

哈爾濱工業大學

題 目 从 OS 角度漫游 Hello

姓 名 张天一

2025 年 1 月

摘 要

本文以 从 OS 角度漫游 Hello 为题，揭示了编程界的传奇 `hello` 的一生。当 `hello.c` 源程序被编写完毕时，`hello` 开启了它的传奇一生——从预处理、编译、汇编、链接生成可执行文件到在系统上运行 `hello`，再到运行完毕被回收，`hello` 结束了它的传奇一生，完成了 P2P 与 020 的过程。`hello` 的传奇一生反映了计算机系统底层的基本内容与基本原理，跟踪 `hello` 的一生就是深入理解计算机系统的过程。

关键词：操作系统；程序生命周期；Linux；Hello 程序

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	5 -
1.4 本章小结	6 -
第 2 章 预处理	7 -
2.1 预处理的概念与作用	7 -
2.2 在 UBUNTU 下预处理的命令	7 -
2.3 HELLO 的预处理结果解析	8 -
2.4 本章小结	10 -
第 3 章 编译	11 -
3.1 编译的概念与作用	11 -
3.2 在 UBUNTU 下编译的命令	11 -
3.3 HELLO 的编译结果解析	12 -
3.4 本章小结	16 -
第 4 章 汇编	17 -
4.1 汇编的概念与作用	17 -
4.2 在 UBUNTU 下汇编的命令	17 -
4.3 可重定位目标 ELF 格式	17 -
4.4 HELLO.O 的结果解析	20 -
4.5 本章小结	22 -
第 5 章 链接	23 -
5.1 链接的概念与作用	23 -
5.2 在 UBUNTU 下链接的命令	23 -
5.3 可执行目标文件 HELLO 的格式	24 -
5.4 HELLO 的虚拟地址空间	25 -
5.5 链接的重定位过程分析	25 -
5.6 HELLO 的执行流程	28 -
5.7 HELLO 的动态链接分析	29 -
5.8 本章小结	30 -
第 6 章 HELLO 进程管理	31 -
6.1 进程的概念与作用	31 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	31 -
6.3 HELLO 的 FORK 进程创建过程	31 -
6.4 HELLO 的 EXECVE 过程	31 -

6.5 HELLO 的进程执行	- 32 -
6.6 HELLO 的异常与信号处理	- 32 -
6.7 本章小结	- 39 -
第 7 章 HELLO 的存储管理	- 40 -
7.1 HELLO 的存储器地址空间	- 40 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 40 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 40 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	错误！未定义书签。
7.5 三级 CACHE 支持下的物理内存访问	错误！未定义书签。
7.6 HELLO 进程 FORK 时的内存映射	错误！未定义书签。
7.7 HELLO 进程 EXECVE 时的内存映射	错误！未定义书签。
7.8 缺页故障与缺页中断处理	错误！未定义书签。
7.9 动态存储分配管理	错误！未定义书签。
7.10 本章小结	- 41 -
第 8 章 HELLO 的 IO 管理	- 46 -
8.1 LINUX 的 IO 设备管理方法	- 46 -
8.2 简述 UNIX IO 接口及其函数	- 46 -
8.3 PRINTF 的实现分析	- 46 -
8.4 GETCHAR 的实现分析	- 47 -
8.5 本章小结	- 47 -
结论	- 47 -
附件	- 49 -
参考文献	- 50 -

第 1 章 概述

1.1 Hello 简介

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。

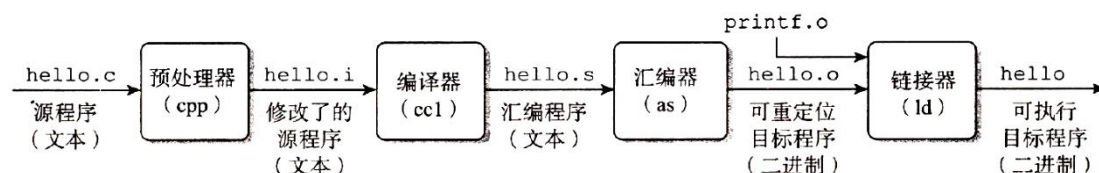


图 1.1.2 编译系统

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 预处理

2.1 预处理的概念与作用

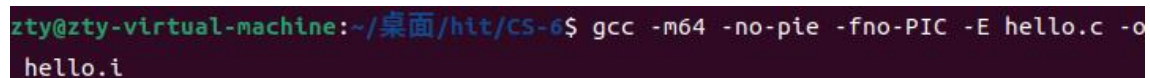
(一) **概念**: 预处理也叫做预编译, 是指在正式开始编译前的操作。预处理器(`cpp`)根据以字符`#`开头的命令, 修改原始的 C 程序, 读取系统头文件中的内容并把它直接插入程序文本, 获得另一个文件后缀为`.i`的文件。

(二) **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, 有选择地执行相关操作 (如`#if`, `#elif`, `#else`)。

2.2 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`



```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ gcc -m64 -no-pie -fno-PIC -E hello.c -o  
hello.i
```

图 2.2.1 预处理命令

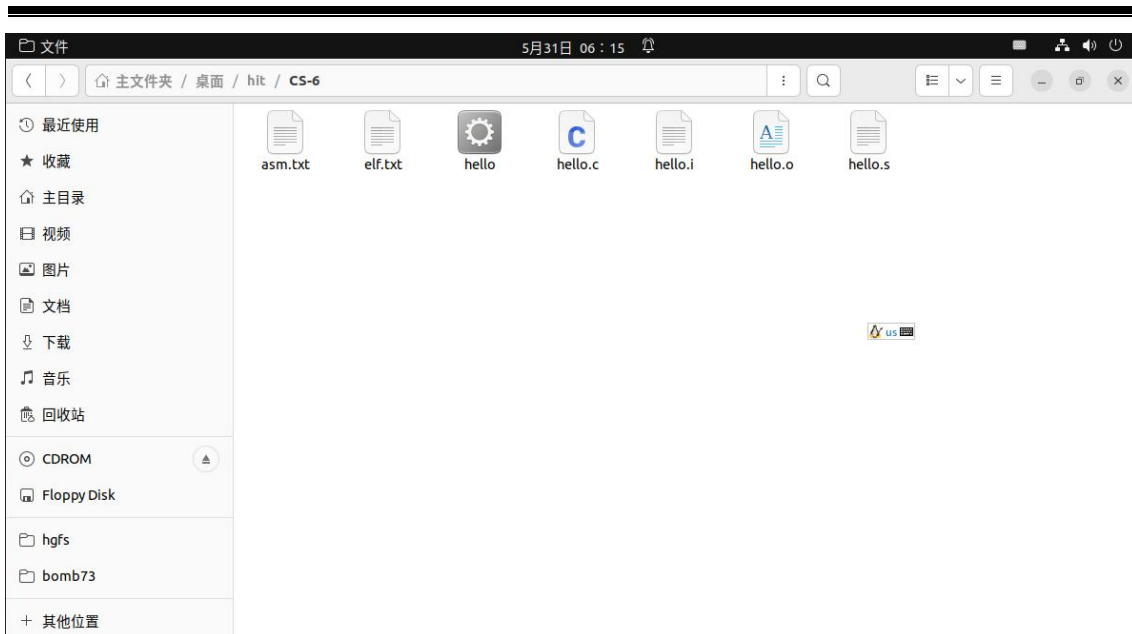


图 2.2.2 hello.i 文件展示

生成的 hello.i 文件截图:



图 2.2.3 hello.i

2.3 Hello 的预处理结果解析

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 3092 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。



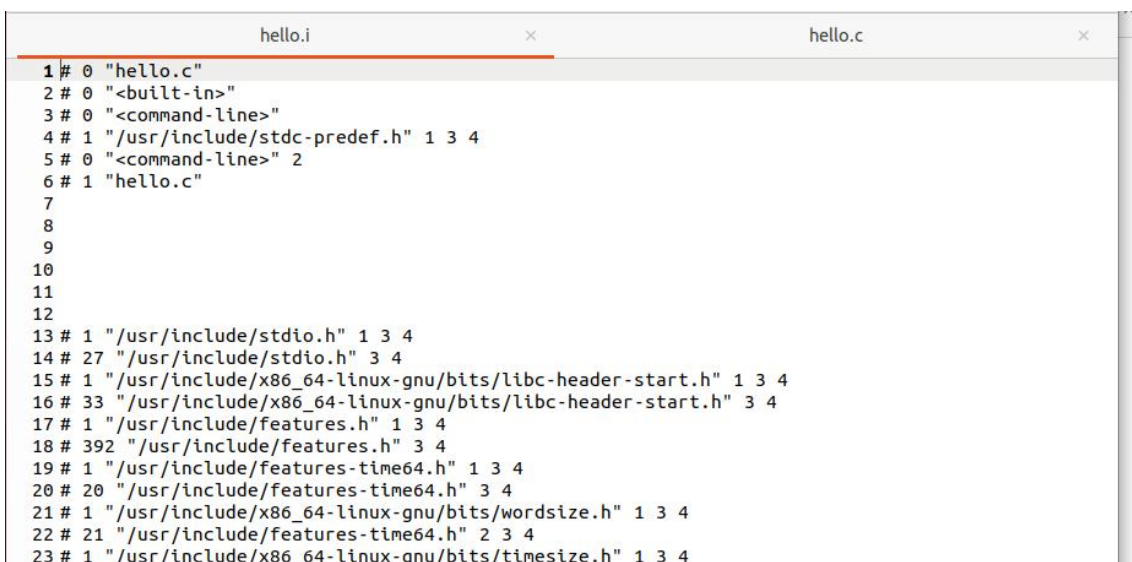
```
3070 # 1023 "/usr/include/stdlib.h" 3 4
3071 # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
3072 # 1024 "/usr/include/stdlib.h" 2 3 4
3073 # 1035 "/usr/include/stdlib.h" 3 4
3074
3075 # 10 "hello.c" 2
3076
3077
3078 # 11 "hello.c"
3079 int main(int argc, char *argv[]){
3080     int i;
3081
3082     if(argc!=5){
3083         printf("用法: Hello 2022112042 張天一 18245706985 秒数! \n");
3084         exit(1);
3085     }
3086     for(i=0;i<10;i++){
3087         printf("Hello %s %s %s\n",argv[1],argv[2],argv[3]);
3088         sleep(atoi(argv[4]));
3089     }
3090     getchar();
3091     return 0;
3092 }
```

图 2.3.1 预处理文件代码段截图



```
1 // 大作业的hello.c 程序
2 // gcc -m64 -Og -no-pie -fno-stack-protector -fno-PIC hello.c -o hello
3 // 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。
4 // 可以运行ps jobs pstree fg 等命令
5 // 秒数=手机号%5
6
7 #include <stdio.h>
8 #include <unistd.h>
9 #include <stdlib.h>
10
```

图 2.3.2 源 c 代码文件头文件截图



```
1 # 0 "hello.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "hello.c"
7
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 # 392 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/features-time64.h" 1 3 4
20 # 20 "/usr/include/features-time64.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 21 "/usr/include/features-time64.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
```

图 2.3.3 预处理文件头文件截图

从下图我们还能看到，程序的最开始的几行给出了这个对该文件的解释。此外，程序中把stdio.h, unistd.h, stdlib.h三个文件解析出来，找到它们的实际地址，把其

中的语句直接插入到hello.i文件中，这也是导致hello.i文件有约三千行代码，远远多于hello.c的几十行代码的最主要原因。



```
3058
3059
3060
3061 extern int rpmatch (const char *__response) __attribute__ ((__nothrow__ , __leaf__))
      __attribute__ ((__nonnull__ (1)));
3062 # 967 "/usr/include/stdlib.h" 3 4
3063 extern int getsubopt (char **__restrict __optionp,
3064                      char *const *__restrict __tokens,
3065                      char **__restrict __valuep)
3066      __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1, 2, 3))) ;
3067 # 1013 "/usr/include/stdlib.h" 3 4
3068 extern int getloadavg (double __loadavg[], int __nelem)
3069      __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1)));
3070 # 1023 "/usr/include/stdlib.h" 3 4
3071 # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
3072 # 1024 "/usr/include/stdlib.h" 2 3 4
3073 # 1035 "/usr/include/stdlib.h" 3 4
3074
3075 # 10 "hello.c" 2
3076
3077
3078 # 11 "hello.c"
3079 int main(int argc, char *argv[]){
3080     int i;
```

