

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 計算機系

學 號 1170300703

班 級 1703007

學 生 蘭鴻兵

指 導 教 師 鄭貴濱

計算機科學與技術學院

2018 年 12 月

## 摘 要

本文运用计算机系统知识，通过各类工具和辅助输出信息，分析 hello 程序在 Linux 系统下的 P2P 和 O2O 过程，对 hello 程序的整个生命周期进行了解析，对计算机系统这门课程内容有了更深刻的认识。

**关键词：**P2P；O2O；操作系统；进程；内存分配；系统 IO；

（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

<b>第 1 章 概述.....</b>	<b>- 4 -</b>
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.....	- 4 -
<b>第 2 章 预处理.....</b>	<b>- 5 -</b>
2.1 预处理的概念与作用.....	- 5 -
2.2 在 UBUNTU 下预处理的命令.....	- 5 -
2.3 HELLO 的预处理结果解析.....	- 6 -
2.4 本章小结.....	- 6 -
<b>第 3 章 编译.....</b>	<b>- 7 -</b>
3.1 编译的概念与作用.....	- 7 -
3.2 在 UBUNTU 下编译的命令.....	- 7 -
3.3 HELLO 的编译结果解析.....	- 8 -
3.4 本章小结.....	- 13 -
<b>第 4 章 汇编.....</b>	<b>- 14 -</b>
4.1 汇编的概念与作用.....	- 14 -
4.2 在 UBUNTU 下汇编的命令.....	- 14 -
4.3 可重定位目标 ELF 格式.....	- 14 -
4.4 HELLO.O 的结果解析.....	- 16 -
4.5 本章小结.....	- 17 -
<b>第 5 章 链接.....</b>	<b>- 18 -</b>
5.1 链接的概念与作用.....	- 18 -
5.2 在 UBUNTU 下链接的命令.....	- 18 -
5.3 可执行目标文件 HELLO 的格式.....	- 19 -
5.4 HELLO 的虚拟地址空间.....	- 20 -
5.5 链接的重定位过程分析.....	- 21 -
5.6 HELLO 的执行流程.....	- 23 -
5.7 HELLO 的动态链接分析.....	- 23 -
5.8 本章小结.....	- 24 -
<b>第 6 章 HELLO 进程管理.....</b>	<b>- 25 -</b>
6.1 进程的概念与作用.....	- 25 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 25 -
6.3 HELLO 的 FORK 进程创建过程.....	- 25 -
6.4 HELLO 的 EXECVE 过程.....	- 25 -
6.5 HELLO 的进程执行.....	- 26 -
6.6 HELLO 的异常与信号处理.....	- 27 -
6.7 本章小结.....	- 30 -
<b>第 7 章 HELLO 的存储管理.....</b>	<b>- 31 -</b>
7.1 HELLO 的存储器地址空间.....	- 31 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 31 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 32 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 32 -
7.5 三级 CACHE 支持下的物理内存访问.....	- 33 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 33 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 34 -
7.8 缺页故障与缺页中断处理.....	- 35 -
7.9 动态存储分配管理.....	- 35 -
7.10 本章小结.....	- 36 -
<b>第 8 章 HELLO 的 IO 管理.....</b>	<b>- 37 -</b>
8.1 LINUX 的 IO 设备管理方法.....	- 37 -
8.2 简述 UNIX IO 接口及其函数.....	- 37 -
8.3 PRINTF 的实现分析.....	- 38 -
8.4 GETCHAR 的实现分析.....	- 39 -
8.5 本章小结.....	- 39 -
<b>结论.....</b>	<b>- 40 -</b>
<b>附件.....</b>	<b>- 41 -</b>
<b>参考文献.....</b>	<b>- 42 -</b>

## 第 1 章 概述

### 1.1 Hello 简介

P2P 过程: From Program to Process, 是将 `hello.c` 的文本代码, 经过预处理->编译->汇编->链接四个步骤, 生成一个二进制可执行文件 `hello`, 通过 `Bash` 为其 `fork` 一个子进程, 并最终使用 `execve` 载入进程执行。

020 过程: From Zero-0 to Zero-0, `shell` 执行, 进程管理 OS 为 `hello` 分配虚拟内存并映射, 在开始执行进程时分配载入物理内存, 执行 `hello` 程序。再进行取值, 译码, 访存, 计算, 写回, 更新 `pc` 等操作, 流水线执行 `hello` 的每条指令。将其输出内容显示到屏幕, 然后 `hello` 进程结束, `shell` 回收其内存。

### 1.2 环境与工具

硬件环境: X64CPU; 4.5GHz; 8GRAM; 1TBHD Disk

软件环境: Windows10 64 位; VMware14.12; Ubuntu 18.04 LTS 64 位; WPS

使用工具: codeblocks, Visual Studio 2017, objdump, gdb, edb, hexedit

### 1.3 中间结果

<code>hello.i</code>	<code>hello.c</code> 预处理后的程序文本
<code>hello.s</code>	<code>hello.i</code> 编译成汇编语言后的程序文本
<code>hello.o</code>	<code>hello.s</code> 生成的二进制文件
<code>hello.objdump</code>	<code>hello.o</code> 反汇编后的文件
<code>hello</code>	<code>hello</code> 通过链接操作后生成的二进制可执行文件
<code>hello_back.txt</code>	<code>hello</code> 反汇编后重定向生成的程序文本
<code>helloelf.txt</code>	<code>hello.o</code> 的 <code>readelf</code> 输出的重定向文本
<code>helloelf_second.txt</code>	<code>hello</code> 的 <code>readelf</code> 输出的重定向文本

### 1.4 本章小结

本章简单介绍了 P2P 与 020 的过程, 并列出实验中所使用的环境与工具, 以及整个过程产生的中间结果。

(第 1 章 0.5 分)

## 第 2 章 预处理

### 2.1 预处理的概念与作用

概念：预处理是在编译之前进行的处理。C 语言的预处理主要有三个方面的内容： 1.宏定义； 2.文件包含； 3.条件编译。 预处理命令以符号“#”开头。具体操作是在程序编译之前，根据这些以符号“#”开头的命令，修改原始的 c 程序。

例如大作业的 hello.c 文件，就有以下命令：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

预处理要做的是从系统头文件包中找到这三个头文件，把需要的内容插入 hello.c 的文本中，生成 hello.i 文件。

作用：预处理将一些宏进行文本替换，将需要的代码写入文件，成为一份完整能够进行编译的代码，方便编译器对程序进行编译。

### 2.2 在 Ubuntu 下预处理的命令

命令：gcc -E -o name.i name.c

截图：

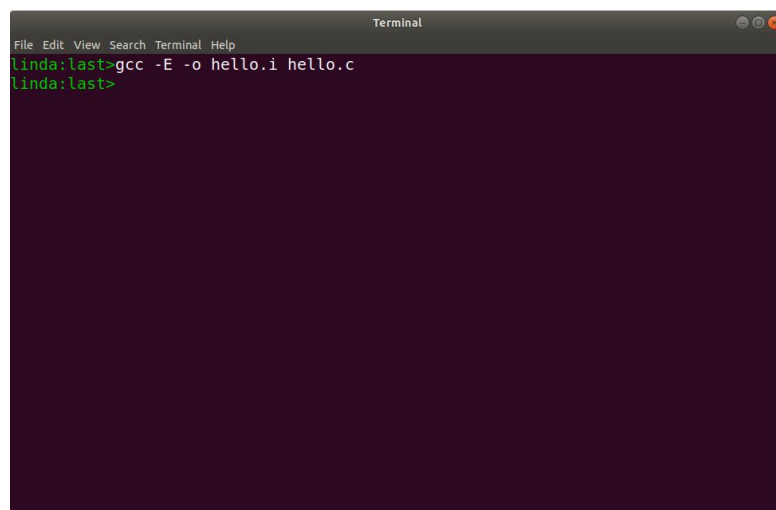


图 2.2.1

## 2.3 Hello 的预处理结果解析



```
extern int getsuopt (char *__restrict __optionp,
char *const *__restrict __tokens,
char *__restrict __valuep)
__attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1, 2, 3))) ;
# 1006 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
__attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));
# 1016 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1017 "/usr/include/stdlib.h" 2 3 4
# 1026 "/usr/include/stdlib.h" 3 4

# 9 "hello.c" 2

# 10 "hello.c"
int sleepsecs=2.5;

int main(int argc, char *argv[])
{
    int i;

    if(argc!=3)
    {
        printf("Usage: Hello 学号 姓名!\n");
        exit(1);
    }
    for(i=0; i<10; i++)
    {
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
```

图 2.3.1

预处理后得到的 `hello.i` 如上。可以看到，代码依然是 `c` 语言的代码，`main` 函数的内容没有发生变化，但行数却增加到了 3118 行，这是因为预编译将 `#include` 的头文件中的代码展开插入到了 `hello.c` 中。并添加了许多注释，用来描述使用的这些代码所在计算机中的位置。

预编译所得到的代码就是完整的可以进行编译的代码，能够进行下一步编译。

## 2.4 本章小结

本章介绍了预编译的概念与作用，简单介绍了其过程，对预编译过程进行演示，并根据演示结果进行了结果分析，验证其过程与作用。

(第 2 章 0.5 分)

