

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 計算機科學與技術

學 號 L170300901

班 級 1703009

學 生 盧兌琬

指 導 教 師 史先俊

計算機科學與技術學院

2018 年 12 月

摘 要

本论文旨在研究 hello 在 linux 系统下的整个生命周期。结合 CSAPP 课本，通过 gcc 等工具进行实验，从而将课本知识落实、融会贯通，通过一个程序深入挖掘知识点，对于学生对于课程的理解以及知识的升华有很大帮助。

关键词： CSAPP；HIT；大作业；Hello 程序；生命周期；

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

第 1 章 概述	- 4 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 5 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 7 -
2.4 本章小结	- 7 -
第 3 章 编译	- 8 -
3.1 编译的概念与作用	- 8 -
3.2 在 UBUNTU 下编译的命令	- 8 -
3.3 HELLO 的编译结果解析	- 9 -
3.4 本章小结	- 14 -
第 4 章 汇编	- 15 -
4.1 汇编的概念与作用	- 15 -
4.2 在 UBUNTU 下汇编的命令	- 15 -
4.3 可重定位目标 ELF 格式	- 15 -
4.4 HELLO.O 的结果解析	- 18 -
4.5 本章小结	- 18 -
第 5 章 链接	- 20 -
5.1 链接的概念与作用	- 20 -
5.2 在 UBUNTU 下链接的命令	- 20 -
5.3 可执行目标文件 HELLO 的格式	- 21 -
5.4 HELLO 的虚拟地址空间	- 21 -
5.5 链接的重定位过程分析	- 22 -
5.6 HELLO 的执行流程	- 25 -
5.7 HELLO 的动态链接分析	- 26 -
5.8 本章小结	- 27 -
第 6 章 HELLO 进程管理	- 29 -
6.1 进程的概念与作用	- 29 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 29 -
6.3 HELLO 的 FORK 进程创建过程	- 29 -
6.4 HELLO 的 EXECVE 过程	- 30 -
6.5 HELLO 的进程执行.....	- 31 -
6.6 HELLO 的异常与信号处理	- 32 -
6.7 本章小结	- 34 -
第 7 章 HELLO 的存储管理.....	- 35 -
7.1 HELLO 的存储器地址空间	- 35 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 35 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 37 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 39 -
7.5 三级 CACHE 支持下的物理内存访问	- 40 -
7.6 HELLO 进程 FORK 时的内存映射	- 41 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 41 -
7.8 缺页故障与缺页中断处理.....	- 42 -
7.9 动态存储分配管理	- 43 -
7.10 本章小结	- 45 -
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.....	- 47 -
8.4 GETCHAR 的实现分析.....	- 49 -
8.5 本章小结	- 49 -
结论	- 49 -
附件	- 51 -
参考文献.....	- 52 -

第 1 章 概述

1.1 Hello 简介

在 Editor 中键入代码得到 `hello.c` 程序。在 linux 中, `hello.c` 经过 `cpp` 的预处理、`ccl` 的编译、`as` 的汇编、`ld` 的链接最终成为可执行目标程序 `hello`, 在 `shell` 中键入启动命令后, `shell` 为其 `fork`, 产生子进程, 于是 `hello` 便从 Program 摇身一变成为 Process, 这便是 P2P 的过程。之后 `shell` 为其 `execve`, 映射虚拟内存, 进入程序入口后程序开始载入物理内存, 然后进入 `main` 函数执行目标代码, CPU 为运行的 `hello` 分配时间片执行逻辑控制流。当程序运行结束后, `shell` 父进程负责回收 `hello` 进程, 内核删除相关数据结构, 以上全部便是 020 的过程。

1.2 环境与工具

硬件环境: Intel Core i7-6700HQ x64CPU, 16G RAM, 256G SSD + 1T HDD. 软件环境: Ubuntu 18.04.1 LTS 开发与调试工具: vim, gcc, as, ld, edb, readelf, HexEdit

1.3 中间结果

文件名称	文件作用
<code>hello.i</code>	预处理之后文本文件
<code>hello.s</code>	编译之后的汇编文件
<code>hello.o</code>	汇编之后的可重定位目标执行
<code>hello</code>	链接之后的可执行目标文件
<code>hello2.c</code>	测试程序代码
<code>hello2</code>	测试程序
<code>hello.o.objdump</code>	<code>hello.o</code> 的反汇编代码
<code>hello.elf</code>	<code>hello.o</code> 的 ELF 格式
<code>hello.objdump</code>	<code>hello</code> 的反汇编代码
<code>hello.elf</code>	<code>hello</code> 的 ELF 格式
<code>tmp.txt</code>	存放临时数据

1.4 本章小结

本章主要简单介绍了 hello 的 p2p, 020 过程, 列出了本次实验信息: 环境、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

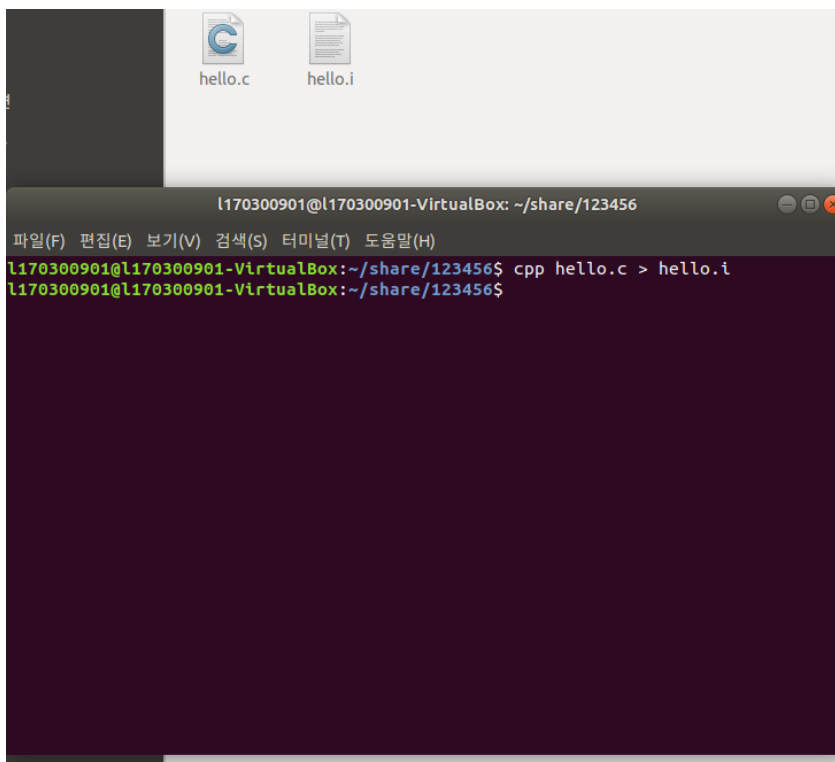
(以下格式自行编排, 编辑时删除)

概念: 预处理器 `cpp` 根据以字符`#`开头的命令(宏定义、条件编译), 修改原始的 C 程序, 将引用的所有库展开合并成为一个完整的文本文件。

主要功能如下:

- 1、将源文件中用`#include` 形式声明的文件复制到新的程序中。比如 `hello.c` 第 6-8 行中的`#include<stdio.h>` 等命令告诉预处理器读取系统头文件 `stdio.h` `unistd.h` `stdlib.h` 的内容, 并把它直接插入到程序文本中。
- 2、用实际值替换用`#define` 定义的字符串
- 3、根据`#if` 后面的条件决定需要编译的代码

2.2 在 Ubuntu 下预处理的命令



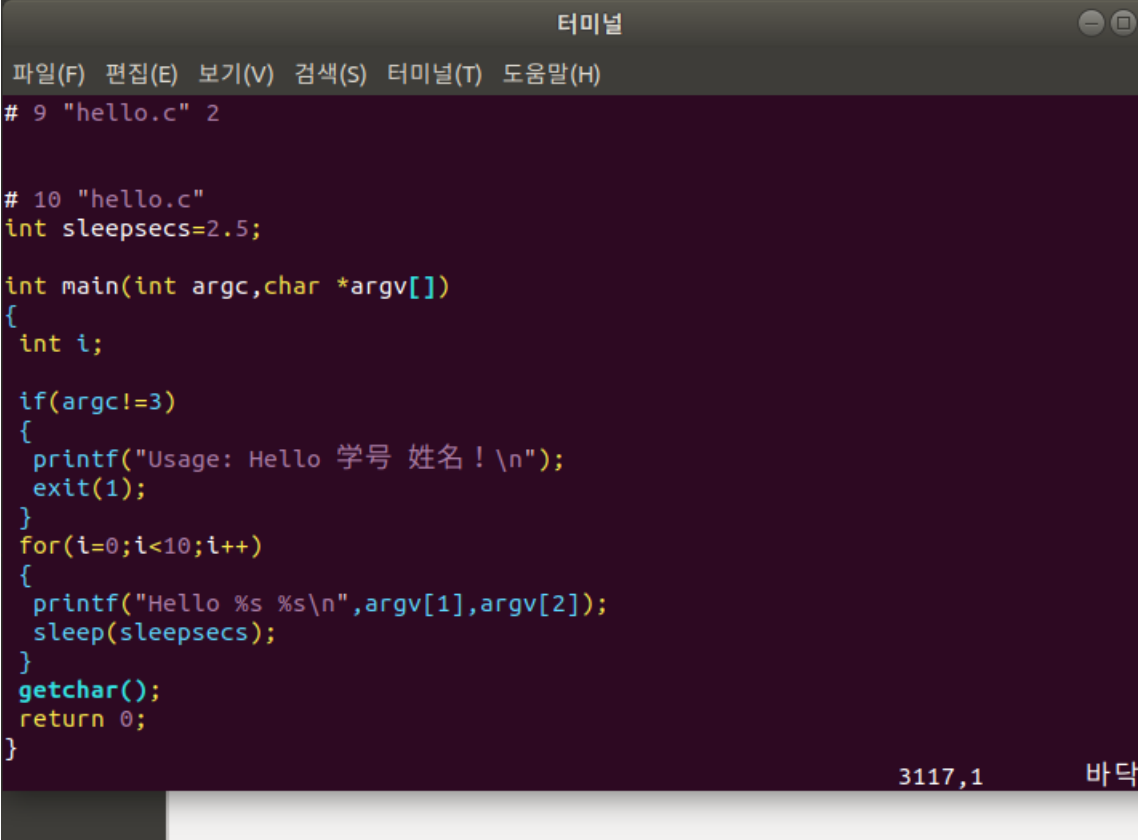
命令: `cpp hello.c > hello.i`

使用 `cpp` 命令生成 `hello.i` 文件

2.3 Hello 的预处理结果解析

使用 vim 打开 hello.i 之后发现，整个 hello.i 程序已经拓展为 3188 行，main 函数出现在 hello.c 中的代码自 3099 行开始。

如下：



```
# 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;
}
```

2.4 本章小结

hello.c 需要用到许多不是自身的“前件儿”，在真正投入造程序的汪洋大海之前还需要装备整齐，体体面面……

本章主要介绍了预处理的定义与作用、并结合预处理之后的程序对预处理结果进行了解析。

(第2章 0.5分)

第 3 章 编译

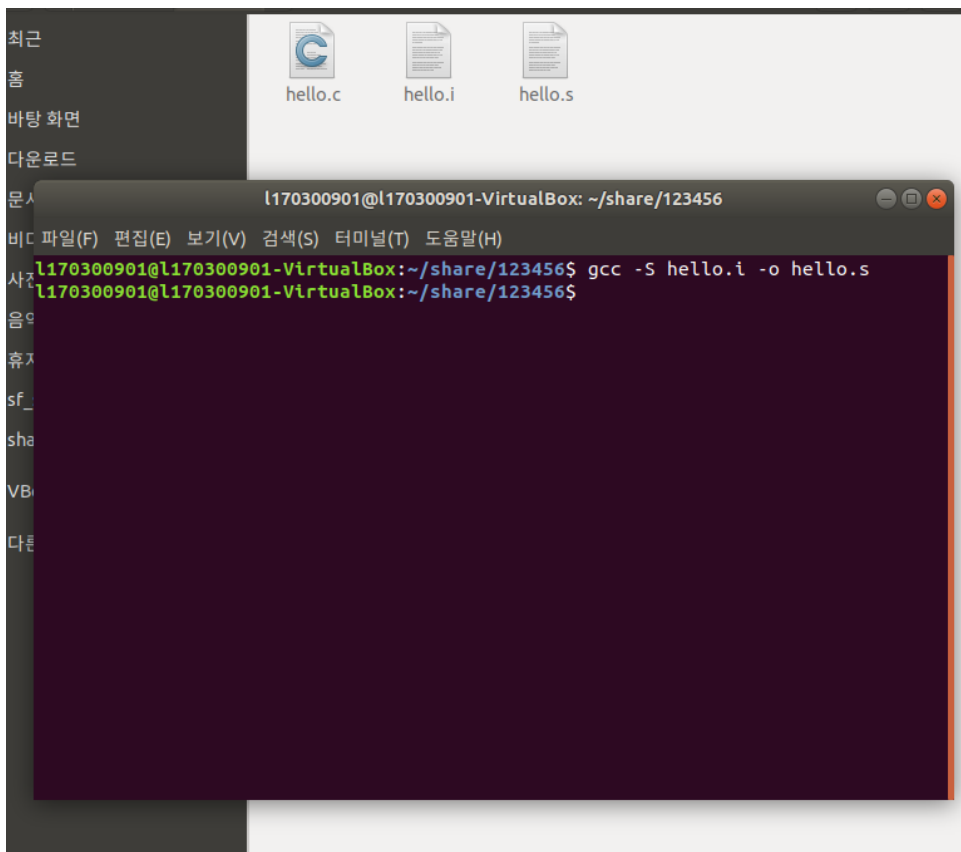
3.1 编译的概念与作用

编译器将文本文件 `hello.i` 翻译成文本文件 `hello.s`，它包含一个汇编语言程序。这个过程称为编译，同时也是编译的作用。

编译器的构建流程主要分为 3 个步骤：

1. 词法分析器，用于将字符串转化成内部的表示结构。
2. 语法分析器，将词法分析得到的标记流（token）生成一棵语法树。
3. 目标代码的生成，将语法树转化成目标代码。