图 2.3.4 预处理文件头文件解析截图

2.4 本章小结

本章详细介绍了预处理的概念和作用，并对 Ubuntu 系统下生成的预处理文件 hello.i 和源 hello.c 文件进行了比较，分析出其中的联系与不同。

第 3 章 编译

3.1 编译的概念与作用

(一) **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。

(二) **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。

3.2 在 Ubuntu 下编译的命令

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

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ gcc -m64 -no-pie -fno-PIC -S hello.i -o hello.s
```

图 3.2.1 编译命令

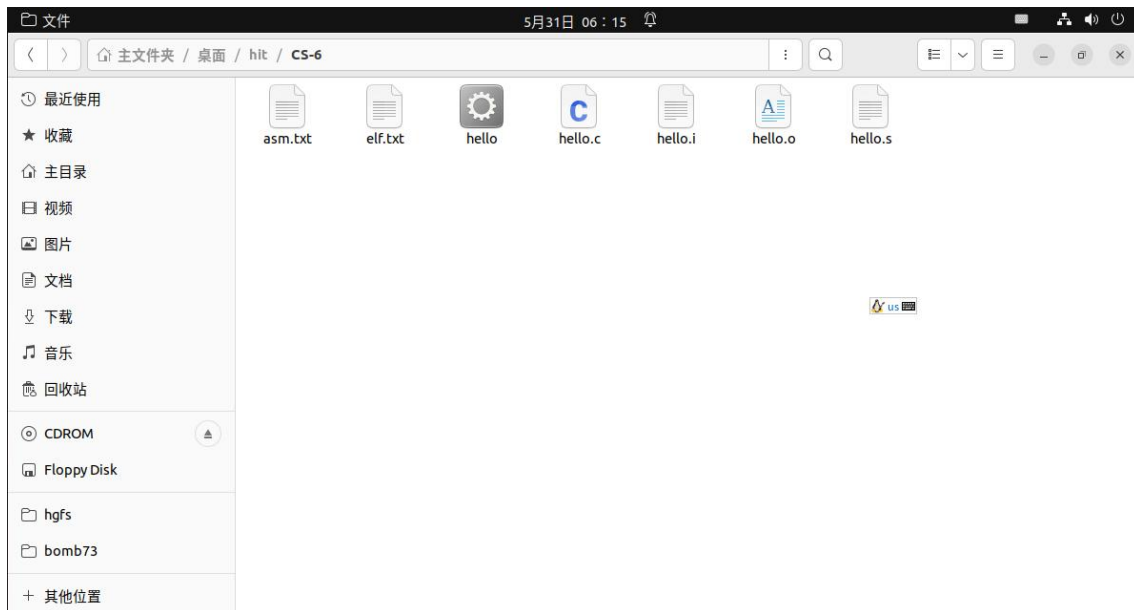


图 3.2.2 hello.s 文件展示

```
hello.i      x      hello.c      x      hello.s
1      .file      "hello.c"
2      .text
3      .section      .rodata
4      .align 8
5      .LC0:
6      .string "\347\224\250\346\263\225: Hello 2022112042 \345\274\265\345\244\251\344\270\200
18245706985 \347\247\222\346\225\260\357\274\201"
7      .LC1:
8      .string "Hello %s %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)
```

图 3.2.3 hello.s 部分代码

3.3 Hello 的编译结果解析

3.3.1 数据

(一) 常量：我们可以看到第 3 行的 `.rodata`（read only data）表明下面的第六行的 `.string` 数据为只读，也即源程序两个 `printf` 函数中的字符串常量。而第 21 行中的 `$32` 表示的是 32 这个立即数，是数字常量。

(二) 局部变量：局部变量通常保存在寄存器或栈中。我们的程序中有三个局部变量：`i`、`argc`、`argv`。我们以 `argc` 为例，分析其过程。程序的第 22 行表明如下信息：`%edi` 中为传入的参数 `argc`，`-20(%rbp)` 表示要把其值送入栈中，接下来可以通过 `rbp` 的相对偏移量进行访问。

```
1      .file      "hello.c"
2      .text
3      .section      .rodata      ← 只读 (read only data)
4      .align 8
5      .LC0:
6      .string "\347\224\250\346\263\225: Hello 2022112042 \345\274\265\345\244\251\344\270\200 18245706985 \347\247\222\346\225\260\357\274\201" ← 字符串常量
7      .LC1:
8      .string "Hello %s %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      ← 立即数, 数字常量
22     movl %edi, -20(%rbp) ← 局部变量
23     movq %rsi, -32(%rbp)
24     cmpl $5, -20(%rbp)
25     je .L2
26     movl $.LC0, %edi
27     call puts
```

图 3.3.1 hello.s 数据和代码段

3.3.2赋值

在汇编语言中，通常使用 `mov` 指令进行赋值，根据赋值的字长不同，后面跟上不同的字母。如汇编代码中第 22 行把 `edi` 寄存器的值赋值给 `rbp` 寄存器地址-20 对应内存地址处，第 28 行把 1 赋值给 `%edi` 寄存器内部等等。

```
22      movl    %edi, -20(%rbp)
23      movq    %rsi, -32(%rbp)
24      cmpl    $5, -20(%rbp)
25      je      .L2
26      movl    $.LC0, %edi
27      call    puts
28      movl    $1, %edi
29      call    exit
30 .L2:
31      movl    $0, -4(%rbp)
32      jmp     .L3
33 .L4:
34      movq    -32(%rbp), %rax
35      addq    $24, %rax
36      movq    (%rax), %rcx
37      movq    -32(%rbp), %rax
38      addq    $16, %rax
```

图 3.3.2 编译代码

3.3.3类型转换

在程序的第 51 行，跳入 `atoi` 函数把字符串显式转换成整型 `int`

```
49      movq    (%rax), %rax
50      movq    %rax, %rdi
51      call    atoi
52      movl    %eax, %edi
```

图 3.3.3 `call` 指令

3.3.4算术操作

在程序的第 51 行，完成了一个加法操作，并且结合下面的代码可以看出，这是对存放于栈中-4 (`%rbp`) 位置的局部变量 `i` 的值进行加法操作，每次加 1 直到它的值达到 5 才不发生跳转而是继续执行。

```
51      call    atoi
52      movl    %eax, %edi
53      call    sleep
54      addl    $1, -4(%rbp)
55 .L3:
56      cmpl    $9, -4(%rbp)
57      jle     .L4
58      call    getchar
```

图 3.3.4 `addl` 指令

3.3.5关系操作

使用 `cmpl` 指令，比较 `rbp` 寄存器-20 的内存地址处的值和立即数 5 的大小，如果相等则跳转到 `.L2` 继续执行，如果不相等则跳过该指令向下执行。这是一个关系操作。


```

24      cmpl    $5, -20(%rbp)
25      je      .L2

```

图 3.3.5 cmpl 指令

3.3.6 数组、指针、结构操作

如图所示，源代码中对数组的操作在汇编程序中变为对地址的加减操作。其中 -32(%rbp) 存放数组首地址即 argv[0]，后续在其基础上进行地址偏移，完成了对数组的访问。

```

34      movq    -32(%rbp), %rax
35      argv[3] addq    $24, %rax
36      movq    (%rax), %rcx
37      movq    -32(%rbp), %rax
38      argv[2] addq    $16, %rax
39      movq    (%rax), %rdx
40      movq    -32(%rbp), %rax
41      argv[1] addq    $8, %rax
42      movq    (%rax), %rax
43      movq    %rax, %rsi
44      movl    $.LC1, %edi
45      movl    $0, %eax
46      call    printf
47      argv[4] movq    -32(%rbp), %rax
48      addq    $32, %rax
49      movq    (%rax), %rax
50      movq    %rax, %rdi

```

图 3.3.6 数组访问

3.3.7 控制转移

如图所示，.L3 的第一二行是一个有条件跳转，只有当 -4(%rbp) 的值小于等于 9 时才会发生跳转，跳转到.L4。

```

.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

```

图 3.3.7.1 有条件跳转

如下图所示，展示的是一个无条件跳转，程序执行到第 32 行直接执行跳转指令，跳转到.L3。

```
30 .L2:
31     movl    $0, -4(%rbp)
32     jmp     .L3
```

图 3.3.7.2 无条件跳转

3.3.8函数操作

(1)参数传递

调用函数之前，编译器会将参数存储在寄存器中，以供调用的函数使用。

(2) 函数调用

如图所示，在 `hello.c` 源程序中共有 5 个函数调用，由于 `printf` 函数调用了 2 次，而其他函数都是 1 次，所以总的调用次数是 6 次。



```
1 // 大作业的hello.c 程序
2 // gcc -m64 -Og -no-pie -fno-stack-protector -fno-PIC hello.c -o hello
3 // 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。
4 // 可以运行 ps jobs pstree fg 等命令
5 // 秒数=手机号%5
6
7 #include <stdio.h>
8 #include <unistd.h>
9 #include <stdlib.h>
10
11 int main(int argc, char *argv[]){
12     int i;
13
14     if(argc!=5){
15         printf("用法: Hello 2022112042 张天一 18245706985 秒数! \n");
16         exit(1);
17     }
18     for(i=0; i<10; i++){
19         printf("Hello %s %s %s\n", argv[1], argv[2], argv[3]);
20         sleep(atol(argv[4]));
21     }
22     getchar();
23     return 0;
24 }
```

图 3.3.8.1 函数调用

① 15 行，调用 `printf` 函数，编译器采用调用 `puts` 函数

```
28     movl    $1, %edi
29     call    exit
```

图 3.3.8.2 `hello.s` 的 `puts` 函数实现 `printf` 函数

② 16 行，调用 `exit` 函数，结束进程。

```
28     movl    $1, %edi
29     call    exit@PLT
```

图 3.3.8.2 .3.8.2 `hello.s` 的 `exit` 函数

③ 19 行，调用 `printf` 函数，实现 `printf("Hello %s %s %s\n", argv[1], argv[2], argv[3])`


```

33 .L4:
34     movq    -32(%rbp), %rax
35     addq    $24, %rax
36     movq    (%rax), %rcx
37     movq    -32(%rbp), %rax
38     addq    $16, %rax
39     movq    (%rax), %rdx
40     movq    -32(%rbp), %rax
41     addq    $8, %rax
42     movq    (%rax), %rax
43     movq    %rax, %rsi
44     movl    $.LC1, %edi
45     movl    $0, %eax
46     call    printf

```

图 3.3.8.3 hello.s 的 printf 函数

- ④ 调用 atoi 函数, 将字符串类型转化为整型(int)

```

47     movq    -32(%rbp), %rax
48     addq    $32, %rax
49     movq    (%rax), %rax
50     movq    %rax, %rdi
51     call    atoi

```

图 3.3.8.4 hello.s 的 atoi 函数

- ⑤ 20 行,调用 sleep 函数, 将 atoi 函数的返回作为 sleep 的参数调用,实现让函数进入 x 秒休眠, 如果在休眠结束前停止则会返回剩余时间。

```

53     call    sleep
54     addl    $1, -4(%rbp)

```

图 3.3.8.5 hello.s 的 sleep 函数

- ⑤ 22 行,调用 getchar 函数

```

55 .L3:
56     cmpl    $9, -4(%rbp)
57     jle     .L4
58     call    getchar
59     movl    $0, %eax

```

图 3.3.8.6 hello.s 的 getchar 函数

(3)函数返回

程序使用汇编指令 `ret` 从调用的函数中返回, 并且还原栈帧, 并且函数的返回值存放在寄存器 `%rax` 中。

```

61     .cfi_def_cfa 7, 8
62     ret
63     .cfi_endproc

```

图 3.3.8.7 hello.s 的函数返回

3.4 本章小结

本章介绍了编译的概念与作用, 并在 linux 下对 `hello.i` 进行了编译操作,分析了编译的生成结果 `hello.s` 的数据, 赋值,算术操作,关系操作,数组/指针/结构操作,控制

转移,函数操作方面的内容。

第 4 章 汇编

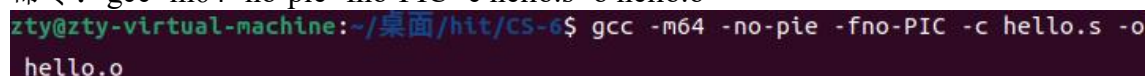
4.1 汇编的概念与作用

（一）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。

（二）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。

4.2 在 Ubuntu 下汇编的命令

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

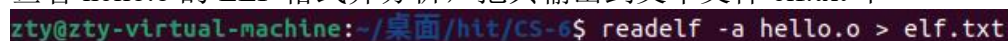