## 第 3 章 编译

### 3.1 编译的概念与作用

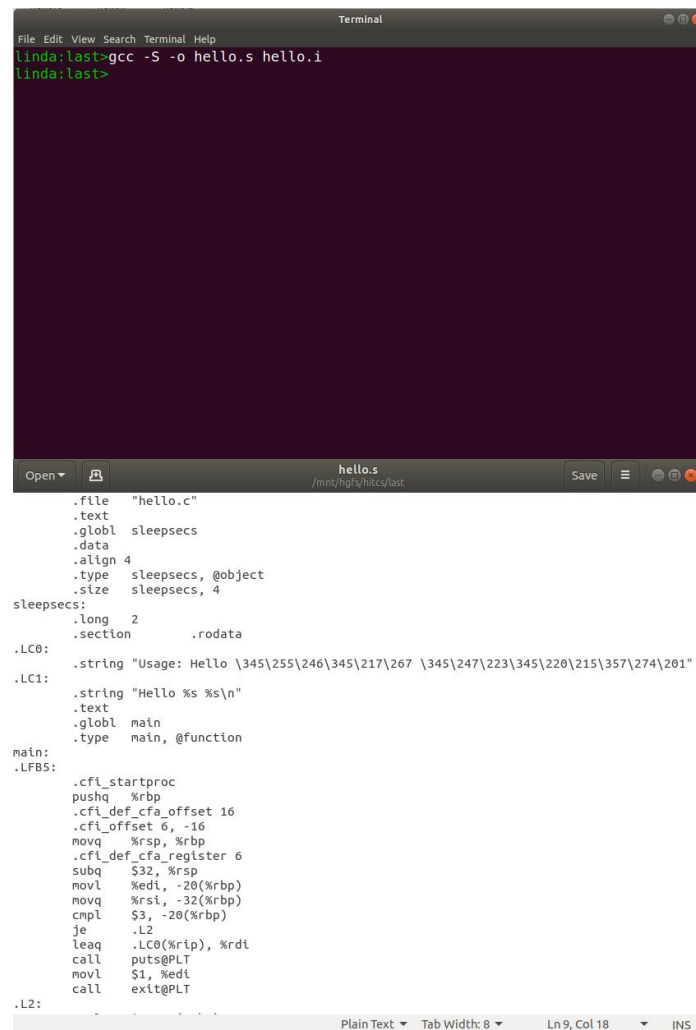
概念：编译，是将高级语言文本程序翻译成相应汇编语言文本程序的过程。

作用：编译将机器所不能读懂的高级语言文本翻译成更加接近机器语言的汇编文本程序，方便将程序最终翻译成二进制可执行文件。

### 3.2 在 Ubuntu 下编译的命令

命令：gcc -S -o name.s name.i

截图：



```
File Edit View Search Terminal Help
linda:last>gcc -S -o hello.s hello.i
linda:last>

hello.s
/mnt/hgfs/hiccs/last

.file "hello.c"
.text
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
.long 2
.section .rodata
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
.string "Hello %s %s\n"
.text
.globl main
.type main, @function
main:
.LFB5:
.cfi_startproc
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 $3, -20(%rbp)
je .L2
leaq .LC0(%rip), %rdi
call puts@PLT
movl $1, %edi
call exit@PLT
.L2:
```

图 3.3.1



### 3.3 Hello 的编译结果解析

编译所得到的.s 文件其长度只有 65 行，所进行的工作是将 main()函数进行了翻译，变成汇编语言。

#### 3.3.1 变量处理

全局变量：

```
int sleepsecs=2.5;

int main(int argc, char *argv[])
```

图 3.3.2

在我们的 C 语言源程序中，有以上全局变量 sleepsecs。

```
.file    "hello.c"
.text
.globl   sleepsecs
.data
.align   4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2
.section          .rodata
```

图 3.3.3

在编译后的.s 文件中，我们找到其相应的代码段。其被称为 sleepsecs，存在.data 段。并记录其相关信息，类型为 object，长度为 4 个字节。并且因为源程序中是 int 类型，赋值为 2.5 发生了隐式转换，最终其值为 2。

局部变量：

```
int main(int argc, char *argv[])
{
    int i;
```

图 3.3.4

再来看局部变量，下图中可以看到，在 c 语言源程序的 main 函数中有一个局部 int 类型变量 i，用来对循环进行计次。

我们来寻找其在汇编代码中的具体位置：

```
.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
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
    → addl    $1, -4(%rbp)
.L3:
    → cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

图 3.3.4

图中箭头所指的操作即是对局部变量 `i` 的操作。可以看到，在汇编代码中，局部变量 `i` 只是被存储在了内存空间 `-4(%rbp)` 中，在使用到它时访问这个内存位置即可。

### 常量：

接着来说常量，在图 3.3.4 中，几个对内存 `-4(%rbp)` 的操作都使用到了常量。在汇编代码中，常量即立即数，其以 `$` 开头，后跟上具体的数值，可以在指令中直接使用。

**数组：**

在图 3.3.4 中可以看到，main 函数在传参时传入了一个 argv[] 数组。

对于数组的访问，是通过首地址加偏移量的方式进行的。

```
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
```

图 3.3.5

如图是汇编代码中对 argv 的访问，首先将首地址传给 %rax，之后偏移 16 个字节，就能够访问到我们所需要的位置。

**字符串：**

```
if(argc!=3)
{
    printf("Usage: Hello 学号 姓名! \n");
    exit(1);
}
for(i=0;i<10;i++)
{
    printf("Hello %s %s\n", argv[1], argv[2]);
    sleep(sleepsecs);
}
```

图 3.3.6

在 c 语言源程序中，有以上两段字符串。同样的我们找到它们在汇编代码中的位置：

```
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
.string "Hello %s %s\n"
.text
.globl main
.type main, @function
```

图 3.3.7

可以看到，字符串被存储在了 .rodata 的只读存储段中，在使用的时候，访问其存储位置首地址即可。

### 3.3.2 操作处理

#### 赋值与四则运算操作：

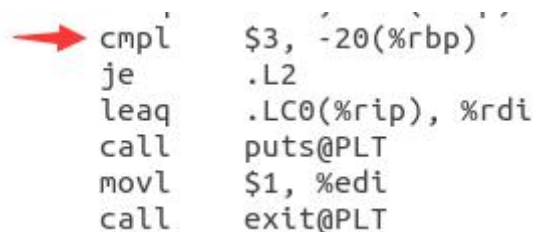
在图 3.3.6 中，我们可以看到，局部变量 `i` 在循环过程中进行了赋值和运算操作。循环开始时赋值为 0，之后逐次加 1。

再看回图 3.3.4，我们找到这些操作在汇编代码中的位置。从图 3.3.4 中可以看到，代码 `movl $0, -4(%rbp)` 是对 `i` 的赋值操作，将立即数 0 赋值给 `-4(%rbp)` 内存。代码 `addl $1, -4(%rbp)` 是对 `i` 的加法操作，将 `-4(%rbp)` 内存加 1。

#### 对于关系操作符与控制语句的处理：

图 3.3.6 中，有 `if(argc!=3)` 这一行代码，其中 `!=` 就是关系操作符。

在汇编代码中没有这样的关系操作符，取而代之的是如下操作：



```

→ cmpl    $3, -20(%rbp)
   je      .L2
   leaq    .LC0(%rip), %rdi
   call    puts@PLT
   movl    $1, %edi
   call    exit@PLT

```

图 3.3.8

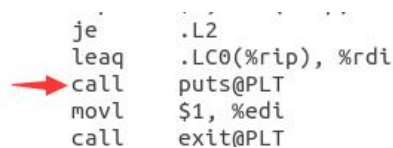
从 `cmpl` 操作开始分析，`cmpl` 是比较的操作，判断 3 与 `-20(%rbp)` 是否相等，不难看出 `-20(%rbp)` 就是 `argc`。`cmpl` 判断是否相等，设置条件码后执行下一句代码，`je` 是跳转操作，根据条件码来决定是否跳转到后面的地址。此处是若 `argc` 等于 3，就跳过 `if` 语句中的操作，直接执行之后的代码。

用 `cmpl` 和 `je` 的结合表示关系操作，这样就形成了控制语句。

### 3.3.3 函数操作

#### 函数调用：

在汇编代码中，函数调用使用 `call` 指令，通过 `call` 指令来访问所跟的相应函数代码。如图，是调用输出函数。



```

   je      .L2
   leaq    .LC0(%rip), %rdi
→  call    puts@PLT
   movl    $1, %edi
   call    exit@PLT

```

图 3.3.9

### 函数传参：

汇编代码中函数传参，需要先找几个寄存器，将参数传给这些寄存器。

x86-64 Linux 平台下，约定前 6 个参数存到寄存器 `rdi, rsi, rdx, rcx, r8, r9` 中，其余参数按照传参顺序依次压入栈中。

如图是 `printf` 函数的传参过程：

```

.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

```

图 3.3.10



在调用 `printf` 函数之前，先进行了传参，将参数传给了 `rdi, rsi, rdx`，以供 `printf` 函数使用。

### 函数返回值：

函数返回值一般存在寄存器 `eax` 中，如果要设定返回值的话，那就先将返回值传入 `eax`，再用 `ret` 语句返回。

如图是 `main` 函数的返回：