3.2 在 Ubuntu 下编译的命令



命令： `gcc -S hello.i -o hello.s`

使用 `gcc` 命令生成 64 位的 `hello.s` 文件

3.3 Hello 的编译结果解析

3.3.0 汇编指令

指令	含义
.file	声明源文件
.text	以下是代码段
.section .rodata	以下是 rodata 节
.globl	声明一个全局变量
.type	用来指定是函数类型或是对象类型
.size	声明大小
.long、.string	声明一个 long、string 类型
.align	声明对指令或者数据的存放地址进行对齐的方式

3.3.1 数据 hello.s 中用到的 C 数据类型有：整数、字符串、数组。

一、字符串

程序中的字符串分别是：

1) “Usage: Hello 学号 姓名! \n”，第一个 printf 传入的输出格式化参数，在 hello.s 中声明如图 3.2，可以发现字符串被编码成 UTF-8 格式，一个汉字在 utf-8 编码中占三个字节，一个\代表一个字节。

2) “Hello %s %s\n”，第二个 printf 传入的输出格式化参数，在 hello.s 中声明如图 3.2。

其中后两个字符串都声明在了.rodata 只读数据节。

```

.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

```

二、整数

程序中涉及的整数有：

1) int sleepsecs: sleepsecs 在 C 程序中被声明为全局变量，且已经被赋值，编译器处理时在.data 节声明该变量，.data 节存放已经初始化的全局和静态 C 变量。在图 3.3 中，可以看到，编译器首先将 sleepsecs 在.text 代码段中声明为全局变量，其次在.data 段中，设置对齐方式为 4、设置类型为对象、设置大小为 4

字节、设置为 long 类型其值为 2（long 类型在 linux 下与 int 相同为 4B，将 int 声明为 long 应该是编译器偏好）。

```

.text
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
.long 2
.section .rodata

```

2) int i: 编译器将局部变量存储在寄存器或者栈空间中，在 hello.s 中 编译器将 i 存储在栈上空间-4(%rbp)中，可以看出 i 占据了栈中的 4B。

3) int argc: 作为第一个参数传入。

4) 立即数: 其他整形数据的出现都是以立即数的形式出现的，直接 硬编码在汇编代码中。

三、 数组

程序中涉及数组的是: char *argv[] main, 函数执行时输入的命令行, argv 作为存放 char 指针的数组同时是第二个参数传入。

argv 单个元素 char*大小为 8B, argv 指针指向已经分配好的、一片存 放着字符指针的连续空间,起始地址为 argv,main 函数中访问数组元素 argv[1],argv[2] 时,按照起始地址 argv 大小 8B 计算数据地址取数据,在 hello.s 中,使用两次 (%rax) (两次 rax 分别为 argv[1]和 argv[2]的地址) 取 出其值。

```

.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)

```

3.3.2 赋值

程序中涉及的赋值操作有:

1) `int sleepsecs=2.5` : 因为 `sleepsecs` 是全局变量, 所以直接在 `.data` 节中 将 `sleepsecs` 声明为值 2 的 `long` 类型数据。

2) `i=0`: 整型数据的赋值使用 `mov` 指令完成, 根据数据的大小不同使用不同后缀, 分别为:

指令	b	w	l	q
大小	8b (1B)	16b (2B)	32b (4B)	64b (8B)

因为 `i` 是 4B 的 `int` 类型, 所以使用 `movl` 进行赋值, 汇编代码 3.5

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

图 3.5 `hello.s` 中变量 `i` 的赋值

3.3.3 类型转换

程序中涉及隐式类型转换的是: `int sleepsecs=2.5`, 将浮点数类型的 2.5 转换为 `int` 类型。

当在 `double` 或 `float` 向 `int` 进行类型转换的时候, 程序改变数值和位模式的原则是: 值会向零舍入。例如 1.999 将被转换成 1, -1.999 将被转换成 -1。进一步来讲, 可能会产生值溢出的情况, 与 Intel 兼容的微处理器指定模式 [10...000] 为整数不确定值, 一个浮点数到整数的转换, 如果不能为该浮点数找到一个合适的整数近似值, 就会产生一个整数不确定值。

浮点数默认类型为 `double`, 所以上述强制转化是 `double` 强制转化为 `int` 类型。遵从向零舍入的原则, 将 2.5 舍入为 2。

3.3.4 算数操作

进行数据算数操作的汇编指令有:

指令	效果
<code>leaq S,D</code>	<code>D=&S</code>
<code>INC D</code>	<code>D+=1</code>
<code>DEC D</code>	<code>D-=1</code>
<code>NEG D</code>	<code>D=-D</code>
<code>ADD S,D</code>	<code>D=D+S</code>
<code>SUB S,D</code>	<code>D=D-S</code>
<code>IMULQ S</code>	<code>R[%rdx]:R[%rax]=S*R[%rax]</code> (有符号)
<code>MULQ S</code>	<code>R[%rdx]:R[%rax]=S*R[%rax]</code> (无符号)
<code>IDIVQ S</code>	<code>R[%rdx]=R[%rdx]:R[%rax] mod S</code> (有符号) <code>R[%rax]=R[%rdx]:R[%rax] div S</code>
<code>DIVQ S</code>	<code>R[%rdx]=R[%rdx]:R[%rax] mod S</code> (无符号) <code>R[%rax]=R[%rdx]:R[%rax] div S</code>

程序中涉及的算数操作有：

1) $i++$ ，对计数器 i 自增，使用程序指令 `addl`，后缀 `l` 代表操作数是一个 4B 大小的数据。

2) 汇编中使用 `leaq .LC1(%rip),%rdi`，使用了加载有效地址指令 `leaq` 计算 `LC1` 的段地址 `%rip+.LC1` 并传递给 `%rdi`

3.3.5 关系操作

进行关系操作的汇编指令有：

指令	效果	描述
<code>CMP S1,S2</code>	<code>S2-S1</code>	比较-设置条件码
<code>TEST S1,S2</code>	<code>S1&S2</code>	测试-设置条件码
<code>SET** D</code>	<code>D=**</code>	按照**将条件码设置 D
<code>J**</code>	——	根据**与条件码进行跳转

程序中涉及的关系运算为：

1) `argc!=3`：判断 `argc` 不等于 3。`hello.s` 中使用 `cmpl $3,-20(%rbp)`，计算 `argc-3` 然后设置条件码，为下一步 `je` 利用条件码进行跳转作准备。

2) `i<10`：判断 i 小于 10。`hello.s` 中使用 `cmpl $9,-4(%rbp)`，计算 `i-9` 然后设置条件码，为下一步 `jle` 利用条件码进行跳转做准备。

3.3.6 控制转移

程序中涉及的控制转移有：

1) `if(argv!=3)`：当 `argv` 不等于 3 的时候执行程序段中的代码。如图 3.6，对于 `if` 判断，编译器使用跳转指令实现，首先 `cmpl` 比较 `argv` 和 3，设置条件码，使用 `je` 判断 `ZF` 标志位，如果为 0，说明 `argv-3=0` `argv==3`，则不执行 `if` 中的代码直接跳转到 `.L2`，否则顺序执行下一条语句，即执行 `if` 中的代码。

```

    cmpl    $3, -20(%rbp)
    je      .L2
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT
.L2:
    movl    $0, -4(%rbp)
    jmp     .L3

```

图 3.6 `if` 语句的编译

2) `for(i=0;i<10;i++)`：使用计数变量 i 循环 10 次。如图 3.7，编译器的编译逻辑是，首先无条件跳转到位于循环体 `.L4` 之后的比较代码，使用 `cmpl` 进行比较，如果 `i<=9`，则跳入 `.L4` `for` 循环体执行，否则说明循环结束，顺序执行 `for` 之后的逻辑。

```

.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

```

图 3.7 for 循环的编译

3.3.7 函数操作

函数是一种过程，过程提供了一种封装代码的方式，用一组指定的参数和可选的返回值实现某种功能。P 中调用函数 Q 包含以下动作：

- 1) 传递控制：进行过程 Q 的时候，程序计数器必须设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。
- 2) 传递数据：P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 中返回一个值。
- 3) 分配和释放内存：在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些空间。

64 位程序参数存储顺序（浮点数使用 xmm，不包含）：

1	2	3	4	5	6	7
%rdi	%rsi	%rdx	%rcx	%r8	%r9	栈空间

程序中涉及函数操作的有：

1) main 函数：

a) 传递控制，main 函数因为被调用 call 才能执行（被系统启动函数 `__libc_start_main` 调用），call 指令将下一条指令的地址 dest 压栈，然后跳转到 main 函数。

b) 传递数据，外部调用过程向 `main` 函数传递参数 `argc` 和 `argv`，分别使用 `%rdi` 和 `%rsi` 存储，函数正常出口为 `return 0`，将 `%eax` 设置 0 返回。

c) 分配和释放内存，使用 `%rbp` 记录栈帧的底，函数分配栈帧空间在 `%rbp` 之上，程序结束时，调用 `leave` 指令，`leave` 相当于 `mov %rbp,%rsp, pop %rbp`，恢复栈空间为调用之前的状态，然后 `ret` 返回，`ret` 相当 `pop IP`，将下一条要执行指令的地址设置为 `dest`。

2) `printf` 函数：

a) 传递数据：第一次 `printf` 将 `%rdi` 设置为 “Usage: Hello 学号 姓名! \n” 字符串的首地址。第二次 `printf` 设置 `%rdi` 为 “Hello %s %s\n” 的首地址，设置 `%rsi` 为 `argv[1]`，`%rdx` 为 `argv[2]`。

b) 控制传递：第一次 `printf` 因为只有一个字符串参数，所以 `call puts@PLT`；第二次 `printf` 使用 `call printf@PLT`。

3) `exit` 函数：

a) 传递数据：将 `%edi` 设置为 1。

b) 控制传递：`call exit@PLT`。

4) `sleep` 函数：

a) 传递数据：

将 `%edi` 设置为 `sleepsecs`。

b) 控制传递：`call sleep@PLT`。

5) `getchar` 函数：

a) 控制传递：`call gethcar@PLT`

3.4 本章小结

造程序首先要承受 `ccl` 的“降维打击”，如果你问为什么，还不是因为要让 `as` 看得懂……

本章主要阐述了编译器是如何处理 C 语言的各个数据类型以及各类操作的，基本都是先给出原理然后结合 `hello.c` C 程序到 `hello.s` 汇编代码之间的映射关系作出合理解释。编译器将 `.i` 的拓展程序编译为 `.s` 的汇编代码。经过编译之后，我们的 `hello` 自 C 语言解构为更加低级的汇编语言。

(第3章2分)