```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ gcc -m64 -no-pie -fno-PIC -c hello.s -o  
hello.o
```

图 4.2.1 汇编命令

4.3 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



```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ readelf -a hello.o > elf.txt
```

图 4.3.1 查看 elf 格式

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

如图 4.3.2 所示，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 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:                                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:                1088 (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
21

```

图 4.3.2 elf 头

(2) 节头表

如图 4.3.3 所示，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	节头:						
23	[号]	名称	类型	地址	偏移量		
24	大小	全体大小	旗标	链接	信息	对齐	
25	[0]		NULL		0000000000000000	00000000	
26		0000000000000000	0000000000000000		0	0	0
27	[1]	.text	PROGBITS		0000000000000000	00000040	
28		00000000000000a3	0000000000000000	AX	0	0	1
29	[2]	.rela.text	RELA		0000000000000000	000002f0	
30		00000000000000c0	0000000000000018	I	11	1	8
31	[3]	.data	PROGBITS		0000000000000000	000000e3	
32		0000000000000000	0000000000000000	WA	0	0	1
33	[4]	.bss	NOBITS		0000000000000000	000000e3	
34		0000000000000000	0000000000000000	WA	0	0	1
35	[5]	.rodata	PROGBITS		0000000000000000	000000e8	
36		0000000000000040	0000000000000000	A	0	0	8
37	[6]	.comment	PROGBITS		0000000000000000	00000128	
38		000000000000002c	0000000000000001	MS	0	0	1
39	[7]	.note.GNU-stack	PROGBITS		0000000000000000	00000154	
40		0000000000000000	0000000000000000		0	0	1
41	[8]	.note.gnu.pr[...]	NOTE		0000000000000000	00000158	
42		0000000000000020	0000000000000000	A	0	0	8
43	[9]	.eh_frame	PROGBITS		0000000000000000	00000178	
44		0000000000000038	0000000000000000	A	0	0	8
45	[10]	.rela.eh_frame	RELA		0000000000000000	000003b0	
46		0000000000000018	0000000000000018	I	11	9	8
47	[11]	.symtab	SYMTAB		0000000000000000	000001b0	
48		00000000000000108	0000000000000018		12	4	8
49	[12]	.strtab	STRTAB		0000000000000000	000002b8	
50		0000000000000032	0000000000000000		0	0	1
51	[13]	.shstrtab	STRTAB		0000000000000000	000003c8	
52		0000000000000074	0000000000000000		0	0	1
53	Key to Flags:						
54	W (write), A (alloc), X (execute), M (merge), S (strings), I (info),						
55	L (link order), O (extra OS processing required), G (group), T (TLS),						
56	C (compressed), x (unknown), o (OS specific), E (exclude),						

图 4.3.3 节头表

(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。如图 4.3.4 所示，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	重定位节 '.rela.text' at offset 0x2f0 contains 8 entries:						
66	偏移量	信息	类型	符号值	符号名称 + 加数		
67	0000000000001c	000300000002	R_X86_64_PC32	0000000000000000	.rodata - 4		
68	00000000000024	000500000004	R_X86_64_PLT32	0000000000000000	puts - 4		
69	0000000000002e	000600000004	R_X86_64_PLT32	0000000000000000	exit - 4		
70	00000000000062	000300000002	R_X86_64_PC32	0000000000000000	.rodata + 2c		
71	0000000000006f	000700000004	R_X86_64_PLT32	0000000000000000	printf - 4		
72	00000000000082	000800000004	R_X86_64_PLT32	0000000000000000	atoi - 4		
73	00000000000089	000900000004	R_X86_64_PLT32	0000000000000000	sleep - 4		
74	00000000000098	000a00000004	R_X86_64_PLT32	0000000000000000	getchar - 4		
75							
76	重定位节 '.rela.eh_frame' at offset 0x3b0 contains 1 entry:						
77	偏移量	信息	类型	符号值	符号名称 + 加数		
78	00000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0		
79	No processor-specific unwind information to decode						
--							

图 4.3.4 重定位节

(4) 符号表

如图 4.3.5 所示，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：

```
80
81 Symbol table '.symtab' contains 11 entries:
82   Num:      Value                Size Type Bind Vis      Ndx Name
83   0: 00000000000000000000      0 NOTYPE LOCAL DEFAULT UND
84   1: 00000000000000000000      0 FILE  LOCAL DEFAULT ABS hello.c
85   2: 00000000000000000000      0 SECTION LOCAL DEFAULT 1 .text
86   3: 00000000000000000000      0 SECTION LOCAL DEFAULT 5 .rodata
87   4: 00000000000000000000    163 FUNC  GLOBAL DEFAULT 1 main
88   5: 00000000000000000000      0 NOTYPE GLOBAL DEFAULT UND puts
89   6: 00000000000000000000      0 NOTYPE GLOBAL DEFAULT UND exit
90   7: 00000000000000000000      0 NOTYPE GLOBAL DEFAULT UND printf
91   8: 00000000000000000000      0 NOTYPE GLOBAL DEFAULT UND atoi
92   9: 00000000000000000000      0 NOTYPE GLOBAL DEFAULT UND sleep
93  10: 00000000000000000000      0 NOTYPE GLOBAL DEFAULT UND getchar
94
95 No version information found in this file.
96
97 Displaying notes found in: .note.gnu.property
98 所有者      Data size  Description
99  GNU          0x00000010      NT_GNU_PROPERTY_TYPE_0
100      Properties: x86 feature: IBT, SHSTK
```

图 4.3.5 符号表

4.4 Hello.o 的结果解析

(以下格式自行编排，编辑时删除)

objdump -d -r hello.o 分析 hello.o 的反汇编，并请与第 3 章的 hello.s 进行对照分析。

说明机器语言的构成，与汇编语言的映射关系。特别是机器语言中的操作数与汇编语言不一致，特别是分支转移函数调用等。

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：

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ objdump -d -r hello.o >asm.txt
```

图 4.4.1 hello.o 的结果解析

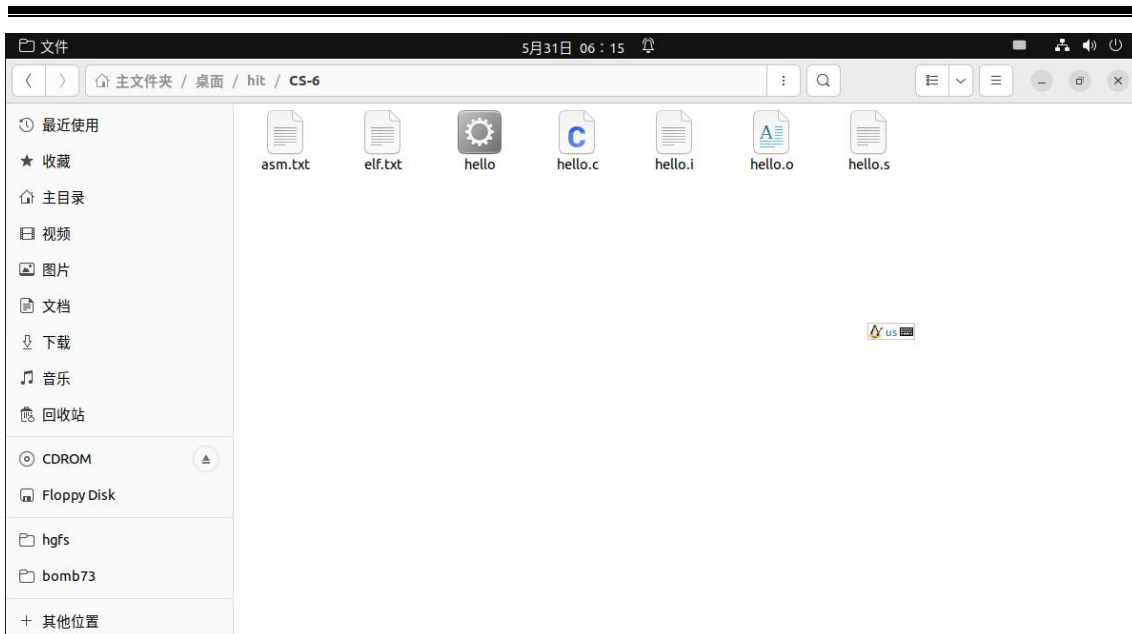


图 4.4.2 asm.txt 文件展示

对反汇编文件 asm.txt 和汇编程序 hello.s 对比分析如下：

如图所示左侧是 asm.txt 右侧是 hello.s 文件

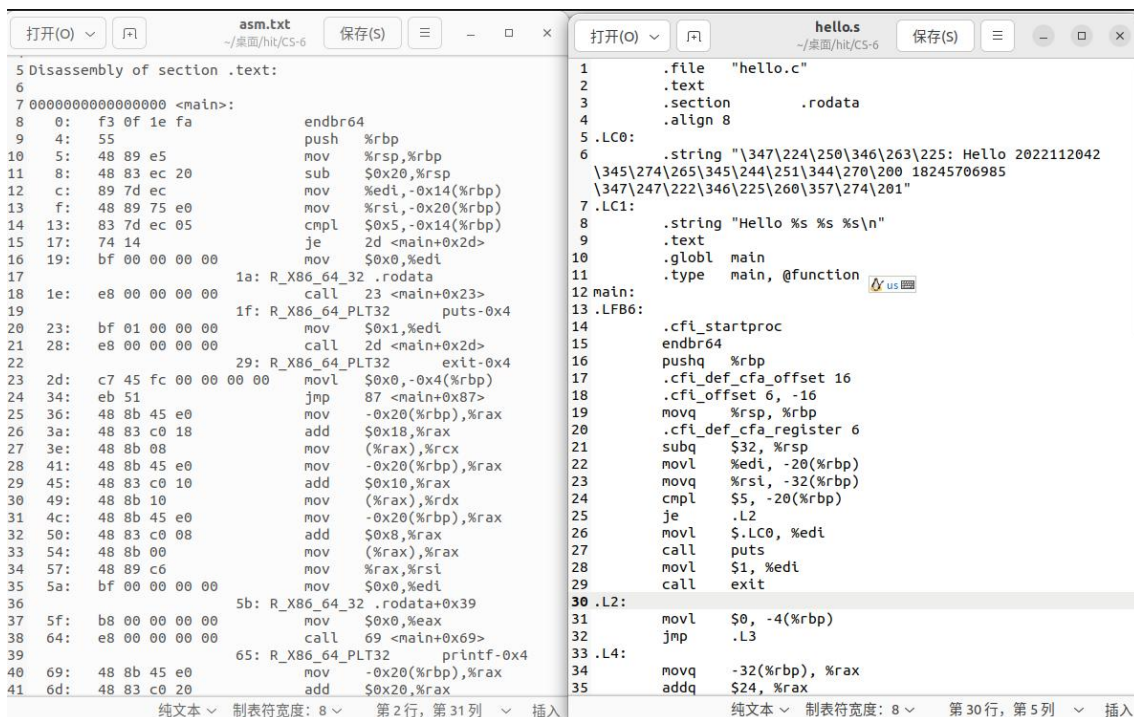


图 4.4.3 asm.txt 与 hello.s 对比

- (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。

4.5 本章小结

本章介绍了汇编的概念与作用,并在 Ubuntu 下实际操作进行汇编,将 `hello.s` 被转化为 `hello.o`,通过 `readelf` 分析了 `hello.o` 的 ELF 格式,分析了其各节的基本信息。同时还使用 `objdump` 将 `hello.o` 反汇编生成 `asm.txt` 反汇编文件,并与汇编程序 `hello.s` 文件进行比对分析。

第 5 章 链接

5.1 链接的概念与作用

（一）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。

（二）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。

5.2 在 Ubuntu 下链接的命令

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
```