```

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

```

图 3.3.11

图中，先对 `eax` 赋值，最后使用 `ret` 返回。

### 3.4 本章小结

本章主要介绍了编译这一过程。对编译的概念与作用进行了简述，并根据 `hello.c` 从 C 语言源码编译到机器语言的案例，进行了各部分的过程解析。具体讲述了从高级语言逐渐接近机器语言的这一过程。

**(第 3 章 2 分)**

## 第 4 章 汇编

### 4.1 汇编的概念与作用

概念：汇编，是指将汇编语言翻译成机器指令，并将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 .o 类型文件中。

作用：通过汇编过程，汇编语言被翻译成一条条机器可以直接读取分析的机器指令。

### 4.2 在 Ubuntu 下汇编的命令

命令：as name.s -o name.o

截图：

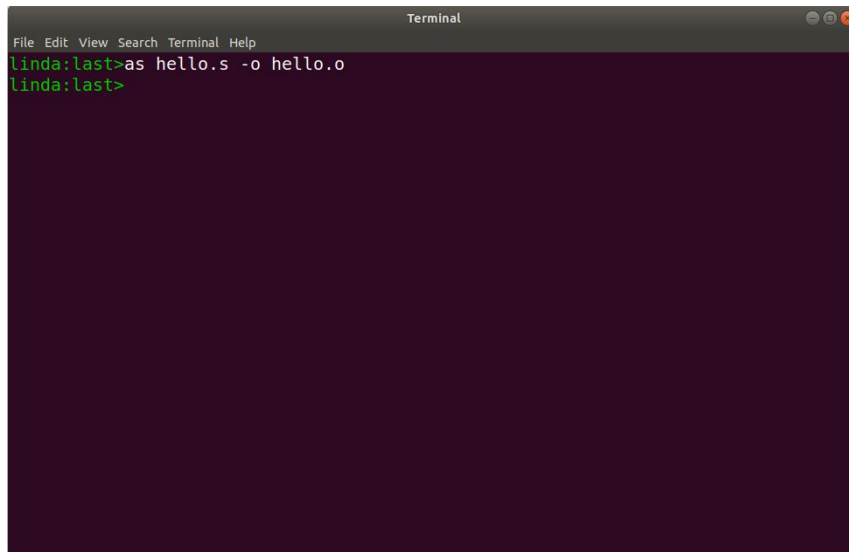


图 4.2.1

### 4.3 可重定位目标 elf 格式

先使用指令 `readelf -a hello.o > helloelf.txt` 读取 main.o 的 elf 文件并重定向输出到 helloelf.txt 中，方便阅读与分析。

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                           0
  Type:                                REL (Relocatable file)
  Machine:                             Advanced Micro Devices X86-64
  Version:                             0x1
  Entry point address:                  0x0
  Start of program headers:             0 (bytes into file)
  Start of section headers:            1152 (bytes into file)
  Flags:                                0x0
  Size of this header:                  64 (bytes)
  Size of program headers:              0 (bytes)
  Number of program headers:            0
  Size of section headers:              64 (bytes)
  Number of section headers:            13
  Section header string table index:    12

```

图 4.3.1

ELF Header：用于总的描述 ELF 文件各个信息的段，包含生成该文件的系统字的大小和字节顺序。

```

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
       00000000000000081  0000000000000000    AX   0   0   1
  [ 2] .rela.text         RELA              0000000000000000    00000340
       000000000000000c0  00000000000000018    I   10   1   8
  [ 3] .data              PROGBITS          0000000000000000    000000c4
       00000000000000004  0000000000000000    WA   0   0   4
  [ 4] .bss               NOBITS            0000000000000000    000000c8
       00000000000000000  0000000000000000    WA   0   0   1
  [ 5] .rodata             PROGBITS          0000000000000000    000000c8
       0000000000000002b  0000000000000000    A   0   0   1
  [ 6] .comment            PROGBITS          0000000000000000    000000f3
       0000000000000002b  00000000000000001    MS   0   0   1
  [ 7] .note.GNU-stack     PROGBITS          0000000000000000    0000011e
       00000000000000000  0000000000000000          0   0   1
  [ 8] .eh_frame            PROGBITS          0000000000000000    00000120
       00000000000000038  0000000000000000    A   0   0   8
  [ 9] .rela.eh_frame       RELA              0000000000000000    00000400
       00000000000000018  00000000000000018    I   10   8   8
  [10] .symtab              SYMTAB            0000000000000000    00000158
       000000000000000198  00000000000000018          11   9   8
  [11] .strtab              STRTAB            0000000000000000    000002f0
       0000000000000004d  0000000000000000          0   0   1
  [12] .shstrtab           STRTAB            0000000000000000    00000418
       00000000000000061  0000000000000000          0   0   1

```

图 4.3.2

Section Header：描述.o 文件中出现的各个节的类型、位置、所占空间大小等信息。



```

Relocation section '.rela.text' at offset 0x340 contains 8 entries:
  Offset          Info          Type          Sym. Value      Sym. Name + Addend
0000000000018    000500000002 R_X86_64_PC32 0000000000000000 .rodata - 4
000000000001d    000c00000004 R_X86_64_PLT32 0000000000000000 puts - 4
0000000000027    000d00000004 R_X86_64_PLT32 0000000000000000 exit - 4
0000000000050    000500000002 R_X86_64_PC32 0000000000000000 .rodata + 1a
000000000005a    000e00000004 R_X86_64_PLT32 0000000000000000 printf - 4
0000000000060    000900000002 R_X86_64_PC32 0000000000000000 sleepsecs - 4
0000000000067    000f00000004 R_X86_64_PLT32 0000000000000000 sleep - 4
0000000000076    001000000004 R_X86_64_PLT32 0000000000000000 getchar - 4

Relocation section '.rela.eh_frame' at offset 0x400 contains 1 entry:
  Offset          Info          Type          Sym. Value      Sym. Name + Addend
0000000000020    000200000002 R_X86_64_PC32 0000000000000000 .text + 0

```

图 4.3.3

.rela.text：重定位节，这个节包含了.text 节中需要进行重定位的信息——偏移量（offset），信息（info），类型（type），符号值（sym.value）。这些信息描述的位置，在由.o 文件生成可执行文件的时候需要被重定位。在 hello.o 里需要被重定位的有 printf, puts, exit, sleepsecs, getchar, sleep 以及 rodata 里的两个字符串。

## 4.4 Hello.o 的结果解析

使用 `objdump -d -r hello.o > hello.objdump` 获得反汇编代码。将文件 hello.s 与文件 hello.objdump 进行比较。

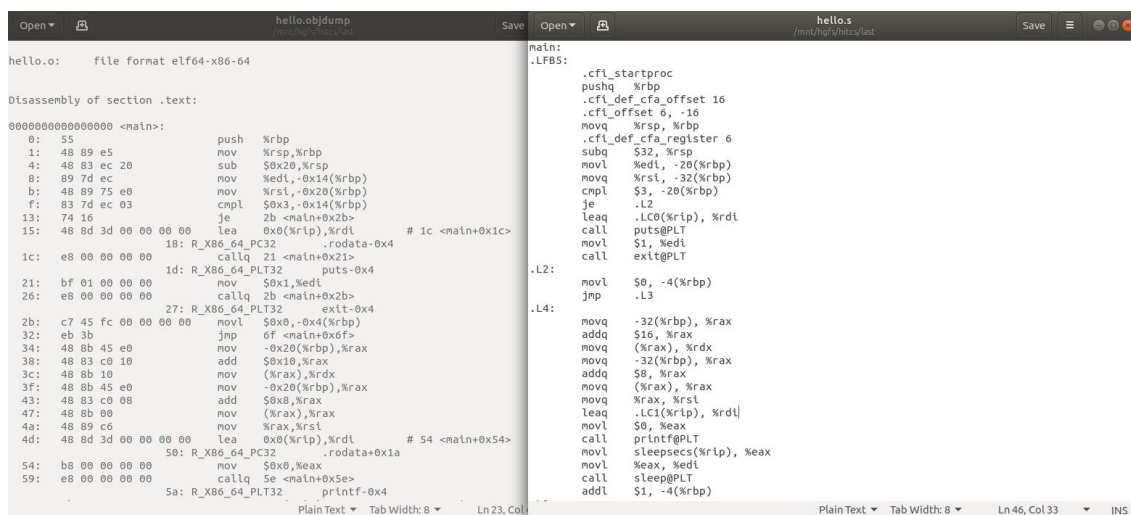


图 4.4.1

对比可以看出，hello.o 反汇编之后，与原来的汇编代码相比，左测多了机器指令，在冒号前面的是运行时机器指令的位置，冒号后的是每一行汇编语句所对应的机器指令。机器指令完全由 0/1 构成，此处显示为 16 进制。

分支跳转语句：我们可以发现，在 `hello.s` 中跳转是用 `.L3/.L4` 等助记符来表示的，而 `hello.o` 反汇编之后，这些助记符消失了，改用具体的地址位置代替。

函数调用：在 `hello.s` 中，调用一个函数使用的是 `call+函数名`，但在 `hello.o` 反汇编结果中，`call` 后所跟的是一个具体的地址位置。

## 4.5 本章小结

本章主要介绍了汇编这一过程，简介了汇编的概念和作用，并通过 `readelf` 查看 `hello.o` 的 ELF，反汇编查看 `hello.o` 反汇编内容并比较其与 `hello.s` 之间差别，这两个过程，分析了汇编指令映射到机器指令的这一过程。