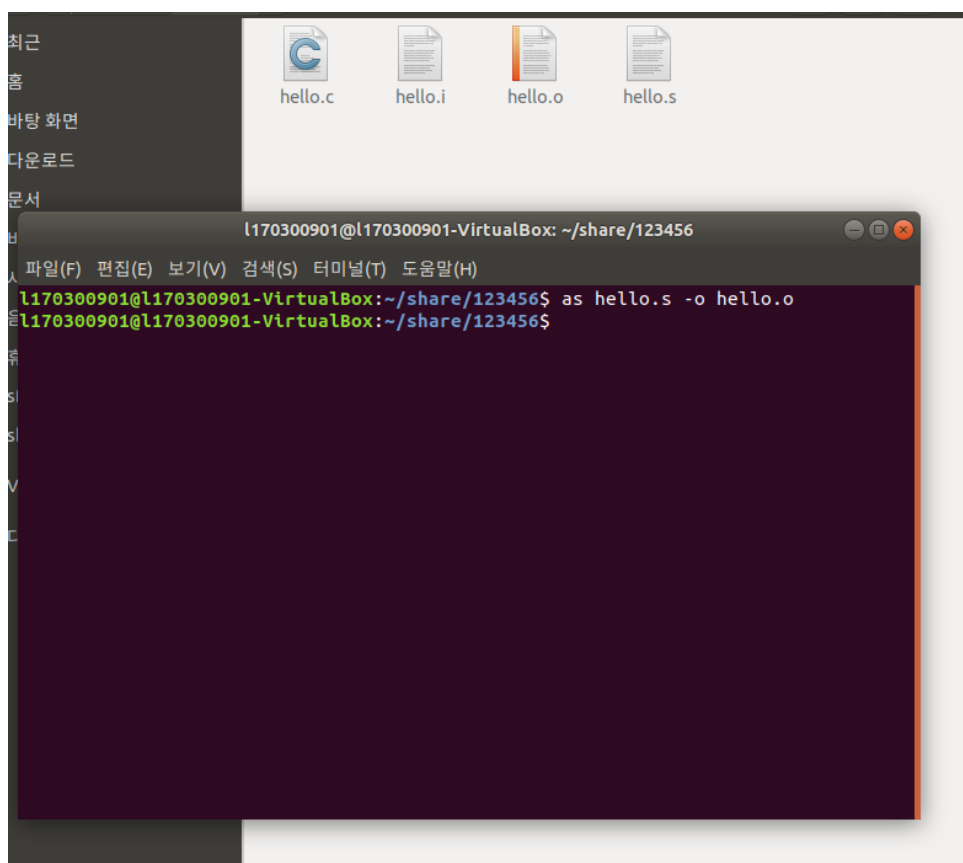
第 4 章 汇编

4.1 汇编的概念与作用

汇编器(as)将.s 汇编程序翻译成机器语言指令,把这些指令打包成可重定位目标程序的格式,并将结果保存在.o 目标文件中,.o 文件是一个二进制文件,它包含程序的指令编码。这个过程称为汇编,亦即汇编的作用。

4.2 在 Ubuntu 下汇编的命令

指令: `as hello.s -o hello.o`



指令: `as hello.s -o hello.o`

使用 `as` 指令生成 `hello.o` 文件

4.3 可重定位目标 elf 格式

使用 `readelf -a hello.o > helloo.elf` 指令获得 `hello.o` 文件的 ELF 格式。其组成如下:

1) **ELF Header:** 以 16B 的序列 Magic 开始, Magic 描述了生成该文件的系统 的字的大小和字节顺序, ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息, 其中包括 ELF 头的大小、目标文件的类型、机器类型、 字节头部表 (section header table) 的文件偏移, 以及节头部表中条目的大小和数量等信息。

```
l170300901@l170300901-VirtualBox:~/share/123456$ readelf -a hello.o
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
```

ELF Header

2) **Section Headers:** 节头部表, 包含了文件中出现的各个节的语义, 包括节的类型、位置和大小等信息。

```
Section Headers:
 [Nr] Name              Type              Address            Offset
      Size              EntSize          Flags Link Info Align
 [ 0]                      NULL              0000000000000000  0 0 0
      0000000000000000  0000000000000000
 [ 1] .text                PROGBITS          0000000000000000  00000040
      0000000000000081  0000000000000000  AX  0  0  1
 [ 2] .rela.text          RELA              0000000000000000  00000340
      00000000000000c0  0000000000000018  I   10  1  8
 [ 3] .data                PROGBITS          0000000000000000  000000c4
      0000000000000004  0000000000000000  WA  0  0  4
 [ 4] .bss                 NOBITS            0000000000000000  000000c8
      0000000000000000  0000000000000000  WA  0  0  1
 [ 5] .rodata              PROGBITS          0000000000000000  000000c8
      000000000000002b  0000000000000000  A   0  0  1
 [ 6] .comment             PROGBITS          0000000000000000  000000f3
      000000000000002b  0000000000000001  MS  0  0  1
 [ 7] .note.GNU-stack      PROGBITS          0000000000000000  0000011e
      0000000000000000  0000000000000000  0   0  1
 [ 8] .eh_frame            PROGBITS          0000000000000000  00000120
      0000000000000038  0000000000000000  A   0  0  8
 [ 9] .rela.eh_frame        RELA              0000000000000000  00000400
      0000000000000018  0000000000000018  I   10  8  8
[10] .symtab              SYMTAB            0000000000000000  00000158
      0000000000000198  0000000000000018  11  9  8
[11] .strtab              STRTAB            0000000000000000  000002f0
      000000000000004d  0000000000000000  0   0  1
[12] .shstrtab            STRTAB            0000000000000000  00000418
      0000000000000061  0000000000000000  0   0  1
```

节头部表 Section Headers

3) 重定位节.rela.text,一个.text 节中位置的列表,包含.text 节中需要进行重定位的信息,当链接器把这个目标文件和其他文件组合时,需要修改这些位置。如图 4.4,图中 8 条重定位信息分别是对.L0(第一个 printf 中的字符串)、puts 函数、exit 函数、.L1(第二个 printf 中的字符串)、printf 函数、sleepsecs、sleep 函数、getchar 函数进行重定位声明。

```
Relocation section '.rela.text' at offset 0x340 contains 8 entries:
  Offset          Info          Type           Sym. Value      Sym. Name + Addend
000000000018     000500000002 R_X86_64_PC32  0000000000000000 .rodata - 4
00000000001d     000c00000004 R_X86_64_PLT32 0000000000000000 puts - 4
000000000027     000d00000004 R_X86_64_PLT32 0000000000000000 exit - 4
000000000050     000500000002 R_X86_64_PC32 0000000000000000 .rodata + 1a
00000000005a     000e00000004 R_X86_64_PLT32 0000000000000000 printf - 4
000000000060     000900000002 R_X86_64_PC32 0000000000000000 sleepsecs - 4
000000000067     000f00000004 R_X86_64_PLT32 0000000000000000 sleep - 4
000000000076     001000000004 R_X86_64_PLT32 0000000000000000 getchar - 4
```

重定位节.rela.text

.rela 节的包含的信息有(readelf 显示与 hello.o 中的编码不同,以 hello.o 为准):

offset	需要进行重定向的代码在.text 或.data 节中的偏移位置,8 个字节。
Info	包括 symbol 和 type 两部分,其中 symbol 占前 4 个字节,type 占后 4 个字节,symbol 代表重定位到的目标在.symtab 中的偏移量,type 代表重定位的类型
Addend	计算重定位位置的辅助信息,共占 8 个字节
Type	重定位到的目标的类型
Name	重定向到的目标的名称

下面以.L1 的重定位为例阐述之后的重定位过程:链接器根据 info 信息向.symtab 节中查询链接目标的符号,由 info.symbol=0x05,可以发现重定位目标链接到.rodata 的.L1,设重定位条目为 r,根据图 4.5 知 r 的构造为:

r.offset=0x18, r.symbol=.rodata, r.type=R_X86_64_PC32, r.addend=-4,

重定位一个使用 32 位 PC 相对地址的引用。计算重定位目标地址的算法如下(设需要重定位的.text 节中的位置为 src,设重定位的目的位置 dst):

refptr = s + r.offset (1)

refaddr = ADDR(s) + r.offset (2)

*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr) (3)

其中(1)指向 src 的指针(2)计算 src 的运行地址,(3)中,

ADDR(r.symbol)计算 dst 的运行地址,在本例中,ADDR(r.symbol)获得的是 dst 的运行地址,因为需要设置的是绝对地址,即 dst 与下一条指令之间的地址之差,所以需要加上 r.addend=-4。

之后将 `src` 处设置为运行时值 `*refptr`，完成该处重定位。

00000340	18 00 00 00	00 00 00 00	02 00 00 00	05 00 00 00
00000350	FC FF FF FF	FF FF FF FF	1D 00 00 00	00 00 00 00
00000360	04 00 00 00	0C 00 00 00	FC FF FF FF	FF FF FF FF
00000370	27 00 00 00	00 00 00 00	04 00 00 00	0D 00 00 00
00000380	FC FF FF FF	FF FF FF FF	50 00 00 00	00 00 00 00
00000390	02 00 00 00	05 00 00 00	1A 00 00 00	00 00 00 00
000003A0	5A 00 00 00	00 00 00 00	04 00 00 00	0E 00 00 00
000003B0	FC FF FF FF	FF FF FF FF	60 00 00 00	00 00 00 00
000003C0	02 00 00 00	09 00 00 00	FC FF FF FF	FF FF FF FF
000003D0	67 00 00 00	00 00 00 00	04 00 00 00	0F 00 00 00
000003E0	FC FF FF FF	FF FF FF FF	76 00 00 00	00 00 00 00
000003F0	04 00 00 00	10 00 00 00	FC FF FF FF	FF FF FF FF
00000400	20 00 00 00	00 00 00 00	02 00 00 00	02 00 00 00

通过 HexEdit 查看 `hello.o` 中的 `.rela.text` 节

对于其他符号的重定位过程，情况类似。

3) `.rela.eh_frame` : `eh_frame` 节的重定位信息。

4) `.symtab`: 符号表，用来存放程序中定义和引用的函数和全局变量的信息。重定位需要引用的符号都在其中声明。

4.4 Hello.o 的结果解析

使用 `objdump -d -r hello.o > hello.o.objdump` 获得反汇编代码。

总体观察图 4.6 后发现，除去显示格式之外两者差别不大，主要差别如下：

- 1) 分支转移：反汇编代码跳转指令的操作数使用的不是段名称如 `L3`，因为段名称只是在汇编语言中便于编写的助记符，所以在汇编成机器语言之后显然不存在，而是确定的地址。
- 2) 函数调用：在 `.s` 文件中，函数调用之后直接跟着函数名称，而在反汇编程序中，`call` 的目标地址是当前下一条指令。这是因为 `hello.c` 中调用的函数都是共享库中的函数，最终需要通过动态链接器才能确定函数的运行时执行地址，在汇编成为机器语言的时候，对于这些不确定地址的函数调用，将其 `call` 指令后的相对地址设置为全 0（目标地址正是下一条指令），然后在 `.rela.text` 节中为其添加重定位条目，等待静态链接的进一步确定。
- 3) 全局变量访问：在 `.s` 文件中，访问 `rodata`（`printf` 中的字符串），使用段名称 `rodata`，在反汇编代码中 `0+rip`，因为 `rodata` 中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全 0 并添加重定位条目。

4.5 本章小结

啥，还要“降维打击”，我……

本章介绍了 `hello` 从 `hello.s` 到 `hello.o` 的汇编过程，通过查看 `hello.o` 的 `elf` 格式 和使用 `objdump` 得到反汇编代码与 `hello.s` 进行比较的方式，间接了解到从汇编语 言映射到机器语言汇编器需要实现的转换。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

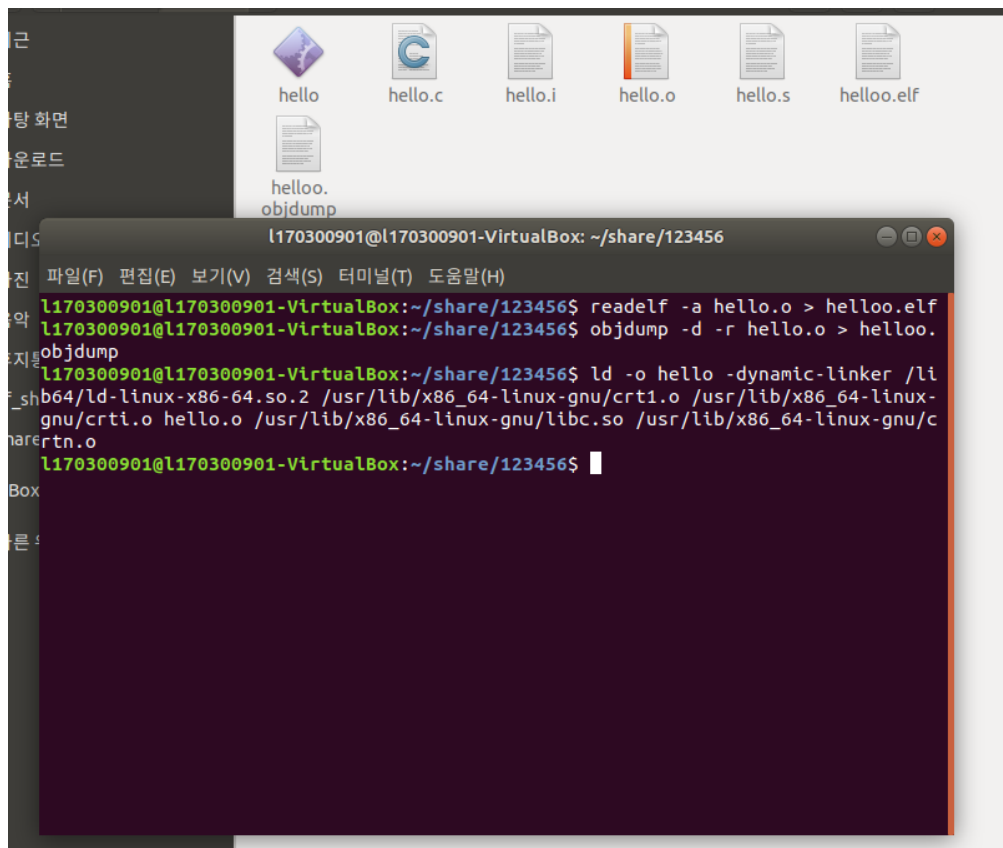
链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