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ 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
```

图 5.2.1 链接命令

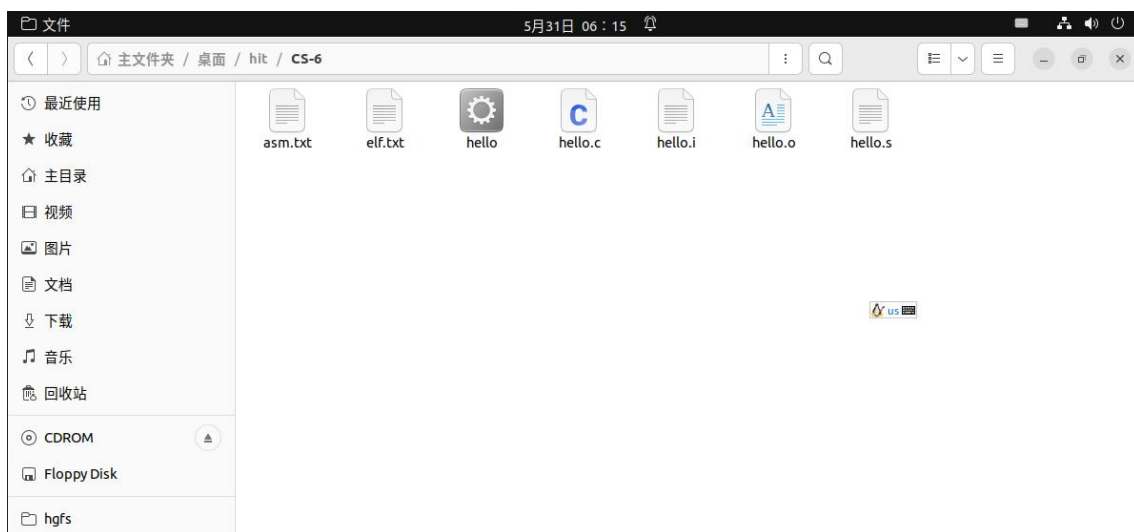


图 5.2.2 hello 链接文件展示

5.3 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：

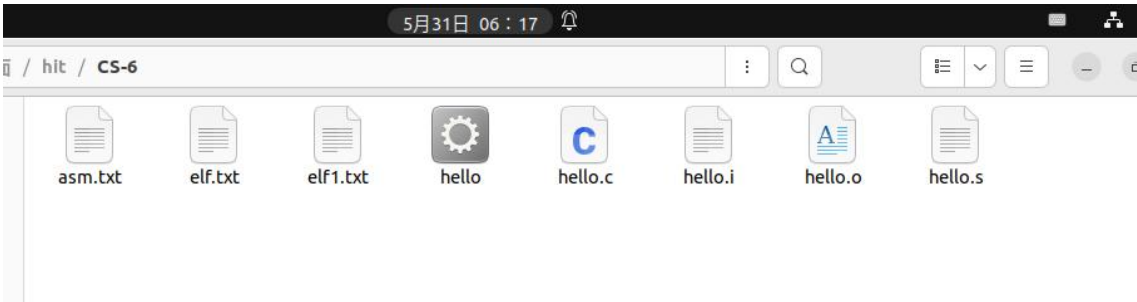


图 5.3.1 elf1.txt 文件展示

打开 elf1.txt，如图 5.3.2 所示，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 节头:

23	[号]	名称	类型	地址	偏移量
24	大小	全体大小	旗标	链接	信息 对齐
25	[0]		NULL	0000000000000000	00000000
26		0000000000000000	0000000000000000	0	0 0
27	[1]	.interp	PROGBITS	00000000004002e0	000002e0
28		000000000000001c	0000000000000000	A	0 0 1
29	[2]	.note.gnu.pr[...]	NOTE	0000000000400300	00000300
30		0000000000000030	0000000000000000	A	0 0 8
31	[3]	.note.ABI-tag	NOTE	0000000000400330	00000330
32		0000000000000020	0000000000000000	A	0 0 4
33	[4]	.hash	HASH	0000000000400350	00000350
34		0000000000000038	0000000000000004	A	6 0 8
35	[5]	.gnu.hash	GNU_HASH	0000000000400388	00000388
36		000000000000001c	0000000000000000	A	6 0 8
37	[6]	.dynsym	DYNSYM	00000000004003a8	000003a8
38		00000000000000d8	0000000000000018	A	7 1 8
39	[7]	.dynstr	STRTAB	0000000000400480	00000480
40		0000000000000067	0000000000000000	A	0 0 1
41	[8]	.gnu.version	VERSYM	00000000004004e8	000004e8
42		0000000000000012	0000000000000002	A	6 0 2
43	[9]	.gnu.version_r	VERNEED	0000000000400500	00000500
44		0000000000000030	0000000000000000	A	7 1 8
45	[10]	.rela.dyn	RELA	0000000000400530	00000530
46		0000000000000030	0000000000000018	A	6 0 8
47	[11]	.rela.plt	RELA	0000000000400560	00000560
48		0000000000000090	0000000000000018	AI	6 21 8
49	[12]	.init	PROGBITS	0000000000401000	00001000
50		000000000000001b	0000000000000000	AX	0 0 4
51	[13]	.plt	PROGBITS	0000000000401020	00001020
52		0000000000000070	0000000000000010	AX	0 0 16
53	[14]	.plt.sec	PROGBITS	0000000000401090	00001090
54		0000000000000060	0000000000000010	AX	0 0 16
55	[15]	.text	PROGBITS	00000000004010f0	000010f0

图 5.3.2 elf1.txt 节头

5.4 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 5.3 对照分析说明。

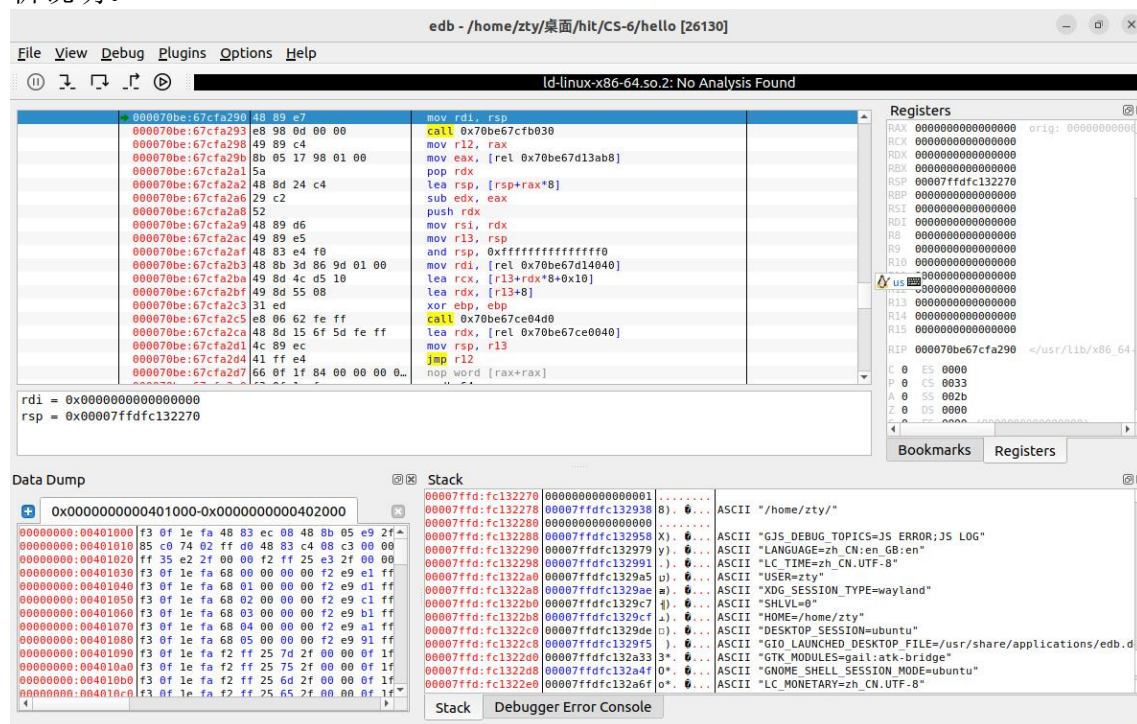


图 5.4.1 edb 查看虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息，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。

偏移	名称	类型	地址	值	偏移
0x00007738b37e620	ld-2.31.so! rld_global ro	全球大小	0000000000000000	0	0
0x00007738b37e750	ld-2.31.so! libc_stack sec	大小	NULL	0000000000000000	0
0x00007738b37e760	ld-2.31.so! libc_enable sec	全局	0000000000000000	0	0
0x00007738b37ef000	ld-2.31.so! rld_global	0000000000000000	0000000000000000	0	0
0x00007738b37ef160	ld-2.31.so! r debug	PROGBITS	0000000000049820	000000260	0
0x0000000000000000	.init	0000000000000000	0000000000000000	0	0
0x0000000000000000	.note.gnu.property	NOTE	0000000000040030	000000320	0
0x0000000000000000	.note.gnu.property	PROGBITS	0000000000040030	000000320	0
0x0000000000000000	hello.note.gnu.property#x20	0000000000000000	A 0 0 0	0	0
0x0000000000000000	hello.note.ABI-tag#x26	NOTE	0000000000040030	000000320	0
0x0000000000000000	hello.libc.so.8	0000000000000000	A 0 0 4	0	0
0x0000000000000000	hello.dynsym	HASH	0000000000040030	000000340	0
0x0000000000000000	hello.gnu.version-d	00000000000000000004	0000000000040030	000000340	0
0x0000000000000000	hello.dynstr#x5c	GNU HASH	0000000000040030	000000378	0
0x0000000000000000	hello.gnu.version r	00000000000000000001	A 0 0 0	0	0
0x0000000000000000	hello.gnu.version-d#x20	DYNSTR	0000000000049830	000000398	0
0x0000000000000000	hello.rela.dyn#x26	0000000000000000001B	0000000000040040	000000470	0
0x0000000000000000	hello.init	STRTAB	0000000000040040	000000470	0
0x0000000000000000	hello.gnu.version	0000000000000000005c	A 0 0 0	0	0
0x0000000000000000	hello.plt	VERSYM	000000000004004c	0000004cc	0
0x0000000000000000	hello.plt	00000000000000000012	A 0 0 2	0	0
0x0000000000000000	hello.puts@plt	00000000000000000012	000000000004004c	0000004cc	0
0x0000000000000000	hello.get@plt	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.getchar@plt	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.toupper@plt	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.toupper@plt	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.sleep@plt	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.plt#x676	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.plt.sec#x676	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.start	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.dl_relocate static pie	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.libc.so.8 init	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.libc.so.8 fini	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.fini	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.rodata	00000000000000000020	000000000004004c	0000004cc	0
0x0000000000000000	hello.fini	00000000000000000020	000000000004004c	0000004cc	0

图 5.4.2 虚拟地址空间对比

.init 节: 该段代表程序的开始, 存放着指令的机器码. 可知该段地址从 0x401000 开始, 偏移量为 0x1000.

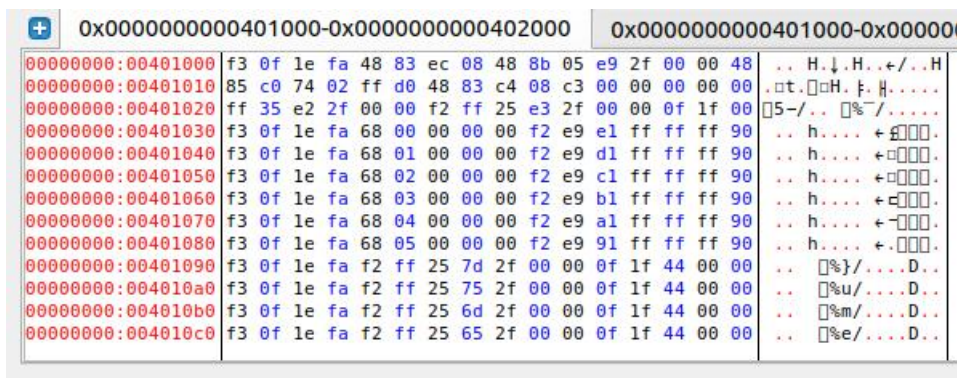


图 5.4.3 edb 查看.init 节

5.5 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` 如图 5.5.1 和图 5.5.2 所示，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：

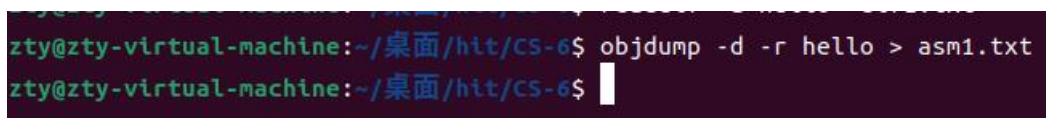


图 5.5.1 重定位命令

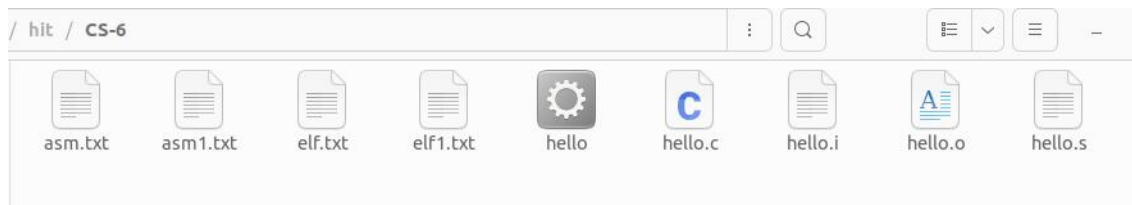


图 5.5.2 asm1.txt 文件展示

（1）为指令分配虚拟地址

如图 5.5.3 所示，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。

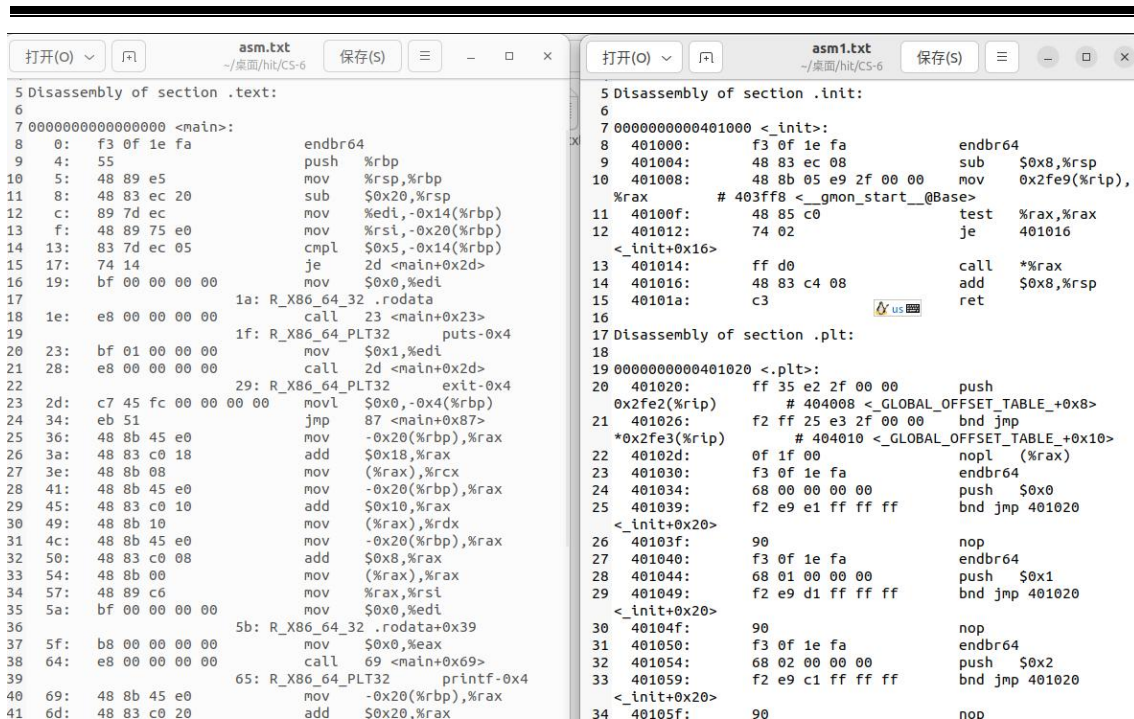


图 5.5.3 为指令分配虚拟地址

(2) 函数调用

如图 5.5.4 所示，在左侧 hellico.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，而右侧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。

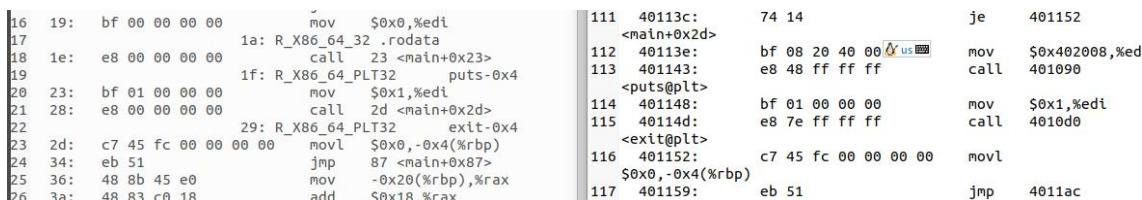


图 5.5.4 函数调用

(3) 跳转指令

同理函数调用，链接后可以采用虚拟地址跳转，如图 5.5.5 所示

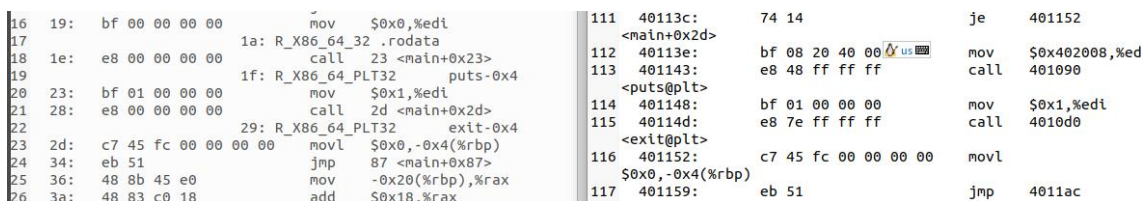


图 5.5.5 跳转指令

(4) 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，如图 5.5.6 所示。

