

哈爾濱工業大學

题 目 从 OS 角度漫游 Hello

姓 名 汤昕润

指 导 教 师 郑铁然

2025 年 1 月

摘 要

本文详细探讨了在操作系统视角下运行一个简单的“Hello World”程序的执行过程，分析了其中涉及的各项操作系统机制。通过逐步跟踪程序的生命周期，从用户输入命令到进程的创建、程序加载、内存管理、IO 操作、缓冲区管理、缺页中断、进程退出等多个关键环节，深入讲解了操作系统在各个阶段如何高效地调度与管理计算机硬件资源。本文不仅帮助理解操作系统内部的工作原理，还为进一步优化内核性能和编写高效程序提供了理论基础。

关键词：操作系统；程序生命周期；Linux；Hello 程序

第 1 章 概述

1.1 Hello 简介

“Hello, World!” 程序是许多编程语言教程中的经典入门程序，它的目的只是将字符串 "Hello, World!" 输出到终端。尽管它的功能非常简单，但这个程序却是了解编程语言、操作系统以及开发工具链的重要起点。

```
1. #include <stdio.h>
2.
3. int main() {
4.     printf("Hello, World!\n");
5.     return 0;
6. }
7.
```

程序的结构非常简单，包括以下几个部分：

- **#include <stdio.h>:** 这行代码包含了标准输入输出库（Standard Input Output），它定义了程序中用来进行输入和输出的函数，如 `printf()`。
- **int main():** 这是程序的入口点，操作系统在程序启动时会首先执行 `main()` 函数。它是所有 C 程序的基础结构。
- **printf("Hello, World!\n");:** `printf` 函数用于将文本输出到终端屏幕。"`\n`" 是换行符，确保输出后光标跳到下一行。
- **return 0;:** 表示程序成功结束。返回值 0 是一个约定，表示程序没有错误地完成执行。

尽管这段代码非常简短，它涵盖了程序运行时的一系列重要步骤，包括如何通过标准库函数进行输出、程序的入口点、以及程序的结束处理。

1.2 程序执行的基本流程

程序从输入命令到执行完毕，涉及操作系统的多个环节。我们可以从操作系统的

角度分析执行 Hello World 程序时所经历的步骤：

1. 编译源代码：

首先，C 语言的源代码需要经过编译器（如 gcc）编译为机器代码。编译过程将源代码转化为中间代码（通常是 .o 文件），然后通过链接器将所有必要的库（如 libc）链接起来，生成一个可执行文件（通常是 hello_world）。

2. 操作系统的加载：

用户在命令行输入 ./hello_world 后，操作系统首先创建一个进程来执行该程序。

操作系统会为这个进程分配内存空间，加载程序的执行文件（包括程序的代码段、数据段、堆栈等）到内存中。

3. 进程创建：

操作系统调用 fork() 创建一个新的子进程（通常由 Shell 进程触发），然后用 exec() 将当前进程的内容替换为 hello_world 程序的内容。

此时，hello_world 程序就被加载进内存，并开始执行。

4. 程序执行：

CPU 开始执行程序机器指令，首先从 main() 函数开始，执行程序中的指令。

当程序执行到 printf("Hello, World!\n"); 语句时，操作系统会通过标准库函数 printf() 来进行输出。printf() 会将数据写入标准输出缓冲区。

5. 输出缓冲与显示：

程序通过标准输出（stdout）将 "Hello, World!" 字符串写入缓冲区。当遇到换行符（\n）时，缓冲区会被刷新，并将内容输出到终端显示屏上。

6. 程序退出：

程序执行完 main() 函数后，通过 return 0; 退出。操作系统回收该进程的资源，清理内存和打开的文件描述符等。

操作系统通过 exit() 系统调用结束进程，返回给父进程或 Shell，完成整个程序的执行流程。

1.3 Hello 的 P2P (Program to Process) 过程

Hello 的 P2P (Program to Process) 过程是指从程序到进程的转变过程。首先, 编写 `hello.c` 源文件, 然后经过预处理、编译、汇编 (和链接的过程, 生成可执行文件。具体过程如下:

1. 预处理阶段: 预处理器(`cpp`)根据以字符`#`开头的命令,修改原始的 C 程序。比如 `hello.c` 中第 1 行的`#include <stdio.h>`命令告诉预处理器读取系统头文件 `stdio.h` 的内容,并把它直接插入程序文本中。结果就得到了另一个 C 程序, 通常是以`.i` 作为文件扩展名。

2. 编译阶段: 编译器 (`ccl`) 将文本文件 `hello.i` 翻译成文本文件 `hello.s`, 他包含一个汇编语言程序。该程序包含函数 `main` 的定义, 如下所示:

```
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    $5, -20(%rbp)
25     je      .L2
26     leaq    .LC0(%rip), %rax
27     movq    %rax, %rdi
28     call    puts@PLT
29     movl    $1, %edi
30     call    exit@PLT
```

图 1.1.1 编译程序

定义中 14~30 行的每一条语句都以一种文本格式描述了一条低级机器语言指令。汇编语言是非常有用的, 因为它以不同高级语言的不同编译器提供了一种通用的输出语言。

3. 汇编阶段: 接下来, 汇编器(`as`)将 `hello`, 翻译成机器语言指令, 把这些指令打包成一种叫做可重定位目标程序(`relocatable object program`)的格式, 并将结果保存在目标文件 `hello.o` 中。`hello.o` 文件是一个二进制文件, 它包含的 17 个字节是函数 `main` 的指令编码。如果我们在文本编辑器中打开 `hello.o` 文件, 将看到

一堆乱码。

4. 链接阶段：请注意，`hello` 程序调用了 `printf` 函数，它是每个 C 编译器都提供的标准 C 库中的一个函数。`printf` 函数存在于一个名为 `printf.o` 的单独的预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的 `hello.o` 程序中。链接器(`ld`)就负责处理这种合并。结果就得到 `hello` 文件，它是一个可执行目标文件(或者简称为可执行文件)，可以被加载到内存中，由系统执行。

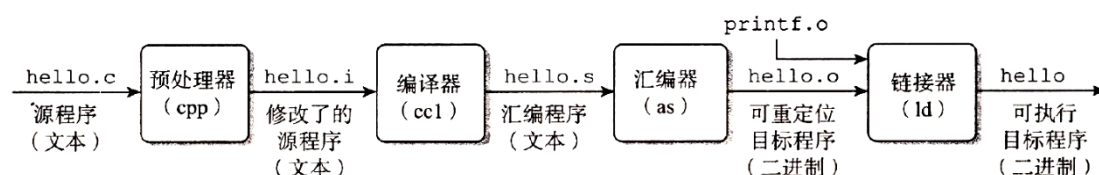


图 1.1.2 编译系统

1.4 Hello 的 020 (From Zero to Zero) 过程

Hello 的 020 (From Zero to Zero) 过程是从无到有再到无的。指的是用 `fork()` 函数在 `shell` 中创建子进程，再用 `exeve` 语句加载可执行程序 `hello`，此时，操作系统为其分配虚拟内存，映射到物理内存，完成了从无到有。内存管理器和中央处理器在执行过程中调用三级 `cache`、快表 (TLB)、内存等行物理内存上的取数据操作，再通过 I/O 根据代码指令输出。程序执行完后，对其进行回收，操作系统内核把它从操作系统清除，完成了从有到无。这就是整个 020 的过程。

1.2 环境与工具

1.2.1 硬件环境

X64 CPU; 2.6GHz; 16G RAM; 512GHD Disk

1.2.2 软件环境

Windows11 64 位; Vmware16; Ubuntu 22.04

1.2.3 开发工具

Visual Studio 2022 64 位 ,Code Blocks 64 位, vi/vim/gedit+gcc

1.3 中间结果

表 1 程序运行中间结果

文件名	作用
hello.i	预处理 hello.c 生成的文件
hello.s	对 hello.i 进行编译生成的文件
hello.o	对 hello.s 进行汇编生成的文件
hello.elf.txt	对 hello.o 进行 readelf 生成的文件
asm.txt	对 hello 进行 objdump 反汇编生成的文件
hello	可执行目标文件
asm1.txt	对 hello 进行 objdump 反汇编生成的文件
hello.txt	对 hello 进行 readelf 生成的文件

第 2 章 P2P

2.1 预处理

1. **概念：**预处理也叫做预编译，是指在正式开始编译前的操作。预处理器（`cpp`）根据以字符`#`开头的命令，修改原始的 C 程序，读取系统头文件中的内容并把它直接插入程序文本，获得另一个文件后缀为`.i`的文件。