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
```

注意：因为需要生成的是 64 位的程序，所以，使用的动态链接器和链接的目标文件都应该是 64 位的。



5.3 可执行目标文件 hello 的格式

使用 `readelf -a hello > hello.elf` 命令生成 hello 程序的 ELF 格式文件。

在 ELF 格式文件中，Section Headers 对 hello 中所有的节信息进行了声明，其中包括大小 Size 以及在程序中的偏移量 Offset，因此根据 Section Headers 中的信息我们就可以用 HexEdit 定位各个节所占的区间（起始位置，大小）。其中 Address 是程序被载入到虚拟地址的起始地址。

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info	Align
[0]	0000000000000000	NULL	0000000000000000	00000000
			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
	00000000000000132	0000000000000000	AX 0 0	16
[14]	.fini	PROGBITS	0000000000400634	00000634
	0000000000000009	0000000000000000	AX 0 0	4
[15]	.rodata	PROGBITS	0000000000400640	00000640
	000000000000002f	0000000000000000	A 0 0	4
[16]	.eh_frame	PROGBITS	0000000000400670	00000670
	00000000000000fc	0000000000000000	A 0 0	8
[17]	.dynamic	DYNAMIC	0000000000600e50	00000e50
	000000000000001a0	0000000000000010	WA 6 0	8
[18]	.got	PROGBITS	0000000000600ff0	00000ff0
	000000000000001a0	0000000000000010	WA 0 0	56,1 9%
[18]	.got	PROGBITS	0000000000600ff0	00000ff0
	0000000000000010	0000000000000008	WA 0 0	8
[19]	.got.plt	PROGBITS	0000000000601000	00001000
	0000000000000040	0000000000000008	WA 0 0	8
[20]	.data	PROGBITS	0000000000601040	00001040
	0000000000000008	0000000000000000	WA 0 0	4
[21]	.comment	PROGBITS	0000000000000000	00001048
	000000000000002a	0000000000000001	MS 0 0	1
[22]	.symtab	SYMTAB	0000000000000000	00001078
	00000000000000498	0000000000000018	23 28	8
[23]	.strtab	STRTAB	0000000000000000	00001510
	00000000000000150	0000000000000000	0 0	1
[24]	.shstrtab	STRTAB	0000000000000000	00001660
	00000000000000c5	0000000000000000	0 0	1

图 5.2 hello ELF 格式中的 Section Headers Table

5.4 hello 的虚拟地址空间

使用 edb 打开 hello 程序，通过 edb 的 Data Dump 窗口查看加载到虚拟地址中的 hello 程序。

在 0x400000~0x401000 段中，程序被载入，自虚拟地址 0x400000 开始，自

0x400fff 结束，这之间每个节（开始 ~ .eh_frame 节）的排列即开始结束同图 5.2 中 Address 中声明。

如图 5.3，查看 ELF 格式文件中的 Program Headers，程序头表在执行的时候被使用，它告诉链接器运行时加载的内容并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。在下面可以看出，程序包含 8 个段：

- 1) PHDR 保存程序头表。
- 2) INTERP 指定在程序已经从可执行文件映射到内存之后，必须调用的解释器（如动态链接器）。
- 3) LOAD 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量数据（如字符串）、程序的目标代码等。
- 4) DYNAMIC 保存了由动态链接器使用的信息。
- 5) NOTE 保存辅助信息。
- 6) GNU_STACK：权限标志，标志栈是否是可执行的。
- 7) GNU_RELRO：指定在重定位结束之后那些内存区域是需要设置只读。

通过 Data Dump 查看虚拟地址段 0x600000~0x602000，在 0~fff 空间中，与 0x400000~0x401000 段的存放的程序相同，在 fff 之后存放的是.dynamic~.shstrtab 节。

5.5 链接的重定位过程分析

使用 `objdump -d -r hello > hello.objdump` 获得 hello 的反汇编代码。

与 hello.o 反汇编文本 hello.o.objdump 相比，在 hello.objdump 中多了许多节，列在下面。

节名称	描述
.interp	保存 ld.so 的路径
.note.ABI-tag	Linux 下特有的 section
.hash	符号的哈希表
.gnu.hash	GNU 拓展的符号的哈希
.dynsym	运行时/动态符号表
.dynstr	存放.dynsym 节中的符号名称
.gnu.version	符号版本
.gnu.version_r	符号引用版本
.rela.dyn	运行时/动态重定位表
.rela.plt	.plt 节的重定位条目
.init	程序初始化需要执行的代码
.plt	动态链接-过程链接表
.fini	当程序正常终止时需要执行的代码
.eh_frame	contains exception unwinding and source language information.
.dynamic	存放被 ld.so 使用的动态链接信
.got	动态链接-全局偏移量表-存放变
.got.plt	动态链接-全局偏移量表-存放函数
.data	初始化了的数据
.comment	一串包含编译器的 NULL-terminated 字符串

通过比较 hello.objdump 和 hello.o.objdump 了解链接器。

1) 函数个数：在使用 ld 命令链接的时候，指定了动态链接器为 64 的 /lib64/ld-linux-x86-64.so.2, crt1.o、crti.o、crtm.o 中主要定义了程序入口_start、初始化函数_init, _start 程序调用 hello.c 中的 main 函数, libc.so 是动态链接共享库，其中定义了 hello.c 中用到的 printf、sleep、getchar、exit 函数和_start 中调用的 __libc_csu_init, __libc_csu_fini, __libc_start_main。链接器将上述函数加入。

2) 函数调用：链接器解析重定条目时发现对外部函数调用的类型为 R_X86_64_PLT32 的重定位，此时动态链接库中的函数已经加入到了 PLT 中，.text 与.plt 节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为 PLT 中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位 链接器为其构造.plt 与.got.plt。

3) .rodata 引用：链接器解析重定条目时发现两个类型为 R_X86_64_PC32 的对.rodata 的重定位（printf 中的两个字符串），.rodata 与.text 节之间的相对距离确定，因此链接器直接修改 call 之后的值为目标地址与下一条指令的地址之差，

指向相应的字符串。这里以计算第一条字符串相对地址为例说明计算相对地址的算法（算法说明同 4.3 节）：

$\text{refptr} = s + r.\text{offset} = \text{Pointer to } 0x40054A$
 $\text{refaddr} = \text{ADDR}(s) + r.\text{offset} = \text{ADDR}(\text{main}) + r.\text{offset} = 0x400532 + 0x18 = 0x40054A$
 $*\text{refptr} = (\text{unsigned}) (\text{ADDR}(r.\text{symbol}) + r.\text{addend} - \text{refaddr}) =$
 $\text{ADDR}(\text{str1}) + r.\text{addend} - \text{refaddr} = 0x400644 + (-0x4) - 0x40054A = (\text{unsigned}) 0xF6,$

观察反汇编验证计算：

