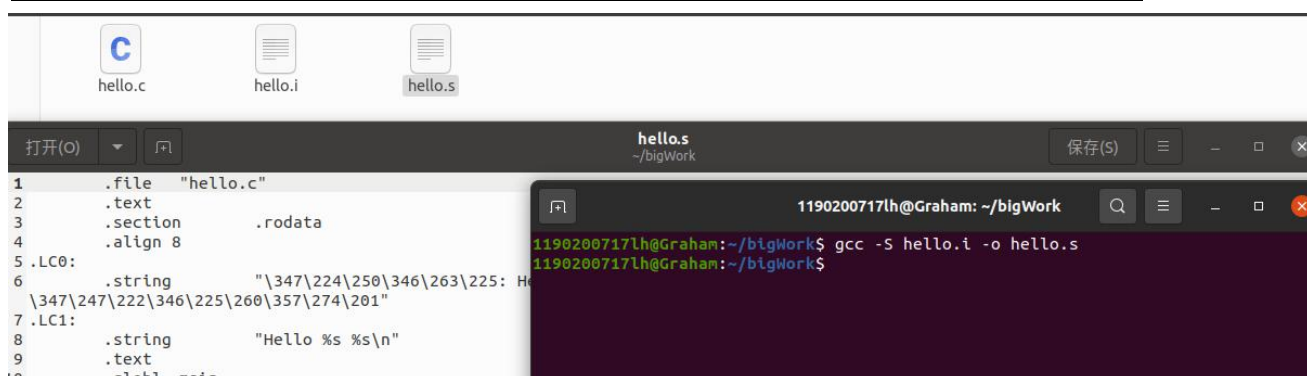


图3.2 hello.i编译后结果

3.3 Hello 的编译结果解析

先展示 hello.s 的截图：

```
1 .file "hello.c"
2 .text
3 .section .rodata
4 .align 8
5 .LC0:
6 .string "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201"
7 .LC1:
8 .string "Hello %s %s\n"
9 .text
10 .globl main
11 .type main, @function
12 main:
13 .LFB6:
14 .cfi_startproc
15 endbr64
16 pushq %rbp
17 .cfi_def_cfa_offset 16
18 .cfi_offset 6, -16
19 movq %rsp, %rbp
20 .cfi_def_cfa_register 6
21 subq $32, %rsp
22 movl %edi, -20(%rbp)
23 movq %rsi, -32(%rbp)
24 cmpl $4, -20(%rbp)
25 je .L2
26 leaq .LC0(%rip), %rdi
27 call puts@PLT
28 movl $1, %edi
29 call exit@PLT
30 .L2:
31 movl $0, -4(%rbp)
32 jmp .L3
33 .L4:
34 movq -32(%rbp), %rax
35 addq $16, %rax
36 movq (%rax), %rdx
37 movq -32(%rbp), %rax
38 addq $8, %rax
39 movq (%rax), %rax
40 movq %rax, %rsi
41 leaq .LC1(%rip), %rdi
42 movl $0, %eax
```

图3.3.1 hello.s的截图上

```

42     movl    $0, %eax
43     call   printf@PLT
44     movq    -32(%rbp), %rax
45     addq    $24, %rax
46     movq    (%rax), %rax
47     movq    %rax, %rdi
48     call   atoi@PLT
49     movl    %eax, %edi
50     call   sleep@PLT
51     addl    $1, -4(%rbp)
52 .L3:
53     cmpl    $7, -4(%rbp)
54     jle     .L4
55     call   getchar@PLT
56     movl    $0, %eax
57     leave
58     .cfi_def_cfa 7, 8
59     ret
60     .cfi_endproc
61 .LFE6:
62     .size   main, .-main
63     .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
64     .section .note.GNU-stack,"",@progbits
65     .section .note.gnu.property,"a"
66     .align  8
67     .long   1f - 0f
68     .long   4f - 1f
69     .long   5
70 0:
71     .string  "GNU"
72 1:
73     .align  8
74     .long   0xc0000002
75     .long   3f - 2f
76 2:
77     .long   0x3
78 3:
79     .align  8
80 4:

```

图3.3.2 hello.s的截图下

3.3.1 head.s 开头部分

hello.s 的开头部分如下：

```

.file    "hello.c"
.text
.section    .rodata
.align 8
.LC0:
.string    "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201"
.LC1:
.string    "Hello %s %s\n"
.text
.globl    main
.type    main, @function

```

图3.3.1 hello.s开头部分

.file 表示源文件，这里是 hello.c

.text 声明下面是代码段

.section .rodata 声明下面是 rodata 节

.align 声明指令、数据存放地址的对齐方式，这里是 8

.long .string 声明 long、string 类型

`.globl` 声明全局变量 `main`

`.type` 声明是函数类型还是对象类型，这里是 `main @function`

3.3.2 各种数据类型

`hello.s` 用到的 C 语言数据类型有：整数、字符串、数组。

1) 整数

对于 `int argc`:

`argc` 是 `main` 函数的第一个参数，代表了 `argv` 指针数组的数组元素个数。从 `hello.s` 里可以找到 `argc` 被存入了 `-20(%rbp)` 地址下，如下图所示：

```
main:
.LFB6:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl     %edi, -20(%rbp)
    movq    %rsi, -32(%rbp)
    cmpl    $4, -20(%rbp)
    je      .L2
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT
```

图3.3.2.1 存储`argc`参数

对于 `int i`:

C 语言中的局部变量在运行时被保存在栈或者是寄存器里。对于这里的 `int i`，编译器将它存储在了栈空间中，具体是 `-4(%rbp)` 下，如图所示：

```
call    printf@PLT
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi
call    atoi@PLT
movl    %eax, %edi
call    sleep@PLT
addl    $1, -4(%rbp)
```

图3.3.2.2 存储`int`变量

对于常数立即数：

源程序中出现的常数如 0, 1, 2, 10 等是直接在汇编代码中出现的，汇编代码是允许立即数以\$常数形式存在的。

```

38      addq    $8, %rax
39      movq    (%rax), %rax
40      movq    %rax, %rsi
41      leaq    .LC1(%rip), %rdi
42      movl    $0, %eax
43      call    printf@PLT
44      movq    -32(%rbp), %rax
45      addq    $24, %rax
46      movq    (%rax), %rax
47      movq    %rax, %rdi
48      call    atoi@PLT
49      movl    %eax, %edi
50      call    sleep@PLT
51      addl    $1, -4(%rbp)
52 .L3:
53      cmpl    $7, -4(%rbp)
54      jle     .L4
55      call    getchar@PLT
56      movl    $0, %eax
57      leave
58      .cfi_def_cfa 7, 8
59      ret
60      .cfi_endproc

```

图3.3.2.3 常数立即数

2) 字符串

程序中的字符串如图所示：

```

.string    "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201"
.string    "Hello %s %s\n"

```

图3.3.2.4 字符串

"用法: Hello 学号 姓名 秒数! \n"是第一个 printf 函数传入的输出格式化参数，在 hello.s 中表示如下：

```

"\347\224\250\346\263\225:           Hello           \345\255\246\345\217\267
\345\247\223\345\220\215 \347\247\222\346\225\260\357\274\201"

```

汉字采用 UTF-8 格式，一个汉字在 UTF-8 编码中用三个字节来表示，用\区分每个字节。

"Hello %s %s\n"，是第二个 printf 函数传入的输出格式化参数，也是只读的。

3) 数组

对于数组 `char *argv[]`:

它是 `main` 函数中的第二个形式参数，来源于执行时输入的参数。`argv` 每个指针元素 `char*` 大小为 8 个字节，`argv` 指针指向已经分配好的、一片存放着字符指针的连续空间，起始地址为 `argv`。在 `main` 函数内部，对 `argv[1]`，`argv[2]` 的访问来源于对数组首地址 `argv` 进行加法计算得到相应的地址。

下图是引用 `argv` 数组内容的部分截图：

```
.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movq    -32(%rbp), %rax
    addq    $24, %rax
    movq    (%rax), %rax
    movq    %rax, %rdi
    call    atoi@PLT
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)
```

图3.3.2.5 `argv`数组的使用

3.3.3 程序中的赋值

程序中的赋值一般情况下都用 `mov` 指令实现，数据长度不同，所用到的指令也就不同。举个例子，为 `int i` 赋值 `i=0`，汇编语句如下：

可以看到由于 `i` 是 `int` 类型的数据，故要用的是 `movl` 指令传送 4 字节大小的值。


```
.L2:      movl    $0, -4(%rbp)
        jmp     .L3
.L4:      movq    -32(%rbp), %rax
        addq    $16, %rax
        movq    (%rax), %rdx
        movq    -32(%rbp), %rax
        addq    $8, %rax
        movq    (%rax), %rax
        movq    %rax, %rsi
```

图3.3.3 程序中变量i的赋值

3.3.4 程序中的类型转换

常见的几种转换方式及其后果：

- 1、从 int 转换为 float 时，不会发生溢出，但可能有数据被舍入
- 2、从 int 或 float 转换为 double 时，因为 double 的有效位数更多，故能保留精确值
- 3、从 double 转换为 float 和 int 时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 4、从 float 或 double 转换为 int 时，因为 int 没有小数部分，所以数据可能会向 0 方向被截断

程序中调用了 `atoi` 函数，把字符串转换成整型数。其作用是将用户在命令行上输入的 `argv[3]` 给转换成 int 类型值传递给 `sleep` 函数，即将秒数传递给 `sleep` 函数。它的头文件包含在 `#include <stdlib.h>` 中。

3.3.5 程序中的算数操作

下面列举书上常见的整数算术操作：

指令	效果	描述
<code>leaq S, D</code>	$D \leftarrow \&S$	加载有效地址
<code>INC D</code>	$D \leftarrow D + 1$	加1
<code>DEC D</code>	$D \leftarrow D - 1$	减1
<code>NEG D</code>	$D \leftarrow -D$	取负
<code>NOT D</code>	$D \leftarrow \sim D$	取补
<code>ADD S, D</code>	$D \leftarrow D + S$	加
<code>SUB S, D</code>	$D \leftarrow D - S$	减
<code>IMUL S, D</code>	$D \leftarrow D * S$	乘
<code>XOR S, D</code>	$D \leftarrow D \wedge S$	异或
<code>OR S, D</code>	$D \leftarrow D \vee S$	或
<code>AND S, D</code>	$D \leftarrow D \& S$	与
<code>SAL k, D</code>	$D \leftarrow D \ll k$	左移
<code>SHL k, D</code>	$D \leftarrow D \ll k$	左移（等同于SAL）
<code>SAR k, D</code>	$D \leftarrow D \gg_A k$	算术右移
<code>SHR k, D</code>	$D \leftarrow D \gg_L k$	逻辑右移

图 3-10 整数算术操作。加载有效地址(`leaq`)指令通常用来执行简单的算术操作。其余的指令是更加标准的一元或二元操作。我们用 \gg_A 和 \gg_L 来分别表示算术右移和逻辑右移。注意，这里的操作顺序与 ATT 格式的汇编代码中的相反

图3.3.5.1 常见整数算术操作

下面举点 `hello.s` 中的例子：

1) 每次循环之后对 `i` 执行+1 操作

```

50      call    sleep@PLT
51      addl    $1, -4(%rbp)

```

图3.3.5.2 `i++`操作

2) 对栈顶指针的操作

```

20      retq    0
21      subq    $32, %rsp
22      movl    %edi, -20(%rbp)