**(第 4 章 1 分)**

## 第 5 章 链接

### 5.1 链接的概念与作用

概念：链接是将多个代码段和数据片段收集并拼接合并成一个可执行文件的过程。

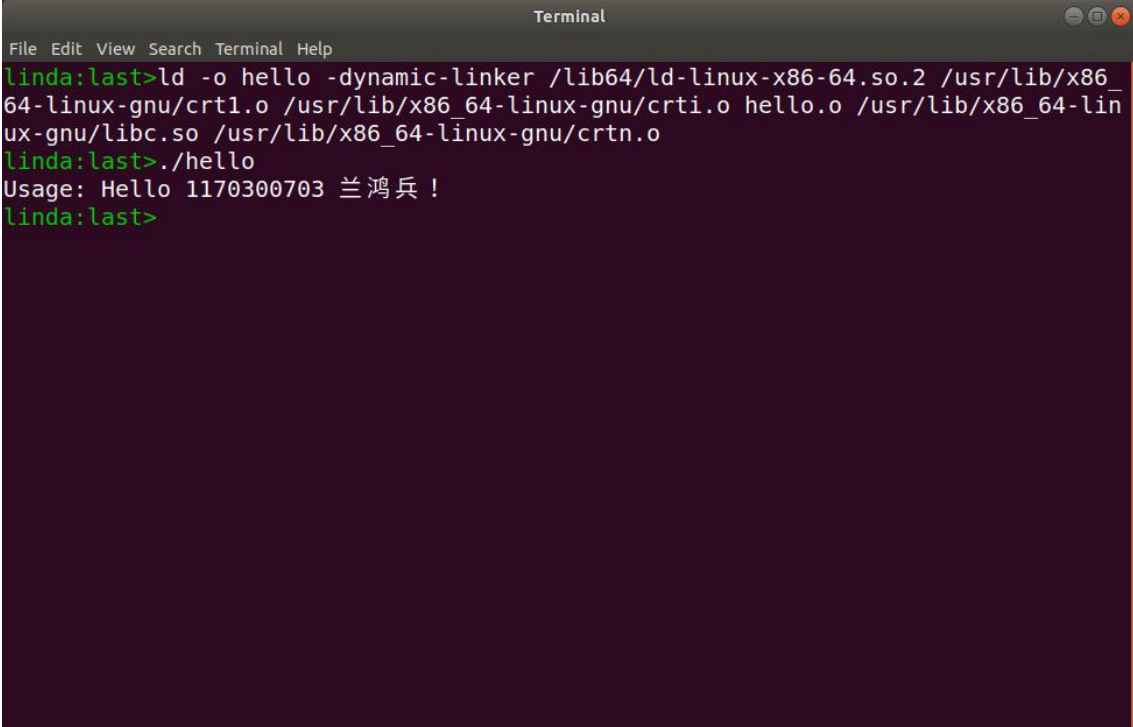
作用：由于存在链接这一过程，可以分段编写程序，最后再进行链接，成为一个完整的程序，降低了模块化编程的难度。

### 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
```

截图：

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
linda:last>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
linda:last>./hello
Usage: Hello 1170300703 兰鸿兵 !
linda:last>
```

图 5.2.1

## 5.3 可执行目标文件 hello 的格式

使用 `readelf -a hello > helloelf_second.txt` 命令输出 ELF 内容。

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                           0
  Type:                                 EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x400500
  Start of program headers:              64 (bytes into file)
  Start of section headers:              5928 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               56 (bytes)
  Number of program headers:              8
  Size of section headers:               64 (bytes)
  Number of section headers:              25
  Section header string table index:      24

```

图 5.3.1

以上是 ELF Header 的信息，可以看出节头表的数量增加了。

```

Section Headers:
[Nr] Name           Type           Address          Offset
     Size           EntSize          Flags Link Info  Align
[ 0]                 NULL           0000000000000000 00000000
     0000000000000000 0000000000000000 0 0 0
[ 1] .interp          PROGBITS       0000000000400200 00000200
     000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag    NOTE           000000000040021c 0000021c
     0000000000000020 0000000000000000 A 0 0 4
[ 3] .hash            HASH           0000000000400240 00000240
     0000000000000034 0000000000000004 A 5 0 8
[ 4] .gnu.hash         GNU_HASH       0000000000400278 00000278
     000000000000001c 0000000000000000 A 5 0 8
[ 5] .dynsym           DYNSYM         0000000000400298 00000298
     00000000000000c0 0000000000000018 A 6 1 8
[ 6] .dynstr           STRTAB         0000000000400358 00000358
     0000000000000057 0000000000000000 A 0 0 1
[ 7] .gnu.version      VERSYM         00000000004003b0 000003b0
     0000000000000010 0000000000000002 A 5 0 2
[ 8] .gnu.version_r    VERNEED       00000000004003c0 000003c0
     0000000000000020 0000000000000000 A 6 1 8
[ 9] .rela.dyn          RELA           00000000004003e0 000003e0
     0000000000000030 0000000000000018 A 5 0 8
[10] .rela.plt          RELA           0000000000400410 00000410
     0000000000000078 0000000000000018 AI 5 19 8
[11] .init             PROGBITS       0000000000400488 00000488
     0000000000000017 0000000000000000 AX 0 0 4
[12] .plt             PROGBITS       00000000004004a0 000004a0
     0000000000000060 0000000000000010 AX 0 0 16
[13] .text            PROGBITS       0000000000400500 00000500
     0000000000000132 0000000000000000 AX 0 0 16
[14] .fini            PROGBITS       0000000000400634 00000634
     0000000000000009 0000000000000000 AX 0 0 4
[15] .rodata           PROGBITS       0000000000400640 00000640
     000000000000003a 0000000000000000 A 0 0 8
[16] .eh_frame         PROGBITS       0000000000400680 00000680
     00000000000000fc 0000000000000000 A 0 0 8

```

图 5.3.2

[17]	.dynamic	DYNAMIC	00000000000600e50	00000e50
	000000000000001a0	0000000000000010	WA 6 0 8	
[18]	.got	PROGBITS	00000000000600ff0	00000ff0
	00000000000000010	0000000000000008	WA 0 0 8	
[19]	.got.plt	PROGBITS	00000000000601000	00001000
	00000000000000040	0000000000000008	WA 0 0 8	
[20]	.data	PROGBITS	00000000000601040	00001040
	00000000000000008	0000000000000000	WA 0 0 4	
[21]	.comment	PROGBITS	00000000000000000	00001048
	0000000000000002a	00000000000000001	MS 0 0 1	
[22]	.symtab	SYMTAB	00000000000000000	00001078
	00000000000000498	00000000000000018	23 28 8	
[23]	.strtab	STRTAB	00000000000000000	00001510
	00000000000000150	00000000000000000	0 0 1	
[24]	.shstrtab	STRTAB	00000000000000000	00001660
	000000000000000c5	00000000000000000	0 0 1	

图 5.3.3

以上是 ELF 节头表的内容，包含各段类型、位置、所占空间大小等信息

## 5.4 hello 的虚拟地址空间

使用 edb 加载 hello，在 Data Dump 中可以看到如下虚拟地址信息。

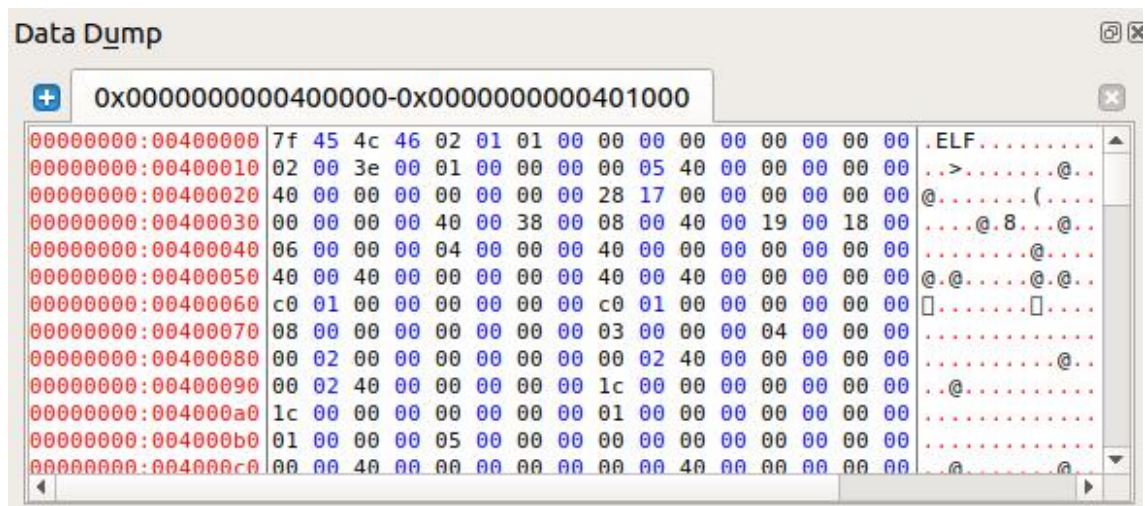


图 5.4.1

看出程序在 0x00400000 地址开始加载，在 0x00400fff 结束。



再看 ELF 中的程序头表：

Program Headers:					
Type	Offset	VirtAddr	PhysAddr	Flags	Align
	FileSiz	MemSiz			
PHDR	0x0000000000000040	0x000000000000400040	0x000000000000400040		
	0x00000000000001c0	0x00000000000001c0	R	0x8	
INTERP	0x0000000000000200	0x000000000000400200	0x000000000000400200		
	0x000000000000001c	0x000000000000001c	R	0x1	
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x000000000000400000	0x000000000000400000		
	0x0000000000000077c	0x0000000000000077c	R E	0x200000	
LOAD	0x00000000000000e50	0x000000000000600e50	0x000000000000600e50		
	0x000000000000001f8	0x000000000000001f8	RW	0x200000	
DYNAMIC	0x00000000000000e50	0x000000000000600e50	0x000000000000600e50		
	0x000000000000001a0	0x000000000000001a0	RW	0x8	
NOTE	0x0000000000000021c	0x00000000000040021c	0x00000000000040021c		
	0x00000000000000020	0x00000000000000020	R	0x4	
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x0000000000000000	0x0000000000000000	RW	0x10	
GNU_RELRO	0x00000000000000e50	0x000000000000600e50	0x000000000000600e50		
	0x000000000000001b0	0x000000000000001b0	R	0x1	

图 5.4.2

各信息如下：

PHDR：程序头表

INTERP：程序执行前需要调用的解释器

LOAD：程序目标代码和常量信息

DYNAMIC：动态链接器所使用的信息

NOTE:: 辅助信息

GNU\_EH\_FRAME：保存异常信息

GNU\_STACK：使用系统栈所需要的权限信息

GNU\_RELRO：保存在重定位之后只读信息的位置

## 5.5 链接的重定位过程分析

先使用命令 `objdump -d -r hello > hello_back.txt`，将 `hello` 的反汇编的代码输出到 `hello_back.txt` 中。