```
400547: 48 8d 3d f6 00 00 00 lea    0xf6(%rip),%rdi    # 400644 < 10>
```

其他.rodata 引用，函数调用原理类似。

5.6 hello 的执行流程

使用 edb 执行 hello，观察函数执行流程，将过程中执行的主要函数列在下面：

程序名称	程序地址
ld-2.27.so!_dl_start	0x7fce 8cc38ea0
ld-2.27.so!_dl_init	0x7fce 8cc47630
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

对于动态共享链接库中 PIC 函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表 PLT+全局偏移量表 GOT 实现函数的动态链接，GOT 中存放函数目标地址，PLT 使用 GOT 中地址 跳转到目标函数。

在 `dl_init` 调用之前，对于每一条 PIC 函数调用，调用的目标地址都实际指向 PLT 中的代码逻辑，GOT 存放的是 PLT 中函数调用指令的下一条指令地址。如在 图 5.4 (a)。

在 `dl_init` 调用之后，如图 5.4 (b)，`0x601008` 和 `0x601010` 处的两个 8B 数据分别发生改变为 `0x7fd9 d3925170` 和 `0x7fd9 d3713680`，如图 5.4 (c) 其中 GOT[1]指向重定位表（依次为.plt 节需要重定位的函数的运行时地址）用来确定调用的函数 地址，如图 5.4 (d) GOT[2]指向动态链接器 `ld-linux.so` 运行时地址。

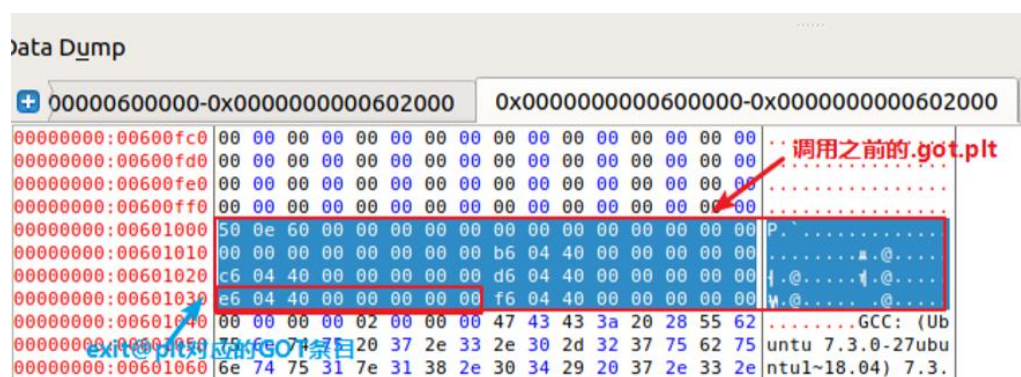


图 5.4 (a) 没有调用 `dl_init` 之前的全局偏移量表.got.plt (根据.plt 中 `exit@plt jmp` 的引用地址 `0x601030` 可以得到其.got.plt 条目为 `0x4004e6`，正是其下条指令地址)

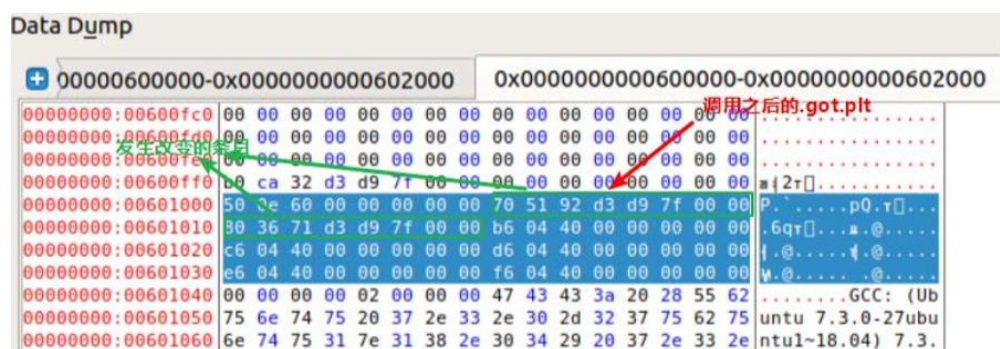


图 5.4 (b) 调用 `dl_init` 之后的全局偏移量表.got.plt

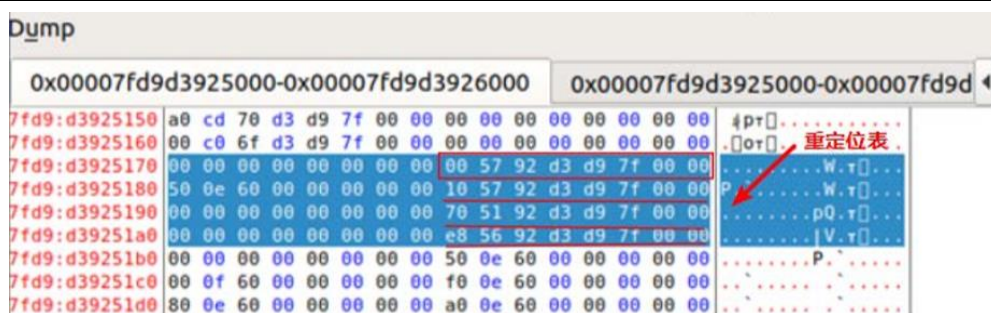


图 5.3 (c) 0x7fd9 d3925170 指向的重定位表

00007fd9:d3713680	53	pushq %rbx
00007fd9:d3713681	48 89 e3	movq %rsp, %rbx
00007fd9:d3713684	48 83 e4 c0	andq \$0xffffffffc0, %rsp
00007fd9:d3713688	48 2b 25 79 01 21 00	subq 0x210179(%rip), %rsp
00007fd9:d371368f	48 89 04 24	movq %rax, (%rsp)
00007fd9:d3713693	48 89 4c 24 08	movq %rcx, 8(%rsp)
00007fd9:d3713698	48 89 54 24 10	movq %rdx, 0x10(%rsp)
00007fd9:d371369d	48 89 74 24 18	movq %rsi, 0x18(%rsp)
00007fd9:d37136a2	48 89 7c 24 20	movq %rdi, 0x20(%rsp)
00007fd9:d37136a7	4c 89 44 24 28	movq %r8, 0x28(%rsp)
00007fd9:d37136ac	4c 89 4c 24 30	movq %r9, 0x30(%rsp)
00007fd9:d37136b1	b8 ee 00 00 00	movl \$0xee, %eax
00007fd9:d37136b6	31 d2	xorl %edx, %edx
00007fd9:d37136b8	48 89 94 24 40 02 00 00	movq %rdx, 0x240(%rsp)
00007fd9:d37136c0	48 89 94 24 48 02 00 00	movq %rdx, 0x248(%rsp)
00007fd9:d37136c8	48 89 94 24 50 02 00 00	movq %rdx, 0x250(%rsp)
00007fd9:d37136d0	48 89 94 24 58 02 00 00	movq %rdx, 0x258(%rsp)
00007fd9:d37136d8	48 89 94 24 60 02 00 00	movq %rdx, 0x260(%rsp)
00007fd9:d37136e0	48 89 94 24 68 02 00 00	movq %rdx, 0x268(%rsp)
00007fd9:d37136e8	48 89 94 24 70 02 00 00	movq %rdx, 0x270(%rsp)
00007fd9:d37136f0	48 89 94 24 78 02 00 00	movq %rdx, 0x278(%rsp)
00007fd9:d37136f8	0f ae 64 24 40	xsave 0x40(%rsp)
00007fd9:d37136fd	48 8b 73 10	movq 0x10(%rbx), %rsi

在之后的函数调用时，首先跳转到 PLT 执行，plt 中逻辑，第一次访问跳转到 GOT 地址为下一条指令，将函数序号压栈，然后跳转到 PLT[0]，在 PLT[0]中将重定位表地址压栈，然后访问动态链接器，在动态链接器中使用函数序号和重定位表确定函数运行时地址，重写 GOT，再将控制传递给目标函数。之后如果对同样函数调用，第一次访问跳转直接跳转到目标函数。

因为在 PLT 中使用的 jmp，所以执行完目标函数之后的返回地址为最近 call 指令下一条指令地址，即在 main 中的调用完成地址。

5.8 本章小结

“大贤者” ld 赋予了 hello.o “捕食者”技能，hello.o 不是仇家 crt1.o, crt0.o, crt0.o, libc.so 中需要的技能成为可执行程序 hello，最终消灭“仇人笔记本”上所有仇家，但真正的链接不止于此，对于 hello 中需要的 PIC 函数调用则需要动态链接器 /lib64/ld-linux-x86-64.so.2，ld.so 是个懒家伙，只有在函数调用的时候才会进行实际上的重定位，这就是动态链接中的延迟绑定。历经艰辛，hello 可算诞生了呦 QWQ

在本章中主要介绍了链接的概念与作用、hello 的 ELF 格式,分析了 hello 的虚拟地址空间、重定位过程、执行流程、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

进程是一个执行中的程序的实例，每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域、和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用：Shell 是一个用 C 语言编写的程序，他是用户使用 Linux 的桥梁。Shell 是指一种应用程序，Shell 应用程序提供了一个界面，用户通过这个界面访问 操作系统内核的服务。

处理流程：

- 1) 从终端读入输入的命令。
- 2) 将输入字符串切分获得所有的参数
- 3) 如果是内置命令则立即执行
- 4) 否则调用相应的程序为其分配子进程并运行
- 5) shell 应该接受键盘输入信号，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 Gnome-Terminal 中键入 `./hello L170300901 lidaxin`，运行的终端程序会对输入的命令行进行解析，因为 `hello` 不是一个内置的 `shell` 命令所以解析之后终端 程序判断`./hello` 的语义为执行当前目录下的可执行目标文件 `hello`，之后终端程序 首先会调用 `fork` 函数创建一个新的运行的子进程，新创建的子进程几乎但不完全 与父进程相同，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，这就意味着，当父进程调用 `fork` 时，子进程可以读写父进程中打开的 任何文件。父进程与子进程之间最大的区别在于它们拥有不同的 `PID`。

父进程与子进程是并发运行的独立进程，内核能够以任意方式交替执行它们的逻辑控制流的指令。在子进程执行期间，父进程默认选项是显示等待子进程的完成。

简单进程图如下：

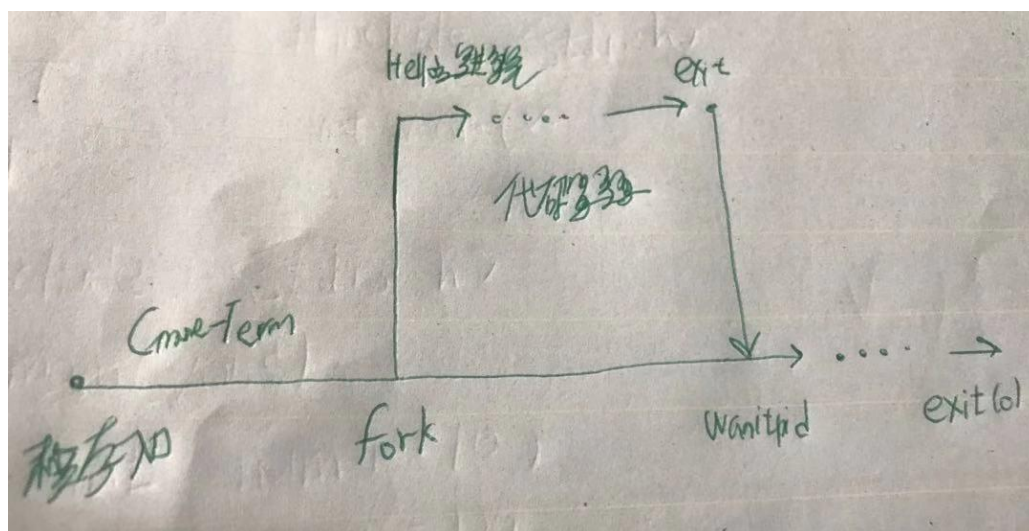


图 6.1 终端程序的简单进程图

6.4 Hello 的 execve 过程

当 `fork` 之后，子进程调用 `execve` 函数（传入命令行参数）在当前进程的上下文中加载并运行一个新程序即 `hello` 程序，`execve` 调用驻留在内存中的被称为启动加载器的操作系统代码来执行 `hello` 程序，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置 `PC` 指向 `_start` 地址，`_start` 最终调用 `hello` 中的 `main` 函数。除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到 `CPU` 引用一个被映射的虚拟页时才会进行复制，这时，操作系统利用它的页面调度机制自动将页面从磁盘传送到内存。

加载器创建的内存映像如下：

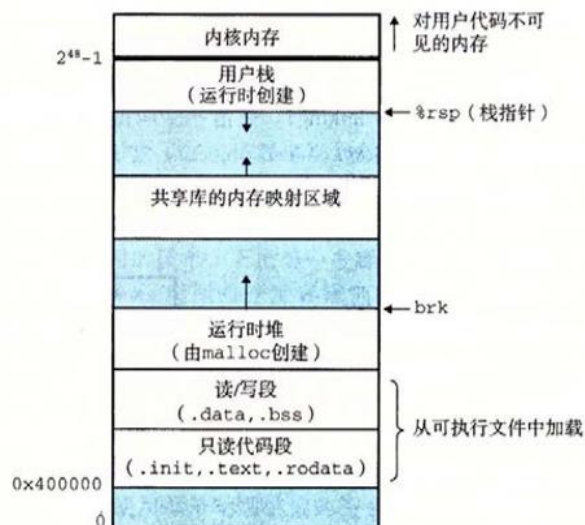


图 6.2 启动加载器创建的系统映像

6.5 Hello 的进程执行

逻辑控制流：一系列程序计数器 PC 的值的序列叫做逻辑控制流，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

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。

用户模式和内核模式：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程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；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

上下文信息：上下文就是内核重新启动一个被抢占的进程所需要的状态，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值构成。

简单看 hello sleep 进程调度的过程：当调用 sleep 之前，如果 hello 程序不被抢占则顺序执行，假如发生被抢占的情况，则进行上下文切换，上下文切换是由内核中调度器完成的，当内核调度新的进程运行后，它就会抢占当前进程，并进行 1) 保存以前进程的上下文 2) 恢复新恢复进程被保存的上下文，3) 将控制传递给这个新恢复的进程，来完成上下文切换。如图 6.3，hello 初始运行在用户模式，在 hello 进程调用 sleep 之后陷入内核模式，内核处理休眠请求主动释放当前进程，并将 hello 进程从运行队列中移出加入等待队列，定时器开始

计时，内核进行上下文切换将当前进程的控制权交给其他进程，当定时器到时（2.5secs）发送一个中断信号，此时进入内核状态执行中断处理，将 hello 进程从等待队列中移出重新加入到运行队列，成为就绪状态，hello 进程就可以继续进行自己的控制逻辑流了。

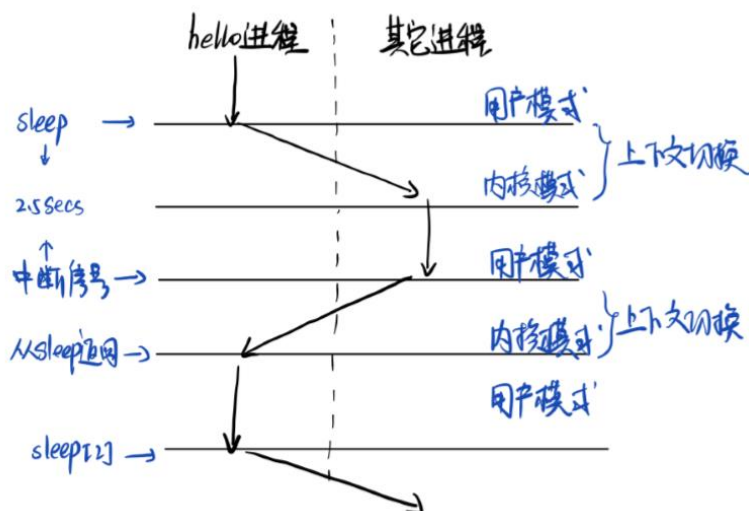


图 6.3 hello 进程 sleep 上下文切换的简单理解

之后的 9 个 sleep 进程调度如上。当 hello 调用 getchar 的时候，实际落脚到执行输入流是 stdin 的系统调用 read，hello 之前运行在用户模式，在进行 read 调用之后陷入内核，内核中的陷阱处理程序请求来自键盘缓冲区的数据传输，并且安排在完成从键盘缓冲区到内存的数据传输后，中断处理器。此时进入内核模式，内核执行上下文切换，切换到其他进程。当完成键盘缓冲区到内存的数据传输时，引发一个中断信号，此时内核从其他进程进行上下文切换回 hello 进程。进程切换如图 6.3，省略。

6.6 hello 的异常与信号处理

如图 6.4 (a)，是正常执行 hello 程序的结果，当程序执行完成之后，进程被回收。

如图 6.4(b)，是在程序输出 2 条 info 之后按下 ctrl-z 的结果，当按下 ctrl-z 之后，shell 父进程收到 SIGSTP 信号，信号处理函数的逻辑是打印屏幕回显、将 hello 进程挂起，通过 ps 命令我们可以看出 hello 进程没有被回收，此时他的后台 job 号是 1，调用 fg 1 将其调到前台，此时 shell 程序首先打印 hello 的命令，hello 继续运行打印剩下的 8 条 info，之后输入字符串，程序结束，同时进程被回收。

如图 6.4 (c) 是在程序输出 3 条 info 之后按下 ctrl-c 的结果，当按下 ctrl-c 之后，shell 父进程收到 SIGINT 信号，信号处理函数的逻辑是结束 hello，并回收 hello 进程。

如图 6.4 (d) 是在程序运行中途乱按的结果，可以发现，乱按只是将屏幕的输入缓存到 stdin，当 getchar 的时候读出一个 '\n' 结尾的字串（作为一次输入），其他字串会当做 shell 命令行输入。