2. **作用：**

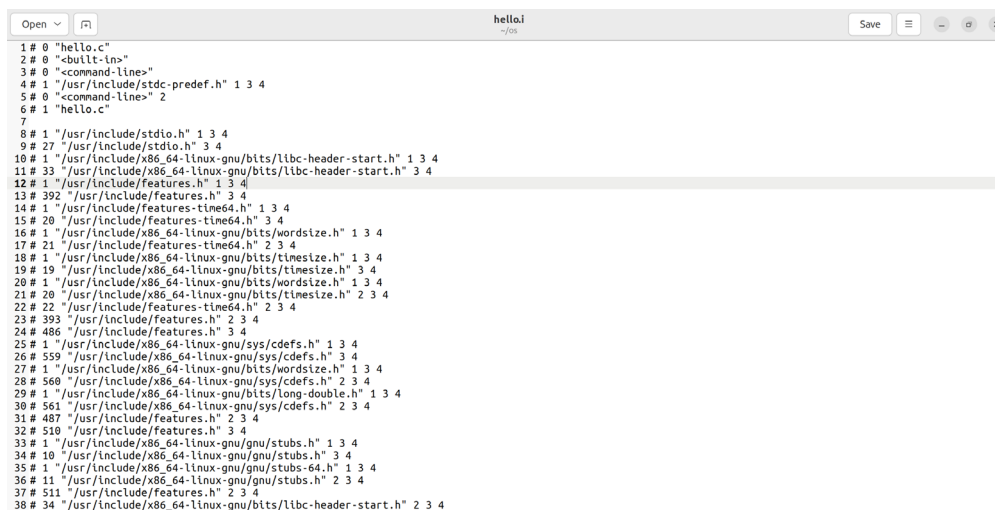
- (1) 能够完成头文件的包含，将包含的文件插入到程序文本中；
- (2) 可以进行宏替换，把它的符号用实际存在的常量加以替换；
- (3) 删除注释部分；
- (4) 实现特殊控制指令（如`#error`）；
- (5) 选择符合条件的代码送至编译器编译，完成条件编译，有选择地执行相关

3. 在 Ubuntu 下预处理的命令

预处理的命令为：`gcc -m64 -no-pie -fno-PIC -E hello.c -o hello.i`



生成的 `hello.i` 文件截图：



打开 `hello.c` 和 `hello.i` 程序，注意到预处理文件（`hello.i`）最后的代码段与原来

的 hello.c 源程序的代码是相同的，即预处理没有对代码段进行处理。并且 hello 的预处理结果 hello.i 共有 745 行,远多于 hello.c。同时我们发现 hello.i 文件中删除了.c 文件中原本的注释部分，也即预处理会删除注释部分。(对比如下图)

```
738 # 3 "hello.c" 2
739
740
741 # 4 "hello.c"
742 int main() {
743     printf("Hello, World!\n");
744     return 0;
745 }
```

```
1 //汤昕润os必胜队源代码
2 #include <stdio.h>
3
4 int main() {
5     printf("Hello, World!\n");
6     return 0;
7 }
8
```

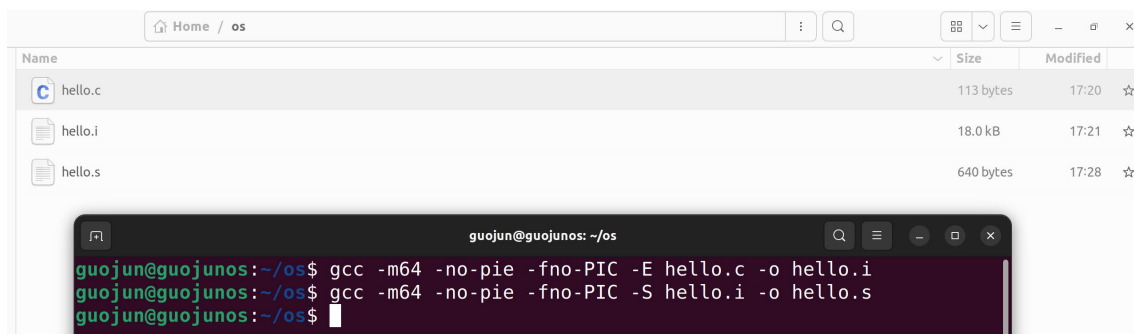
2.2 编译

1. **概念：**编译是把源程序直接变为机器可识别的目标代码。（将源程序一次性转换成目标代码的过程）。执行编译过程的程序叫做编译器。

2. **作用：**把适合人阅读的高级程序代码，转变成机器易于阅读的机器语言。同时人也可以读懂机器语言，从而对程序进行性能上的提升。

3. 在 Ubuntu 下编译的命令

命令：gcc -m64 -no-pie -fno-PIC -S hello.i -o hello.s



hello.s 部分代码展示：

```
hello.s
~/os
Save
hello.i x hello.c x hello.s x
1 .file "hello.c"
2 .text
3 .section .rodata
4 .LC0:
5 .string "Hello, World!"
6 .text
7 .globl main
8 .type main, @function
9 main:
10 .LFB0:
11 .cfi_startproc
12 endbr64
13 pushq %rbp
14 .cfi_def_cfa_offset 16
15 .cfi_offset 6, -16
16 movq %rsp, %rbp
17 .cfi_def_cfa_register 6
18 movl $.LC0, %edi
19 call puts
20 movl $0, %eax
21 popq %rbp
22 .cfi_def_cfa 7, 8
23 ret
24 .cfi_endproc
25 .LFE0:
26 .size main, .-main
27 .ident "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
28 .section .note.gnu-stack,"",@progbits
29 .section .note.gnu.property,"a"
30 .align 8
31 .long 1f - 0f
32 .long 4f - 1f
33 .long 5
34 0:
35 .string "GNU"
```

4. 编译结果解析:

(1). 文件信息

```
1 .file "hello.c"
```

这是汇编文件的源信息，指明此汇编文件是由 `hello.c` 源代码生成的。

(2). 只读数据段

```
3 .section .rodata
4 .LC0:
5 .string "Hello, World!"
```

.section .rodata: 定义一个只读数据段，存储程序中的常量字符串。

.LC0: 是常量字符串 "Hello, World!" 的标签（标识符），供程序中引用。在后面的代码中，程序会使用 `movl $.LC0, %edi` 将字符串地址加载到寄存器 `%edi` 中。

(3). 全局入口函数

```
6 .text
7 .globl main
8 .type main, @function
9 main:
```

.text: 指明以下代码属于程序的代码段（可执行段）。

.globl main: 将 `main` 声明为全局函数，供链接器识别。

.type main, @function: 标识 `main` 是一个函数。

main:: 定义 main 函数的入口点，汇编程序的控制流从这里开始。

(4) . 函数栈帧设置

```
10 .LFB0:
11     .cfi_startproc
12     endbr64
13     pushq    %rbp
14     .cfi_def_cfa_offset 16
15     .cfi_offset 6, -16
16     movq     %rsp, %rbp
17     .cfi_def_cfa_register 6
```

(4.1) 栈帧初始化:

pushq %rbp: 将调用者的栈帧指针 (%rbp) 保存到栈中。保存调用者的栈帧是为了在函数返回时还原调用者的环境。

movq %rsp, %rbp: 将当前栈指针 (%rsp) 的值赋值给基址指针 (%rbp)，为当前函数创建一个新的栈帧。

(4.2) CFI (Call Frame Information) 指令:

.cfi_startproc: 标记函数开始，供调试器和异常处理器使用。

.cfi_def_cfa_offset 16: 定义栈帧的 Canonical Frame Address (CFA) 偏移量为 16。

.cfi_offset 6, -16: 记录寄存器 %rbp 在栈中的偏移量为 -16。

.cfi_def_cfa_register 6: 将栈帧寄存器定义为 %rbp。

功能总结：初始化函数的栈帧，为后续局部变量的操作提供基础。

(5) . 函数主体

```
18     movl     $.LC0, %edi
19     call     puts
20     movl     $0, %eax
```

赋值操作: **movl \$.LC0, %edi:** 将只读数据段中 "Hello, World!" 的地址 (标签 .LC0) 赋值给寄存器 %edi。%edi 是传递给 puts 函数的第一个参数的寄存器。

函数调用: **call puts:** 调用标准库函数 puts，将字符串 Hello, World! 输出到标准输出 (终端)。这里调用的是 puts，因为 GCC 编译器可能将 printf 的简单调用优化为 puts。