```

图3.3.5.3 栈顶指针操作

3.3.6 程序中的比较操作

常见的指令有：

指令	基于	描述
CMP S_1, S_2	$S_2 - S_1$	比较
cmpb		比较字节
cmpw		比较字
cmpl		比较双字
cmpq		比较四字

图3.3.6.1 cmp指令

指令	同义名	跳转条件	描述
jmp <i>Label</i>		1	直接跳转
jmp <i>*Operand</i>		1	间接跳转
je <i>Label</i>	jz	ZF	相等/零
jne <i>Label</i>	jnz	~ZF	不相等/非零
js <i>Label</i>		SF	负数
jns <i>Label</i>		~SF	非负数
jg <i>Label</i>	jnle	~(SF ^ OF) & ~ZF	大于 (有符号>)
jge <i>Label</i>	jnl	~(SF ^ OF)	大于或等于 (有符号>=)
jl <i>Label</i>	jnge	SF ^ OF	小于 (有符号<)
jle <i>Label</i>	jng	(SF ^ OF) ZF	小于或等于 (有符号<=)
ja <i>Label</i>	jnbe	~CF & ~ZF	超过 (无符号>)
jae <i>Label</i>	jnb	~CF	超过或相等 (无符号>=)
jb <i>Label</i>	jnae	CF	低于 (无符号<)
jbe <i>Label</i>	jna	CF ZF	低于或相等 (无符号<=)

图 3-15 jmp 指令。当跳转条件满足时，这些指令会跳转到一条带标号的目的地。
有些指令有“同义名”，也就是同一条机器指令的别名

图3.3.6.2 jump指令

hello.s 中用到了对 jle 和 je 的条件码判断，如下图所示：

判断 argc 是否等于 4

```

23      movq    %rsi, -32(%rbp)
24      cmpl    $4, -20(%rbp)
25      je      .L2

```

图3.3.6.3 判断argc是否等于4

判断 i 是否小于 8

```
cmpl    $7, -4(%rbp)
jle     .L4
call    getchar@PLT
```

图3.3.6.4 判断i是否小于8

3.3.7 程序中的控制转移

对于 if 判断)

举例如下：

首先 `cmpl` 比较 `argv` 和 4，并设置条件码，使用 `je` 判断 ZF 标志位，如果为 ZF 为 1，说明 `argv` 等于 4，直接跳转到.L2，否则顺序执行下一条语句。

```
22      movl    %edi, -20(%rbp)
23      movq    %rsi, -32(%rbp)
24      cmpl    $4, -20(%rbp)
25      je      .L2
26      leaq    .LC0(%rip), %rdi
27      call    puts@PLT
28      movl    $1, %edi
29      call    exit@PLT
```

图3.3.7.1 if判断

对于 for 循环)

举例如下：

首先对 `i` 赋初值为 0，然后用 `cmpl` 进行比较，如果 `i` ≤ 7，则跳入.L4，执行循环体内部的操作，否则循环结束，执行 for 循环之后的代码。

```

.L2:
    movl    $0, -4(%rbp) int i=0
    jmp     .L3

.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movq    -32(%rbp), %rax
    addq    $24, %rax
    movq    (%rax), %rax
    movq    %rax, %rdi
    call    atoi@PLT
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp) 对 i++

.L3:
    cmpl    $7, -4(%rbp)
    jle     .L4 如果 i ≤ 7, 直接跳到 L4
    call    getchar@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

for 循环体里的内容

图3.3.7.2 for循环

3.3.8 函数操作

函数提供了一种封装代码的方式，用一组指定的参数和一个可选的返回值实现了某种功能。

源代码中涉及的函数有：

1) main 函数

传递控制：call 指令将下一条指令的地址压栈，然后跳转到 main 函数；传递数

据：向 main 函数传递参数 argc 和 argv，分别使用 %rdi 和 %rsi 存储，将 %eax 作为

返回值，设置为 0。

2) printf 函数

传递数据：将%rdi 设置为字符串的首地址

```
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $4, -20(%rbp)
je      .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
```

图3.3.8.1 printf函数调用1

```
movq    (%rax), %rax
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi
leaq    .LC1(%rip), %rdi
movl    $0, %eax
call    printf@PLT
```

图3.3.8.2 printf函数调用2

控制传递：第一次 call puts；第二次 call printf。

3) atoi 函数

传递参数

```
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi
call    atoi@PLT
```

图3.3.8.3 atoi函数调用

控制转移：call atoi

4) sleep 函数

传递参数：将%edi 设置为 atoi(argv[3])

控制传递：call sleep

5) getchar 函数：

控制传递: call getchar

```
call    getchar@PLT
movl    $0, %eax
leave
.cfi def cfa 7, 8
```

图3.3.8.4 getchar函数调用

6) exit 函数:

传递数据: 将%edi 设置为 1。

控制传递: call exit

```
:8      movl    $1, %edi
:9      call    exit@PLT
```

图3.3.8.5 exit函数调用

3.3.9 最终结果

经过编译后, hello.s 还是文本文件的形式, CPU 无法直接执行, 编译只是将高级语言程序转化为机器可直接识别处理执行的的机器码的中间步骤。

3.4 本章小结

本章详细介绍了编译的概念与作用以及 Ubuntu 下编译的指令, 然后从数据类型、赋值操作、类型转换、算术操作、控制转移和函数操作等 9 个方面细致分析了 hello.s 文件的内容。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

概念：

驱动程序运行汇编器 `as`，将汇编语言翻译成机器语言指令，并把这些指令打包成可重定位目标程序的格式，并将结果保存在二进制目标文件的过程称为汇编。

作用：

汇编就是将高级语言转化为机器可直接识别执行的代码文件的过程，汇编器将 `.s` 文件翻译成机器语言指令，并将这些指令打包成可重定位目标程序的格式，将结果保存在 `.o` 二进制目标文件中。

4.2 在 Ubuntu 下汇编的命令

`gcc -o hello.s -o hello.o`

执行结果如图所示：

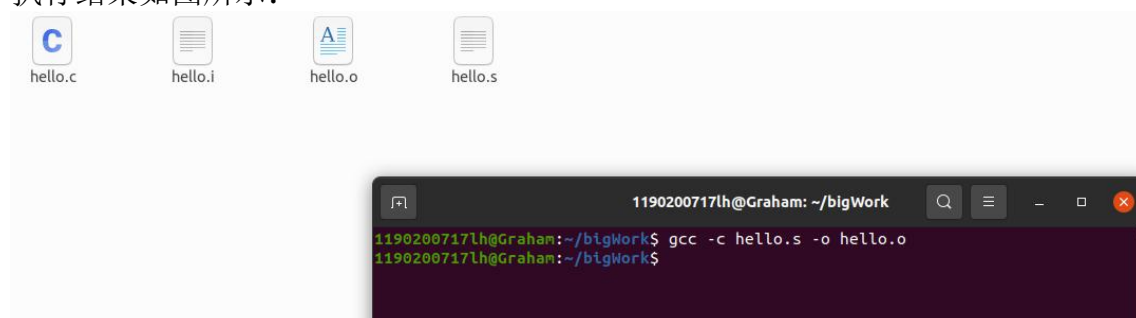
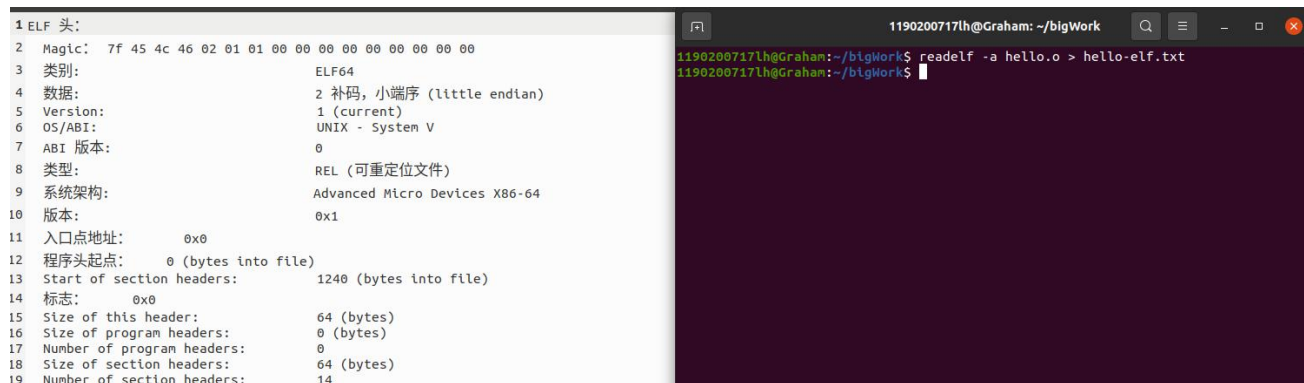


图4.2 执行完`gcc -o hello.s -o hello.o`之后的结果

4.3 可重定位目标 `elf` 格式

4.3.1 读取可重定位目标文件

输入命令 `readelf -a hello.o > hello-elf.txt` 将 `elf` 可重定位目标文件输出定向到文本文件 `hello.elf` 中，截图如下：



```

1 ELF 头:
2 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 类别: ELF64
4 数据: 2 补码, 小端序 (little endian)
5 Version: 1 (current)
6 OS/ABI: UNIX - System V
7 ABI 版本: 0
8 类型: REL (可重定位文件)
9 系统架构: Advanced Micro Devices X86-64
10 版本: 0x1
11 入口点地址: 0x0
12 程序头起点: 0 (bytes into file)
13 Start of section headers: 1240 (bytes into file)
14 标志: 0x0
15 Size of this header: 64 (bytes)
16 Size of program headers: 0 (bytes)
17 Number of program headers: 0
18 Size of section headers: 64 (bytes)
19 Number of section headers: 14

```

图4.3.1 hello-elf.txt

4.3.2 典型的 ELF 可重定位目标文件的表格

ELF 头
.text
.rodata
.data
.bss
.symtab
.rel.text
.rel.data
.debug
.line
.strtab
节头部表

4.3.3 列出 ELF 文件各个节的内容

ELF 头:

以 16 字节的序列开始,这个序列描述了生成该文件的系统的字的大小和字节顺序。ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息,其中包括 ELF 头的大小、目标文件的类型、机器类型、节头部表的文件偏移,以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的,其中目标文件中每个节都有一个固定大小的条目。截图如下:

```

1 ELF 头:
2 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 类别: ELF64
4 数据: 2 补码, 小端序 (little endian)
5 Version: 1 (current)
6 OS/ABI: UNIX - System V
7 ABI 版本: 0
8 类型: REL (可重定位文件)
9 系统架构: Advanced Micro Devices X86-64
10 版本: 0x1
11 入口点地址: 0x0
12 程序头起点: 0 (bytes into file)
13 Start of section headers: 1240 (bytes into file)
14 标志: 0x0
15 Size of this header: 64 (bytes)
16 Size of program headers: 0 (bytes)
17 Number of program headers: 0
18 Size of section headers: 64 (bytes)
19 Number of section headers: 14
20 Section header string table index: 13

```

图4.3.3.1 ELF头

下面是具体分析:

Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

Magic 是魔数, 确定文件的类型或格式, 加载或读取文件时, 可用魔数确认文件类型是否正确

类别: ELF64

代表 ELF64 格式

数据: 2 补码, 小端序 (little endian)

二进制补码类型, 小端

Version: 1 (current)

版本信息 Version

OS/ABI

UNIX - System V

ABI 版本: 0

ABI 版本

类型: REL (可重定位文件)

目标文件格式 可重定位文件

系统架构: Advanced Micro Devices X86-64

Advanced Micro Devices X86-64 的机器

版本: 0x1

版本

入口点地址: 0x0

程序执行的入口地址

程序头起点: 0 (bytes into file)

段头部表的开始

Start of section headers: 1240 (bytes into file)

节头部表的开始

标志: 0x0

标志

Size of this header: 64 (bytes)

头大小

Size of program headers: 0 (bytes)

段头部表大小

Number of program headers: 0

段头部表数量

Size of section headers: 64 (bytes)

节头部表大小

Number of section headers: 14

节头部表数量

Section header string table index: 13

字符串表在节头部表中的索引

.text: 已编译程序的机器代码。

.rodata: 只读数据, 比如 printf 语句中的格式串和开关语句的跳转表。

.data: 已初始化的全局和静态 C 变量。局部 C 变量在运行时被保存在栈中, 既不出现在 .data 节中, 也不出现在 .bss 中。

.bss: 未初始化的全局和静态 C 变量, 以及所有被初始化为 0 的全局或静态 C 变量。

.symtab: 一个符号表, 他存放在程序中定义和引用的函数和全局变量的信息

.rel.text: 一个 .text 节中位置的列表, 链接时修改

.rel.data: 被模块引用或定义的所有全局变量的重定位信息

.debug: 条目是局部变量、类型定义、全局变量及 C 源文件

.line: C 源程序中行号和.text 节机器指令的映射

.strtab: .symtab 和.debug 中符号表及节头部中节的名字

节头部表: 节头表包括节名称, 节的类型, 节的属性(读写权限), 节在 ELF 文件中所占的长度以及节的对齐方式和偏移量。

节头:

[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标	链接 信息 对齐
[0]	0000000000000000	NULL	0000000000000000	00000000
[1]	.text	PROGBITS	0000000000000000	00000040
[2]	.rela.text	RELA	0000000000000000	00000388
[3]	.data	PROGBITS	0000000000000000	000000d2
[4]	.bss	NOBITS	0000000000000000	000000d2
[5]	.rodata	PROGBITS	0000000000000000	000000d8
[6]	.comment	PROGBITS	0000000000000000	0000010b
[7]	.note.gnu-stack	PROGBITS	0000000000000000	00000136
[8]	.note.gnu.property	NOTE	0000000000000000	00000138
[9]	.eh_frame	PROGBITS	0000000000000000	00000158
[10]	.rela.eh_frame	RELA	0000000000000000	00000448
[11]	.symtab	SYMTAB	0000000000000000	00000190
[12]	.strtab	STRTAB	0000000000000000	00000340
[13]	.shstrtab	STRTAB	0000000000000000	00000460

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

图4.3.3.2 节头部表

4.4 Hello.o 的结果解析

命令行输入 `objdump -d -r hello.o`

结果如图所示:

```

1190200717lh@Graham:~/bigWork$ objdump -d -r hello.o
hello.o: 文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0: f3 0f 1e fa          endbr64
 4: 55                   push  %rbp
 5: 48 89 e5             mov   %rsp,%rbp
 8: 48 83 ec 20          sub   $0x20,%rsp
 c: 89 7d ec             mov   %edi,-0x14(%rbp)
 f: 48 89 75 e0          mov   %rsi,-0x20(%rbp)
13: 83 7d ec 04          cmpl  $0x4,-0x14(%rbp)
17: 74 16               je    2f <main+0x2f>
19: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 20 <main+0x20>
                1c: R_X86_64_PC32      .rodata-0x4
20: e8 00 00 00 00      callq 25 <main+0x25>
                21: R_X86_64_PLT32      puts-0x4
25: bf 01 00 00 00      mov   $0x1,%edi
2a: e8 00 00 00 00      callq 2f <main+0x2f>
                2b: R_X86_64_PLT32      exit-0x4
2f: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
36: eb 48               jmp    80 <main+0x80>
38: 48 8b 45 e0          mov   -0x20(%rbp),%rax
3c: 48 83 c0 10          add   $0x10,%rax
40: 48 8b 10             mov   (%rax),%rdx
43: 48 8b 45 e0          mov   -0x20(%rbp),%rax
47: 48 83 c0 08          add   $0x8,%rax
4b: 48 8b 00             mov   (%rax),%rax
4e: 48 89 c6             mov   %rax,%rsi
51: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 58 <main+0x58>

```

图4.4.1 objdump -d -r hello.o的执行结果

4.4.1 分支转移

hello.s 文件中分支转移是使用段名称进行跳转的，而 hello.o 文件中分支转移是通过地址进行跳转的。

```

.L2:
    movl    $0, -4(%rbp)
    jmp     .L3

```

图4.4.1.1 hello.s中的分支转移

```

17: 74 16               je    2f <main+0x2f>
19: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 20 <main+0x20>
                1c: R_X86_64_PC32      .rodata-0x4
20: e8 00 00 00 00      callq 25 <main+0x25>
                21: R_X86_64_PLT32      puts-0x4
25: bf 01 00 00 00      mov   $0x1,%edi
2a: e8 00 00 00 00      callq 2f <main+0x2f>
                2b: R_X86_64_PLT32      exit-0x4
2f: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
36: eb 48               jmp    80 <main+0x80>

```

图4.4.1.2 hello.o中的分支转移

4.4.2 函数调用

hello.s 文件中，函数调用 call 后跟的是函数名称；而在 hello.o 文件中，call 后跟的是下一条指令。而当这些函数都是共享库函数时，地址是不确定的，因此 call 指令将相对地址设置为全 0，然后在 .rela.text 节中为其添加重定位条目，等待链接的进一步确定。

```

26      leaq    .LC0(%rip), %rdi
27      call   puts@PLT
28      movl    $1, %edi
29      call   exit@PLT

```

图4.4.2.1 hello.s调用函数

```

86:  e8 00 00 00 00      callq  8b <main+0x8b>
87:  R_X86_64_PLT32     getchar-0x4

```

图4.4.2.2 hello.o调用函数

4.4.3 全局变量

hello.s文件中，全局变量是通过语句：段地址+%rip完成的；对于hello.o来说，则是：0+%rip，因为.rodata节中的数据是在运行时确定的，也需要重定位，现在填0占位，并为其在.rela.text节中添加重定位条目。

```
leaq    .LC0(%rip), %rdi
```

图4.4.3.1 hello.s访问全局变量

```

48 8d 3d 00 00 00 00      lea     0x0(%rip),%rdi      # 58 <main+0x58>

```

图4.4.3.2 hello.o访问全局变量

4.4.4 机器语言

机器语言程序是二进制的机器指令序列集合，完全由二进制数表示，由操作码与操作数组成。机器语言与汇编语言具有一一对应的映射关系，每一行机器代码对应一行汇编代码。机器代码中的操作数是能被机器直接理解的代码，而汇编代码是能使人理解CPU操作的低级语言代码。

4.5 本章小结

本章对汇编结果进行了详细的介绍，介绍了汇编的概念与作用以及在 Ubuntu 下汇编的命令。同时本章主要部分在于对可重定位目标 elf 格式进行了详细的分析。同时对 hello.o 文件进行反汇编，与之前生成的 hello.s 文件进行了对比分析。

(第4章1分)

第 5 章 链接

5.1 链接的概念与作用

概念：

链接程序将分别在不同的目标文件中编译或汇编的代码收集到一个可直接执行的文件中。它还连接目标程序和用于标准库函数的代码，以及连接目标程序和由计算机的操作系统提供的资源。

作用：

- 1) 链接可以执行于编译时，也就是源代码被翻译成机器代码时；也可以执行于加载时，即程序被加载器加载到内存并执行时；甚至执行于运行时，也就是由应用程序来执行。
- 2) 链接使得分离编译成为可能。便于维护管理，可以独立的修改和编译我们需要修改的小的模块。

5.2 在 Ubuntu 下链接的命令

命令如下：

```
ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o  
/usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so  
/usr/lib/x86_64-linux-gnu/crtn.o
```

效果如下图所示：

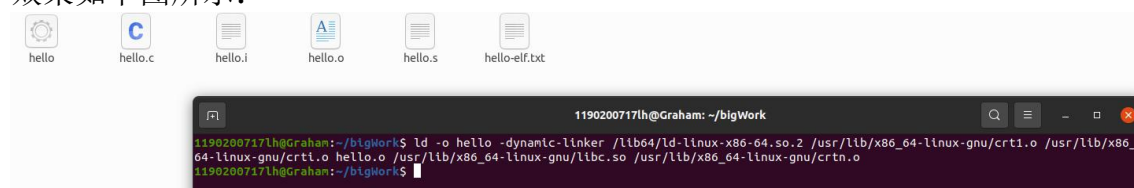


图5.2 hello.o链接后得到hello

5.3 可执行目标文件 hello 的格式

获取 hello 的 elf 格式文件：`readelf -a hello > hello-out-elf.txt`