```

hello.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16            je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata+0x4
1c: e8 00 00 00 00    callq  21 <main+0x21>
1d: R_X86_64_PLT32    puts+0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq  2b <main+0x2b>
27: R_X86_64_PLT32    exit+0x4
2b: c7 45 fc 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     0f <main+0x0f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
43: 48 83 c0 08       add     $0x8,%rax
47: 48 8b 00          mov     (%rax),%rax
4a: 48 89 c6          mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 54 <main+0x54>
50: R_X86_64_PC32    .rodata+0x1a
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq  5e <main+0x5e>
5a: R_X86_64_PLT32    printf+0x4
5e: 8b 05 00 00 00    mov     0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs+0x4
64: 89 c7            mov     %eax,%edi
Plain Text  Tab Width: 8  Ln 23, Col 47

hello_back.txt: file format elf64-x86-64

Disassembly of section .init:

0000000000004088 <_init>:
400488: 48 83 ec 08       sub     $0x8,%rsp
40048c: 48 8b 05 65 0b 00 mov     0x200b65(%rip),%rax    # 600ff8
<_gmon_start__>
400493: 48 85 c0          test    %rax,%rax
400496: 74 02            je      40049a <_init+0x12>
400498: ff d0            callq   *%rax
40049a: 48 83 c4 08       add     $0x8,%rsp
40049e: c3              retq

Disassembly of section .plt:

00000000000040a0 <.plt>:
4004a0: ff 35 62 0b 20 00 pushq   0x200b62(%rip)    # 601008
<_GLOBAL_OFFSET_TABLE_+0x8>
4004a6: ff 25 64 0b 20 00 jmpq    *0x200b64(%rip)    # 601010
<_GLOBAL_OFFSET_TABLE_+0x10>
4004ac: 0f 1f 40 00       nopl    0x0(%rax)

00000000000040b0 <puts@plt>:
4004b0: ff 25 62 0b 20 00 jmpq    *0x200b62(%rip)    # 601018
<puts@GLIBC_2.2.5>
4004b6: 68 00 00 00 00    pushq   $0x0
4004bb: e9 e0 ff ff ff    jmpq    4004a0 <.plt>

00000000000040c0 <printf@plt>:
4004c0: ff 25 5a 0b 20 00 jmpq    *0x200b5a(%rip)    # 601020
<printf@GLIBC_2.2.5>
4004c6: 68 01 00 00 00    pushq   $0x1
4004cb: e9 d0 ff ff ff    jmpq    4004a0 <.plt>

00000000000040d0 <getchar@plt>:

```

图 5.5.1

与 hello.o 的反汇编代码进行比较可以发现, hello 中是从 hello.elf 开始, 而 hello.o 是从 .text 节开始; 另外 hello 的反汇编比 hello.o 的反汇编要多出许多函数, 在 hello.s 中导入了诸如 puts、printf、getchar、sleep 等在 hello 程序中使用的函数。

```

00000000000040532 <main>:
400532: 55                push    %rbp
400533: 48 89 e5          mov     %rsp,%rbp
400536: 48 83 ec 20       sub     $0x20,%rsp
40053a: 89 7d ec          mov     %edi,-0x14(%rbp)
40053d: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
400541: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
400545: 74 16            je      40055d <main+0x2b>
400547: 48 8d 3d fa 00 00 lea     0xfa(%rip),%rdi    # 400648
<_IO_stdin_used+0x8>
40054e: e8 5d ff ff ff    callq   4004b0 <puts@plt>
400553: bf 01 00 00 00    mov     $0x1,%edi
400558: e8 83 ff ff ff    callq   4004e0 <exit@plt>

```

图 5.5.2

再看 main 函数, 不难看出, 链接调用函数的方式的是 call+<main+偏移量>。

可以得出重定位过程如下: 链接器完成地址和空间的分配后, 可以根据表中的内容确定所有符号的虚拟地址。根据这些符号地址, 就能够对需要重定位的符号进行修正。而在 elf 文件中有一个重定位表, 其中存储了需要重定位的位置, 描述了如何修改相应的段中的内容。重定位表是一个结构体数组, 每个数组元素对应一个重定位入口。

## 5.6 hello 的执行流程

hello 的执行流程如下：

```

_dl_start 0x00007fff6f0674a0)
0x00007f0625d5e630 <ld-2.27.so!_dl_init+0>
hello!_start 0x400500
libc-2.27.so!__libc_start_main 0x7fce 8c867ab0
-libc-2.27.so!__cxa_atexit 0x7fce 8c889430
-libc-2.27.so!__libc_csu_init 0x4005c0
hello!_init 0x400488
libc-2.27.so!_setjmp 0x7fce 8c884c10
libc-2.27.so!_sigsetjmp 0x7fce 8c884b70
libc-2.27.so!__sigjmp_save 0x7fce 8c884bd0
hello!main 0x400532
hello!puts@plt 0x4004b0
hello!exit@plt 0x4004e0
*hello!printf@plt
*hello!sleep@plt
*hello!getchar@plt
ld-2.27.so!_dl_runtime_resolve_xsave 0x7fce 8cc4e680
ld-2.27.so!_dl_fixup 0x7fce 8cc46df0
ld-2.27.so!_dl_lookup_symbol_x 0x7fce 8cc420b0
libc-2.27.so!exit 0x7fce 8c889128

```

## 5.7 Hello 的动态链接分析

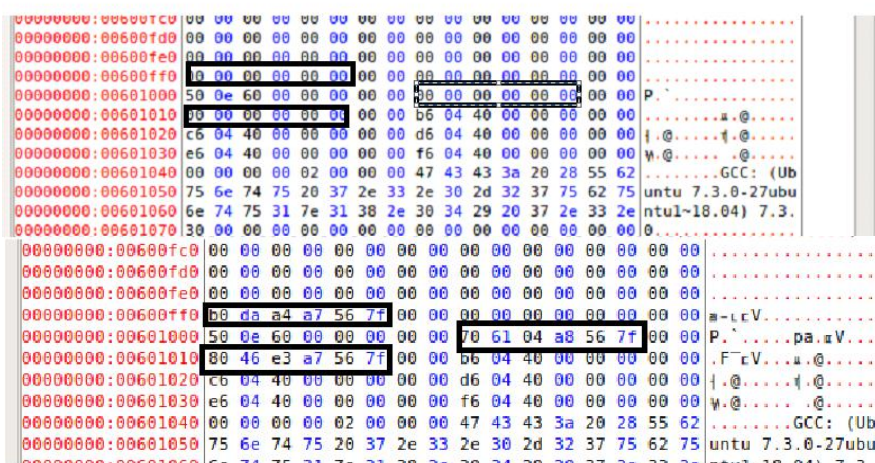


图 5.7.1



前后对比我们发现原先 `global_offset` 表是全 0 的，但在执行 `_dl_init` 后被赋上了相应的偏移量值。这表明 `_dl_init` 操作是给程序赋上当前执行内存地址的偏移量，以初始化 `hello` 程序。

## 5.8 本章小结

本章主要介绍了链接这一过程，简介了链接的概念和作用，分析了 `hello` 的 ELF 格式、虚拟地址空间分配、重定向过程、执行流程和动态链接过程。

**(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：进程是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

作用：程序是指令、数据及其组织形式的描述，进程是程序的实体。

### 6.2 简述壳 Shell-bash 的作用与处理流程

shell 是 linux 下的一个应用程序，提供用户直接和操作系统内核进行交互的界面，其处理流程如下：

- 1、读取用户的输入。
- 2、切分命令以获得用户输入的相关参数。
- 3、判断是否是内置命令。
- 4、如果是内置命令，则直接执行。
- 5、否则使用 fork 和 execve 创建一个进程并替换。
- 6、程序运行期间，shell 需要监视键盘的输入，并且做出相应反应。

### 6.3 Hello 的 fork 进程创建过程

- 1、用户输入命令 ./hello
- 2、判断非内置命令
- 3、使用 fork 函数创建一个子进程。子进程得到与父进程相同的虚拟地址空间，拥有与父进程相同参数等信息和与父进程相同的副本。区别在于子进程和父进程的 PID 不同。
- 4、系统并发运行，内核交替执行父子进程的逻辑控制流。

### 6.4 Hello 的 execve 过程

子进程调用 execve 函数，将子进程内容完全替换为 hello 可执行文件执行。

具体操作是将子进程的虚拟内存段全部替换为 hello 的虚拟内存段，不改变 pid 的前提下实现内容的替换，再令 PC 指向 hello 程序开始位置，在 hello 时间片内执行指令。execve 调用一次从不返回。

## 6.5 Hello 的进程执行