返回值设置: **movl \$0, %eax**: 将整数值 0 赋值给寄存器 `%eax`。在 x86-64 系统中, 函数返回值通常存储在 `%rax` 或 `%eax` 中。这里 0 表示程序正常退出。

(6). 函数栈帧清理与返回

```
21      popq    %rbp
22      .cfi_def_cfa 7, 8
23      ret
```

栈帧还原: **popq %rbp**: 从栈中弹出保存的调用者栈帧指针, 恢复调用者的栈帧。

返回指令: **ret**: 返回到调用者程序的地址 (从栈顶弹出返回地址并跳转)。

(7). 调试器信息

```
24      .cfi_endproc
25 .LFE0:
26      .size   main, .-main
27      .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
28      .section .note.GNU-stack,"",@progbits
29      .section .note.gnu.property,"a"
```

CFI 结束信息: **.cfi_endproc**: 标记函数栈帧信息的结束。

调试信息: **.ident "GCC: ..."**: 说明程序是由 GCC 编译器编译的, 并标记编译器版本。

编译选项和段信息: **.section .note.GNU-stack**: 声明当前程序未启用可执行栈。**.section .note.gnu.property**: 记录程序的一些元信息。

(8). 关键点解析

根据分析, `hello.s` 的主要操作如下:

赋值操作

指令: **movl \$.LC0, %edi**

将字符串地址赋值到 `%edi` 中, 为 `puts` 函数调用提供参数。

函数调用

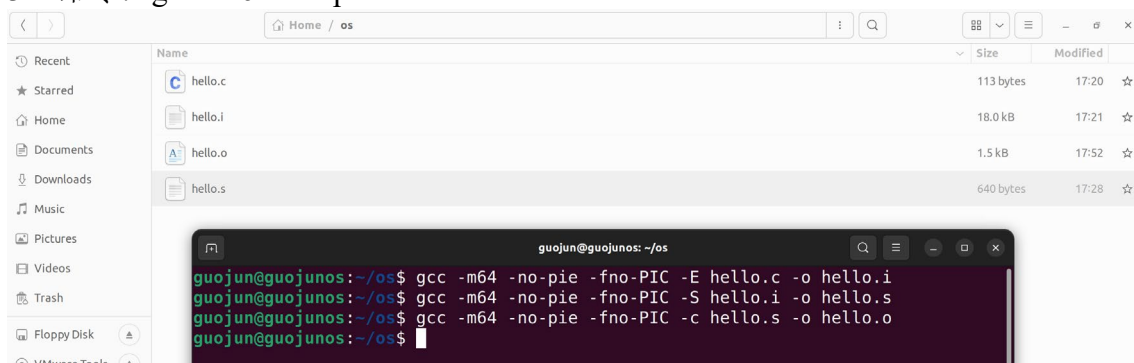
指令: **call puts** 调用 `puts` 函数, 将字符串 "Hello, World!" 输出到标准输出。函数返回

指令: **movl \$0, %eax** 和 **ret**

设置返回值为 0 (正常退出), 然后返回调用者。

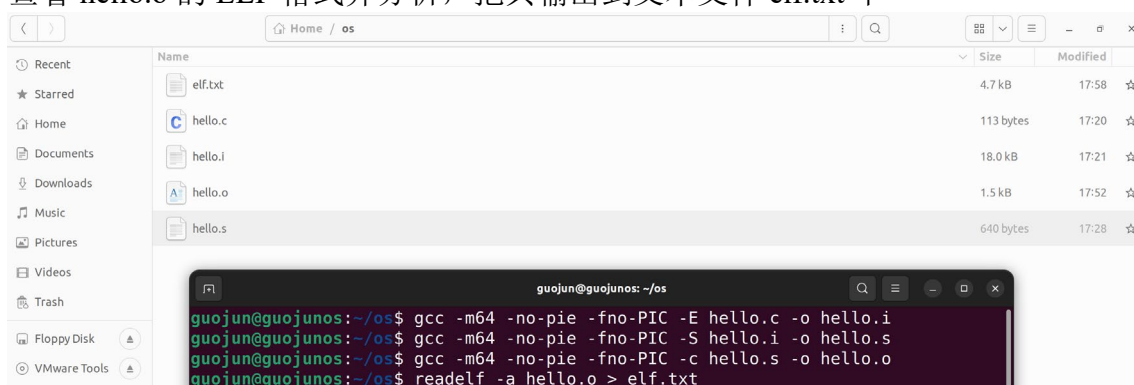
2.3 汇编

1. **概念**: 汇编是指经过汇编器 (as) 把汇编语言程序 (hello.s) 翻译成机器指令, 并把这些指令打包成可重定位目标程序的形式, 并保存在 hello.o 文件中。
2. **作用**: 把汇编语言翻译成计算机能够直接执行的 0、1 机器语言, 把文本文件转化成二进制文件。
3. **命令**: `gcc -m64 -no-pie -fno-PIC -c hello.s -o hello.o`



4. 可重定位目标 elf 格式

查看 hello.o 的 ELF 格式并分析, 把其输出到文本文件 elf.txt 中



ELF 文件包括 ELF 头、节头表、重定位节、符号表等, 并且在其中列出了各节的基本信息, 包括类型、地址等等, 具体的分析如下:

(1) ELF 头

ELF 头中描述了文件类别、数据存放方式、版本号、操作系统等信息。并且给出了入口点地址和程序头起点以及节头表的偏移。

1 ELF Header:

```
2 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 Class: ELF64
4 Data: 2's complement, little endian
5 Version: 1 (current)
6 OS/ABI: UNIX - System V
7 ABI Version: 0
8 Type: REL (Relocatable file)
9 Machine: Advanced Micro Devices X86-64
10 Version: 0x1
11 Entry point address: 0x0
12 Start of program headers: 0 (bytes into file)
13 Start of section headers: 600 (bytes into file)
14 Flags: 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
```

(2) 节头表

节头表中描述了文件中各个节的类型、大小、地址、偏移、读写访问权限等信息。

22 Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0	0
[1]	.text	PROGBITS	0000000000000000	00000040
	000000000000000019	0000000000000000	AX 0 0	1
[2]	.rela.text	RELA	0000000000000000	00000198
	000000000000000030	000000000000000018	I 11 1	8
[3]	.data	PROGBITS	0000000000000000	00000059
	0000000000000000	0000000000000000	WA 0 0	1
[4]	.bss	NOBITS	0000000000000000	00000059
	0000000000000000	0000000000000000	WA 0 0	1
[5]	.rodata	PROGBITS	0000000000000000	00000059
	00000000000000000e	0000000000000000	A 0 0	1
[6]	.comment	PROGBITS	0000000000000000	00000067
	00000000000000002c	00000000000000001	MS 0 0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	00000093
	0000000000000000	0000000000000000	0 0	1
[8]	.note.gnu.pr[...]	NOTE	0000000000000000	00000098
	000000000000000020	0000000000000000	A 0 0	8
[9]	.eh_frame	PROGBITS	0000000000000000	000000b8
	000000000000000038	0000000000000000	A 0 0	8
[10]	.rela.eh_frame	RELA	0000000000000000	000001c8
	000000000000000018	000000000000000018	I 11 9	8
[11]	.symtab	SYMTAB	0000000000000000	000000f0
	000000000000000090	000000000000000018	12 4	8
[12]	.strtab	STRTAB	0000000000000000	00000180
	000000000000000013	0000000000000000	0 0	1
[13]	.shstrtab	STRTAB	0000000000000000	000001e0
	000000000000000074	0000000000000000	0 0	1

(3) 重定位节

文件中有两个重定位节，分别是.rela.text 和.rela.eh_frame 节。不同的重定位节对应

不同节的重定位信息。连接器在处理目标是要对目标文件进行重定位，这时需要重定位节中的信息才能知道如何进行重定位。偏移量指出重定位的字节偏移量，类型指出重定位类型等。有了这些信息才能正确地进行重定位。

```
65 Relocation section '.rela.text' at offset 0x198 contains 2 entries:
66   Offset      Info      Type           Sym. Value  Sym. Name + Addend
67 0000000000009 000300000000a R_X86_64_32    0000000000000000 .rodata + 0
68 000000000000e 0005000000004 R_X86_64_PLT32 0000000000000000 puts - 4
69
70 Relocation section '.rela.eh_frame' at offset 0x1c8 contains 1 entry:
71   Offset      Info      Type           Sym. Value  Sym. Name + Addend
72 0000000000020 0002000000002 R_X86_64_PC32  0000000000000000 .text + 0
73 No processor specific unwind information to decode
```