```

1 ELF 头:
2 Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
3 类别:                                ELF64
4 数据:                                2 补码, 小端序 (little endian)
5 Version:                            1 (current)
6 OS/ABI:                             UNIX - System V
7 ABI 版本:                           0
8 类型:                                EXEC (可执行文件)
9 系统架构:                           Advanced Micro Devices X86-64
10 版本:                                0x1
11 入口点地址:                          0x4010f0
12 程序头起点:                          64 (bytes into file)
13 Start of section headers:            14208 (bytes into file)
14 标志:                                0x0
15 Size of this header:                 64 (bytes)
16 Size of program headers:             56 (bytes)
17 Number of program headers:           12
18 Size of section headers:             64 (bytes)
19 Number of section headers:           27
20 Section header string table index: 26
21

```

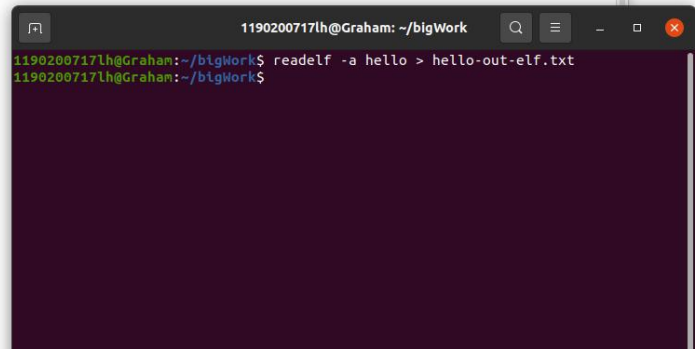


图5.3.1 hello的elf格式文件

各节的基本信息均在节头表中进行了声明，节头表（包括名称，大小，类型，全体大小，地址，旗标，偏移量，对齐等信息），截图如下：

节头:

[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标 链接 信息	对齐
[0]	0000000000000000	NULL	0000000000000000	00000000
	0000000000000000		0	0
[1]	.interp	PROGBITS	00000000004002e0	000002e0
	000000000000001c		A 0 0	1
[2]	.note.gnu.propert	NOTE	0000000000400300	00000300
	0000000000000020		A 0 0	8
[3]	.note.ABI-tag	NOTE	0000000000400320	00000320
	0000000000000020		A 0 0	4
[4]	.hash	HASH	0000000000400340	00000340
	0000000000000038		A 6 0	8
[5]	.gnu.hash	GNU_HASH	0000000000400378	00000378
	000000000000001c		A 6 0	8
[6]	.dynsym	DYNSYM	0000000000400398	00000398
	00000000000000d8		A 7 1	8
[7]	.dynstr	STRTAB	0000000000400470	00000470
	000000000000005c		A 0 0	1
[8]	.gnu.version	VERSYM	00000000004004cc	000004cc
	0000000000000012		A 6 0	2
[9]	.gnu.version_r	VERNEED	00000000004004e0	000004e0
	0000000000000020		A 7 1	8
[10]	.rela.dyn	RELA	0000000000400500	00000500
	0000000000000030		A 6 0	8
[11]	.rela.plt	RELA	0000000000400530	00000530
	0000000000000090		AI 6 21	8
[12]	.init	PROGBITS	0000000000401000	00001000
	000000000000001b		AX 0 0	4
[13]	.plt	PROGBITS	0000000000401020	00001020
	0000000000000070		AX 0 0	16
[14]	.plt.sec	PROGBITS	0000000000401090	00001090
	0000000000000060		AX 0 0	16
[15]	.text	PROGBITS	00000000004010f0	000010f0
	00000000000000145		AX 0 0	16
[16]	.fini	PROGBITS	0000000000401238	00001238
	000000000000000d		AX 0 0	4
[17]	.rodata	PROGBITS	0000000000402000	00002000
	000000000000003b		A 0 0	8
[18]	.eh_frame	PROGBITS	0000000000402040	00002040
	00000000000000fc		A 0 0	8
[19]	.dynamic	DYNAMIC	0000000000403e50	00002e50
	000000000000001a0		WA 7 0	8
[20]	.got	PROGBITS	0000000000403ff0	00002ff0
	0000000000000010		WA 0 0	8
[21]	.got.plt	PROGBITS	0000000000404000	00003000

图5.3.2 节头表

程序头:

程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040
	0x00000000000002a0	0x00000000000002a0	R 0x8
INTERP	0x00000000000002e0	0x00000000004002e0	0x00000000004002e0
	0x00000000000001c	0x00000000000001c	R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000
	0x00000000000005c0	0x00000000000005c0	R 0x1000
LOAD	0x00000000000001000	0x0000000000401000	0x0000000000401000
	0x0000000000000245	0x0000000000000245	R E 0x1000
LOAD	0x00000000000002000	0x0000000000402000	0x0000000000402000
	0x000000000000013c	0x000000000000013c	R 0x1000
LOAD	0x00000000000002e50	0x0000000000403e50	0x0000000000403e50
	0x00000000000001fc	0x00000000000001fc	RW 0x1000
DYNAMIC	0x00000000000002e50	0x0000000000403e50	0x0000000000403e50
	0x00000000000001a0	0x00000000000001a0	RW 0x8
NOTE	0x0000000000000300	0x0000000000400300	0x0000000000400300
	0x0000000000000020	0x0000000000000020	R 0x8
NOTE	0x0000000000000320	0x0000000000400320	0x0000000000400320
	0x0000000000000020	0x0000000000000020	R 0x4
GNU_PROPERTY	0x0000000000000300	0x0000000000400300	0x0000000000400300
	0x0000000000000020	0x0000000000000020	R 0x8
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x00000000000002e50	0x0000000000403e50	0x0000000000403e50
	0x00000000000001b0	0x00000000000001b0	R 0x1

图5. 3. 3 程序头

5.4 hello 的虚拟地址空间

1、找到 edb 所在位置并运行，截图如下：

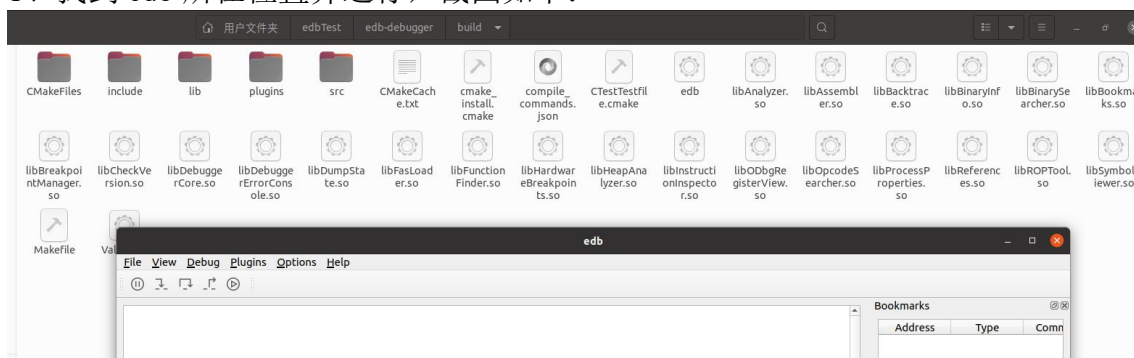


图5. 4. 1 找到edb并运行

2、在 edb 中打开 hello

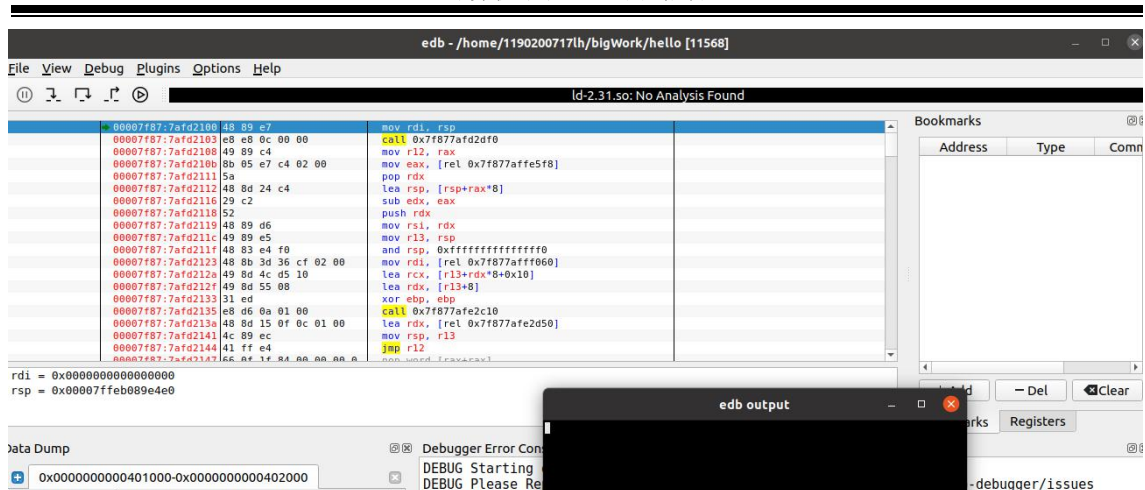


图5. 4. 2 在edb中打开hello

3、打开 Symbol Viewer 窗口

可以看到各节的虚拟地址，截图如下：

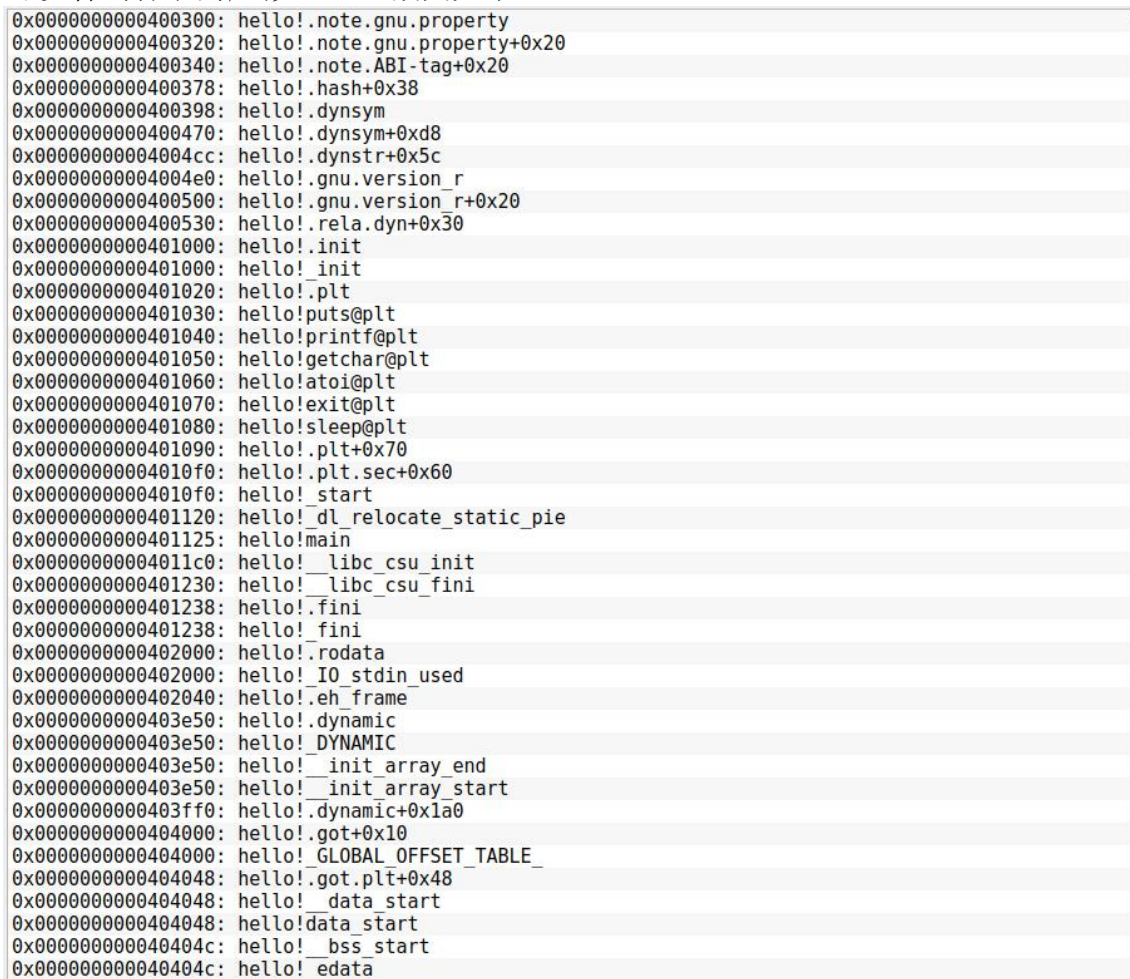


图5. 4. 3 各节的虚拟地址

5.5 链接的重定位过程分析

1、获得 hello.o 和 hello 的反汇编文件

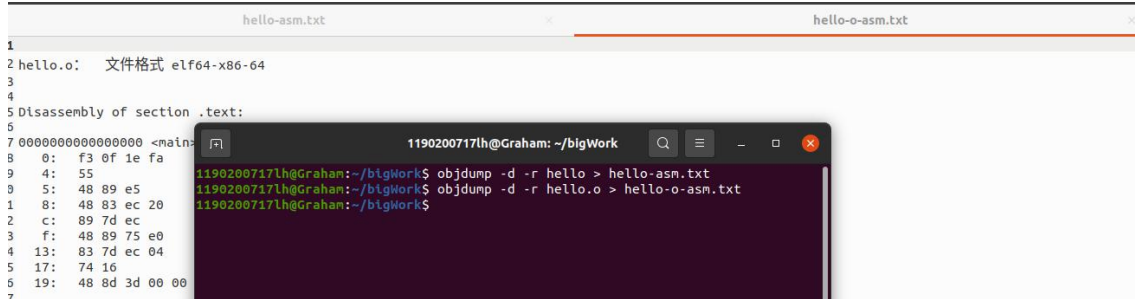


图5.5.1 获得两个反汇编文件

2、对比两个反汇编文件

1) hello-asm.txt 比 hello-o-asm.txt 多了几个节，例如下面的 section .init

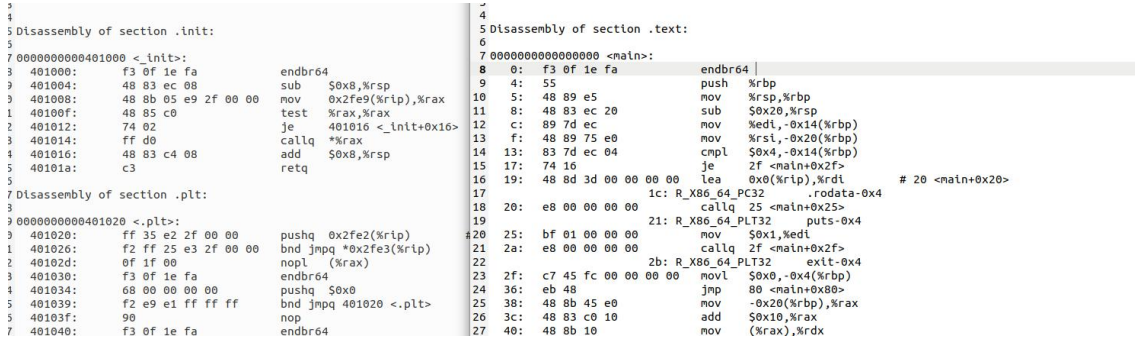


图5.5.2 区别1：多了一些节

2) hello-asm.txt 中用的是虚拟地址，hello-o-asm.txt 中用的是相对偏移地址

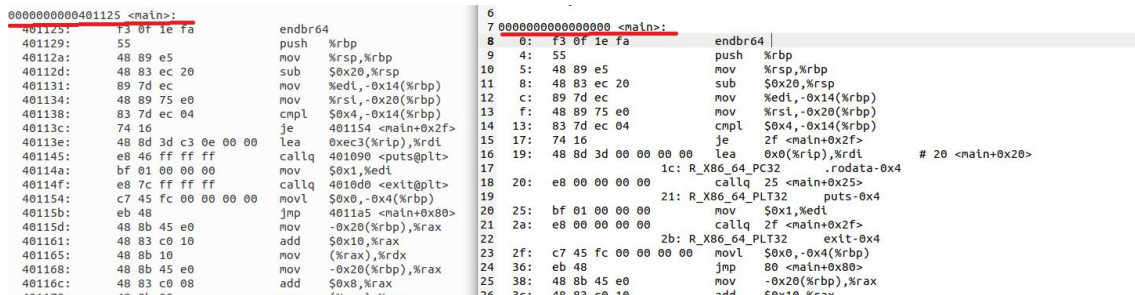


图5.5.3 区别2：虚拟地址和相对偏移地址

3) hello-asm.txt 中增加了许多外部链接的共享库函数。如 puts@plt 共享库函数，printf@plt 共享库函数以及 getchar@plt 函数等。