```

l170300901@l170300901-VirtualBox:~/share/123456$ ./hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
^Z
[1]+  정지됨                  ./hello L170300901 luduiyun
l170300901@l170300901-VirtualBox:~/share/123456$ ps
  PID TTY          TIME CMD
 2198 pts/0        00:00:00 bash
 2267 pts/0        00:00:00 hello
 2274 pts/0        00:00:00 ps
l170300901@l170300901-VirtualBox:~/share/123456$ fg 1
./hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
bfgf
l170300901@l170300901-VirtualBox:~/share/123456$ jobs

```

图 6.4 (b) 运行中途按下 ctrl-z

```

l170300901@l170300901-VirtualBox:~/share/123456$ ./hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
^C
l170300901@l170300901-VirtualBox:~/share/123456$ ps
  PID TTY          TIME CMD
 2198 pts/0        00:00:00 bash
 2279 pts/0        00:00:00 ps

```

图 6.4 (c) 运行中途按下 ctrl-c

```
l170300901@l170300901-VirtualBox:~/share/123456$ ./hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
Hello L170300901 luduiyun
f
Hello L170300901 luduiyun
f
Hello L170300901 luduiyun
asdf
Hello L170300901 luduiyun
Hello L170300901 luduiyun
asd
adsHello L170300901 luduiyun
ad
Hello L170300901 luduiyun
l170300901@l170300901-VirtualBox:~/share/123456$ f
f: 명령을 찾을 수 없습니다
l170300901@l170300901-VirtualBox:~/share/123456$ asdf

Command 'asdf' not found, did you mean:

  command 'adsf' from deb ruby-adsf
  command 'sdf' from deb sdf
  command 'asdfg' from deb aoewi
  command 'sadf' from deb sysstat