```

48 Disassembly of section .plt.sec:
49
50 0000000000401090 <puts@plt>:
51 401090: f3 0f 1e fa          endbr64
52 401094: f2 ff 25 7d 2f 00 00 bnd jmp *0x2f7d(%rip)    # 404018 <puts@GLIBC_2.2.5>
53 40109b: 0f 1f 44 00 00       nopl 0x0(%rax,%rax,1)
54
55 00000000004010a0 <printf@plt>:
56 4010a0: f3 0f 1e fa          endbr64
57 4010a4: f2 ff 25 75 2f 00 00 bnd jmp *0x2f75(%rip)    # 404020 <printf@GLIBC_2.2.5>
58 4010ab: 0f 1f 44 00 00       nopl 0x0(%rax,%rax,1)
59
60 00000000004010b0 <getchar@plt>:
61 4010b0: f3 0f 1e fa          endbr64
62 4010b4: f2 ff 25 6d 2f 00 00 bnd jmp *0x2f6d(%rip)    # 404028 <getchar@GLIBC_2.2.5>
63 4010bb: 0f 1f 44 00 00       nopl 0x0(%rax,%rax,1)
64
65 00000000004010c0 <atoi@plt>:
66 4010c0: f3 0f 1e fa          endbr64
67 4010c4: f2 ff 25 65 2f 00 00 bnd jmp *0x2f65(%rip)    # 404030 <atoi@GLIBC_2.2.5>
68 4010cb: 0f 1f 44 00 00       nopl 0x0(%rax,%rax,1)
69
70 00000000004010d0 <exit@plt>:
71 4010d0: f3 0f 1e fa          endbr64
72 4010d4: f2 ff 25 5d 2f 00 00 bnd jmp *0x2f5d(%rip)    # 404038 <exit@GLIBC_2.2.5>
73 4010db: 0f 1f 44 00 00       nopl 0x0(%rax,%rax,1)
74
75 00000000004010e0 <sleep@plt>:
76 4010e0: f3 0f 1e fa          endbr64
77 4010e4: f2 ff 25 55 2f 00 00 bnd jmp *0x2f55(%rip)    # 404040 <sleep@GLIBC_2.2.5>
78 4010eb: 0f 1f 44 00 00       nopl 0x0(%rax,%rax,1)
79

```

图 5.5.6 链接过程插入 C 语言标准库函数

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。

5.6 `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：

<_init>: 401000

<.plt>: 401020

<puts@plt> : 401090