```
for(i=0;i<10;i++)
{
    printf("Hello %s %s\n",argv[1],argv[2]);
    sleep(sleepsecs);
}
getchar();
```

图 6.5.1

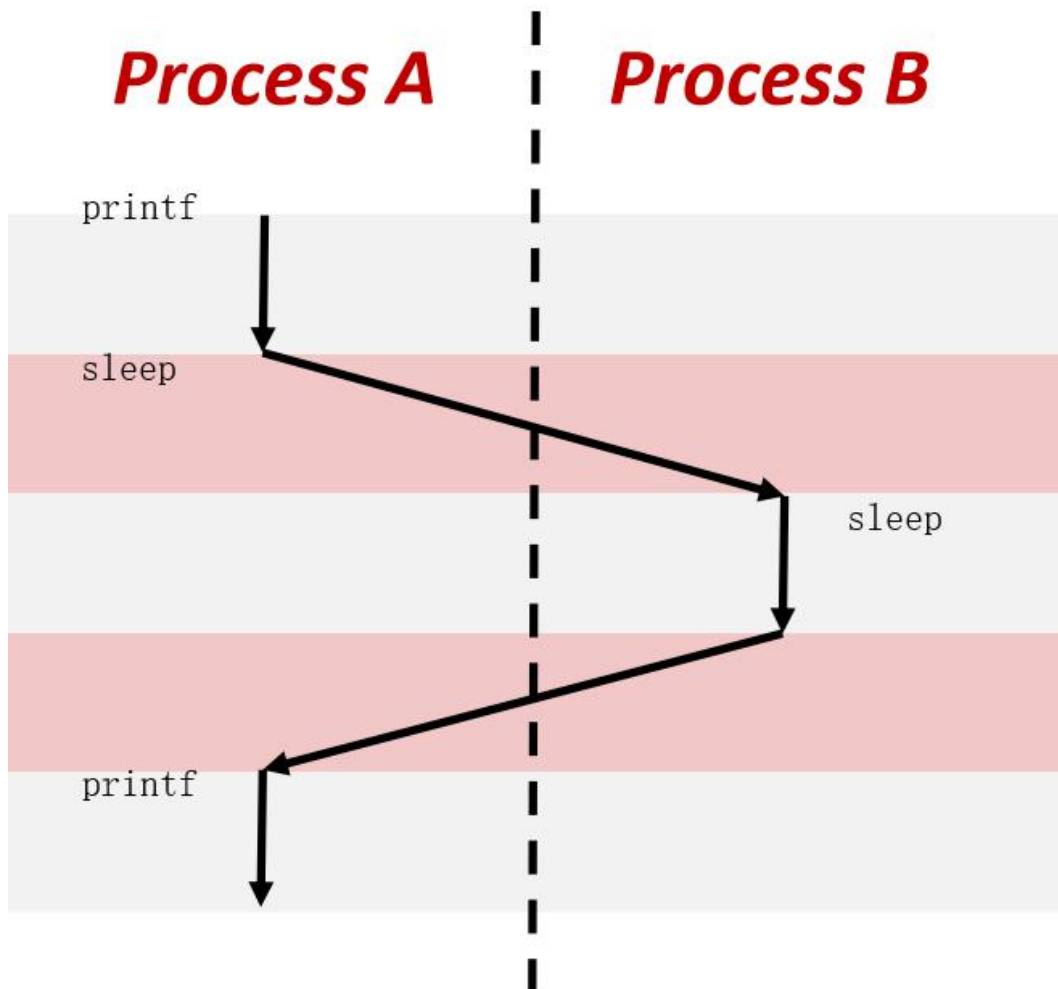


图 6.5.2

图 6.5.2 是 6.5.1 中 hello 的 C 语言代码片段的执行过程样例，操作系统在进程 A 到进程 B 之间切换，进程 A 是用户模式，进程 A 中断后，操作系统会保存进程 A 的信息，之后进入内核模式，由进程 A 转到进程 B，在进程 B 执行之后引发中断，再转回进程 A。

另外，调用 `getchar()` 时，会先在前端 hello 进程中运行，调用时再切换到内核进程标准读取程序中，从键盘输入中读取一个字符后再回到前端 hello 进程。

像这样在内核和前端之前切换的操作，称为上下文切换。

## 6.6 hello 的异常与信号处理

### 1、Ctrl-Z



A terminal window showing a user running a program named 'hello'. The user enters the command `./hello 1170300703 兰鸿兵`, and the program outputs three lines of 'Hello' messages. Then, the user presses Ctrl-Z, which sends a SIGTSTP signal to the process. The terminal shows `[1]+ Stopped` and the command `./hello 1170300703 兰鸿兵` in the background. The user then enters `ps`, and the terminal displays the following process list:

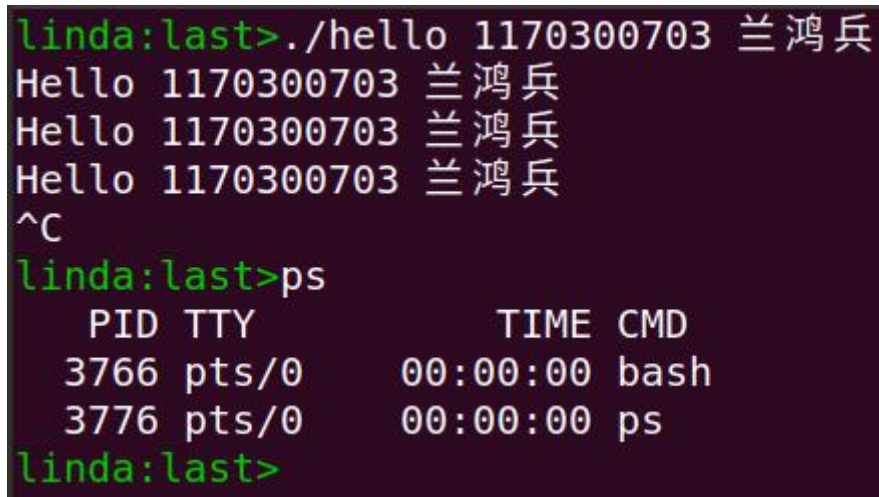
PID	TTY	TIME	CMD
3743	pts/0	00:00:00	bash
3752	pts/0	00:00:00	hello
3753	pts/0	00:00:00	ps

图 6.6.1

Ctrl-Z 是将程序暂时挂起。

由图可以看出，Ctrl-Z 操作向进程发送 `sigstsp` 信号，让进程暂时挂起，输入 `ps` 指令后可以看到 `hello` 进程并没有关闭。

### 2、Ctrl-C



A terminal window showing a user running the same 'hello' program. The user enters the command `./hello 1170300703 兰鸿兵`, and the program outputs three lines of 'Hello' messages. Then, the user presses Ctrl-C, which sends a SIGINT signal to the process. The terminal shows `^C`. The user then enters `ps`, and the terminal displays the following process list:

PID	TTY	TIME	CMD
3766	pts/0	00:00:00	bash
3776	pts/0	00:00:00	ps

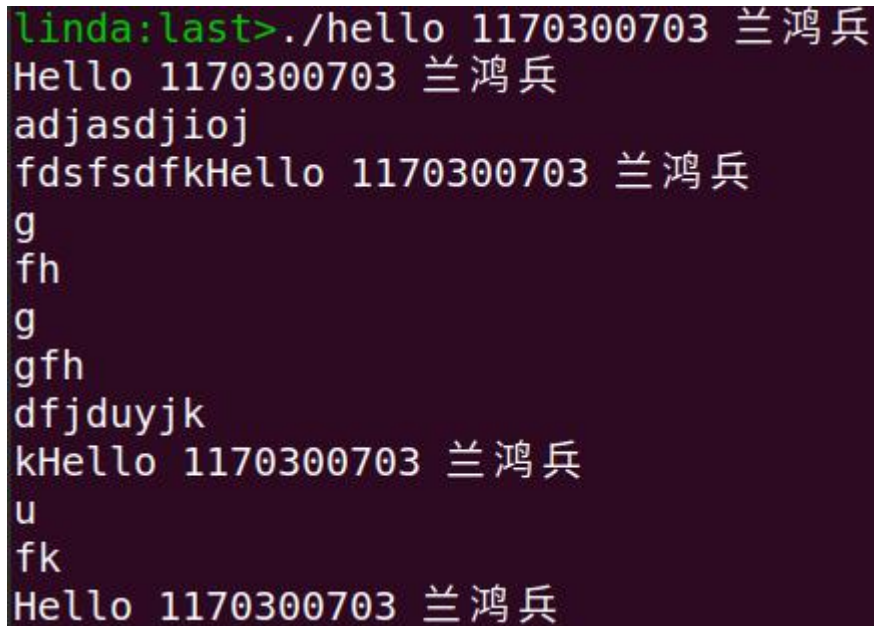
The `hello` process is no longer present in the list.

图 6.6.2

Ctrl-C 是将程序关闭。

由图可以看出，Ctrl-C 操作向进程发送 `sigint` 信号，让进程直接结束。输入 `ps` 指令后可以看到 `hello` 进程已经被关闭。

## 3、不停乱按，包括回车



```
linda:last>./hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
adjasdjioj
fdsfsdfkHello 1170300703 兰鸿兵
g
fh
g
gfh
dfjduyjk
kHello 1170300703 兰鸿兵
u
fk
Hello 1170300703 兰鸿兵
```

图 6.6.3

如图，不停乱按后键盘读入的字符都会到缓存区中，对当前执行的程序没有影响。当按下回车后，会对 `getchar()` 产生影响。

## 4、fg 命令



```
linda:last>./hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
^Z
[1]+  Stopped                  ./hello 1170300703 兰鸿兵
linda:last>fg
./hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
```

图 6.6.4

`fg` 命令可以让挂起的程序继续执行。

由图可以看出，在 `Ctrl-Z` 挂起程序后，输入 `fg` 可以让挂起的程序继续执行。





## 7、kill