```

000000000401090 <puts@plt>:
401090: f3 0f 1e fa      endbr64
401094: f2 ff 25 7d 2f 00 00 bnd jmpq *0x2f7d(%rip) # 404018 <puts@GLIBC_2.2.5>
40109b: 0f 1f 44 00 00    nopl 0x0(%rax,%rax,1)

0000000004010a0 <printf@plt>:
4010a0: f3 0f 1e fa      endbr64
4010a4: f2 ff 25 75 2f 00 00 bnd jmpq *0x2f75(%rip) # 404020 <printf@GLIBC_2.2.5>
4010ab: 0f 1f 44 00 00    nopl 0x0(%rax,%rax,1)

0000000004010b0 <getchar@plt>:
4010b0: f3 0f 1e fa      endbr64
4010b4: f2 ff 25 6d 2f 00 00 bnd jmpq *0x2f6d(%rip) # 404028 <getchar@GLIBC_2.2.5>
4010bb: 0f 1f 44 00 00    nopl 0x0(%rax,%rax,1)

0000000004010c0 <atoi@plt>:
4010c0: f3 0f 1e fa      endbr64
4010c4: f2 ff 25 65 2f 00 00 bnd jmpq *0x2f65(%rip) # 404030 <atoi@GLIBC_2.2.5>
4010cb: 0f 1f 44 00 00    nopl 0x0(%rax,%rax,1)

0000000004010d0 <exit@plt>:
4010d0: f3 0f 1e fa      endbr64

```

图5.5.4 区别3: 新增了共享库函数

4) hello-asm.txt 中跳转以及函数调用都用的虚拟地址, 例如这里的 printf 和 atoi 函数。

```

26 401182: e8 19 ff ff ff    callq 4010a0 <printf@plt>
27 401187: 48 8b 45 e0       mov -0x20(%rbp),%rax
28 40118b: 48 83 c0 18       add $0x18,%rax
29 40118f: 48 8b 00          mov (%rax),%rax
30 401192: 48 89 c7          mov %rax,%rdi
31 401195: e8 26 ff ff ff    callq 4010c0 <atoi@plt>

```

图5.5.5 区别4: 函数调用、跳转都用了虚拟地址

3、重定位过程流程

要合并相同的节, 确定新节中所有定义符号在虚拟地址空间中的地址, 还要对引用符号进行重定位, 修改.text 节和.data 节中对每个符号的引用。

5.6 hello 的执行流程

1、运行 edb, 打开 hello, 如图所示:

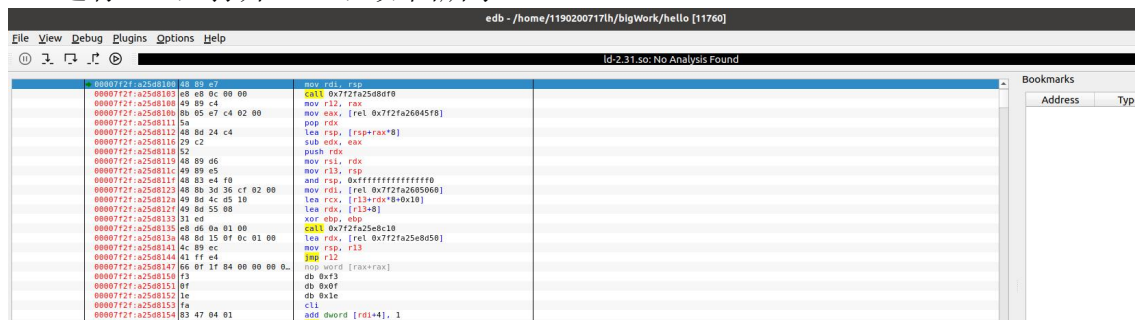


图5.6.1 运行edb

2、列出所有过程

函数名称	函数地址
init	0x401000
.plt	0x401020
puts@plt	0x401090
printf@plt	0x4010a0
getchar@plt	0x4010b0
atoi@plt	0x4010c0

exit@plt	0x4010d0
sleep@plt	0x4010e0
_start	0x4010f0
_dl_relocate_static_pie	0x401120
main	0x401125
_libc_csu_init	0x4011c0
_libc_csu_fini	0x401230
_fini	0x401238

5.7 Hello 的动态链接分析

在 edb 中查找.interp 段的虚拟地址，这里保存了一个字符串，这个字符串就是动态库的地址。



图5.7.1 动态库地址

在 Data dump 中找到此处存放的是/lib64/ld-linux-x86-64.so.2

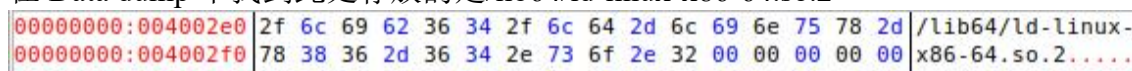


图5.7.2 动态链接用到的库

5.8 本章小结

本章结合实验中的 hello 可执行程序依次介绍了链接的概念及作用以及 Ubuntu 下链接的命令行。然后对 hello 的 elf 格式进行了详细的分析，并对比 hello 的反汇编文件和 hello.o 的反汇编文件，详细了解了重定位的过程。最后遍历了整个 hello 的执行过程，并对 hello 进行了动态链接分析。

(第5章1分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念:

狭义定义: 进程是正在运行的程序的实例

广义定义: 进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元, 在传统的操作系统中, 进程既是基本的分配单元, 也是基本的执行单元。

作用:

- 1、在现代计算机中, 进程为用户提供了以下假象: 我们的程序好像是系统中当前运行的唯一程序 一样, 我们的程序好像是独占的使用处理器和内存, 处理器好像是无间断的执行 我们程序中的指令, 我们程序中的代码和数据好像是系统内存中唯一的对象。
- 2、每次用户通过向 shell 输入一个可执行目标文件的名字, 运行程序时, shell 就会创建一个新的进程, 然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程, 并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。
- 3、进程给每个应用提供了两个非常关键的抽象: 一是逻辑控制流, 二是私有地址空间。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell-bash 的作用:

- 1、管理用户与操作系统之间的交互
- 2、等待用户输入, 向操作系统解释用户的输入
- 3、处理各种各样的操作系统的输出结果

Shell-bash 的处理流程:

- 1、将用户输入的命令行进行解析, 分析是否是内置命令;
- 2、若是内置命令, 直接执行; 若不是内置命令, 则 bash 在初始子进程的上下文中加载和运行它。
- 3、本质上就是 shell 在执行一系列的读和求值的步骤, 在这个过程中, 他同时可以接受来自终端的命令输入。

6.3 Hello 的 fork 进程创建过程

执行中的进程调用 fork() 函数, 就创建了一个子进程。其函数原型为 pid_t

`fork(void)`; 对于返回值, 若成功调用一次则返回两个值, 子进程返回 0, 父进程返回子进程 ID; 否则, 出错返回 -1。

首先对于 `hello` 进程。我们终端的输入被判断为非内置命令, 然后 `shell` 试图在硬盘上查找该命令 (即 `hello` 可执行程序), 并将其调入内存, 然后 `shell` 将其解释为系统功能调用并转交给内核执行。

`shell` 执行 `fork` 函数, 创建一个子进程。这时候我们的 `hello` 程序就开始运行了。值得注意的是, `hello` 子进程是父进程的副本, 它将获得父进程数据空间、堆、栈等资源的副本。但是子进程持有的是上述存储空间的“副本”, 这意味着父子进程间不共享这些存储空间。

同时 `Linux` 将复制父进程的地址空间给予进程, 因此, `hello` 进程就有了独立的地址空间。

6.4 Hello 的 `execve` 过程

`execve` 函数在当前进程的上下文中加载并运行新程序 `hello`。如果成功, 则不返回; 如果错误, 则返回 -1。

在 `execve` 加载了 `hello` 之后, 它调用启动代码。启动代码设置栈, 并将控制传递给 `hello` 的主函数。

`hello` 子进程通过 `execve` 系统调用启动加载器。

加载器删除子进程所有的虚拟地址段, 并创建一组新的代码、数据、堆段。新的栈和堆段被初始化为 0。

通过将虚拟地址空间中的页映射到可执行文件的页大小的片, 新的代码和数据段被初始化为可执行文件中的内容。

最后加载器跳到 `_start` 地址, 它最终调用 `hello` 的 `main` 函数。除了一些头部信息, 在加载过程中没有任何从磁盘到内存的数据复制。直到 CPU 引用一个被映射的虚拟页时才会进行复制, 此时, 操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

6.5 Hello 的进程执行

进程的上下文信息包括通用目的寄存器、浮点寄存器。程序计数器、用户栈、状态寄存器、内核栈和各种数据结构。

在进行程序调度时, 系统保存当前进程的上下文, 载入目标进程的上下文, 并将控制传递到目标程序。这些操作都在内核模式下被执行。

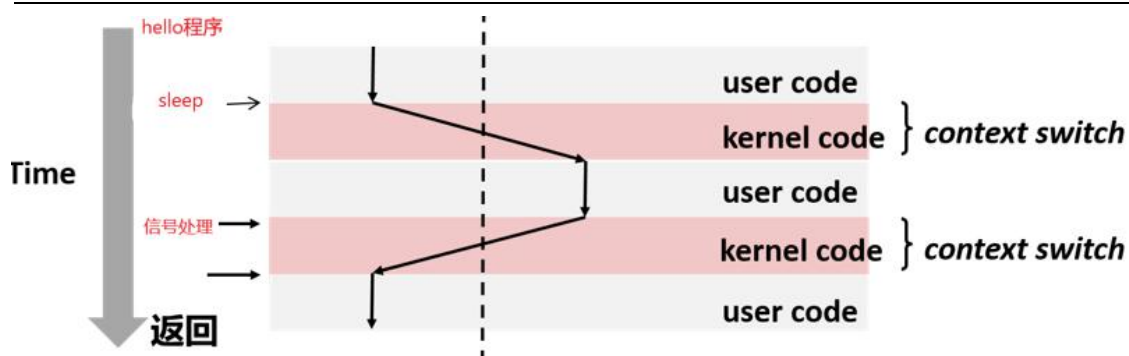


图6.5 CPU在用户态和内核态间的转变

程序调度频率由 CPU 的 clk 时钟周期长度有关，由于 CPU 时钟周期很短，所以会给用户造成一种几个程序同时在进行的感觉。

6.6 hello 的异常与信号处理

异常指的是把控制交给系统内核来响应某些事件（例如处理器状态的变化），其中内核是操作系统常驻内存的一部分，而这类事件包括除以零、数学运算溢出、页错误、I/O 请求完成或用户按下了 ctrl+c 等等系统级别的事件。

异常的分类如图所示：

异常可以分为四类：中断(interrupt)、陷阱(trap)、故障(fault)和终止(abort)。图 8-4 中的表对这些类别的属性做了小结。

类别	原因	异步 / 同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

图 8-4 异常的分类。异步异常是由处理器外部的 I/O 设备中的事件产生的。同步异常是执行一条指令的直接产物

图6.6.1 异常的分类

尝试输入空格，回车或者随便什么数字，如图所示，不影响程序执行。