```
<printf@plt>: 4010a0
<getchar@plt>: 4010b0
<atoi@plt>: 4010c0
<exit@plt>: 4010d0
<sleep@plt>: 4010e0
<_start>: 4010f0
<_dl_relocate_static_pie>: 401120
<main> : 401125
<_libc_scu_init>: 4011c0
<_libc_csu_fini>: 401230
<_fini>: 401238
```

5.7 Hello 的动态链接分析

当程序在调用共享函数库的时候，我们无法预测这个函数的地址，因为定义他的模块可以在再运行时加载到任意位置。此时，使用延迟绑定的策略，把这个过程地址的加载推迟到第一次调用该进程的时刻。

延迟绑定是通过两个数据结构之间的交互来实现的，分别是 GOT 和 PLT, GOT 是数据段的一部分，而 PLT 是代码段的一部分。PLT 与 GOT 的协作可以在运行时解析函数的地址，实现函数的动态链接。

过程链接表 PLT 是一个数组，每个条目是 16 字节代码。PLT[0]是一个特殊条目，跳转到动态链接器中。每个被可执行程序调用的库函数都有它自己的 PLT 条目。

动态连接器使用 GOT 和 PLT 实现函数的动态链接。其中 GOT 存放的是函数的目标地址，PLT 使用 GOT 中地址跳转到目标函数。GOT 和 PLT 信息如下图所示：

	000000000000001a0	0000000000000010	WA	7	0	8
[20]	.got	PROGBITS	0000000000403ff0	00002ff0		
	0000000000000010	0000000000000008	WA	0	0	8
[21]	.got.plt	PROGBITS	0000000000404000	00003000		
	0000000000000048	0000000000000008	WA	0	0	8
[22]	.data	PROGBITS	0000000000404048	00003048		
	0000000000000004	0000000000000008	WA	0	0	8

图 5.7.1 GOT 和 PLT 信息

可以在 edb 中找到该节的内容并观察其在运行 dl_int 前后内容的变化，如图 5.7.2 和 5.7.3 所示，有内容发生变化。

10	0e 60 00 00 00 00 00 00	00 00 00 00 00 00 00 00	b6	04 40 00 00 00 00 00 00	...
c6	04 40 00 00 00 00 00 00	d6	04 40 00 00 00 00 00 00	...	
e6	04 40 00 00 00 00 00 00	f6	04 40 00 00 00 00 00 00	...	
00	00 00 00 00 00 00 00 00	00	00 00 00 00 00 00 00 00	...	

图 5.7.2 调用 dl_init 前的情况

10	0e	60	00	00	00	00	00	70	41	90	63	b5	7f	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																		
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

图 5.7.3 调用 dl_init 后的情况

5.8 本章小结

本章介绍了链接的概念和作用，链接的命令，分析了可执行目标文件 hello 的格式，通过 edb 展示了 hello 的虚拟地址空间，分析链接重定位过程，分析 hello 的执行流程，动态链接过程。

第 6 章 hello 进程管理

6.1 进程的概念与作用

(一) **概念**: 进程的经典定义就是一个执行中程序的实例, 系统中的每个程序都运行在某个进程的上下文中, 上下文是由程序正确运行所需的状态组成的, 这个状态包括存放在内存中的程序的代码和数据、它的栈、通用目的寄存器的内容, 程序计数器、环境变量, 以及打开文件描述的集合。

(二) **作用**: 提供给应用程序两个关键抽象, (1) 一个独立的逻辑控制流, 它提供一个假象, 好像我们的程序独占的使用处理器。(2) 一个私有的地址空间, 它提供一个假象, 好像我们的程序独占的使用内存系统

6.2 简述壳 Shell-bash 的作用与处理流程

(一) **作用**: Shell 是一种命令行解释器, 它为应用程序的执行提供一个界面, 用户通过这个界面访问操作系统内核的服务, shell 读取用户输入的字符解释并执行。

(二) **处理流程**:

(1) shell 从终端读入用户输入的命令。

(2) 将输入字符串进行划分以获得所有参数。

(3) 判断是否是内置命令, 是则立即执行, 否则调用相应的程序为其分配子进程并运行。

(4) Shell 接受键盘输入信号并对这些信号进行相应处理。

6.3 Hello 的 fork 进程创建过程

在终端输入./hello 的输入命令后, shell 进行命令行解释, 由于不是内置 shell 指令, shell 会调用 fork 函数创建一个子进程并执行可执行程序 hello。新创建的子进程几乎与父进程相同, 子进程得到与父进程用户及虚拟地址空间相同的但是独立的一份副本, 包括代码和数据段、堆、共享库以及用户栈, 子进程还获得与父进程任何打开文件, 描述符相同的副本, 这就意味着, 当父进程调用 fork 函数时, 子进程可以读写父进程中打开的任何文件。

6.4 Hello 的 execve 过程

执行 exexve 过程时, 会加载并运行程序。加载器将删除子进程现有的虚拟内

存段，创建一组新的段（该栈堆的初始化为 0），新程序启动后栈的结构如图所示，并将虚拟地址空间中的页映射到可执行文件的页大小的片 chunk 中，新的代码段与数据段被初始化为可执行文件的内容，然后转到_start。执行过程中还会覆盖当前进程的代码段，数据段，栈，保留有相同的 PID，继承已经打开的文件描述符和信号上下文，exexeve 函数调用一次且一般不返回（有错误情况除外）。

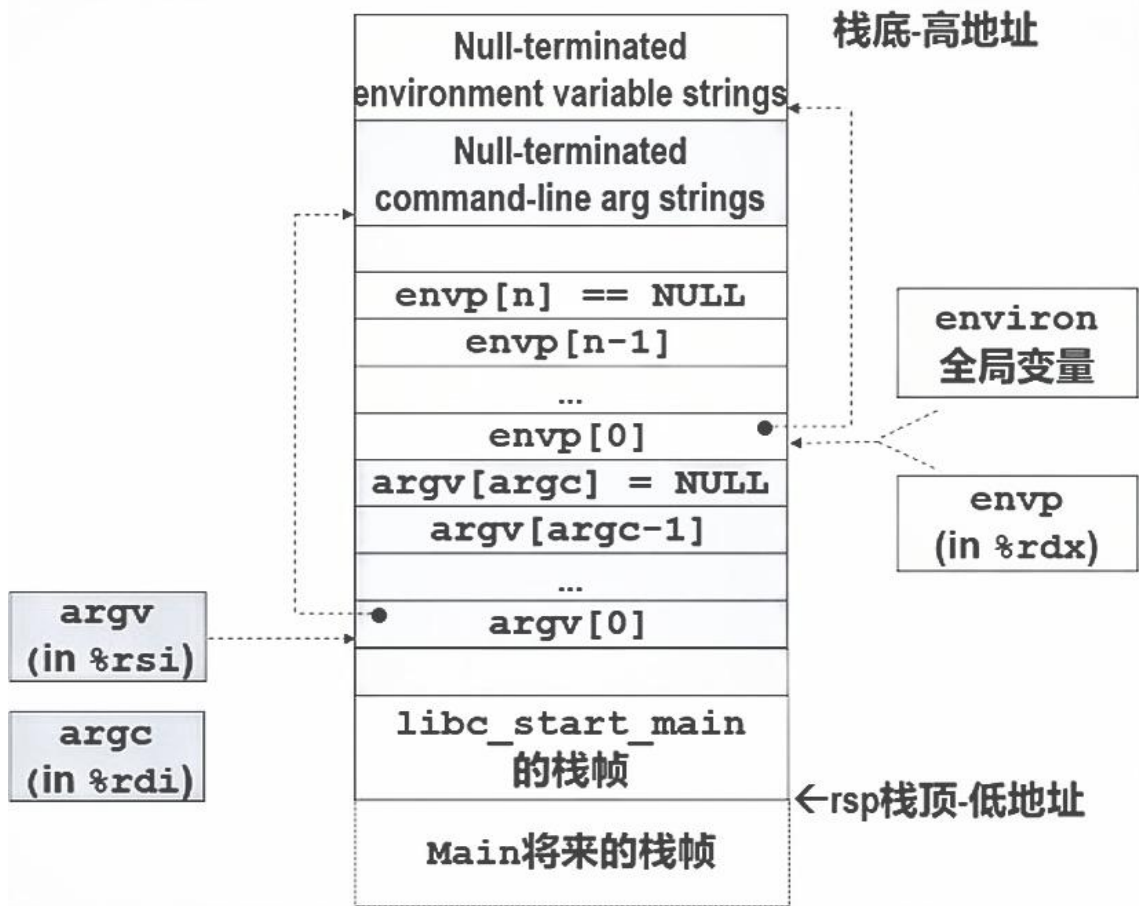


图 6.4.1 新栈的结构

6.5 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

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` 休眠结束，此时再次完成上下文切换，恢复休眠前的上下文信息，此时控制权送回 `hello` 并继续执行。循环结束后，程序调用 `getchar()`，`hello` 从用户模式进入内核模式，并再次上下文切换，控制交付给其他进程。最后，内核会从其他进程回到 `hello` 进程。

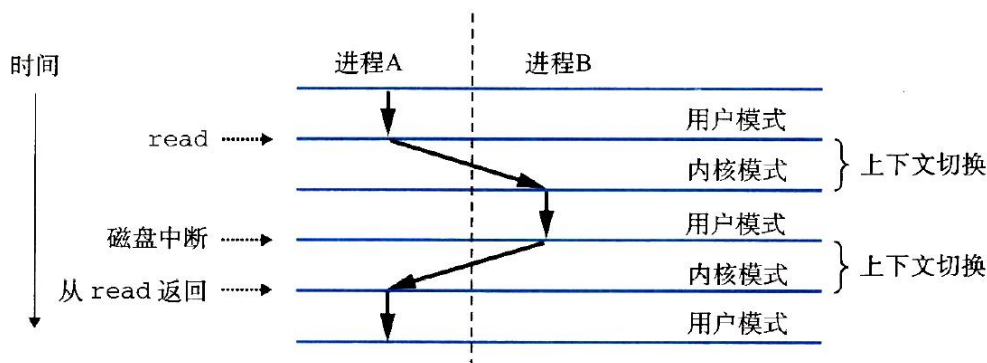


图 6.5.1 进程上下文切换的流程

6.6 `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。

6.6.1 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

过程如图 6.6.1 所示。

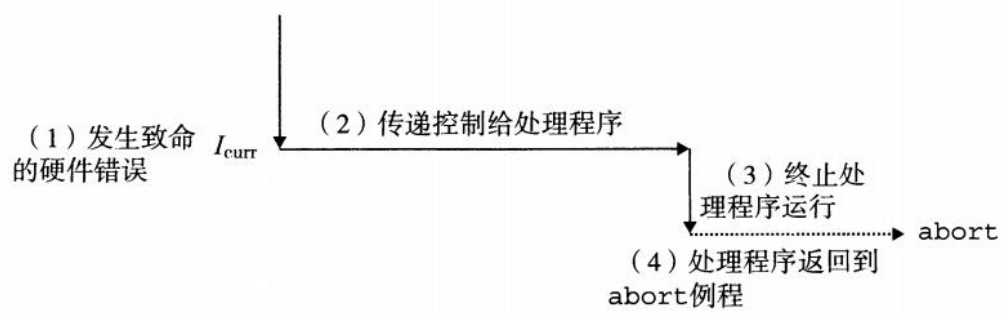


图 6.6.1 中断的处理过程

(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。处理过程如图 6.6.2 所示。

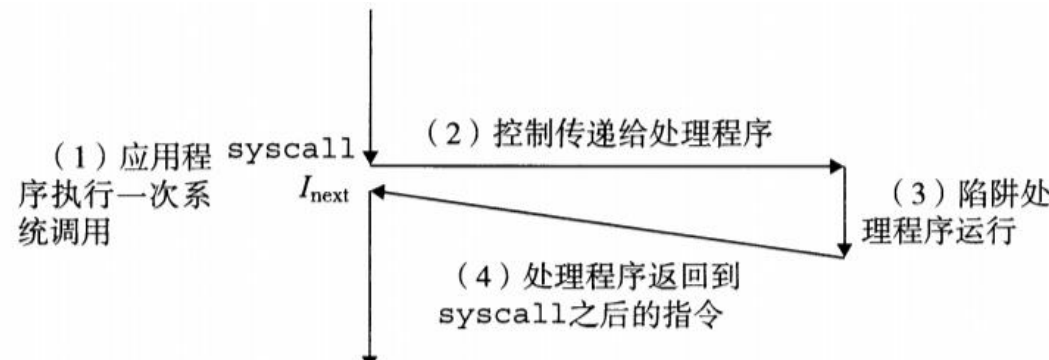


图 6.6.2 陷阱的处理过程

(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，其处理过程如图 6.6.3 所示。

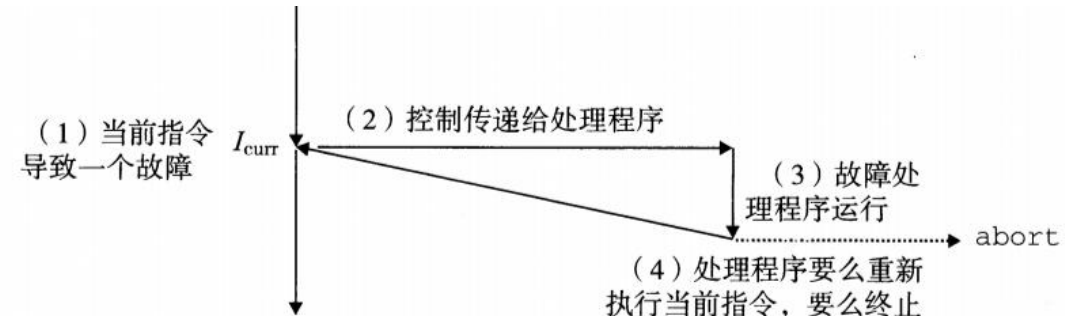


图 6.6.3 故障的处理过程

(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。其处理过程如图 6.6.4 所示。

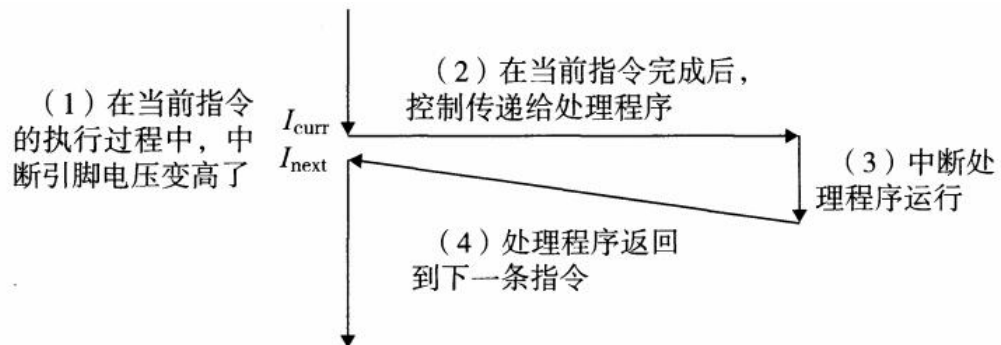


图 6.6.4 终止的处理过程

6.6.2 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。

6.6.3 程序的运行

(1) 正常运行

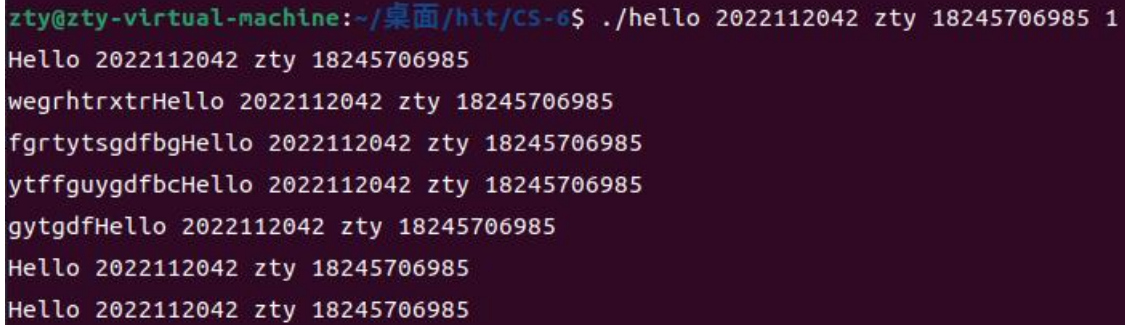
在终端输入 ./hello 2022112042 zty 18245706985 1，终端每隔 1 秒打印输出一次 Hello 2022112042 zty 18245706985，共 5 次。输出后不会立刻结束程序，由于有一个 getchar（）函数的存在，键入一个回车后停止程序执行。如图 6.6.5 所示。

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ./hello 2022112042 zty 18245706985 1
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
zty@zty-virtual-machine:~/桌面/hit/CS-6$
```