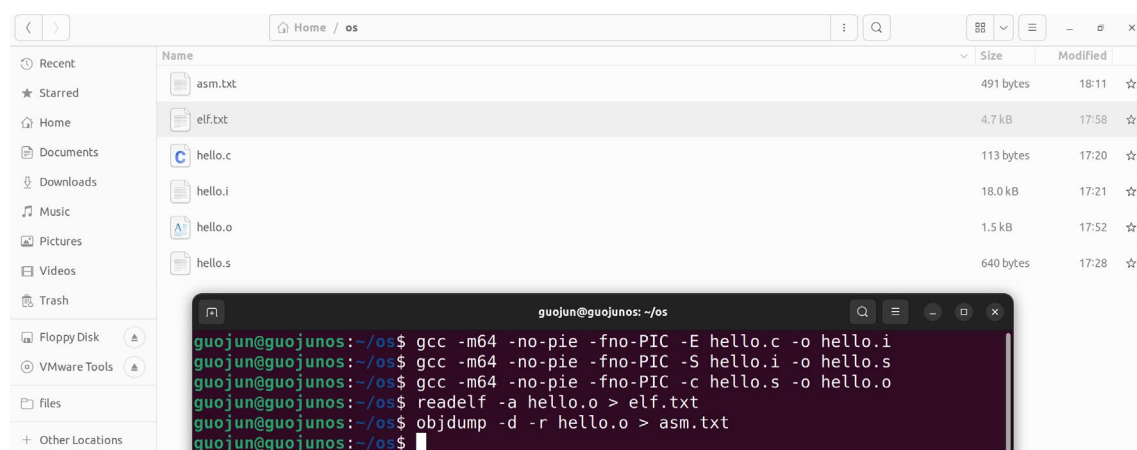
(4) 符号表

在符号表中列出了程序中所有定义和引用的全局变量以及函数的信息：

```
75 Symbol table '.syntab' contains 6 entries:
76   Num:      Value      Size Type      Bind      Vis      Ndx Name
77   0: 0000000000000000    0 NOTYPE   LOCAL   DEFAULT   UND
78   1: 0000000000000000    0 FILE     LOCAL   DEFAULT   ABS hello.c
79   2: 0000000000000000    0 SECTION LOCAL   DEFAULT    1 .text
80   3: 0000000000000000    0 SECTION LOCAL   DEFAULT    5 .rodata
81   4: 0000000000000000   25 FUNC     GLOBAL   DEFAULT    1 main
82   5: 0000000000000000    0 NOTYPE   GLOBAL   DEFAULT   UND puts
83
84 No version information found in this file.
85
86 Displaying notes found in: .note.gnu.property
87   Owner          Data size      Description
88   GNU            0x00000010      NT_GNU_PROPERTY_TYPE_0
89   Properties: x86 feature: IBT, SHSTK
```

结果分析：

在 Ubuntu 下利用反汇编指令生成 asm.txt 反汇编文件，如图 4.4.1 和图 4.4.2 所示：



对反汇编文件 asm.txt 和汇编程序 hello.s 对比分析如下：


```

1 |
2 hello.o:      file format elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8   0:  f3 0f 1e fa          endbr64
9   4:  55                   push   %rbp
10  5:  48 89 e5             mov    %rsp,%rbp
11  8:  bf 00 00 00 00      mov    $0x0,%edi
12                      9: R_X86_64_32 .rodata
13  d:  e8 00 00 00 00      call   12 <main+0x12>
14                      e: R_X86_64_PLT32 puts-0x4
15 12:  b8 00 00 00 00      mov    $0x0,%eax
16 17:  5d                   pop    %rbp
17 18:  c3                   ret

```

- (1) 函数调用：在 `hello.s` 中函数调用直接引用函数名称即可，而在反汇编文件 `asm.txt` 中，函数调用是使用 `call` 指令就加上被调用函数的首地址形成的一条跳转指令。
- (2) 操作数的进制表示：在 `hello.s` 中在进行栈指针移动的时候，使用的是十进制数，而在反汇编文件 `asm.txt` 中使用的是十六进制数。
- (3) 分支转移：在 `hello.s` 中分支转移目标的位置是使用 `.L` 表示的，即给出一个跳转的索引位置，而在反汇编文件 `asm.txt` 中每一个跳转指令的目标位置都是对应的跳转地址。

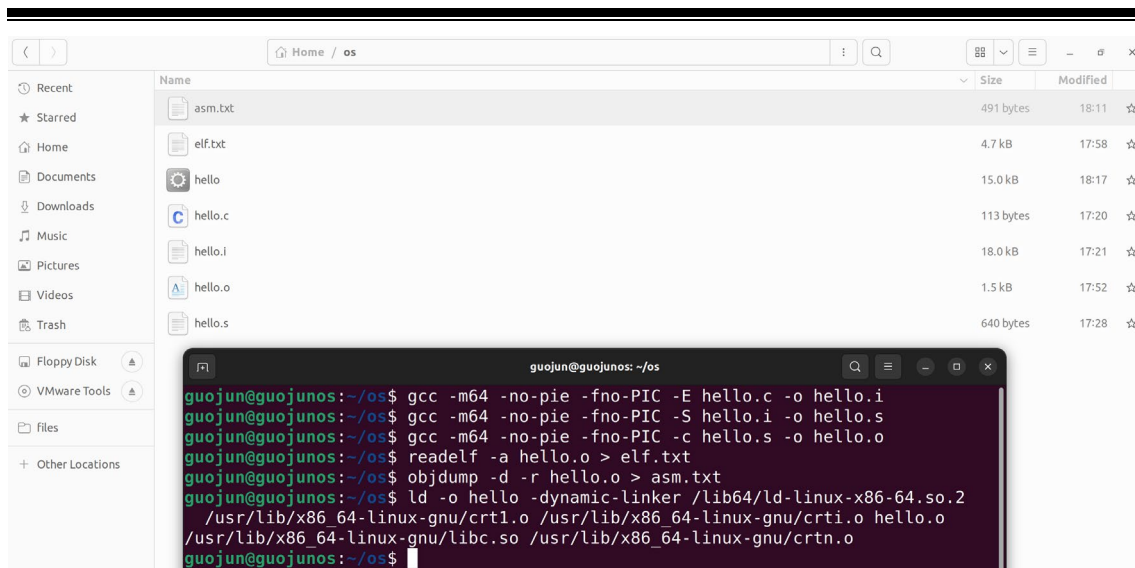
2.4 链接

1. **概念：**链接（linking）是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。
2. **作用：**链接器在软件开发中扮演着一个关键的角色，因为它们使得分离编译（separate compilation）成为可能，我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小，更好管理的模块，可以独立的修改和编译这些模块，当我们改变这些模块中的一个时，只需简单的重新编译它，并重新链接应用，而不必重新编译其他文件。
3. **命令：**

```

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

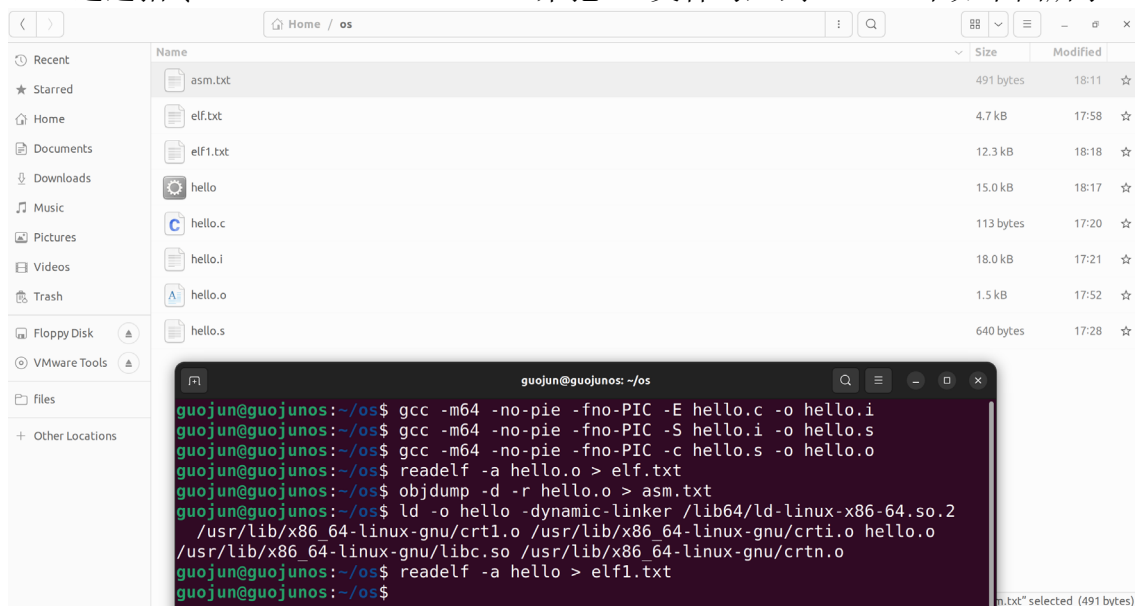
```