Try: sudo apt install <deb name>
l170300901@l170300901-VirtualBox:~/share/123456$ asd
Command 'asd' not found, but there are 24 similar ones.
```

图 6.4 (d) 运行中途乱按

6.7 本章小结

Shell (Gnome-Terminal) 下达命令，进程管理为 `hello` 提供了活动空间，Shell 为其 `fork`，为其 `execve`，为其分配时间片，Linux 是繁忙的，但是却依靠进程调度使得每个进程安稳运行，在庞大但有序的 Linux 都市中，`hello` 还只是个 naive 的孩子呀，too young, too simple……

在本章中，阐明了进程的定义与作用，介绍了 Shell 的一般处理流程，调用 `fork` 创建新进程，调用 `execve` 执行 `hello`，`hello` 的进程执行，`hello` 的异常与信号处理

(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

物理地址：CPU 通过地址总线的寻址，找到真实的物理内存对应地址。CPU 对内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址。

逻辑地址：程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏移部分）组成。

线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。至于虚拟地址，只关注 CSAPP 课本中提到的虚拟地址，实际上就是这里的线性地址。

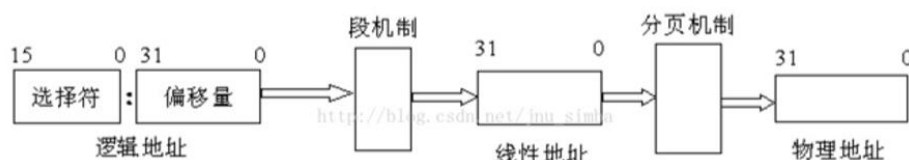


图 7.1[转] 三种地址之间的关系

7.2 Intel 逻辑地址到线性地址的变换-段式管理

最初 8086 处理器的寄存器是 16 位的，为了能够访问更多的地址空间但不改变寄存器和指令的位宽，所以引入段寄存器，8086 共设计了 20 位宽的地址总线，通过将段寄存器左移 4 位加上偏移地址得到 20 位地址，这个地址就是逻辑地址。将内存分为不同的段，段有段寄存器对应，段寄存器有一个栈、一个代码、两个数据寄存器。

分段功能在实模式和保护模式下有所不同。

实模式，即不设防，也就是说逻辑地址=线性地址=实际的物理地址。段寄存器存放真实段基址，同时给出 32 位地址偏移量，则可以访问真实物理内存。

在保护模式下，线性地址还需要经过分页机制才能够得到物理地址，线性地址也需要逻辑地址通过段机制来得到。段寄存器无法放下 32 位段基址，所以它们被称作选择符，用于引用段描述符表中的表项来获得描述符。描述符表中的一个条目描述一个段，构造如下：

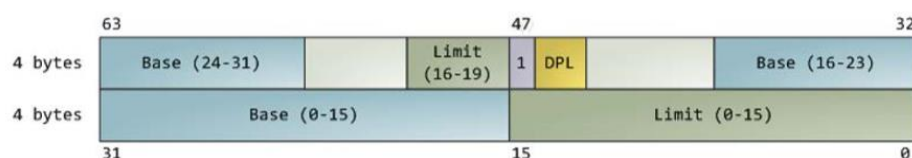


图 7.2[转] 段描述符表中的一个条目的构造

Base: 基地址，32 位线性地址指向段的开始。**Limit:** 段界限，段的大小。**DPL:** 描述符的特权级 0（内核模式）-3（用户模式）。

所有的段描述符被保存在两个表中：全局描述符表 **GDT** 和局部描述符表 **LDT**。**gdt** 寄存器指向 **GDT** 表基址。

段选择符构造如下：



图 7.3[转] 段选择符的构造

TI: 0 为 **GDT**，1 为 **LDT**。**Index** 指出选择描述符表中的哪个条目，**RPL** 请求 特权级。

所以在保护模式下，分段机制就可以描述为：通过解析段寄存器中的段选择符 在段描述符表中根据 **Index** 选择目标描述符条目 **Segment Descriptor**，从目标描述符中提取出目标段的基地址 **Base address**，最后加上偏移量 **offset** 共同构成线性地址 **Linear Address**。保护模式时分段机制图示如下：

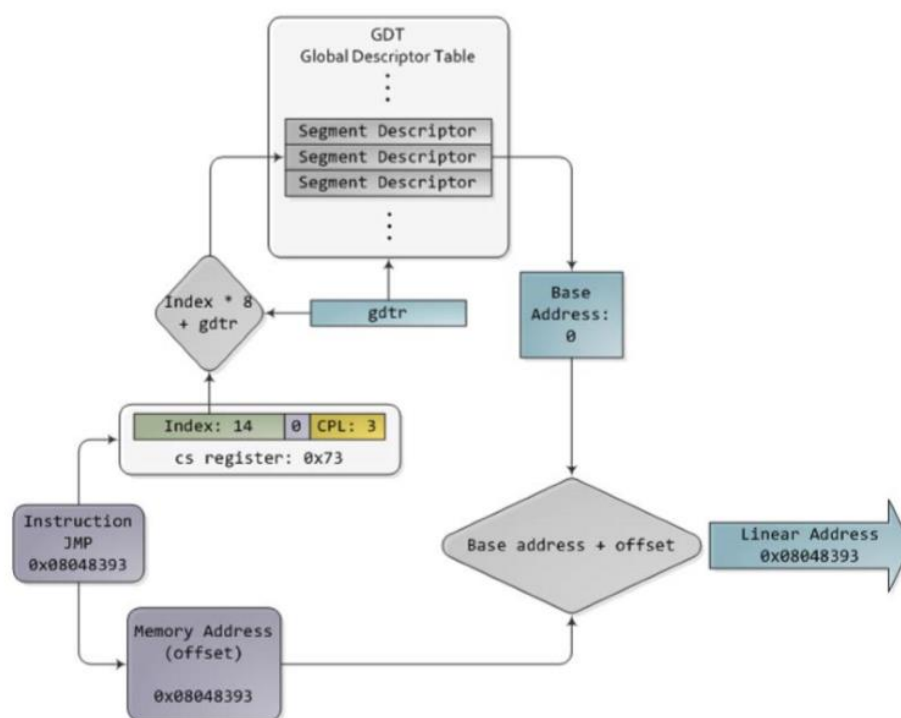


图 7.4[转] 保护模式下分段机制

当 CPU 位于 32 位模式时，内存 4GB，寄存器和指令都可以寻址整个线性地址空间，所以这时候不再需要使用基地址，将基地址设置为 0，此时逻辑地址=描述符=线性地址，Intel 的文档中将其称为扁平模型（flat model），现代的 x86 系统内核使用的是基本扁平模型，等价于转换地址时关闭了分段功能。在 CPU 64 位模式中强制使用扁平的线性空间。逻辑地址与线性地址就合二为一了。所以分段机制也就成为时代的眼泪了

7.3 Hello 的线性地址到物理地址的变换-页式管理

线性地址（书里的虚拟地址 VA）到物理地址（PA）之间的转换通过分页机制完成。而分页机制是对虚拟地址内存空间进行分页。

首先 Linux 系统有自己的虚拟内存系统，其虚拟内存组织形式如图 7.5，Linux 将虚拟内存组织成一些段的集合，段之外的虚拟内存不存在因此不需要记录。内核为 hello 进程维护一个段的任务结构即图中的 task_struct，其中条目 mm 指向一个 mm_struct，它描述了虚拟内存的当前状态，pgd 指向第一级页表的基地址（结合一个进程一串页表），mmap 指向一个 vm_area_struct 的链表，一个链表条目对应一个段，所以链表相连指出了 hello 进程虚拟内存中的所有段。

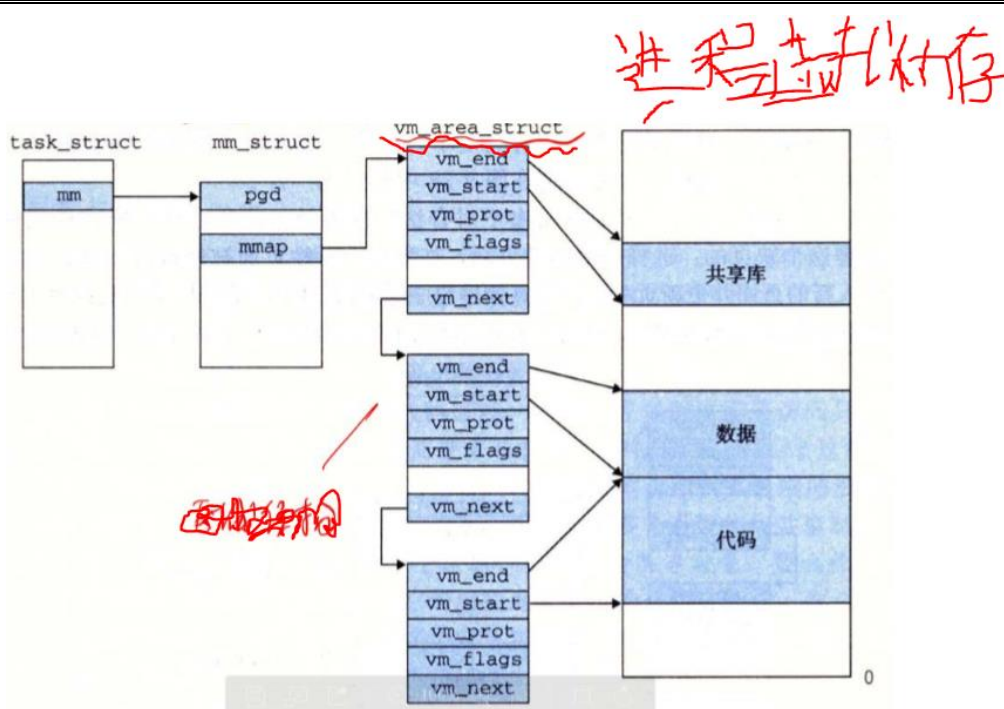


图 7.5 Linux 是如何组织虚拟内存的

系统将每个段分割为被称为虚拟页（VP）的大小固定的块来作为进行数据传输的单元，在 linux 下每个虚拟页大小为 4KB，类似地，物理内存也被分割为物理页（PP/页帧），虚拟内存系统中 MMU 负责地址翻译，MMU 使用存放在物理内存中的被称为页表的数据结构将虚拟页到物理页的映射，即虚拟地址到物理地址的映射。

如图 7.6，不考虑 TLB 与多级页表（在 7.4 节中包含这两者的综合考虑），虚拟地址分为虚拟页号 VPN 和虚拟页偏移量 VPO，根据位数限制分析（可以在 7.4 节中看到分析过程）可以确定 VPN 和 VPO 分别占多少位是多少。通过页表基址寄存器 PTBR+VPN 在页表中获得条目 PTE，一条 PTE 中包含有效位、权限信息、物理页号，如果有效位是 0+NULL 则代表没有在虚拟内存空间中分配该内存，如果是有效位 0+非 NULL，则代表在虚拟内存空间中分配了但是没有缓存到物理内存中，如果有效位是 1 则代表该内存已经缓存在了物理内存中，可以得到其物理页号 PPN，与虚拟页偏移量共同构成物理地址 PA。

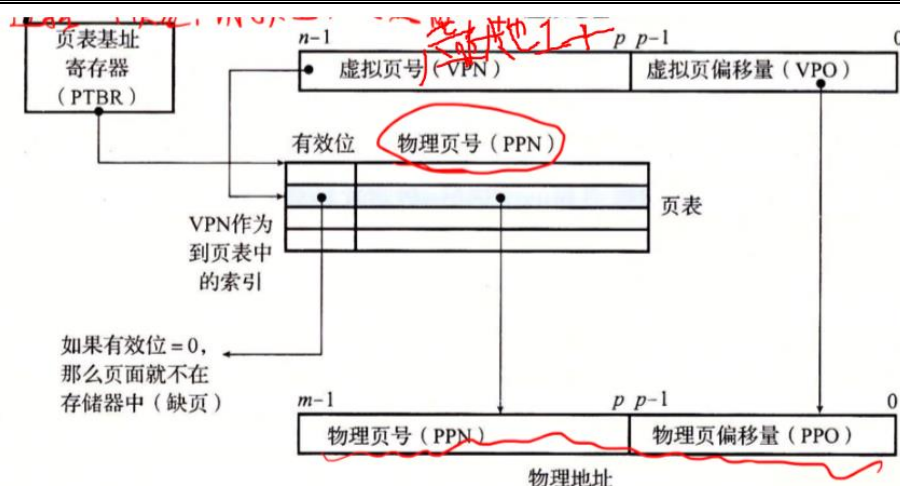


图 7.6 地址翻译

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

在 Intel Core i7 环境下研究 VA 到 PA 的地址翻译问题。前提如下： 虚拟地址空间 48 位，物理地址空间 52 位，页表大小 4KB，4 级页表。TLB 4 路 16 组相联。CR3 指向第一级页表的起始位置（上下文一部分）。 解析前提条件： 由一个页表大小 4KB，一个 PTE 条目 8B，共 512 个条目，使用 9 位二进制索引，一共 4 个页表共使用 36 位二进制索引，所以 VPN 共 36 位，因为 VA 48 位，所以 VPO 12 位；因为 TLB 共 16 组，所以 TLBI 需 4 位，因为 VPN 36 位，所以 TLBT 32 位。

如图 7.7，CPU 产生虚拟地址 VA，VA 传送给 MMU，MMU 使用前 36 位 VPN 作为 TLBT（前 32 位）+TLBI（后 4 位）向 TLB 中匹配，如果命中，则得到 PPN（40bit）与 VPO（12bit）组合成 PA（52bit）。 如果 TLB 中没有命中，MMU 向页表中查询，CR3 确定第一级页表的起始地址，VPN1（9bit）确定在第一级页表中的偏移量，查询出 PTE，如果在物理内存中且权限符合，确定第二级页表的起始地址，以此类推，最终在第四级页表中查询到 PPN，与 VPO 组合成 PA，并且向 TLB 中添加条目。 如果查询 PTE 的时候发现不在物理内存中，则引发缺页故障。如果发现权限不够，则引发段错误。

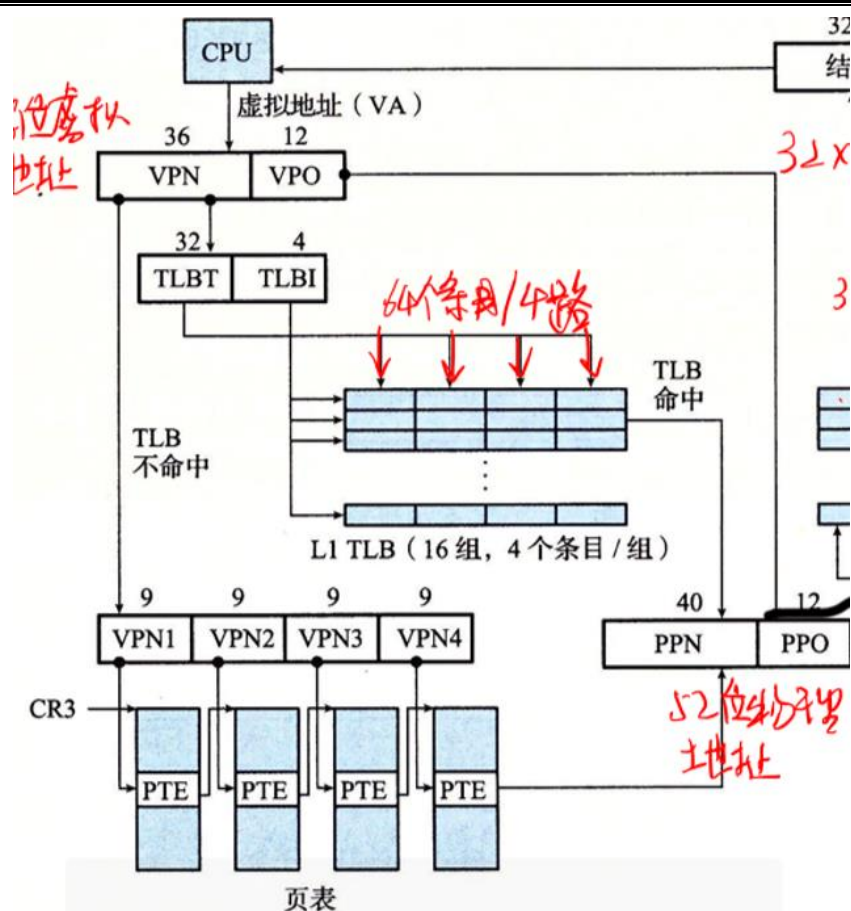


图 7.7 TLB 与 4 级页表下 Core i7 的地址翻译情况

7.5 三级 Cache 支持下的物理内存访问

前提：只讨论 L1 Cache 的寻址细节，L2 与 L3Cache 原理相同。L1 Cache 是 8 路 64 组相联。块大小为 64B。解析前提条件：因为共 64 组，所以需要 6bit CI 进行组寻址，因为共有 8 路，因为块大小为 64B 所以需要 6bit CO 表示数据偏移位置，因为 VA 共 52bit，所以 CT 共 40bit。

在上一步中我们已经获得了物理地址 VA，如图 7.8，使用 CI（后六位再后六位）进行组索引，每组 8 路，对 8 路的块分别匹配 CT（前 40 位）如果匹配成功且块的 valid 标志位为 1，则命中（hit），根据数据偏移量 CO（后六位）取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是 1，则不命中（miss），向下一级缓存中查询数据（L2 Cache->L3 Cache->主存）。查询到数据之后，一种简单的放置策略如下：如果映射到的组内有空闲块，则直接放置，否则组内都是有效块，产生冲突（evict），则采用最近最少使用策略 LFU 进行替换。

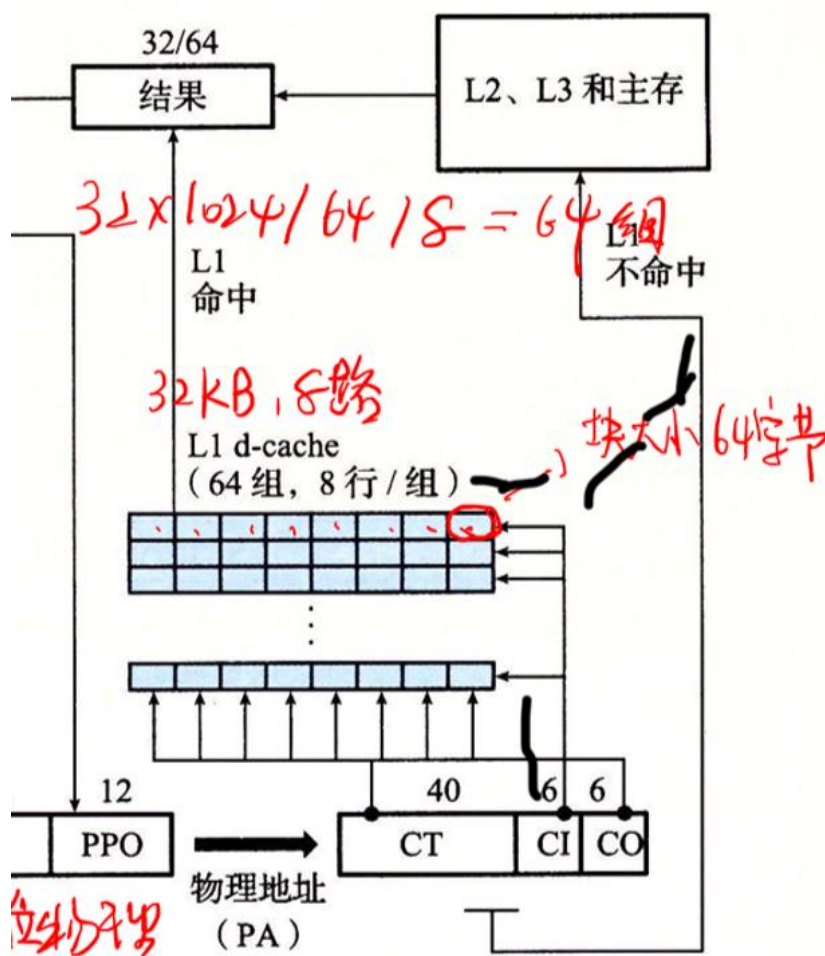


图 7.8 物理内存的访问

7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 进程调用时，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，为了给这个新进程创建虚拟内存，它创建了当前进程的 mm_struct、区域结构和页表的原样副本。它将这两个进程的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

7.7 hello 进程 execve 时的内存映射

execve 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序。加载并运行 hello 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。代码和数据区域被映射为 `hello` 文件中的 `.text` 和 `.data` 区, `bss` 区域是请求二进制零的, 映射到匿名文件, 其大小包含在 `hello` 中, 栈和堆地址也是请求二进制零的, 初始长度为零。

3) 映射共享区域, `hello` 程序与共享对象 `libc.so` 链接, `libc.so` 是动态链接到这个程序中的, 然后再映射到用户虚拟地址空间中的共享区域内。

4) 设置程序计数器 (PC), `execve` 做的最后一件事情就是设置当前进程上下文的程序计数器, 使之指向代码区域的入口点。

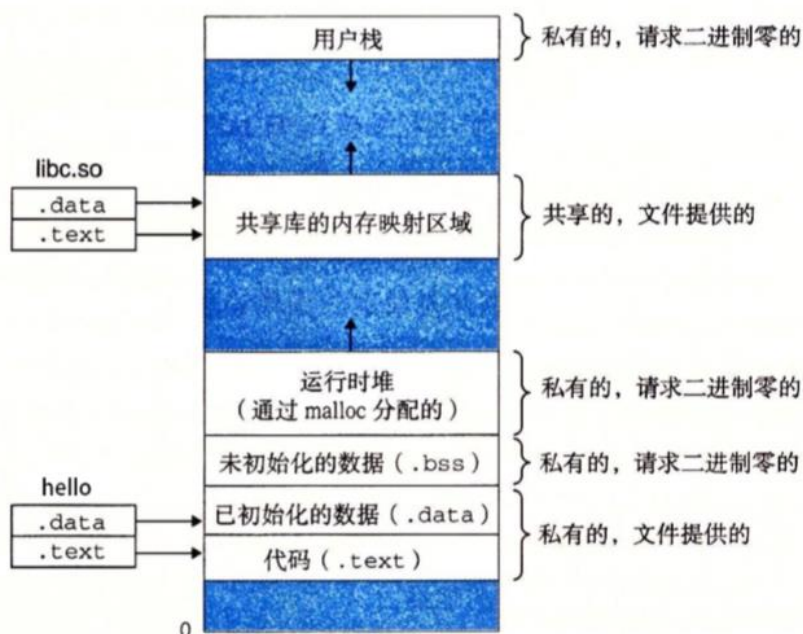


图 7.9 加载器是如何映射用户地址空间区域的

7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障, 当指令引用一个虚拟地址, 在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中, 因此必须从磁盘中取出的时候就会发生故障。其处理流程遵循图 7.10 所示的故障处理流程。

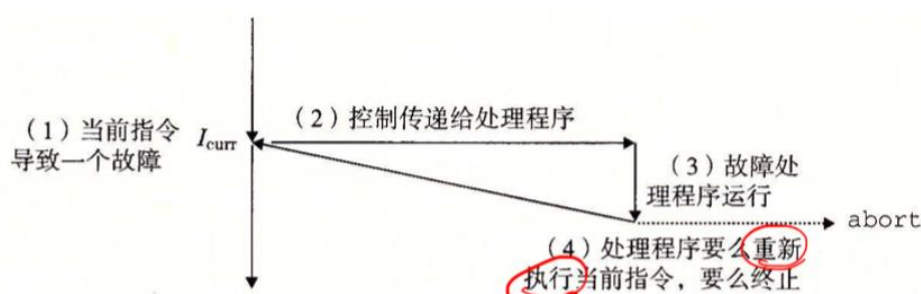


图 7.10 故障处理流程

缺页中断处理：缺页处理程序是系统内核中的代码，选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令再次发送 VA 到 MMU，这次 MMU 就能正常翻译 VA 了。

7.9 动态存储分配管理

printf 函数会调用 malloc，下面简述动态内存管理的基本方法与策略：动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用 来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

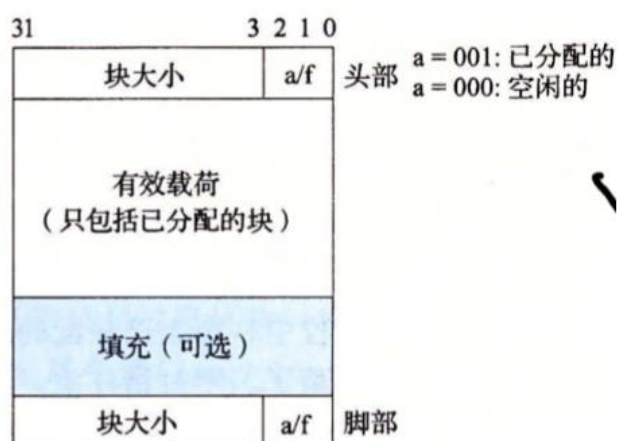
分配器分为两种基本风格：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放任何已分配的块。

隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

一、带边界标签的隐式空闲链表

1) 堆及堆中内存块的组织结构



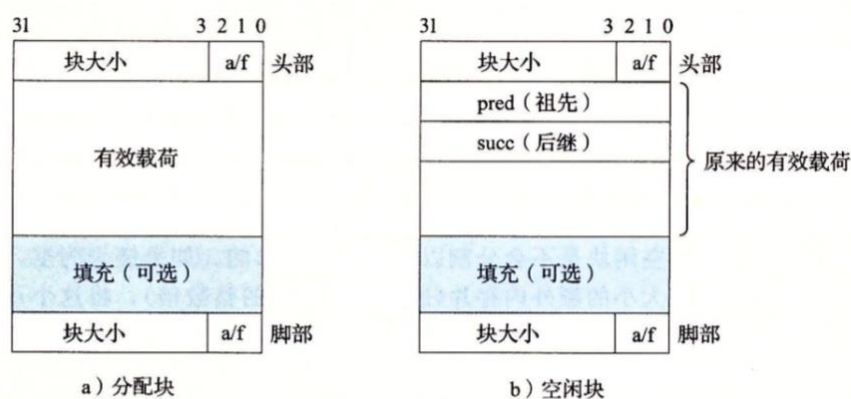
在内存块中增加 4B 的 Header 和 4B 的 Footer，其中 Header 用于寻找下一个 block，Footer 用于寻找上一个 block。Footer 的设计是专门为了合并空闲块方便的。因为 Header 和 Footer 大小已知，所以我们利用 Header 和 Footer 中存

放的块大小就可以寻找上下 block。

2) 隐式链表 所谓隐式空闲链表，对比于显式空闲链表，代表并不直接对空闲块进行链接，而是将对内存空间中的所有块组织成一个大链表，其中 Header 和 Footer 中的 block 大小间接起到了前驱、后继指针的作用。

3) 空闲块合并 因为有了 Footer，所以我们可以方便的对前面的空闲块进行合并。合并的情况一共分为四种：前空后不空，前不空后空，前后都空，前后都不空。对于四种情况分别进行空闲块合并，我们只需要通过改变 Header 和 Footer 中的值就可以完成这一操作。

二、显示空间链表基本原理 将空闲块组织成链表形式的数据结构。堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针，如下图：



使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。

维护链表的顺序有：后进先出（LIFO），将新释放的块放置在链表的开始处，使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块，在这种情况下，释放一个块可以在线性的时间内完成，如果使用了边界标记，那么合并也可以在常数时间内完成。按照地址顺序来维护链表，其中链表中的每个块的地址都小于它的后继的地址，在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序首次适配比 LIFO 排序的首次适配有着更高的内存利用率，接近最佳适配的利用率。

7.10 本章小结

Linux 可不是一个想呆在哪就呆在哪的地方，这个大都市是如此有序，公共活动场所又是如此宝贵，只有当真正要进行活动的时候，hello 才能向 MMU 递交统一格式的虚拟地址来获得自己真正该玩耍的地方，活动时如果需要临时存放个程序物品，还需要特别地调用 malloc 申请堆空间。通过网上冲浪 hello 还了解到一个叫做段式管理的都市传说，也是个令程序摸不到头脑的东西 555。虽然繁琐，但 Linux 可真使程序感到安心呀……

本章主要介绍了 hello 的存储器地址空间、intel 的段式管理、hello 的页式管理，以 intel Core7 在指定环境下介绍了 VA 到 PA 的变换、物理内存访问，还介绍了 hello 进程 fork 时的内存映射、execve 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理

（第 7 章 2 分）

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：所有的 IO 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，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

8.2 简述 Unix IO 接口及其函数

Unix I/O 接口统一操作：

1) 打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备，内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息。

2) Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。

3) 改变当前的文件位置：对于每个打开的文件，内核保持着一个文件位置 k ，初始为 0，这个文件位置是从文件开头起始的字节偏移量，应用程序能够通过执行 `seek`，显式地将改变当前文件位置 k 。

4) 读写文件：一个读操作就是从文件复制 $n>0$ 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 $k+n$ ，给定一个大小为 m 字节的而文件，当 $k \geq m$ 时，触发 EOF。类似一个写操作就是从内存中复制 $n>0$ 个字节到一个文件，从当前文件位置 k 开始，然后更新 k 。

5) 关闭文件，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中去。

Unix I/O 函数：

1) `int open(char* filename,int flags,mode_t mode)`，进程通过调用 `open` 函数来打开一个存在的文件或是创建一个新文件的。`open` 函数将 `filename` 转换为一个文件描述符，并且返回描述符数字，返回的描述符总是在进程中当前没有打开的最小描述符，`flags` 参数指明了进程打算如何访问这个文件，`mode` 参数指定了新文件的访问权限位。

2) `int close(fd)`，`fd` 是需要关闭的文件的描述符，`close` 返回操作结果。

3) `ssize_t read(int fd,void *buf,size_t n)`，`read` 函数从描述符为 `fd` 的当前文

件位置赋值最多 n 个字节到内存位置 `buf`。返回值-1 表示一个错误，0 表示 EOF，否则返回值表示的是实际传送的字节数量。

4) `ssize_t write(int fd, const void *buf, size_t n)`, `write` 函数从内存位置 `buf` 复制至多 n 个字节到描述符为 `fd` 的当前文件位置。

8.3 printf 的实现分析

前提: `printf` 和 `vsprintf` 代码是 windows 下的。

查看 `printf` 代码:

```
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;
}
```

首先 `arg` 获得第二个不定长参数，即输出的时候格式化串对应的值。

查看 `vsprintf` 代码:

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

    for (p = buf; *fmt; fmt++)
    {
        if (*fmt != '%') //忽略无关字符
        {
            *p++ = *fmt;
            continue;
        }

        fmt++;

        switch (*fmt)
        {
            case 'x': //只处理%x一种情况
                itoa(tmp, *((int*)p_next_arg)); //将输入参数值转化为
```


字符串保存在tmp

```

        strcpy(p, tmp); //将tmp字符串复制到p处
        p_next_arg += 4; //下一个参数值地址
        p += strlen(tmp); //放下一个参数值的地址
        break;
    case 's':
        break;
    default:
        break;
}
}

return (p - buf); //返回最后生成的字符串的长度
}

```

则知道 vsprintf 程序按照格式 fmt 结合参数 args 生成格式化之后的字符串，并返回字符串的长度。

在 printf 中调用系统函数 write(buf,i)将长度为 i 的 buf 输出。write 函数如下：

```

write:
    mov eax, _NR_write
    mov ebx, [esp + 4]
    mov ecx, [esp + 8]
    int INT_VECTOR_SYS_CALL

```

在 write 函数中，将栈中参数放入寄存器，ecx 是字符个数，ebx 存放第一个字符地址，int INT_VECTOR_SYS_CALL 代表通过系统调用 syscall，查看 syscall 的实现：

```

sys_call:
    call save

    push dword [p_proc_ready]

    sti

    push ecx
    push ebx
    call [sys_call_table + eax * 4]
    add esp, 4 * 3

    mov [esi + EAXREG - P_STACKBASE], eax

    cli

    ret

```

syscall 将字符串中的字节“Hello 1170300825 lidaxin”从寄存器中通过总线复

制到显卡的显存中，显存中存储的是字符的 ASCII 码。

字符显示驱动子程序将通过 ASCII 码在字模库中找到点阵信息将点阵信息存储到 vram 中。

显示芯片会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

于是我们的打印字符串“Hello 1170300825 lidaxin”就显示在了屏幕上。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：当用户按键时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码转换成 ASCII 码，保存到系统的键盘缓冲区之中。getchar 函数落实到底层调用了系统函数 read，通过系统调用 read 读取存储在键盘缓冲区中的 ASCII 码直到读到回车符然后返回整个字符串，getchar 进行封装，大体逻辑是读取字符串的第一个字符然后返回。

8.5 本章小结

对于 printf 和 getchar，hello 以前只知道调用之后一个能打印字符，一个能读入字符，可究竟为啥，不知道，学习完 Linux 都市的 IO 管理手册之后，hello 多少明白了其中奥妙，原来他们都是 Unix I/O 的封装，而真正调用的是 write 和 read 这样的系统调用函数，而它们又都是由内核完成的，之所以键盘能输入是因为引发了异步异常，之所以屏幕上会有显示是因为字符串被复制到了屏幕赖以显示的显存当中，至于其中细节，也值得好好研究一番……

本章主要介绍了 Linux 的 IO 设备管理方法、Unix IO 接口及其函数，分析了 printf 函数和 getchar 函数。

（第 8 章 1 分）

结论

hello 程序终于完成了它艰辛的一生。hello 的一生大事记如下：

1) 编写，通过 editor 将代码键入 hello.c

- 2) 预处理, 将 `hello.c` 调用的所有外部的库展开合并到一个 `hello.i` 文件中
- 3) 编译, 将 `hello.i` 编译成为汇编文件 `hello.s`
- 4) 汇编, 将 `hello.s` 会变成成为可重定位目标文件 `hello.o`
- 5) 链接, 将 `hello.o` 与可重定位目标文件和动态链接库链接成为可执行目标程序 `hello`
- 6) 运行: 在 `shell` 中输入 `./hello 1170300825 lidaxin`
- 7) 创建子进程: `shell` 进程调用 `fork` 为其创建子进程
- 8) 运行程序: `shell` 调用 `execve`, `execve` 调用启动加载器, 加映射虚拟内存, 进入程序入口后程序开始载入物理内存, 然后进入 `main` 函数。
- 9) 执行指令: `CPU` 为其分配时间片, 在一个时间片中, `hello` 享有 `CPU` 资源, 顺序执行自己的控制逻辑流
- 10) 访问内存: `MMU` 将程序中使用的虚拟内存地址通过页表映射成物理地址。
- 11) 动态申请内存: `printf` 会调用 `malloc` 向动态内存分配器申请堆中的内存。
- 12) 信号: 如果运行途中键入 `ctr-c ctr-z` 则调用 `shell` 的信号处理函数分别停止、挂起。
- 13) 结束: `shell` 父进程回收子进程, 内核删除为这个进程创建的所有数据结构。

这一天, 世界上终于响起那首关于 `Hello` 的歌曲 “只有 `CS` 知道……我曾经…… 来………过……”, 但听到这首歌的绝非仅有 `CS`……
还有我啊 啊 啊 啊 啊……

附件

文件名称	文件作用
hello.i	预处理之后文本文件
hello.s	编译之后的汇编文件
hello.o	汇编之后的可重定位目标执
hello	链接之后的可执行目标文件
hello2.c	测试程序代码
hello2	测试程序
helloo.objdmp	Hello.o 的反汇编代码
helloo.elf	Hello.o 的 ELF 格式
hello.objdmp	Hello 的反汇编代码
hello.elf	Hellode ELF 格式
hmp.txt	存放临时数据

参考文献

[1] ELF 构造:

<https://www.cs.stevens.edu/~jschauma/631/elf.html>

[1] 16 进制计算器:

<http://www.99cankao.com/digital-computation/hex-calculator.php>

[2] Linux 下进程的睡眠唤醒:

<https://blog.csdn.net/shengin/article/details/21530337>

[3]进程的睡眠、挂起和阻塞:

<https://www.zhihu.com/question/42962803>

[4] 虚拟地址、 逻辑地址、线性地址、物理地址:

https://blog.csdn.net/rabbit_in_android/article/details/49976101

[5] printf 函数实现的深入剖析:

https://blog.csdn.net/zhengqijun_/article/details/72454714

[6] 内存地址转换与分段

<https://blog.csdn.net/drshenlei/article/details/4261909>