图 6.6.5 正常运行

(2) 不停乱按

执行过程中不断乱按，会把按的内容显示在屏幕上，最后依然要由 `getchar()` 函数接收到键盘输入的一个回车后结束执行。

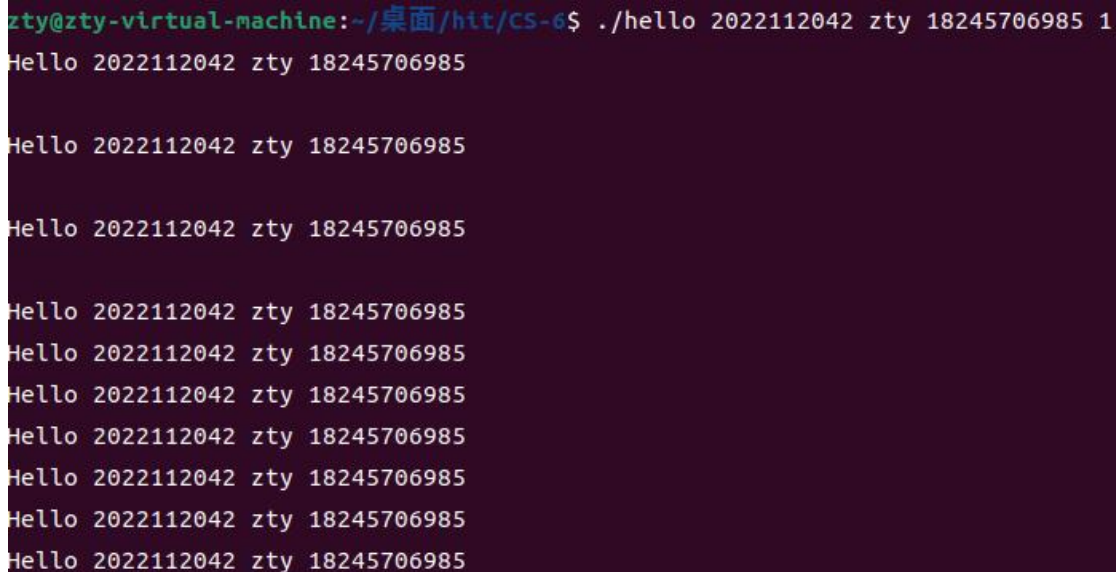


```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ./hello 2022112042 zty 18245706985 1
Hello 2022112042 zty 18245706985
wegrhtrxtrHello 2022112042 zty 18245706985
fgrtytsgdfbgHello 2022112042 zty 18245706985
ytffguygdfbcHello 2022112042 zty 18245706985
gytgdfHello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
```

图 6.6.6 不断乱按

(3) 输入回车

在程序执行过程中按入四个回车，它们被保存在缓冲区中，最后程序结束时，最后一个回车被 `getchar()` 读走，结束程序，还有两个回车会输出出来，如图 6.6.7 所示，程序结束后，会有三个换行输出。



```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ./hello 2022112042 zty 18245706985 1
Hello 2022112042 zty 18245706985

Hello 2022112042 zty 18245706985

Hello 2022112042 zty 18245706985

Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
```

图 6.6.7 键入四个回车

(4) 按下 Ctrl-C

在执行过程中如果按下 `Ctrl-C` 会立即停止程序的执行。因为这个键盘输入会导致发送 `SIGINT` 信号给 `hello`，该信号要求 `hello` 立刻终止该进程，如图 6.6.8 所示。


```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ./hello 2022112042 zty 18245706985 1
Hello 2022112042 zty 18245706985
^C
zty@zty-virtual-machine:~/桌面/hit/CS-6$
```

图 6.6.8 按下 Ctrl-C 结束进程

(5) 按下 Ctrl-Z 及相关操作

程序在运行过程中按下 Ctrl-Z 会产生中断异常，发送 SIGSTP 信号，暂时挂起 hello 进程并打印相关信息。如图 6.6.9 所示。

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ./hello 2022112042 zty 18245706985 1
Hello 2022112042 zty 18245706985
Hello 2022112042 zty 18245706985
^Z
[1]+ 已停止                  ./hello 2022112042 zty 18245706985 1
```

图 6.6.9 暂时挂起进程

且挂起后还可以执行若干命令，如输入 ps，将打印各进程的 PID，如图 6.6.10 所示：

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ps
```

PID	TTY	TIME	CMD
28360	pts/0	00:00:00	bash
28494	pts/0	00:00:00	hello
28495	pts/0	00:00:00	ps

图 6.6.10 打印各进程 PID

输入 jobs 将打印出被挂起的 hello 的相关信息，如图 6.6.11 所示：

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ jobs
[1]+ 已停止                  ./hello 2022112042 zty 18245706985 1
```

图 6.6.11 打印 hello 的相关信息

输入 pstree 指令将打印进程树，如图 6.6.12 所示：


```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ kill -9 28501
zty@zty-virtual-machine:~/桌面/hit/CS-6$ kill -9 28502
bash: kill: (28502) - 没有那个进程
[1]+ 已杀死                  ./hello 2022112042 zty 18245706985 1
```

图 6.6.14 杀死进程

再次用 ps 查看，发现 hello 已经被终止.hello 进程被杀死了。