4. 可执行目标文件 hello 的格式

分析 hello 的 ELF 格式，用 readelf 等列出其各段的基本信息，包括各段的起始地址，大小等信息。

通过指令 readelf -a hello > elf1.txt 来把 elf 文件写入到 elf1.txt 中如下图所示：



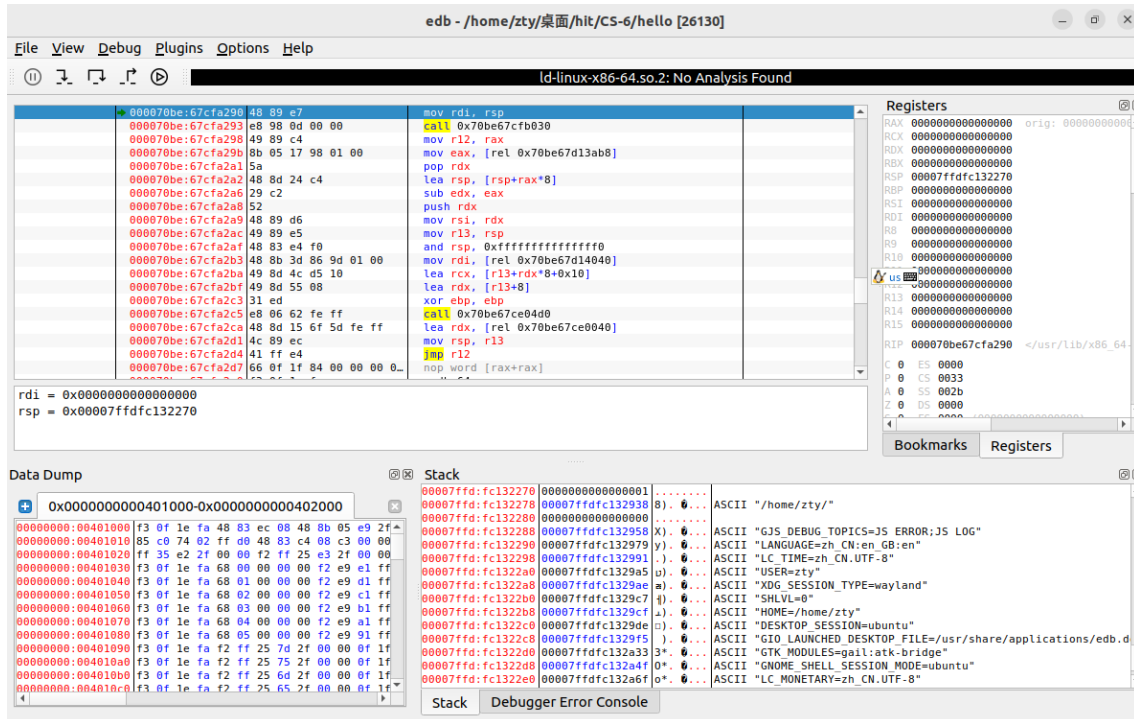
打开 elf1.txt，可以看到可执行文件 hello 的 elf 格式，其中最左侧一列为节名，第二列为节类型，第三列为起始地址，第四列是偏移量。

22 Section Headers:

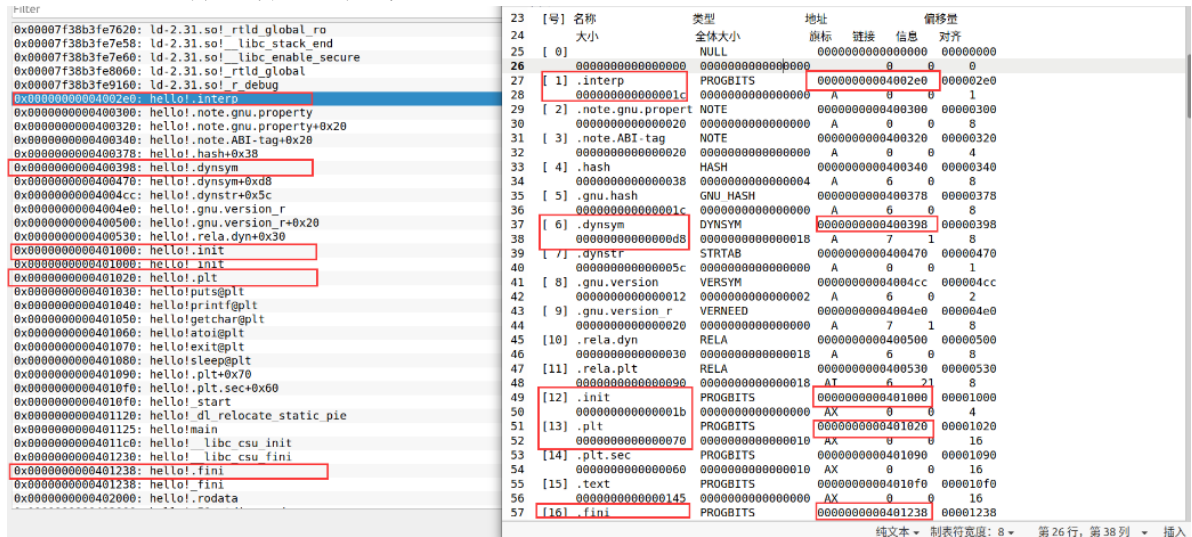
Nr	Name	Type	Address			Offset
		EntSize	Flags	Link	Info	Align
23	[0]	Size	NULL	0000000000000000	00000000	
24						
25	[0]	NULL	0000000000000000	0	0	0
26		0000000000000000	0000000000000000	0	0	0
27	[1]	.interp	PROGBITS	00000000004002e0	000002e0	
28		0000000000000001c	0000000000000000	A	0	0
29	[2]	.note.gnu.pr[...]	NOTE	0000000000400300	00000300	
30		00000000000000030	0000000000000000	A	0	0
31	[3]	.note.ABI-tag	NOTE	0000000000400330	00000330	
32		00000000000000020	0000000000000000	A	0	0
33	[4]	.hash	HASH	0000000000400350	00000350	
34		00000000000000024	0000000000000004	A	6	0
35	[5]	.gnu.hash	GNU_HASH	0000000000400378	00000378	
36		0000000000000001c	0000000000000000	A	6	0
37	[6]	.dynsym	DYNSYM	0000000000400398	00000398	
38		00000000000000060	0000000000000018	A	7	1
39	[7]	.dynstr	STRTAB	00000000004003f8	000003f8	
40		00000000000000048	0000000000000000	A	0	0
41	[8]	.gnu.version	VERSYM	0000000000400440	00000440	
42		00000000000000008	0000000000000002	A	6	0
43	[9]	.gnu.version_r	VERNEED	0000000000400448	00000448	
44		00000000000000030	0000000000000000	A	7	1
45	[10]	.rela.dyn	RELA	0000000000400478	00000478	
46		00000000000000030	0000000000000018	A	6	0
47	[11]	.rela.plt	RELA	00000000004004a8	000004a8	
48		00000000000000018	0000000000000018	AI	6	21
49	[12]	.init	PROGBITS	0000000000401000	00001000	
50		0000000000000001b	0000000000000000	AX	0	0
51	[13]	.plt	PROGBITS	0000000000401020	00001020	
52		00000000000000020	0000000000000010	AX	0	0
53	[14]	.plt.sec	PROGBITS	0000000000401040	00001040	
54		00000000000000010	0000000000000010	AX	0	0
55	[15]	.text	PROGBITS	0000000000401050	00001050	
56		0000000000000004e	0000000000000000	AX	0	0
57	[16]	.fini	PROGBITS	00000000004010a0	000010a0	
58		0000000000000000d	0000000000000000	AX	0	0
59	[17]	.rodata	PROGBITS	0000000000402000	00002000	
60		00000000000000012	0000000000000000	A	0	0
61	[18]	.eh_frame	PROGBITS	0000000000402018	00002018	
62		000000000000000a0	0000000000000000	A	0	0
63	[19]	.dynamic	DYNAMIC	0000000000403e50	00002e50	
64		000000000000001a0	0000000000000010	WA	7	0
65	[20]	.got	PROGBITS	0000000000403ff0	00002ff0	
66		00000000000000010	0000000000000008	WA	0	0
67	[21]	.got.plt	PROGBITS	0000000000404000	00003000	
68		00000000000000020	0000000000000008	WA	0	0
69	[22]	.data	PROGBITS	0000000000404020	00003020	
70		00000000000000004	0000000000000000	WA	0	0
71	[23]	.comment	PROGBITS	0000000000000000	00003024	
72		0000000000000002b	0000000000000001	MS	0	0
73	[24]	.symtab	SYMTAB	0000000000000000	00003050	
74		000000000000001f8	0000000000000018		25	7
75	[25]	.strtab	STRTAB	0000000000000000	00003248	
76		000000000000000d3	0000000000000000		0	0
77	[26]	.shstrtab	STRTAB	0000000000000000	0000331b	
78		000000000000000e1	0000000000000000		0	0