```
linda:last>./hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
Hello 1170300703 兰鸿兵
^Z
[1]+  Stopped                  ./hello 1170300703 兰鸿兵
linda:last>ps
  PID TTY          TIME CMD
  3926 pts/0        00:00:00 bash
  3934 pts/0        00:00:00 hello
  3935 pts/0        00:00:00 ps
linda:last>kill 3934
linda:last>ps
  PID TTY          TIME CMD
  3926 pts/0        00:00:00 bash
  3934 pts/0        00:00:00 hello
  3936 pts/0        00:00:00 ps
linda:last>fg
./hello 1170300703 兰鸿兵
Terminated
```

图 6.6.7

如图，在挂起后输入 kill 指令——kill pid，能够杀死相应进程。

## 6.7 本章小结

本章主要介绍了进程管理相关的内容。介绍了进程的概念和作用，简述了 Shell-bash 的作用与处理流程，并分析了其 fork 和 execve 的具体过程，最后分析了几个异常与信号处理。

**(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：又称相对地址，是程序运行时与段相关的偏移地址，是描述一个程序运行段的地址。

物理地址：程序运行时加载到内存地址寄存器中的地址，是内存单元真正的地址，在前端总线上传输且是唯一的。hello 程序中，它代表程序运行时的指令在内存地址上的哪一个具体位置执行。

线性地址：即虚拟地址，是经过段机制转化之后，用于描述程序分页信息的地址，是对程序运行区块的一个抽象映射。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

首先给定一个完整的逻辑地址——段选择符：段内偏移地址

1、检查段选择符的  $T1=0$  还是 1，以确定要转换是 GDT 中的段，还是 LDT 中的段，再根据相应的寄存器，得到其地址和大小。这样就得到了一个数组。

2、取段选择符中前 13 位，在数组中可以查找到对应段描述符，这样就获取了 Base 基地址。

3、最后把  $Base + offset$ ，就是要转换的线性地址了。

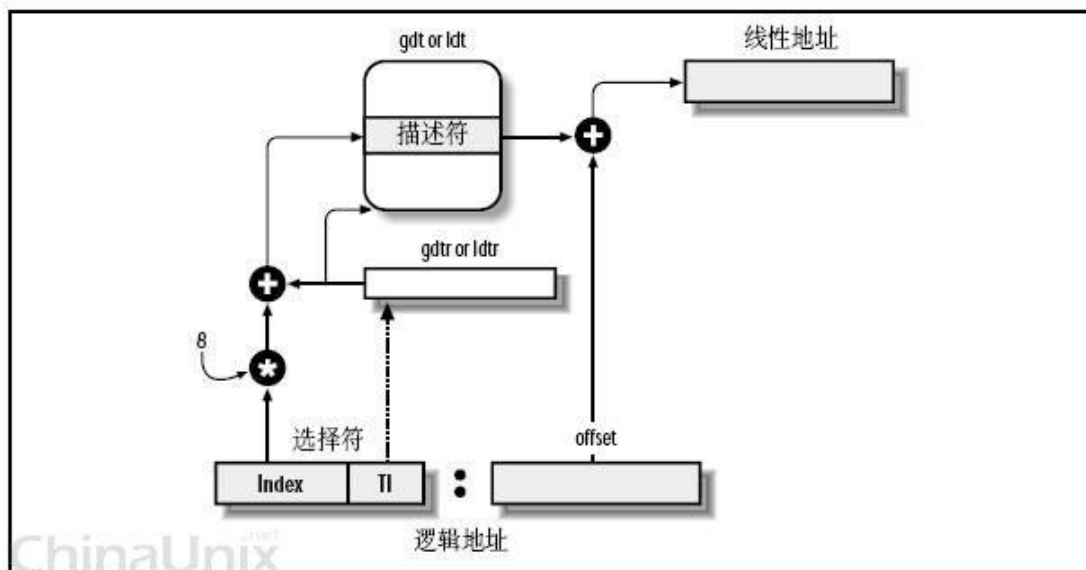


图 7.2.1



## 7.3 Hello 的线性地址到物理地址的变换-页式管理

转换中需要使用翻译后备缓冲器（TLB）。

- 1、首先将线性地址分成 VPN（虚拟页号）+VPO（虚拟页偏移）的形式
- 2、再将 VPN 分成 TLBT（TLB 标记）+TLBI（TLB 索引）的形式
- 3、在 TLB 缓存里寻找其所对应的 PPN（物理页号）
- 4、若发生缺页情况，则查找对应 PPN，将其与 VPO 合并为 PPN+VPO，得到的就是所要生成的物理地址。

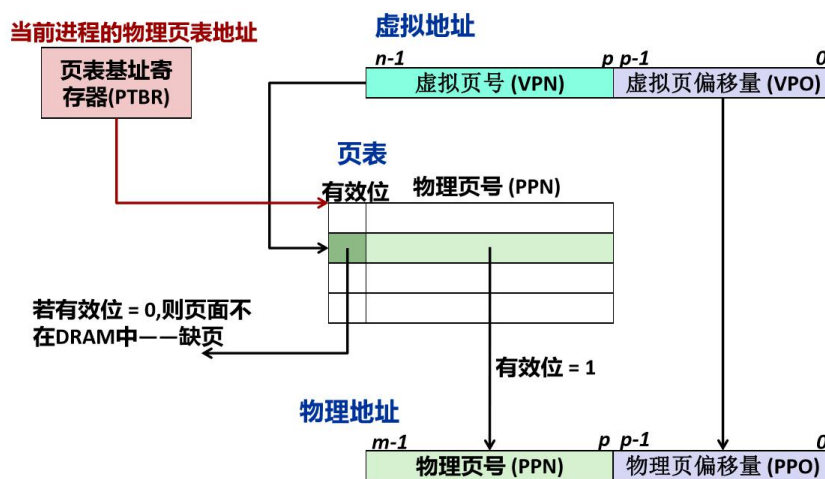


图 7.3.1

## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

从 VA 中划分出 VPN 和 VPO，在 TLB 寻找对应 PPN，若发生内存缺页，就到页表中寻找对应的物理地址，找到就一步步递进往下寻址，越往下一层每个条目所对应区域越小，寻址越细致，经过 4 层寻址后找到相应的 PPN 和 VPO，拼接起来就完成了变换。

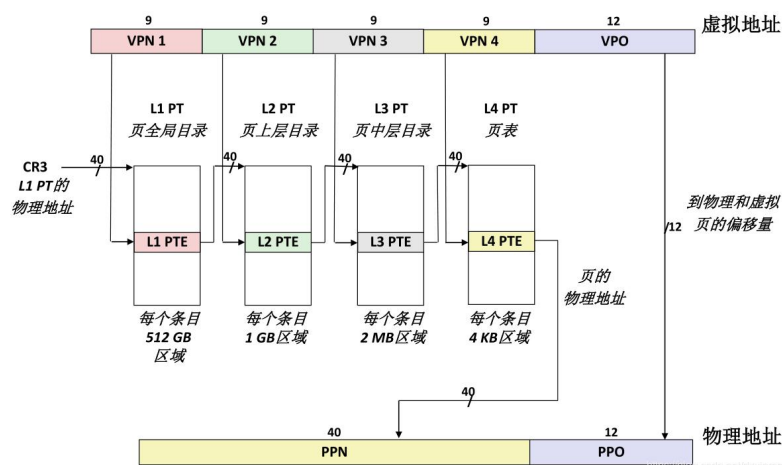


图 7.4.1

## 7.5 三级 Cache 支持下的物理内存访问

- 1、得到物理地址后，将物理地址分成 CT（标记）+CI（索引）+CO（偏移量）。
- 2、先在一级 cache 内部查找。根据给出的地址索引，先找到 Cache 中的组，在这些组中，根据地址中的标记位寻找组中的匹配行。若地址中的标记与组中某一行的标记相等，再根据偏移量得到数据块。
- 3、若未能找到标记位为有效的字节，就逐级到二级和三级 cache 中寻找对应的字节，找到之后返回结果。

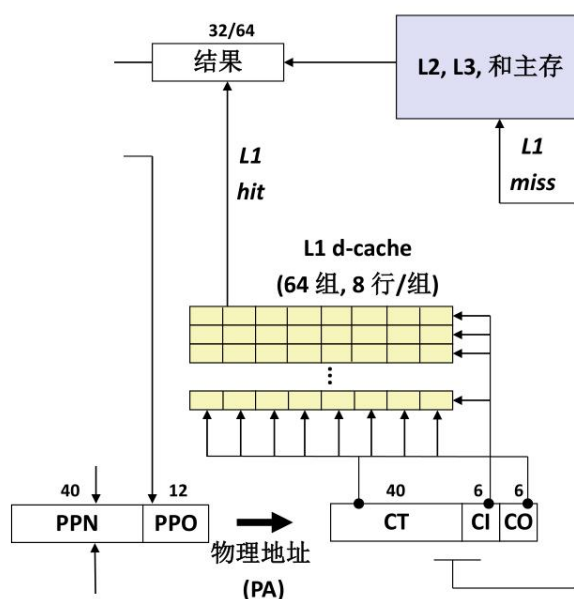


图 7.5.1

## 7.6 hello 进程 fork 时的内存映射

fork 时使用到以下两个信息：

- 1、mm\_struct（内存描述符）：描述一个进程的整个虚拟内存空间
- 2、vm\_area\_struct（区域结构描述符）：描述进程虚拟内存空间的一个区间

使用 fork 创建虚拟内存时，有以下步骤：

- 1、创建当前进程的 mm\_struct, vm\_area\_struct 和页表的原样副本
- 2、将两个进程的页面都标记为只读
- 3、将两个进程的每个 vm\_area\_struct 都标记为私有，只能进行写时复制。

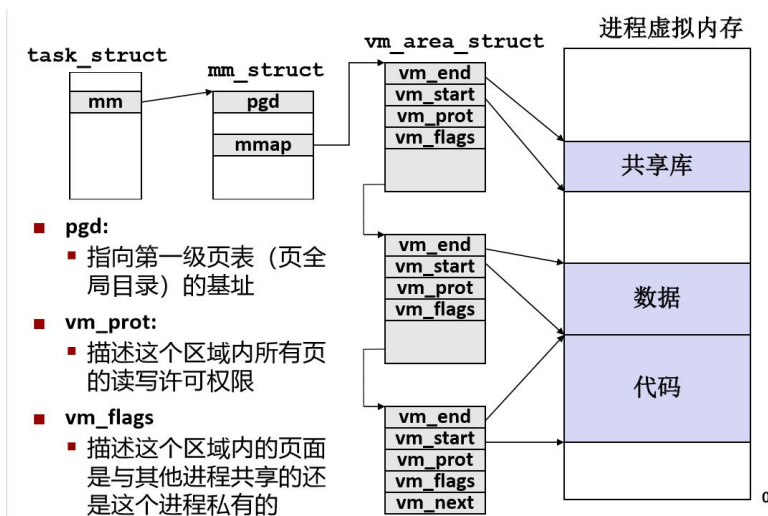


图 7.6.1

## 7.7 hello 进程 execve 时的内存映射

使用 `execve` 载入进程时，有以下步骤：

- 1、删除已存在的用户区域
- 2、创建新的私有区域（`.malloc,.data,.bss,.text`）
- 3、创建新的共享区域（`libc.so.data,libc.so.text`）
- 4、设置 PC，指向代码区域的入口点

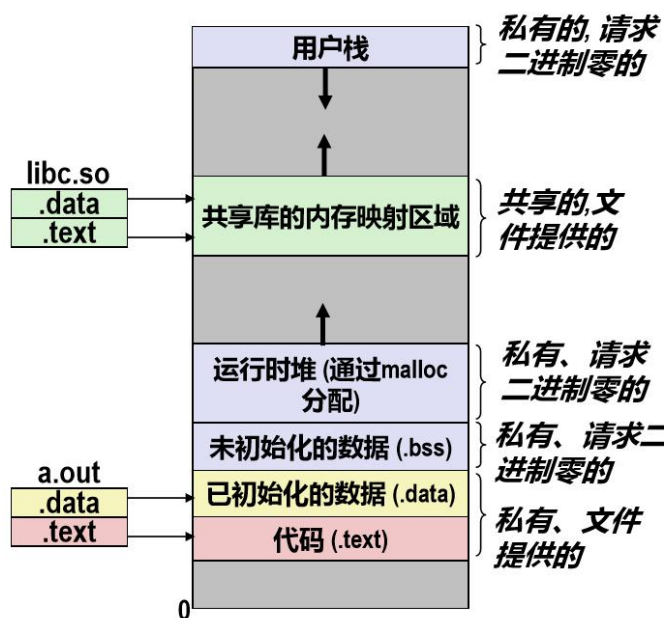


图 7.7.1

## 7.8 缺页故障与缺页中断处理

缺页故障：CPU 给出虚拟地址，根据寄存器寻找相应的物理地址数据时，在页表条目中没有所要找的数据，这种情况就是缺页故障。

缺页中断处理：

情况 1：段错误。先判断缺页的虚拟地址是否合法，遍历所有的合法区域结构，若虚拟地址对所有区域结构都不匹配，就返回一个段错误（segment fault）。

情况 2：非法访问。接着检查这个地址权限，判断进程是否有读写改该地址的权限。

情况 3：若非以上两种情况，就是正常缺页。选择一个页面牺牲换入新的页面，更新到页表即可。

## 7.9 动态存储分配管理



图 7.7.1

在程序运行时，使用上图所示的动态内存分配器为引用程序分配内存，动态内存分配器维护进程的虚拟内存。分配的动态内存在地址段中按照下图中位置为堆栈分配内存空间，称为堆。

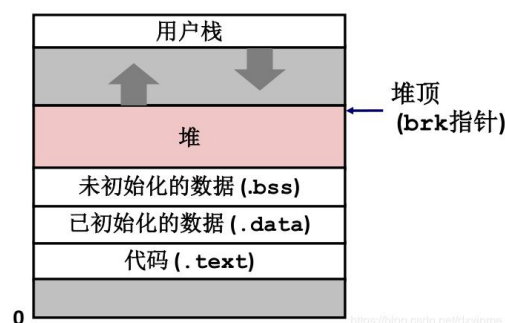


图 7.7.2

hello 程序使用的是 C 语言，其内存分配器为显式分配器，此处讨论显示分配器的内存管理方式。显示空间链表类似双向链表，只维护空闲空间块，不能通过块大小进行索引。

## ■ 方法 2: 显式空闲链表 (Explicit list) 在空闲块中使用指针

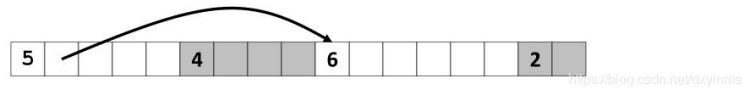


图 7.7.3

使用 `malloc(size_t size)` 函数申请内存时，每次内存空间都保证至少分配 `size_t` 大小的内存，并保证双字对齐。每次都必须从空闲块中分配内存空间，申请内存空间时，要将空闲内存碎片合并，尽可能减少浪费。

## 7.10 本章小结

本章介绍了 `hello` 的存储管理。简述了 `hello` 存储器地址空间，逻辑地址、物理地址、线性地址的概念。介绍了段式管理、页式管理的过程，以及 VA 到 PA 的转换。分析了三级 Cache 物理内存访问，程序 `fork` 和 `execve` 时的内存映射。最后给出了缺页故障的概念和缺页中断处理的流程，并给出了动态存储分配管理过程。

(第 7 章 2 分)

## 第 8 章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

IO 设备管理方法：

UNIX 系统中，所有的 I/O 设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行。这种将设备映射为文件的方式，允许 UNIX 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。

设备的模型化：文件

文件类型：

- 1、普通文件（**regular file**）：包含任意数据的文件。
- 2、目录（**directory**）：包含一组链接的文件，每个链接都将一个文件名映射到一个文件（他还有另一个名字叫做“文件夹”）。
- 3、套接字（**socket**）：用来与另一个进程进行跨网络通信的文件
- 4、命名通道
- 5、符号链接
- 6、字符和块设备

设备管理：unix io 接口

- 1、打开和关闭文件
- 2、读取和写入文件
- 3、改变当前文件的位置

### 8.2 简述 Unix IO 接口及其函数

打开和关闭文件：

1.int open(char \*filename,int flags,mode\_t mode);

该函数会打开一个已经存在的文件或者创建一个新的文件如果 open 的返回值为-1 则说明其打开该文件失败。

2.int close(int fd);

这个函数会关闭一个打开的文件。

读取和写入文件：

1. `ssize_t read(int fd, void *buf, size_t n);`

该函数会从当前文件位置复制字节到内存位置

2. `ssize_t write(int fd, const void *buf, size_t n);`

该函数从内存复制字节到当前文件位置

\*读写文件时，如果返回值 $<0$  则说明出现错误

文件读写位置：

`off_t lseek(int handle, off_t offset, int fromwhere);`

该函数用于指示文件要读写位置的偏移量。

### 8.3 printf 的实现分析

printf 函数代码如下：