```
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ps
  PID TTY          TIME CMD
 28360 pts/0        00:00:00 bash
 28501 pts/0        00:00:00 hello
 28502 pts/0        00:00:00 ps
zty@zty-virtual-machine:~/桌面/hit/CS-6$ kill -9 28501
zty@zty-virtual-machine:~/桌面/hit/CS-6$ kill -9 28502
bash: kill: (28502) - 没有那个进程
[1]+ 已杀死                  ./hello 2022112042 zty 18245706985 1
zty@zty-virtual-machine:~/桌面/hit/CS-6$ ps
  PID TTY          TIME CMD
 28360 pts/0        00:00:00 bash
 28505 pts/0        00:00:00 ps
```

图 6.6.15 查看杀死进程

6.7 本章小结

本章介绍了进程的概念以及作用，详细分析了 shell, fork, execve 以及进程的调度和上下文的切换。最后分析了异常的种类，并且用具体的命令分析了不同情况下信号的处理。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

- 1、**逻辑地址**：程序经过编译后出现在汇编代码中的地址。
- 2、**线性地址**：逻辑地址向物理地址转换时的中间环节，hello 中的偏移地址加上相应段的基地址就是线性地址。
- 3、**虚拟地址**：就是线性地址
- 4、**物理地址**：实际出现在 CPU 外部地址总线上的地址信号，表示实的物理内存所对应的地址。

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。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。

转换的具体过程：首先，给定一个完整的逻辑地址，它的格式如下：[段选择符：段内偏移地址]然后，看段选择符的 T1=0 还是 1，知道当前要转换是全局段描述符表（存储全局的段描述符），还是局部段描述符表（存储进程自己的段描述符），再根据相应寄存器，得到其地址和大小，得到一个数组。最后，拿出段选择符中前 13 位，查找到对应的段描述符，进而找到基地址， $base+offset$ 得到线性地址。

7.3 Hello 的线性地址到物理地址的变换-页式管理

线性地址到物理地址的变换由页式管理实现，它通过分页机制对虚拟内存空间进行分页，然后把页式虚拟地址与物理内存地址建立一一对应页表，并用相应的硬件地址变换机构（MMU）来解决离散地址变换问题。页式管理采用请求调页或预调页技术实现了内外存存储器的统一管理。页表是一个由页表项（PTE）组成的数组，存储在内存中，将虚拟页地址映射到物理页地址。

如下图所示，一个虚拟地址（VA）包含两个部分：虚拟页号（VPN）和虚拟页偏移量（VPO），其中 VPO 和 PPO（物理页偏移量）是相同的。MMU 利用 VPN 选择适当的 PTE，如果 PTE 的有效位为 1，也即 PTE 命中，则直接将 PTE 中存储

的物理页号 (PPN) 和虚拟地址中的虚拟页偏移量 (VPO) 串联起来就得到一个相应的物理地址。如果页表项 (PTE) 不命中, 则会触发缺页故障, 调用缺页处理子程序进行相应处理。

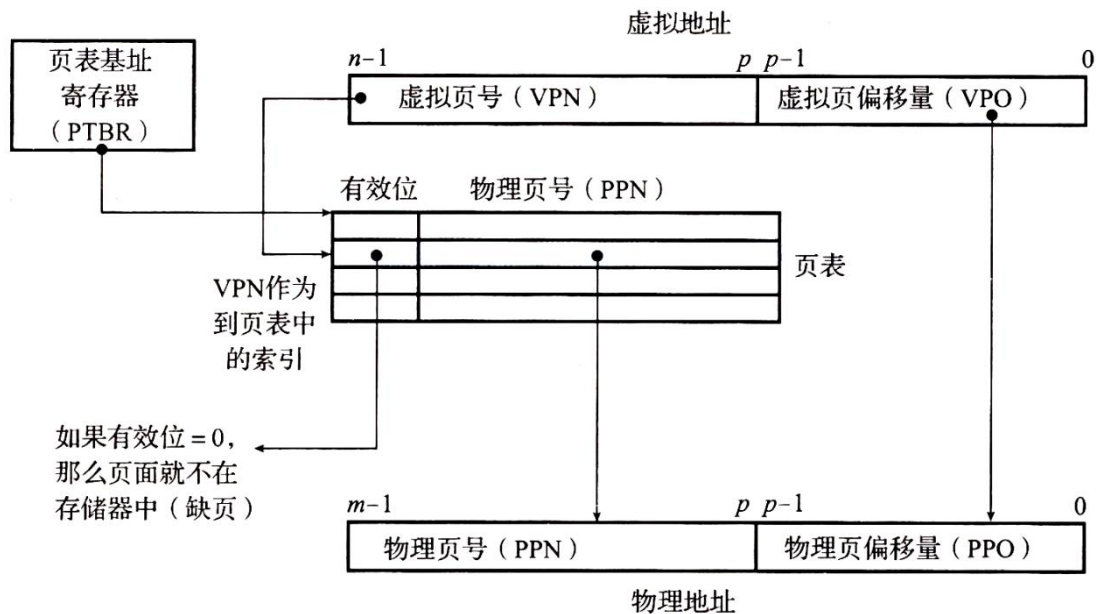


图 7.3.1 使用页表的地址翻译

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

首先在 TLB 中查找 PTE, 若能直接找到则直接得到对应的 PPN, 具体的操作是将 VPN 看作 TLBI 和 TLBT, 前者是组号, 后者是标记, 根据 TLBI 去对应的组找, 如果 TLBT 能够对应的话, 则能够直接得到 PTE, 进而得到 PPN。

其中若是在 TLB 中找不到对应的条目, 则应去多级页表中查找, VPN 被分为了四块。有一个叫做 CR3 的寄存器包含 L1 页表的物理地址, VPN1 提供到了一个 L1

PTE 的偏移量, 这个 PTE 包含 L2 页表的基地址, VPN2 提供一个 L2 PTE 的偏移量。依次类推, 最终找到页表中的 PTE, 得到 PPN。

而 VPO 和 PPO 相等, 最终的 PA 等于 PPN+PPO。

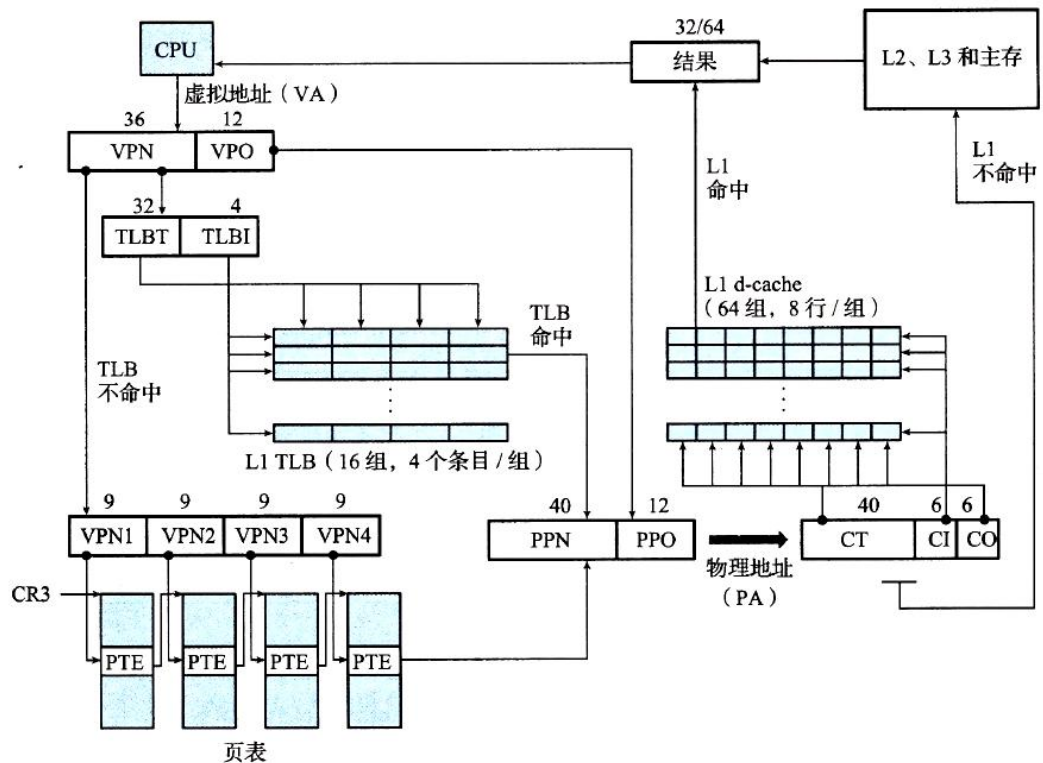


图 7.4.1 TLB 与四级页表支持下的地址翻译

7.5 三级 Cache 支持下的物理内存访问

对于一个虚拟地址请求，首先将去 TLB 寻找，看是否已经在 TLB 中缓存。如果命中的话就直接 MMU 获取，没有命中的话就先在结合多级页表，得到物理地址，去 cache 中找，到了 L1 里面以后，寻找物理地址又要检测是否命中，不命中则紧接着寻找下一级 cache L2，接着 L3。这里就是使用到 CPU 的高速缓存机制了，逐级向下找，直到找到对应的内容。

7.6 hello 进程 fork 时的内存映射

当 fork 函数被当前进程调用时，操作系统内核为新进程创建各种数据结构，并分配给它一个唯一的 PID，然后通过以下步骤为其创建虚拟内存：

- ①创建当前进程的 mm_struct、区域结构和页表的原样副本；
- ②将两个进程中的每个页面都标记为只读；
- ③将两个进程中的每个区域结构都标记为私有的写时复制；

当 fork 在新进程中返回时，新进程现在的虚拟内存刚好和调用 fork 时存在的虚拟内存相同。当这两个进程中的任一个在后面进行写操作时，写时复制机制就

会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

7.7 hello 进程 execve 时的内存映射

execve 函数的函数声明为 `int execve(char *filename, char *argv[], char *envp[])`;加载 hello 并执行需要以下几个步骤：

(1) 删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。

(2) 映射私有区域。为新程序的代码、数据、bss 和栈区域创建新的区域结构。所有这些新的区域都是私有的、写时复制的。

(3) 映射共享区域。如果 filename 程序与共享对象(或目标)链接，那么这些对象是动态链接到这个程序的，然后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)。execve 做的最后一件事情就是设置当前进程上下文中的程序计数器，使之指向代码区域的入口点。

当再次调度这个进程时，它将从入口点开始执行。Linux 将根据需要换入代码和数据页面。

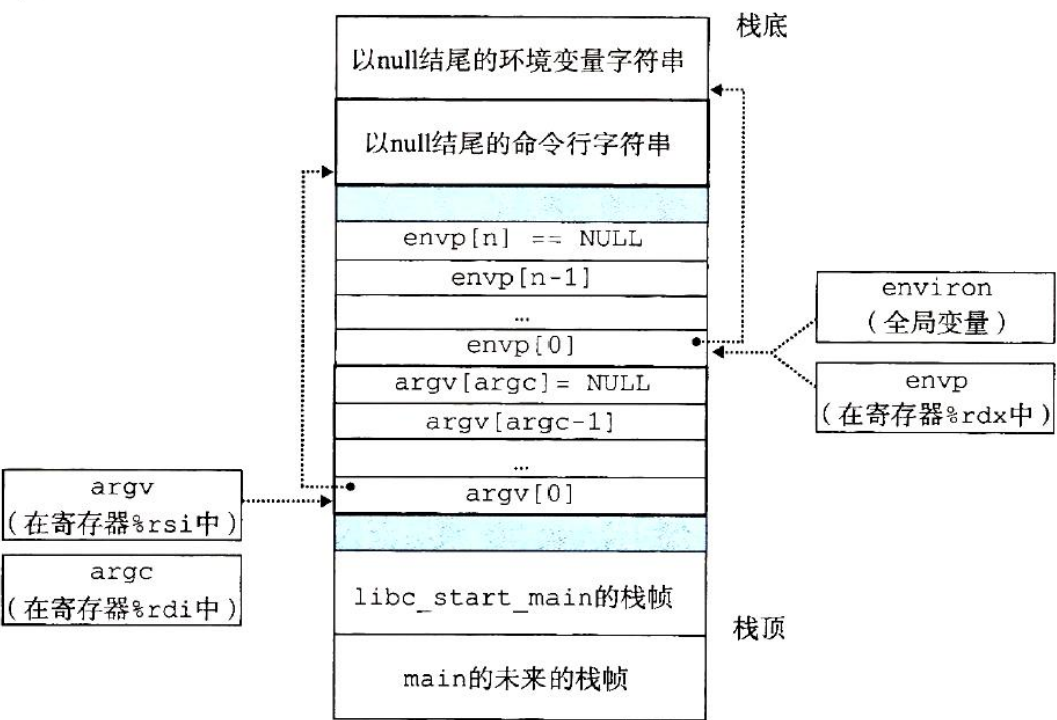


图 7.7.1 用户栈典型组织结构

7.8 缺页故障与缺页中断处理

当发生缺页中断时，系统的处理流程如下图所示：

- ①处理器将虚拟地址发送给 MMU；
- ②MMU 生成 PTE 地址（PTEA），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；
- ④PTE 有效位为 0, MMU 触发缺页异常，传递 CPU 中的控制到操作系统内核的缺页异常处理程序；
- ⑤缺页处理程序确定出物理内存中的牺牲页，若页面被修改，则把它写回到磁盘中。
- ⑥缺页处理程序调入新的页面，并更新内存中的 PTE；
- ⑦缺页处理程序返回到原进程，再次执行导致缺页的指令，CPU 将 VA 重新送给 MMU 并执行相应访问操作，此时将不会再出现缺页的情况。

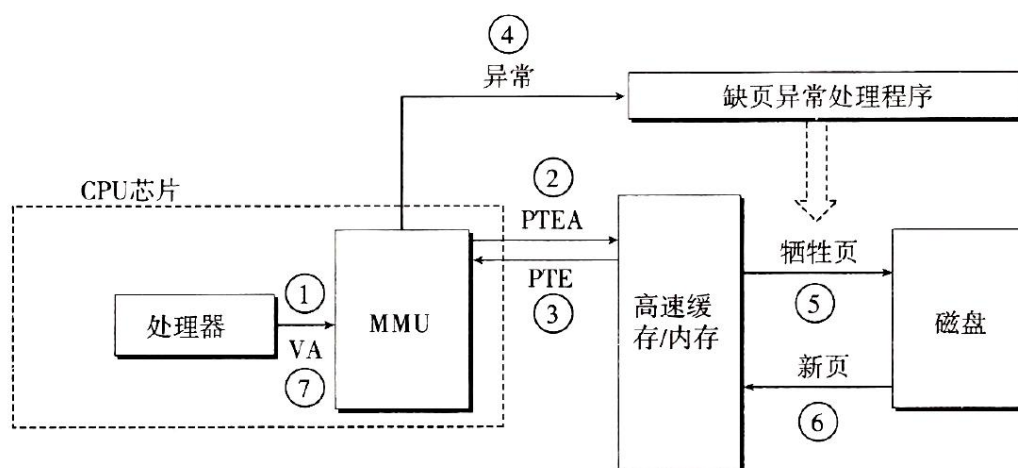


图 7.8.1 缺页操作

7.9 动态存储分配管理

动态内存分配是指在程序运行时程序员使用动态内存分配器（如 `malloc`）获得虚拟内存。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。堆被分配器视为一组不同大小的块的集合，每个块仅会是以下两种状态之一：“已分配”和“空闲”。已分配的块显式保留为应用程序使用；空闲块保持空闲直到被分配使用。

分配器有两种方式：显式分配器和隐式分配器。显式分配器要求应用显式地释放任何已分配的块，如 C 语言中的 `malloc` 和 `free` 的组合对内存进行分配与释放；隐式分配器要求分配器自动完成上述任务。

常用的分配器有显式链表和隐式链表。前者的特点是堆中的空闲块通过头部

中的大小字段隐含地连接；后者则是在在每个空闲块中，都包含一个前驱（pred）与后继（succ）指针，从而减少了搜索与适配的时间。

7.10 本章小结

本章介绍了储存器的地址空间，讲述了虚拟地址、物理地址、线性地址、逻辑地址的概念，还有进程 `fork` 和 `execve` 时的内存映射的内容。描述了系统如何应对缺页异常。

第 8 章 hello 的 IO 管理

8.1 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。

8.2 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

8.3 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。

8.4 getchar 的实现分析

在异步异常-键盘中断的处理过程中，当用户按下键盘时，键盘接口会接收到代表该按键的键盘扫描码，并产生中断请求，系统会调用键盘中断处理子程序。该处理子程序首先从键盘接口获取按键的扫描码，然后将扫描码转换成对应的 ASCII 码，并将 ASCII 码保存到系统的键盘缓冲区中。

当应用程序中调用 `getchar` 函数时，实际上会调用系统函数 `read` 来读取键盘缓冲区中的 ASCII 码，直到读取到回车符为止，然后将读取到的 ASCII 码组成的字符串作为 `getchar` 函数的返回值返回给应用程序。这个过程可以实现从键盘输入字符并将其返回给应用程序的功能。整个过程是异步的，即用户按下键盘引发的中断请求会被及时处理并将按键信息传递给应用程序。

8.5 本章小结

本章阐述了系统级 IO，介绍了 IO 设备的管理方法，UnixIO 接口及其函数，最后简要分析了 `printf` 函数和 `getchar` 函数的实现过程。

结论

Hello 的一生：

1. 源程序编写：Hello 诞在文本编辑器或 IDE 中编写 C 语言代码，生成了最初的 `hello.c` 源程序。Hello 在这里被赋予了它最初的存在
2. 预处理：Hello 进入预处理阶段，预处理器解析宏定义、文件包含和条件编译等指令，生成一个 ASCII 码的中间文件 `hello.i`。
3. 编译：Hello 通过编译器进行编译，将 C 语言代码转换为汇编指令，生成一个 ASCII 汇编语言文件 `hello.s`。
4. 汇编：汇编器将汇编指令翻译成机器语言，并生成重定位信息，生成可重定位目标文件 `hello.o`。
5. 链接：链接器进行符号解析、重定位和动态链接等操作，将可重定位目标文件 `hello.o` 与其他目标文件链接在一起，生成一个可执行目标文件 `hello`。现在，Hello 终于可以真正地被执行了。
6. 在 shell 输入 `./hello 2022112042 zty 12345678910 1` 调用命令行解释功能
7. 创建进程：用 `fork` 函数为 `hello` 创建子进程。
8. 运行阶段：通过 shell 命令运行 `hello` 程序时，shell 调用 `fork` 函数创建子进程，为 `hello` 程序提供执行环境。子进程中通过 `execve` 函数加载 `hello` 程序，

进入 `hello` 的程序入口点，`hello` 开始运行。

9. 执行指令：CPU 为 `hello` 分配时间片，执行控制逻辑流。
10. 内存管理和协作：在运行阶段，操作系统的内核负责调度进程，并对可能产生的异常和信号进行处理。硬件组件如 MMU、TLB、多级页表、`cache` 和 DRAM 内存等与操作系统协作，共同完成内存的管理。
11. 动态申请内存：`hello` 运行过程中 `printf` 中调用 `malloc` 动态的申请堆中的内存空间
12. I/O 交互：`Hello` 程序可以利用操作系统提供的 Unix I/O 功能与文件进行交互。
13. 信号与异常：`hello` 运行过程中可能会产生各种异常信号，系统会做出相应处理。
14. 终止：`Hello` 进程运行结束后，由 `shell` 负责回收终止的 `hello` 进程，操作系统内核删除为 `hello` 进程创建的所有数据结构。`Hello` 短暂又灿烂一生在这里结束。

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



参考文献

[1] Pianistx.printf 函数实现的深入剖析[EB/OL].2013[2021-6-9].

<https://www.cnblogs.com/pianist/p/3315801.html>.

[2] Randal E.Bryant, David O'Hallaron. 深入理解计算机系统[M]. 机械工业出版社.2018.4