5.hello 的虚拟地址空间

使用 edb 加载 hello, 查看本进程的虚拟地址空间各段信息, 并与 4 对照分析说明。



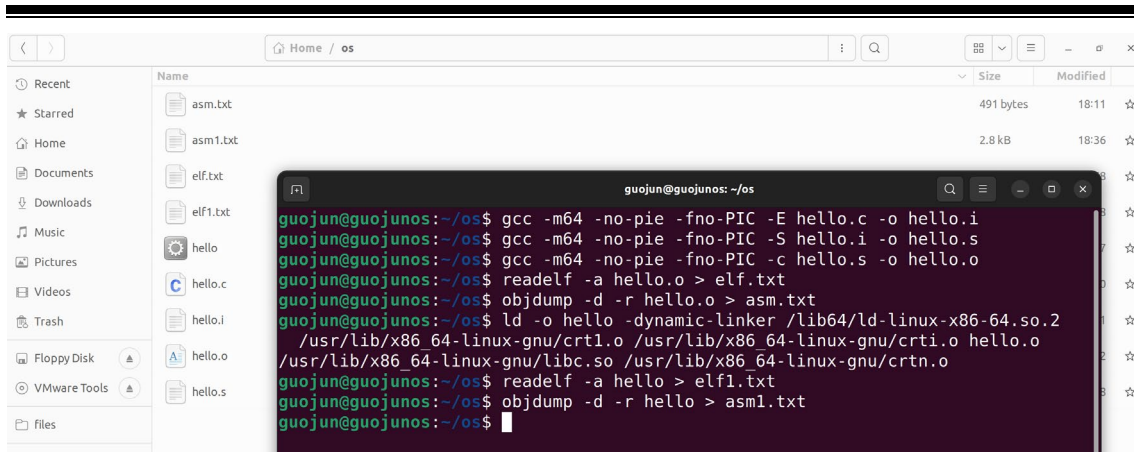
使用 edb 加载 hello, 查看本进程的虚拟地址空间各段信息, edb 中显示的虚拟地址和 elf 文件中看到的虚拟地址是对应相同的。



6.链接的重定位过程分析

运用 objdump -d -r hello > asm1.txt 把 hello 反汇编写入 asm1.txt

接下来我们对 asm1.txt (hello 的反汇编文件) 和 asm.txt (hello.o 的反汇编文件) 进行分析比对, 这实际上就是在比较 hello 与 hello.o 的不同, 具体分析如下:



(1) 为指令分配虚拟地址

左侧是链接前的反汇编，右侧是链接后的。在 `hello.o` 的反汇编程序中，函数中的语句前面的地址都是从函数开始从依次递增的，而不是虚拟地址；而经过链接后，在右侧的返回变种每一条指令都被分配了虚拟地址。

1	hello.o:	file format elf64-x86-64
2		
3		
4		
5	Disassembly of section .text:	
6		
7	0000000000000000 <main>:	
8	0: f3 0f 1e fa	endbr64
9	4: 55	push %rbp
10	5: 48 89 e5	mov %rsp,%rbp
11	8: bf 00 00 00 00	mov \$0x0,%edi
12		9: R_X86_64_32 .rodata
13	d: e8 00 00 00 00	call 12 <main+0x12>
14		e: R_X86_64_PLT32 puts-0x4
15	12: b8 00 00 00 00	mov \$0x0,%eax
16	17: 5d	pop %rbp
17	18: c3	ret

(2) 函数调用

在上面 `hello.o` 的反汇编程序中，图中选中的函数调用指令由于还没有分配虚拟地址，所以只能用偏移量跳转，而下面链接后已经分配好了虚拟地址，可以直接用 `call` 虚拟地址进行跳转。

11	8:	bf 00 00 00 00	mov	\$0x0,%edi
12			9: R_X86_64_32	.rodata
13	d:	e8 00 00 00 00	call	12 <main+0x12>
14			e: R_X86_64_PLT32	puts-0x4
15	12:	b8 00 00 00 00	mov	\$0x0,%eax
16	17:	5d	pop	%rbp
17	18:	c3	ret	


```

59 401085:      f3 0f 1e fa      endbr64
60 401089:      55              push    %rbp
61 40108a:      48 89 e5          mov     %rsp,%rbp
62 40108d:      bf 04 20 40 00    mov     $0x402004,%edi
63 401092:      e8 a9 ff ff ff    call    401040 <puts@plt>
64 401097:      b8 00 00 00 00    mov     $0x0,%eax
65 40109c:      5d              pop     %rbp
66 40109d:      c3              ret

```

(3) 调入 C 标准库函数

在 `hello.o` 中只有 `main` 函数段，还没有把标准库函数插入，经过链接后，调用的 C 标准库函数的代码被插入其中。

```

19 0000000000401020 <.plt>:
20 401020:      ff 35 e2 2f 00 00    push    0x2fe2(%rip)          # 404008 <_GLOBAL_OFFSET_TABLE_+0x8>
21 401026:      f2 ff 25 e3 2f 00 00    bnd jmp *0x2fe3(%rip)        # 404010 <_GLOBAL_OFFSET_TABLE_+0x10>
22 40102d:      0f 1f 00             nopl    (%rax)
23 401030:      f3 0f 1e fa      endbr64
24 401034:      68 00 00 00 00    push    $0x0
25 401039:      f2 e9 e1 ff ff ff    bnd jmp 401020 <_init+0x20>
26 40103f:      90              nop
27
28 Disassembly of section .plt.sec:
29
30 0000000000401040 <puts@plt>:
31 401040:      f3 0f 1e fa      endbr64
32 401044:      f2 ff 25 cd 2f 00 00    bnd jmp *0x2fcd(%rip)        # 404018 <puts@GLIBC_2.2.5>
33 40104b:      0f 1f 44 00 00    nopl    0x0(%rax,%rax,1)
34

```

综上所述，链接的过程主要分为两个过程：符号解析和重定位。

符号解析时解析目标文件定义和引用符号，并建立每个符号引用和符号定义之间的关联。

重定位时分为两个步骤，先重定位节和符号定义，把相同类型的节合并，并为其分配内存。接下来进行符号引用的重定位，修改代码和数据中对符号的引用，使得他们指向正确地址。

7. hello 的执行流程

其调用与跳转的各个子程序名或程序地址如下

调用/跳转地址	目标函数/程序名	描述
403ff8	<code>__gmon_start__</code>	GNU 监控工具函数（可能为空）。
404008	<code>_GLOBAL_OFFSET_TABLE_+0x8</code>	全局偏移表，可能指向动态链接器的初始化数据。
404010	<code>_GLOBAL_OFFSET_TABLE_+0x10</code>	动态链接器函数入口。
404018	<code>puts@GLIBC_2.2.5</code>	标准 C 库函数 <code>puts</code> 的入口地址，通过 PLT 表跳转到实际实现。
403ff0	<code>__libc_start_main@GLIBC_2.34</code>	程序的初始化函数，调用 <code>main</code> 并进行启动。
401040	<code>puts@plt</code>	调用 <code>puts</code> 的 PLT 表入口，通过 PLT 跳转到动态链接库中 <code>puts</code> 的实现。