```
1190200717lh@Graham:~/bigWork$ ./hello 1190200717 梁浩 2
Hello 1190200717 梁浩
Hello 1190200717 梁浩

Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩
123Hello 1190200717 梁浩
```

图6.6.2 空格 回车 乱按

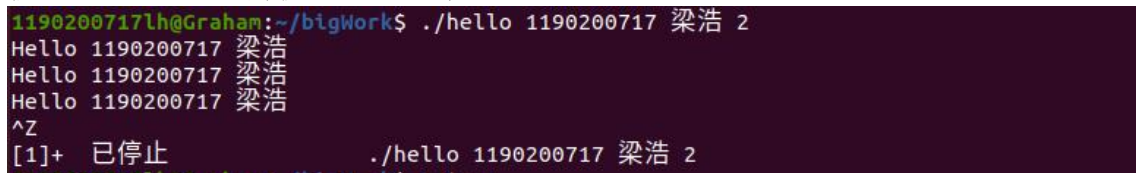
按下ctrl+c，程序接收到SIGSTOP信号，程序终止。



```
1190200717lh@Graham: ~/bigWork
1190200717lh@Graham:~/bigWork$ ./hello 1190200717 梁浩 2
Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩
^C
1190200717lh@Graham:~/bigWork$
```

图6.6.3 ctrl+c导致程序终止

按下ctrl+z，程序会收到SIGINT信号，进程暂停。



```
1190200717lh@Graham:~/bigWork$ ./hello 1190200717 梁浩 2
Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩
^Z
[1]+ 已停止 ./hello 1190200717 梁浩 2
```

图6.6.4 ctrl+z后进程暂停

用ps查看，仍可看到进程。



```
1190200717lh@Graham:~/bigWork$ ps
  PID TTY          TIME CMD
 2686 pts/0    00:00:00 bash
 2692 pts/0    00:00:00 hello
 2695 pts/0    00:00:00 ps
```

图6.6.5 ps查看进程

```

1190200717lh@Graham:~/bigWork$ ./hello 1190200717 梁浩 2
Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩
^Z
[1]+  已停止                  ./hello 1190200717 梁浩 2
1190200717lh@Graham:~/bigWork$ pstree
systemd──ModemManager──2*[{ModemManager}]
        └─NetworkManager──2*[{NetworkManager}]
              └─3*[VBoxClient──VBoxClient──2*[{VBoxClient}]]
                    └─VBoxClient──VBoxClient──3*[{VBoxClient}]
                          └─VBoxService──8*[{VBoxService}]
                                └─accounts-daemon──2*[{accounts-daemon}]
                                      └─acpid
                                            └─anacron
                                                  └─avahi-daemon──avahi-daemon
                                                        └─colord──2*[{colord}]
                                                              └─cron
                                                                    └─cups-browsed──2*[{cups-browsed}]
                                                                          └─cupsd──dbus
                                                                                └─dbus-daemon
                                                                                      └─fwupd──4*[{fwupd}]
                                                                                            └─gdm3──gdm-session-wor──gdm-x-session──Xorg──5*[{Xorg}]
                                                                                                  └─gnome-session-b──ssh-agent
                                                                                                        └─2*[{gnome-+
                                                                                                              └─2*[{gdm-x-session}]
                                                                                                                    └─2*[{gdm-session-wor}]
                                                                                                                          └─2*[{gdm3}]
                                                                                                                                └─gnome-keyring-d──3*[{gnome-keyring-d}]
                                                                                                                                      └─2*[{kerneloops}]
                                                                                                                                            └─networkd-dispat
                                                                                                                                                  └─polkitd──2*[{polkitd}]
                                                                                                                                                        └─rsyslogd──3*[{rsyslogd}]
                                                                                                                                                            └─rtkit-daemon──2*[{rtkit-daemon}]
                                                                                                                                            └─snapd──12*[{snapd}]
                                                                                                                                                └─switcheroo-cont──2*[{switcheroo-cont}]
                                                                                                                                                      └─systemd──(sd-pam)
                                                                                                                                                            └─at-spi-bus-laun──dbus-daemon
                                                                                                                                                                  └─3*[{at-spi-bus-laun}]
                                                                                                                                                                        └─at-spi2-registr──2*[{at-spi2-registr}]
                                                                                                                                                                              └─dbus-daemon
                                                                                                                                                                                    └─dconf-service──2*[{dconf-service}]
                                                                                                                                                                                            └─evolution-addre──5*[{evolution-addre}]
                                                                                                                                                                                                    └─evolution-calen──8*[{evolution-calen}]
                                                                                                                                                                                                            └─evolution-sourc──3*[{evolution-sourc}]
                                                                                                                                                                                                                  └─gjs──4*[{gjs}]
                                                                                                                                                                                                                        └─gnome-session-b──evolution-alarm──5*[{evolution-alarm}]
                                                                                                                                                                                                                              └─gsd-disk-utilit──2*[{gsd-disk-utilit}]
                                                                                                                                                                                                                                    └─update-notificat──3*[{update-notificat}]

```

图6.6.6 pstree查看进程树

输入fg，进程重新在前台工作。

```

1190200717lh@Graham:~/bigWork$ fg
./hello 1190200717 梁浩 2
Hello 1190200717 梁浩
Hello 1190200717 梁浩
Hello 1190200717 梁浩

```

图表6.6.7 fg后进程重新工作

利用kill指令杀死hello进程