```
int printf(const char *fmt, ...)
{
    int i;
    char buf[256];

    va_list arg = (va_list)((char*)&fmt + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);

    return i;
}
```

观察以上代码，可以发现 `printf` 函数中调用了两个外部函数，一个是 `vsprintf`，还有一个是 `write`。`vsprintf` 函数的作用是，将所有参数内容格式化后存入 `buf`，然后返回格式化后数组的长度；`write` 函数的作用是，将 `buf` 中的 `i` 个元素写到终端。

printf 运行过程：

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

getchar 函数代码如下：

```
int getchar(void)
{
    static char buf[BUFSIZ];
    static char* bb=buf;
    static int n=0;
    if(n==0)
    {
        n=read(0,buf,BUFSIZ);
        bb=buf;
    }
    return(--n>=0)?(unsigned char)*bb++:EOF;
}
```

从以上代码可以看到，getchar 函数调用了一个 read 函数，read 函数将缓冲区内容读取到 buf 里，返回值为缓冲区长度。代码中，若 buf 长度为 0，才调用 read 函数，否则直接返回 buf 中最前面的元素。

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 等调用 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

本章讲述了 linux 系统 I/O 设备管理方法，介绍了 Unix IO 接口及相关函数，分析了 printf 和 getchar 函数的实现与操作过程。

**(第 8 章 1 分)**



## 结论

hello 程序的执行，是在计算机系统从硬件到软件支持下才一步步完成的。

回顾 hello 的一生，它经过预处理和编译，才能被汇编成一个二进制文件 hello.o，这样还不够，我们用链接在它身上链接上了一系列函数，才最终让它成为了一个可以被 linux 系统执行的二进制可执行文件。我们在 shell 中使用 fork 和 execve 执行 hello，hello 成为了一个进程，被映射到内存里去，逐条执行，CPU 为其分配时间片，它拥有了自己的逻辑流。我们又暂停挂起它，然后脸滚键盘，打满缓冲区，又试着 kill 掉 hello，看看它出现异常后的反应，调用它的异常处理程序。到了最后，我们才使用 Ctrl-C，结束了它的一生。shell 回收了 hello 的内存，让它消失得无影无踪。我们也借着它的余晖，分析了它的内存管理和系统 IO。

hello 是一个简单的程序，但它背后实现的过程却不简单。做完大作业，回顾了计算机系统的课程，对整个计算机系统有了新的理解。

**（结论 0 分，缺失 -1 分，根据内容酌情加分）**

## 附件

hello.c	hello 的 c 语言源程序
hello.i	hello.c 预处理后的程序文本
hello.s	hello.i 编译成汇编语言后的程序文本
hello.o	hello.s 生成的二进制文件
hello.objdump	hello.o 反汇编后的文件
hello	hello 通过链接操作后生成的二进制可执行文件
hello_back.txt	hello 反汇编后重定向生成的程序文本
helloelf.txt	hello.o 的 readelf 输出的重定向文本
helloelf_second.txt	hello 的 readelf 输出的重定向文本

(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] 林来兴. 空间控制技术[M]. 北京: 中国宇航出版社, 1992: 25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集: A 集[C]. 北京: 中国科学出版社, 1999.
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北: 天下文化出版社, 1998 [1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨: 哈尔滨工业大学, 1992: 8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359) : 2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science , 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.
- [7] <https://blog.csdn.net/tuxedolinux/article/details/80317419>
- [8] <https://www.cnblogs.com/pianist/p/3315801.html>
- [9] <https://blog.csdn.net/gdj0001/article/details/80135196>
- [10] <http://www.techlog.cn/article/list/10182663>
- [11] <https://blog.csdn.net/GDJ0001/article/details/80135196>

(参考文献 0 分, 缺失 -1 分)