第 3 章 用户输入与 Shell 解析

3.1 Shell 命令解析

当用户在 Shell 输入命令并按下回车，Shell 会执行以下操作：

读取输入命令

Shell 从用户输入的缓冲区中读取完整的一行命令（例如 `./hello`）。

用户输入的内容包括：程序名：`./hello`。 参数（有的时候）：例如 `./hello arg1`。

拆分命令：

将命令字符串解析为：程序路径（`./hello`）。参数列表（`argv`）。

查找可执行文件：

如果命令是路径（如 `./hello`），Shell 直接查找该路径。

如果是简单命令（如 `ls`），Shell 会依次查找 `$PATH` 环境变量中定义的路径列表。

检查权限：

Shell 会检查文件是否存在，以及是否具有执行权限。

准备执行：

如果解析成功，Shell 会创建一个子进程，并在子进程中执行该命令。

```
guojun@guojunos:~/os$ ./hello
Hello, World!
```

使用 `strace` 跟踪 Shell 的命令解析过程：

```

guojun@guojunos:~/os$ strace -e trace=openat,access bash -c "./hello"
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libtinfo.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/dev/tty", O_RDWR|O_NONBLOCK) = 3
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY) = 3
access("/usr/bin/bash", X_OK) = 0
access("/usr/bin/bash", R_OK) = 0
access("/usr/bin/bash", X_OK) = 0
access("/usr/bin/bash", R_OK) = 0
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
Hello, World!
+++ exited with 0 +++

```

(1) Shell 命令解析

```

access("/usr/bin/bash", X_OK) = 0
access("/usr/bin/bash", R_OK) = 0
access("/usr/bin/bash", X_OK) = 0
access("/usr/bin/bash", R_OK) = 0

```

access("/usr/bin/bash", X_OK): 检查文件是否可执行 (X_OK)。返回值 0 表示 /usr/bin/bash 是可执行文件。

access("/usr/bin/bash", R_OK): 检查文件是否可读 (R_OK)。返回值 0 表示 /usr/bin/bash 是可读文件。

Shell 在解析命令 ./hello 时，确认 bash 自身具有可执行和可读权限，确保可以继续运行。通过调用 access 系统调用，Shell 检查目标程序的可执行性。

(2) 动态链接器的准备

```

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libtinfo.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

```

access("/etc/ld.so.preload", R_OK): 检查是否存在 /etc/ld.so.preload 文件，该文件通常用于预加载动态链接库。返回值 ENOENT 表示文件不存在，跳过预加载。

openat("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC): 打开 /etc/ld.so.cache 文件，动态链接器使用此缓存文件快速查找共享库路径。

openat("/lib/x86_64-linux-gnu/libtinfo.so.6", O_RDONLY|O_CLOEXEC): 加载 libtinfo.so.6, 这是与终端相关的库。

openat("/lib/x86_64-linux-gnu/libc.so.6",O_RDONLY|O_CLOEXEC): 加载标准 C 库 libc.so.6, 包含 printf 等函数的实现。

动态链接器负责解析并加载程序运行时所需的共享库。如果没有动态链接缓存 /etc/ld.so.cache, 动态链接器会直接从预定义路径 (如 /lib) 加载库。

(3) 程序执行: 调用 ./hello

```
Hello, World!  
+++ exited with 0 +++
```

程序输出:

Hello, World! 是 ./hello 程序的输出结果。

说明程序运行成功, 并使用标准 C 库中的 puts 或 printf 函数完成了输出操作。

程序退出:

+++ exited with 0 +++ 表示程序以退出代码 0 正常结束。

程序在子进程中成功运行并完成任务。返回值 0 表明没有发生任何错误。

(4) local 和终端支持

```
openat(AT_FDCWD, "/dev/tty", O_RDWR|O_NONBLOCK) = 3  
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3  
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY) = 3
```

/dev/tty: 打开当前终端设备, 用于标准输入/输出。

/usr/lib/locale/locale-archive: 动态链接器加载与区域设置 (locale) 相关的文件。

区域设置定义了程序的语言、日期格式等设置。/usr/lib/x86_64-linux-

gnu/gconv/gconv-modules.cache: 动态链接器加载字符编码转换模块。通常用于终端输出或输入时的字符集支持。

总结:

Shell 命令解析:

Shell 使用 `access` 检查 `bash` 和目标程序 (`./hello`) 的可执行权限。确保目标文件存在并可执行。

动态链接库加载:

动态链接器加载程序运行所需的共享库:

`/lib/x86_64-linux-gnu/libc.so.6` (标准 C 库)。

`/lib/x86_64-linux-gnu/libtinfo.so.6` (终端支持库)。

程序执行:

调用 `./hello` 程序并成功输出 "Hello, World!"。

程序退出:

程序正常退出, 返回值为 0。

3.2 Hello 的进程执行

(1) 上下文信息: 上下文信息是操作系统内核重新启动一个挂起的进程所需要恢复的状态。它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的信息构成。

(2) 时间片: 一个进程执行它的控制流的一部分的每一时间段叫做时间片。

(3) 进程调度: 在进程执行过程中, 操作系统内核可以决定抢占当前进程, 并重新开始一个先前被挂起的进程, 这样的一种决策叫做进程调度。当抢占进程时, 要完成以下三个任务: ①保存之前进程的上下文; ②恢复要执行的新进程的上下文; ③把控制转让给新恢复的进程完成上下文切换。

(4) 用户模式和内核模式: 处理器通常使用一个寄存器来区分两种模式, 这个寄存器描述了当前进程的权限情况。简单来说, 两种模式有不同的“权限”, 用户模式权限较低, 不允许执行特权指令, 也不允许直接引用地址空间中内核区内的代码和数据; 内核模式权限较高, 可以执行任何命令, 并且可以访问系统中的任何内存位置。

(5) 进程执行与用户态核心态转换: 当开始运行 `hello` 时, 内存为 `hello` 分配时间片, 若一个系统同时运行多个进程, 则它们轮流使用处理器, 物理控制流被划分成多个交错的逻辑控制流, 存在并发执行的现象。然后在用户态下执行并保存上下文。如果在此期间内发生了异常或系统中断, 则内核会休眠该进程, 并在核心态中进行上下文切换, 把控制权让给其他进程。当 `hello` 进程执行到 `sleep` 时, `hello` 会进入休眠状态, 此时再次进行上下文切换, 控制交付给其他进程, 一段时间后 `hello`

- 子进程（运行 ./hello）的 PID: 15029。
- 父进程（bash -c 调用 Shell）的 PID: 15026。

```
getpid()           = 15029
getppid()          = 15026
getpid()           = 15029
getppid()          = 15026
getpgrp()          = 15026
```

3.4 hello 的异常与信号处理

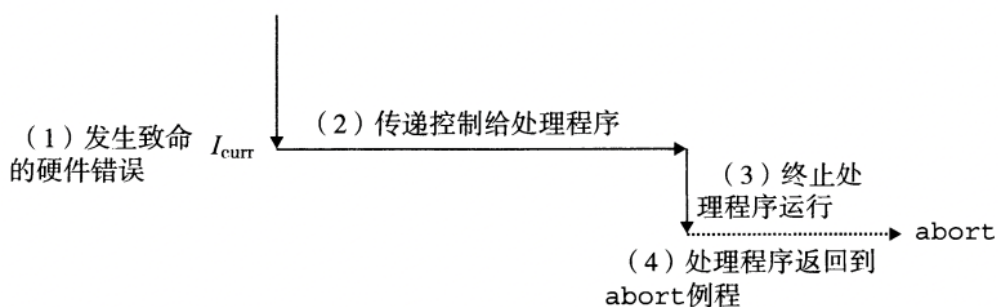
hello 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z，Ctrl-C 等，Ctrl-z 后可以运行 ps jobs pstree fg kill 等命令，请分别给出各命令及运行结果截屏，说明异常与信号的处理。