```

1190200717lh@Graham:~/bigWork$ ./hello 1190200717 梁浩 2
Hello 1190200717 梁浩
Hello 1190200717 梁浩
^Z
[1]+  已停止                  ./hello 1190200717 梁浩 2
1190200717lh@Graham:~/bigWork$ ps
  PID TTY          TIME CMD
 2788 pts/0    00:00:00 bash
 2794 pts/0    00:00:00 hello
 2795 pts/0    00:00:00 ps
1190200717lh@Graham:~/bigWork$ kill -s 9 2794
1190200717lh@Graham:~/bigWork$ ps
  PID TTY          TIME CMD
 2788 pts/0    00:00:00 bash
 2796 pts/0    00:00:00 ps
[1]+  已杀死                  ./hello 1190200717 梁浩 2
1190200717lh@Graham:~/bigWork$

```

图6.6.8 kill指令杀死进程

信号处理：

- 1、对于ctrl+c或者ctrl+z：键盘键入后，内核就会发送SIGINT或者SIGSTP。SIGINT信号默认终止前台job即程序hello，SIGSTP默认挂起前台hello作业。
- 2、对于fg信号：内核发送SIGCONT信号，我们刚刚挂起的程序hello重新在前台运行。
- 3、对于kill指令：内核发送SIGKILL信号给指定的pid，杀死该进程。

6.7 本章小结

这一章中，我们细致的了解了异常的4种类型，包括中断，故障，终止和陷阱，还有异常控制流的四个基本机制：异常、进程切换、信号和非本地跳转。其中异常是最底层的，也是后面几种的基础；而信号是进程间最重要，简单却强大的信使，用来给进程传送信号。

(第6章1分)

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：

逻辑地址是指由程序 `hello` 产生的与段相关的偏移地址部分。

线性地址：

线性地址是逻辑地址到物理地址变换之间的中间层。程序 `hello` 的代码会产生逻辑地址，或者说是段中的偏移地址，它加上相应段的基地址就生成了一个线性地址。

虚拟地址：

我们也把逻辑地址称为虚拟地址。因为与虚拟内存空间的概念类似，逻辑地址也是与实际物理内存容量无关的，是 `hello` 中的虚拟地址。

物理地址：

物理地址是指出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么 `hello` 的线性地址会使用页目录和页表中的项变换成 `hello` 的物理地址；如果没有启用分页机制，那么 `hello` 的线性地址就直接成为物理地址了。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部分组成，段标识符，段内偏移量。段标识符是一个 16 位长的字段组成，称为段选择符，其中前 13 位是一个索引号。后面三位包含一些硬件细节。索引号可以直接理解成数组下标，它对应的“数组”就是段描述符表，段描述符具体描述了一个段地址，这样，很多段描述符就组成段描述符表。可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段。我们只关心 `Base` 字段，它描述了一个段的开始位置的线性地址。

逻辑地址转化的过程如下：

首先给定一个完整的逻辑地址[段选择符：段内偏移地址]

1、看段选择描述符中的 `T1` 字段是 0 还是 1，可以知道当前要转换的是 `GDT` 中的段，还是 `LDT` 中的段，再根据指定的相应的寄存器，得到其地址和大小，我们就有了一个数组了。

2、拿出段选择符中的前 13 位，可以在这个数组中查找到对应的段描述符，这样就有了 `Base`，即基地址就知道了。

3、把基地址 `Base+Offset`，就是要转换的下一个阶段的物理地址。

7.3 Hello 的线性地址到物理地址的变换-页式管理

下图展示了 MMU 如何利用页表完成虚拟地址到物理地址的映射：

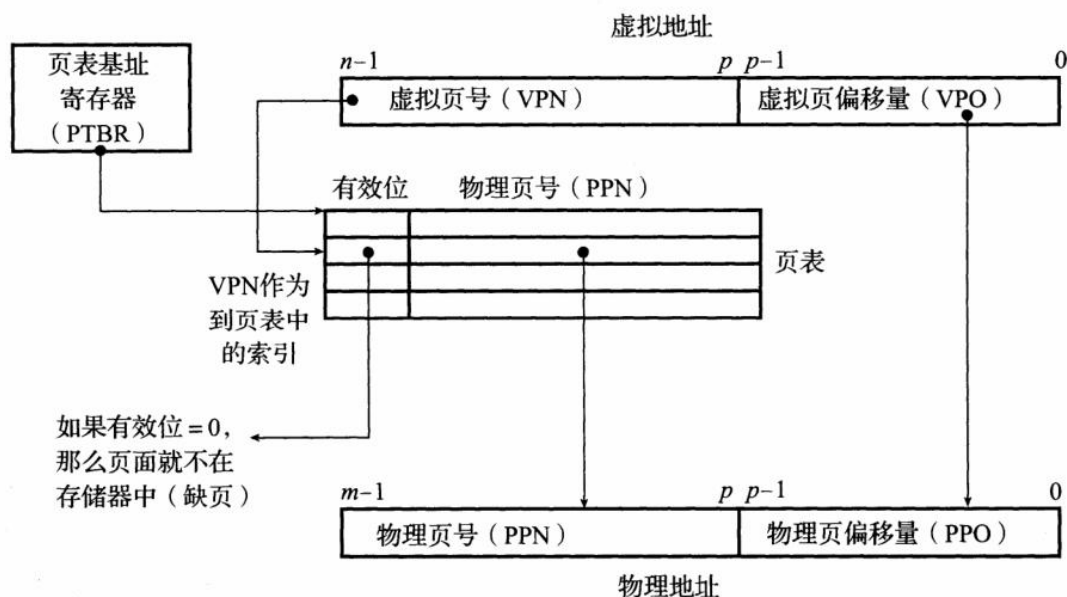


图7.3 使用页表的地址翻译

由图所示，虚拟地址被分为虚拟页号（VPN）与虚拟页偏移量（VPO）。CPU 取出虚拟页号，通过页表基址寄存器（PTBR）来定位页表条目，有效位为 1 时，从页表条目中取出物理页号（PPN），通过将物理页号与虚拟页偏移量（VPO）结合，得到由物理地址（PPN）和物理页偏移量（PPO）组合的物理地址。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

下图给出了第一级、第二级或第三级页表中条目的格式。当 $P=1$ 时 (Linux 中总是如此)，地址字段包含一个 40 位物理页号 (PPN)，它指向适当的页表的开始处。注意，这强加了一个要求，要求物理页表 4KB 对齐。

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	未使用	页表物理基地址				未使用	G	PS		A	CD	WT	U/S	R/W	P=1
OS 可用（磁盘上的页表位置）															P=0

字段	描述
P	子页表在物理内存中（1），不在（0）
R/W	对于所有可访问页，只读或者读写访问权限
U/S	对于所有可访问页，用户或超级用户（内核）模式访问权限
WT	子页表的直写或写回缓存策略
CD	能 / 不能缓存子页表
A	引用位（由 MMU 在读和写时设置，由软件清除）
PS	页大小为 4 KB 或 4 MB（只对第一层 PTE 定义）
Base addr	子页表的物理基地址的最高 40 位
XD	能 / 不能从这个 PTE 可访问的所有页中取指令

图7.4.1 第一级、第二级和第三级页表条目格式

下图给出了第四级页表中条目的格式。当 P=1，地址字段包括一个 40 位 PPN，它指向物理内存中某一页的基地址，这又强加了一个要求，要求物理页 4KB 对齐。

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	未使用	页表物理基地址				未使用	G	0	D	A	CD	WT	U/S	R/W	P=1
OS 可用（磁盘上的页表位置）															P=0

字段	描述
P	子页表在物理内存中（1），不在（0）
R/W	对于子页，只读或者读写访问权限
U/S	对于子页，用户或超级用户（内核）模式访问权限
WT	子页的直写或写回缓存策略
CD	能 / 不能缓存
A	引用位（由 MMU 在读和写时设置，由软件清除）
D	修改位（由 MMU 在读和写时设置，由软件清除）
G	全局页（在任务切换时，不从 TLB 中驱逐出去）
Base addr	子页物理基地址的最高 40 位
XD	能 / 不能从这个子页中取指令

图7.4.2 第四级页表条目格式

PTE 有三个权限位,控制对页的访问。R/W 位确定页的内容是可以读写的还是只读

的。U/S 位确定是否能够在用户模式中访问该页,从而保护操作系统内核中的代码和数据。

当 MMU 翻译每一个虚拟地址时, 它还会更新另外两个内核缺页处理程序会用到的位。每次访问一个页时, MMU 都会设置 A 位, 称为引用位。内核可以用这个引用位来实现它的页替换算法。每次对一个页进行了写之后, MMU 都会设置 D 位, 又称修改位或脏位(dirty bit)。修改位告诉内核在复制替换页之前是否必须写回牺牲页。内核可以通过调用一条特殊的内核模式指令来清除引用位或修改位。

下图给出了 Core i7 MMU 如何使用四级的页表来将虚拟地址翻译成物理地址。36 位 VPN 被划分成四个 9 位的片, 每个片被用作到一个页表的偏移量。CR3 寄存器包含 L1 页表的物理地址。VPN1 提供到一个 L1 PTE 的偏移量,这个 PTE 包含 L2 页表的基地址。VPN2 提供到一个 L2 PTE 的偏移量, 以此类推。

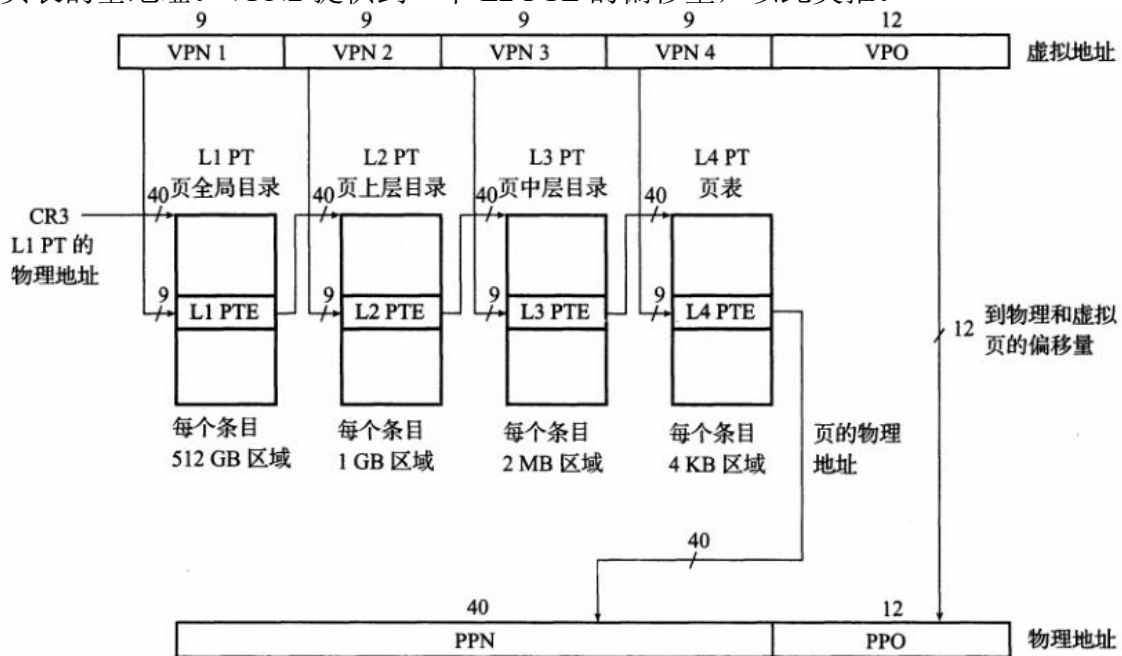


图7.4.3 Corei7 页表翻译

7.5 三级 Cache 支持下的物理内存访问

当 CPU 提出访存请求后给出物理地址, 然后高速缓存根据 CI 找到缓存组, 在缓存组中根据 CT 与缓存行中的标记位进行匹配, 如果匹配成功并且有效位为 1, 为命中, 则按照块偏移对数据块中的数据进行访问, 否则为不命中, 向下一级缓存中寻找数据, 如果找到, 则按照放置策略和替换策略替换该级缓存中的缓存块, 否则继续向下一级缓存或主存中寻找数据。

7.6 hello 进程 fork 时的内存映射

虚拟内存和内存映射解释了 `fork` 函数如何为 `hello` 进程提供私有的虚拟地址空间。`fork` 为 `hello` 的进程创建虚拟内存，创建当前进程的 `mm_struct`, `vm_area_struct` 和页表的原样副本；两个进程中的每个页面都标记为只读；两个进程中的每个区域结构都标记为私有的写时复制。在 `hello` 进程中返回时，`hello` 进程拥有与调用 `fork` 进程相同的虚拟内存，随后的写操作通过写时复制机制创建新页面。

7.7 `hello` 进程 `execve` 时的内存映射

`execve` 函数在当前进程中加载并运行程序 `hello` 的步骤：

- 1、删除已存在的用户区域
- 2、映射私有区域
- 3、映射共享区域：`hello` 程序与共享对象（或目标）链接，比如标准 C 库 `libc.so`，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。
- 4、设置程序计数器（PC）：设置当前进程的上下文中的程序计数器，是指指向代码区域的入口点。而下一次调度这个进程时，它将从这个入口点开始执行。

7.8 缺页故障与缺页中断处理

DRAM 缓存不命中称为缺页，即虚拟内存中的字不在物理内存中。缺页导致页面出错，产生缺页异常。缺页异常处理程序选择一个牺牲页，然后将目标页加载到物理内存中。最后让导致缺页的指令重新启动，页面命中。

下面是整体的处理流程：

- 1、处理器生成一个虚拟地址，并将它传送给 MMU
- 2、MMU 生成 PTE 地址，并从高速缓存/主存请求得到它
- 3、高速缓存/主存向 MMU 返回 PTE
- 4、PTE 中的有效位是 0，所以 MMU 出发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。
- 5、缺页处理程序确认出物理内存中的牺牲页，如果这个页已经被修改了，则把它换到磁盘。
- 6、缺页处理程序页面调入新的页面，并更新内存中的 PTE
- 7、缺页处理程序返回到原来的进程，再次执行导致缺页的命令。CPU 将引起缺页的虚拟地址重新发送给 MMU。因为虚拟页面已经换存在物理内存中，所以就会命中。

7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆(heap)。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长(向更高的地址)。对于每个进程,内核维护着一个变量 brk(读做“break”),它指向堆的顶部。

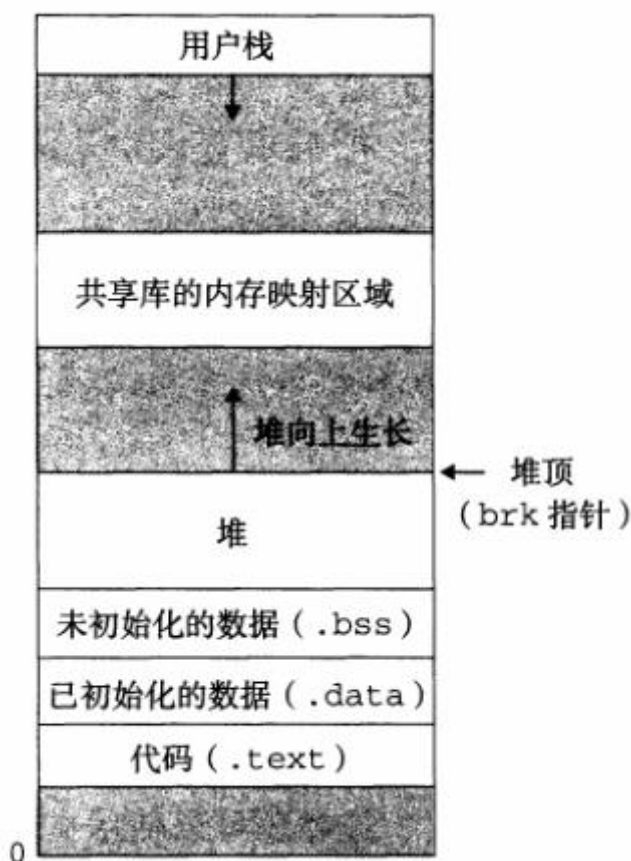


图7.9.1 堆

分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk),要么是已分配的,要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲,直到它显式地被应用所分配。一个已分配的块保持已分配状态,直到它被释放,这种释放要么是应用程序显式执行的,要么是内存分配器自身隐式执行的。

分配器有两种基本风格。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

- 1、显式分配器：要求应用显式地释放任何已分配的块。
- 2、隐式分配器：另一方面，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配的块的过程叫做垃圾收集。例如，诸如 Lisp、ML 以及 Java 之类的高级语言就依

赖垃圾收集来释放已分配的块。

带边界标签的隐式空闲链表分配器管理：

带边界标记的隐式空闲链表的每个块是由一个字的头部、有效载荷、可能的额外填充以及一个字的尾部组成的。在隐式空闲链表中，因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。其中，一个设置了已分配的位而大小为零的终止头部将作为特殊标记的结束块。当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。



图7.9.2 使用边界标记的堆块格式

7.10 本章小结

通过本章，我们阐述了 `hello` 的存储器地址空间、`intel` 的段式管理、`hello` 的页式管理，TLB 与四级页表支持下的 VA 到 PA 的变换、三级 `cache` 下的物理内存访问，还介绍了 `hello` 进程 `fork` 时的内存映射、`execve` 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理等内容。学习到了虚拟内存是对主存的一个抽象。处理器产生一个虚拟地址，在被发送到主存之前，这个地址被翻译成一个物理地址。从虚拟地址空间到物理地址空间的地址翻译要求硬件和软件紧密合作。专门的硬件使用页表来翻译虚拟地址。

(第7章 2分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个字节的序列，所有的 I/O 设备(例如网络，磁盘和终端)都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O，这使得所有的输入和输出都能以一种统一且一致的方式来执行。我们可以对文件的操作有：打开关闭操作 `open` 和 `close`；读写操作 `read` 和 `write`；改变当前文件位置 `lseek` 等

8.2 简述 Unix IO 接口及其函数

接口：

- 1、打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备，内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息，应用程序只需要记住这个描述符。
- 2、linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）。头文件 `<unistd.h>` 定义了常量 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，它们可用来代替显式的描述符值。
- 3、改变当前的文件位置：对于每个打开的文件，内核保持着一个文件位置 `k`，初始为 0，这个文件位置是从文件开头起始的字节偏移量，应用程序能够通过执行 `seek`，显式地将改变当前文件位置 `k`。
- 4、读写文件。一个读操作就是从文件复制 $n>0$ 个字节到内存，从当前文件位置 `k` 开始，然后将 `k` 增加到 `k+n`。给定一个大小为 `m` 字节的文件，当 $k \sim m$ 时执行读操作会触发一个称为 `end-of-file(EOF)` 的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF 符号”。类似地，写操作就是从内存复制 $n>0$ 个字节到一个文件，从当前文件位置 `k` 开始，然后更新 `k`。
- 5、关闭文件。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

函数：

- 1.打开和关闭文件。

打开文件函数原型：`int open(char* filename,int flags,mode_t mode)`

返回值：若成功则为新文件描述符，否则返回-1；

flags: O_RDONLY（只读），O_WRONLY（只写），O_RDWR（可读写）

mode: 指定新文件的访问权限位。

关闭文件函数原型：int close(fd)

返回值：成功返回 0，否则为-1

2. 读和写文件

读文件函数原型：ssize_t read(int fd,void *buf,size_t n)

返回值：成功则返回读的字节数，若 EOF 则为 0，出错为-1

描述：从描述符为 fd 的当前文件位置复制最多 n 个字节到内存位置 buf

写文件函数原型：ssize_t write(int fd,const void *buf,size_t n)

返回值：成功则返回写的字节数，出错则为-1

描述：从内存位置 buf 复制至多 n 个字节到描述符为 fd 的当前文件位置

8.3 printf 的实现分析

先看 printf 函数的函数体：

```

1. int printf(const char *fmt, ...)
2. {
3.     int i;
4.     char buf[256];
5.
6.     va_list arg = (va_list)((char*)&fmt + 4);
7.     i = vsprintf(buf, fmt, arg);
8.     write(buf, i);
9.
10.    return i;
11. }
```

发现里面还调用了函数 vsprintf，在查看一下 vsprintf 的函数实现

```

1. int vsprintf(char *buf, const char *fmt, va_list args)
2. {
3.     char* p;
4.     char tmp[256];
5.     va_list p_next_arg = args;
6.
7.     for (p=buf;*fmt;*fmt++) {
8.         if (*fmt != '%') {
9.             *p++ = *fmt;
10.            continue;
11.        }
```

```
12.
13.     fmt++;
14.
15.     switch (*fmt) {
16.     case 'x':
17.         itoa(tmp, *((int*)p_next_arg));
18.         strcpy(p, tmp);
19.         p_next_arg += 4;
20.         p += strlen(tmp);
21.         break;
22.     case 's':
23.         break;
24.     default:
25.         break;
26.     }
27. }
28.
29.     return (p - buf);
30. }
```

系统函数 write:

```
1.  write:
2.      mov eax, _NR_write
3.      mov ebx, [esp + 4]
4.      mov ecx, [esp + 8]
5.      int INT_VECTOR_SYS_CALL
```

发现 write 里调用了 syscall, 查看 syscall:

```
1.  sys_call:
2.      call save
3.      push dword [p_proc_ready]
4.      sti
5.      push ecx
6.      push ebx
7.      call [sys_call_table + eax * 4]
8.      add esp, 4 * 3
9.      mov [esi + EAXREG - P_STACKBASE], eax
10.     cli
11.     ret
```

分析得知:

printf 函数: 接受一个 `fmt` 的格式, 然后将匹配到的参数按照 `fmt` 格式输出。`printf` 用了两个外部函数, 一个是 `vsprintf`, 还有一个是 `write`。

vsprintf 函数: 接受确定输出格式的格式字符串 `fmt` (输入)。用格式字符串对个数变化的参数进行格式化, 产生格式化输出。

`write` 函数：将 `buf` 中的 `i` 个元素写到终端。

从 `vsprintf` 生成显示信息，到 `write` 系统函数，到陷阱-系统调用 `int 0x80` 或 `syscall`. 字符显示驱动子程序：从 ASCII 到字模库到显示 `vram`（存储每一个点的 RGB 颜色信息）。

显示芯片按照刷新频率逐行读取 `vram`，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

8.4 getchar 的实现分析

当程序调用 `getchar` 时，程序就等着用户按键，用户输入的字符被存放在键盘缓冲区中直到用户按回车为止，回车字符也放在缓冲区中。当用户键入回车之后，`getchar` 才开始从 `stdio` 流中每次读入一个字符。`getchar` 函数的返回值是用户输入的第一个字符的 ASCII 码，如出错返回 -1，且将用户输入的字符回显到屏幕。如用户在按回车之前输入了不止一个字符，其他字符会保留在键盘缓存区中，等待后续 `getchar` 调用读取。

异步异常-键盘中断的处理：键盘中断处理子程序。接受按键扫描码转成 `ascii` 码，保存到系统的键盘缓冲区。

`getchar` 调用 `read` 系统函数，通过系统调用读取按键 `ascii` 码，直到接受到回车键才返回。

8.5 本章小结

本章主要介绍了 Linux 的 I/O 设备管理方法、Unix I/O 接口及其函数，分析了 `printf` 函数和 `getchar` 函数的实现。

（第 8 章 1 分）

结论

`hello.c` 开始是用户编写的 C 语言源程序，之后它经历以下的过程：

- 1、`hello.c` 经过预编译，拓展得到 `hello.i` 文本文件
- 2、`hello.i` 经过编译，得到汇编代码 `hello.s` 汇编文件
- 3、`hello.s` 经过汇编，得到二进制可重定位目标文件 `hello.o`
- 4、`hello.o` 经过链接，生成了可执行文件 `hello`
- 5、在 shell 利用 `./hello` 运行 `hello` 程序，父进程通过 `fork` 函数为 `hello` 创建进程
- 6、由 `execve` 函数加载运行当前进程的上下文中加载并运行新程序 `hello`
- 7、内存管理单元 MMU、TLB、多级页表机制、三级 `cache` 完成对 PA 物理地址的

8、异常处理机制实现了对异常信号的处理

9、hello 最终被 shell 父进程回收，内核会收回为其创建的所有信息

感悟：

计算机真是非常巧妙的存在，内部的设计复杂且充满智慧。我现在的水平还浮于表面，对于底层的掌握还不是特别到位，在以后的学习中不能仅仅停留在高级语言层次，还应该深入到底层代码中。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

列出所有的中间产物的文件名，并予以说明起作用。

生成的中间结果文件名	文件的作用
hello.i	hello.c 预处理后得到的文件
hello.s	hello.i 编译后得到的文件
hello.o	hello.s 汇编后的可重定位目标文件
hello-elf.txt	hello.o 的 elf 格式文件
hello	hello.o 经过链接后的可执行目标文件
hello-out-elf.txt	hello 的 elf 格式文件
hello-asm.txt	hello 的反汇编代码文件
hello-o-asm.txt	hello.o 的反汇编代码文件

(附件 0 分，缺失 -1 分)

参考文献

为完成本次大作业你翻阅的书籍与网站等

[1] 深入理解计算机系统

(参考文献 0 分，缺失 -1 分)