3.3.1 执行过程中可能出现的异常

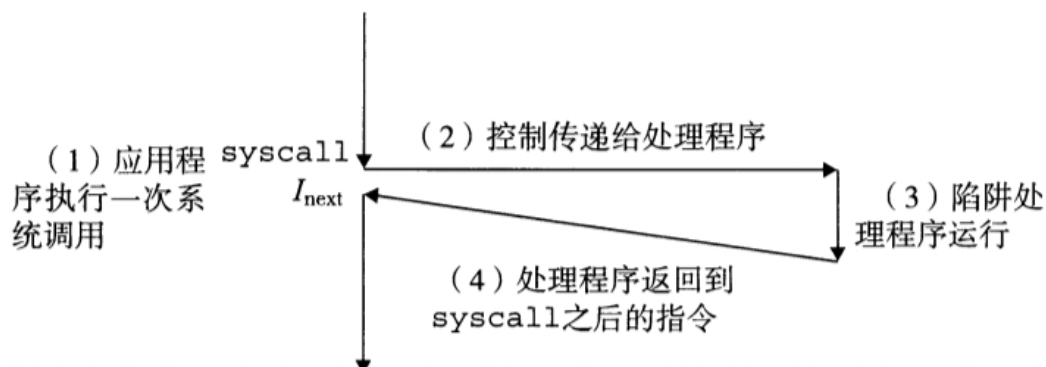
（1）异步异常（中断）

在程序执行过程中由处理器外部 IO 设备引起的异常，如键盘上敲击 Ctrl-C。处理。



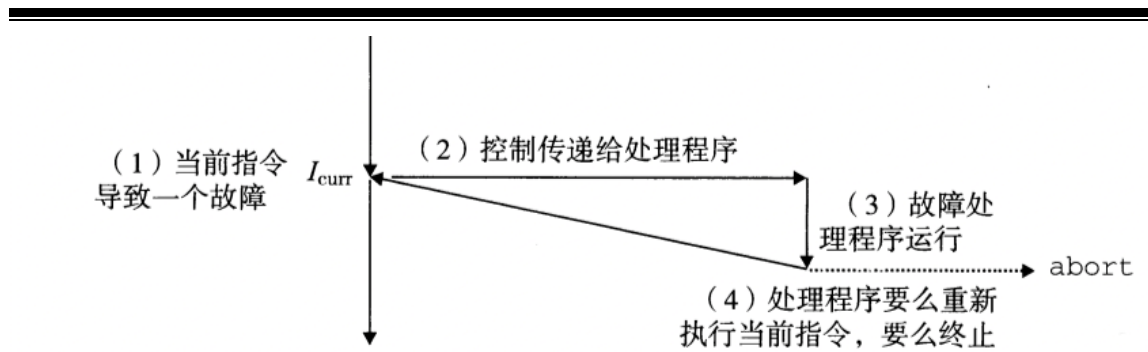
（2）同步异常之陷阱

陷阱是有意的异常，是指令的执行结果，如系统调用。



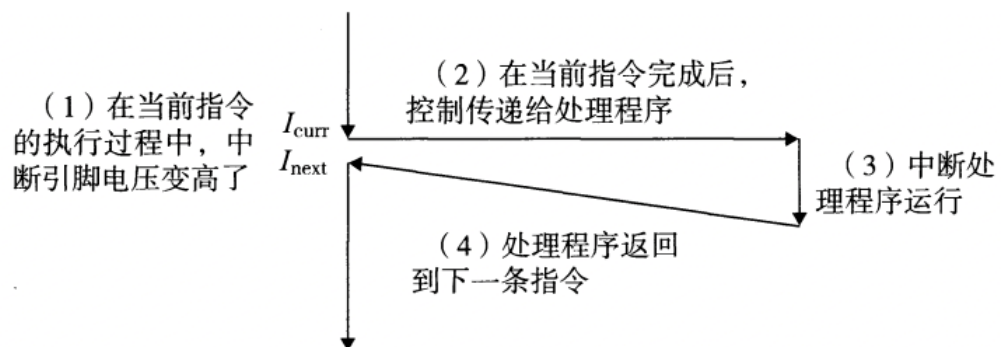
（3）同步异常之故障

故障不是有意的，但可能被修复，如缺页故障是可恢复的，但保护故障是不可恢复的。



(4) 同步异常之终止

终止是非故意的，不可恢复的致命错误造成的，如非法指令。



3.3.2 可能出现的信号

hello 执行过程中可能出现的信号有：SIGINT、SIGKILL、SIGSEGV、SIALARM、SIGCHLD。

第 4 章 hello 的存储管理

1.1 hello 的存储器地址空间

1. 逻辑地址

定义：程序经过编译后，汇编代码中使用的地址被称为逻辑地址。它由一个 **段选择符 (Segment Selector)** 和 **段内偏移量 (Offset)** 组成。

特点：逻辑地址由程序员或编译器指定。并不直接对应物理内存，需要通过 CPU 的分段机制和分页机制转换为物理地址。

2. 线性地址

定义：线性地址是逻辑地址经过分段机制转换后的中间地址。

线性地址 = 段基址 + 段内偏移量

段基址由段选择符指定的段描述符提供。段内偏移量直接由逻辑地址中的偏移部分提供。

特点：线性地址是程序访问内存的统一地址空间。在现代 x86 系统中，线性地址和虚拟地址通常是一致的。

3. 虚拟地址

定义：虚拟地址是程序员和操作系统认为的地址空间。通常，虚拟地址和线性地址是一致的（在启用分页机制时，虚拟地址通过页表被映射到物理地址）。

特点：每个进程都有独立的虚拟地址空间（例如，Linux 中每个用户进程有 4GB 的虚拟地址空间）。在 x86-64 架构中，64 位虚拟地址空间通常限制在前 48 位地址范围内。

4. 物理地址

定义：物理地址是 CPU 外部地址总线上实际出现的地址。它直接对应实际的物理内存单元。

特点：物理地址由虚拟地址通过分页机制（或直接映射）转换而来。程序本身不能直接访问物理地址，而是通过 MMU（内存管理单元）间接访问。

4.2 Intel 逻辑地址到线性地址的变换-段式管理

1. 逻辑地址的组成：

一个逻辑地址由两部分组成：**段选择符 (Segment Selector)**：16 位，前 13 位是段描述符表中的索引号，决定段基址。**段内偏移量 (Offset)**：段内的具体位置。

逻辑地址格式：[段选择符:段内偏移量]

2. 逻辑地址到线性地址的转换步骤

确定段描述符表：

根据段选择符的 TI 位判断是全局段描述符表 (GDT) 还是局部段描述符表 (LDT)。

从相关寄存器（GDTR 或 LDTR）获取描述符表的基地址。

找到段描述符：

使用段选择符的前 13 位索引，从描述符表中查找到对应的段描述符。

段描述符包含段基址（Base Address）和段界限（Limit）。

计算线性地址：

线性地址 = 段基址（Base） + 段内偏移量（Offset）

第 5 章 hello 的 IO 管理

5.1 Linux 的 IO 设备管理方法

设备的模型化：所有 IO 设备都被模型化为文件，所有的输入输出均可作为文件的读写来执行。

设备管理：Linux 内核有一个简单、低级的接口，称为 Unix I/O，使得所有的输入输出都能以统一方式来执行。

5.2 简述 Unix IO 接口及其函数

（一）Unix IO 接口：Unix IO 接口提供了打开、关闭、读取和写入文件的功能。通过打开文件，操作系统内核会返回一个文件描述符，应用程序可以使用这个文件描述符来操作文件。应用程序可以使用 `seek` 操作来改变文件的当前位置，从而可以在文件中进行定位读写操作。读取和写入文件时，操作系统会在文件 and 应用程序之间进行信息传输。当应用程序不再需要访问文件时，可以通知内核关闭文件，从而释放文件占用的资源。

（二）Unix IO 函数：

1. `open()`函数：打开文件
2. `lseek()`函数：将文件指针定位到相应位置
3. `read()`函数 读文件
4. `write()`函数：写文件
5. `close()`函数：关闭文件

5.3 printf 的实现分析

当我们调用 `vsprintf` 函数对需要输出的字符串进行格式化并存放在缓冲区中后，系统函数 `write` 会被调用以在屏幕上打印缓冲区中的前 `i` 个字符，即输出我们要显示的字符串。调用 `write` 系统函数后，程序将进入内核态，并通过系统调用 `int 0x80` 或 `syscall` 等方式，字符驱动子程序会被执行，将我们的字符串内容按照 ASCII 码转换为字模库，然后写入显示 `vram` 中以存储每个点的 RGB 颜色信息。

显示芯片会按照刷新频率逐行读取 `vram` 中的内容，并通过信号线传输 RGB 分量信息到液晶显示器，从而显示出要显示的字符串内容。最后程序将返回实际输出的字符数量 `i`。整个过程实现了将格式化后的字符串内容显示在屏幕上的功能。

附件

hello.c: C 语言源程序文件

hello.i: 预处理后生成的文件

hello.s: 编译产生的汇编程序文件

hello.o: 汇编产生的可重定位目标文件

hello: 链接产生的可执行文件

elf.txt: hello.o 的 elf 格式文件

elf1.txt: hello 的 elf 格式文件

asm.txt: hello.o 反汇编的结果文件

asm1.txt: hello 反汇编的结果文件

