

哈爾濱工業大學

計算機系統

大作業

題	目	<u>程序人生-Hello's P2P</u>
專	業	<u>計算機科學與技術</u>
學	號	<u>1170300224</u>
班	級	<u>1703002</u>
學	生	<u>施子騰</u>
指	導	教
師		<u>史先俊</u>

計算機科學與技術學院
2018 年 12 月

摘 要

本文通过研究示例程序 hello 从代码编写到加载执行的全过程，讨论计算机系统的主要特性与机制。结合《深入理解计算机系统》与哈工大计算机系统课程，并借助 hello 实例总结对计算机系统的学习收获。

关键词：CSAPP; 哈工大；计算机系统；hello 程序

（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	- 6 -
2.4 本章小结	- 9 -
第 3 章 编译	- 10 -
3.1 编译的概念与作用	- 10 -
3.2 在 UBUNTU 下编译的命令	- 10 -
3.3 HELLO 的编译结果解析	- 10 -
3.4 本章小结	- 17 -
第 4 章 汇编	- 18 -
4.1 汇编的概念与作用	- 18 -
4.2 在 UBUNTU 下汇编的命令	- 18 -
4.3 可重定位目标 ELF 格式	- 18 -
4.4 HELLO.O 的结果解析	- 21 -
4.5 本章小结	- 24 -
第 5 章 链接	- 25 -
5.1 链接的概念与作用	- 25 -
5.2 在 UBUNTU 下链接的命令	- 25 -
5.3 可执行目标文件 HELLO 的格式	- 25 -
5.4 HELLO 的虚拟地址空间	- 29 -
5.5 链接的重定位过程分析	- 31 -
5.6 HELLO 的执行流程	- 36 -
5.7 HELLO 的动态链接分析	- 36 -
5.8 本章小结	- 38 -
第 6 章 HELLO 进程管理	- 40 -
6.1 进程的概念与作用	- 40 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 40 -
6.3 HELLO 的 FORK 进程创建过程	- 40 -
6.4 HELLO 的 EXECVE 过程	- 40 -
6.5 HELLO 的进程执行	- 41 -
6.6 HELLO 的异常与信号处理	- 43 -
6.7 本章小结	- 45 -
第 7 章 HELLO 的存储管理.....	- 46 -
7.1 HELLO 的存储器地址空间	- 46 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 46 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 49 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	- 50 -
7.5 三级 CACHE 支持下的物理内存访问	- 51 -
7.6 HELLO 进程 FORK 时的内存映射	- 52 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 52 -
7.8 缺页故障与缺页中断处理	- 53 -
7.9 动态存储分配管理	- 54 -
7.10 本章小结	- 56 -
第 8 章 HELLO 的 IO 管理	- 57 -
8.1 LINUX 的 IO 设备管理方法	- 57 -
8.2 简述 UNIX IO 接口及其函数	- 57 -
8.3 PRINTF 的实现分析	- 58 -
8.4 GETCHAR 的实现分析	- 60 -
8.5 本章小结	- 60 -
结论	- 61 -
附件	- 62 -
参考文献	- 63 -

第 1 章 概述

1.1 Hello 简介

在文本编辑器中，键入 `hello` 程序文本得到源代码 `hello.c`。在 `linux` 下通过 GNU 编译系统中 `cpp` 的预处理，`cl` 的编译，`as` 的汇编，`ld` 的链接得到可执行目标文件 `hello`。

在 `shell` 下键入命令行“`./hello`”，`shell` 为此任务 `fork` 子进程，并 `exeve` 了 `hello` 程序的核心映像，而后将控制转移到 `hello` 程序的 `main` 入口处。`Hello` 在用户态下执行用户命令，陷入内核调用系统代码执行 `I/O` 命令。进程调度程序为 `hello` 分配时间片，控制逻辑控制流；虚拟内存系统为 `hello` 分配独立的地址空间，管理各层次存储系统；内核 `I/O` 模块为 `hello` 提供文件抽象，处理输入输出……程序结束后，`hello` 的父进程 `shell` 为其扫尾，内核代码删除与之相关的数据结果，将之回收。

1.2 环境与工具

硬件环境： Intel® Core(TM) i5-7300HQ CPU @ 2.50GHz
8GB RAM

软件环境： Ubuntu 18.04.1 LT S
Windows 10 家庭中文版 17763.195

开发工具： `gcc`, `vim`, `cpp`, `as`, `ld`, `edb`, `readelf`, `objdump`

1.3 中间结果

<code>Hello.c</code>	源代码文件
<code>Hello.i</code>	预处理后的源代码文件
<code>Hello.s</code>	汇编后的源代码文件
<code>Hello_r.o</code>	可重定位目标文件
<code>Hello</code>	可执行目标文件
<code>Hello_r.elf</code>	<code>hello_r.o</code> 的 <code>readelf</code> 输出文件
<code>Hello.elf</code>	<code>hello.o</code> 的 <code>readelf</code> 输出文件
<code>Hello_r.obj</code>	<code>hello_r.o</code> 的 <code>objdump</code> 输出文件
<code>Hello.obj</code>	<code>hello.o</code> 的 <code>objdump</code> 输出文件

1.4 本章小结

本章介绍了本次大作业的主要内容，列出了实验环境和各中间文件。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

在编译之前进行的处理。

预处理器： C Preprocessor CPP

预处理器处理以 “#” 开头的指令；

主要包括以下三个方面的内容：

1. 宏定义；

可以是宏文本，也可以是带参数的宏函数。

2. 文件包含；

3. 条件编译；

2.2 在 Ubuntu 下预处理的命令

直接调用	<code>cpp hello.c</code>
由 gcc 调用	<code>gcc -c hello.c</code>

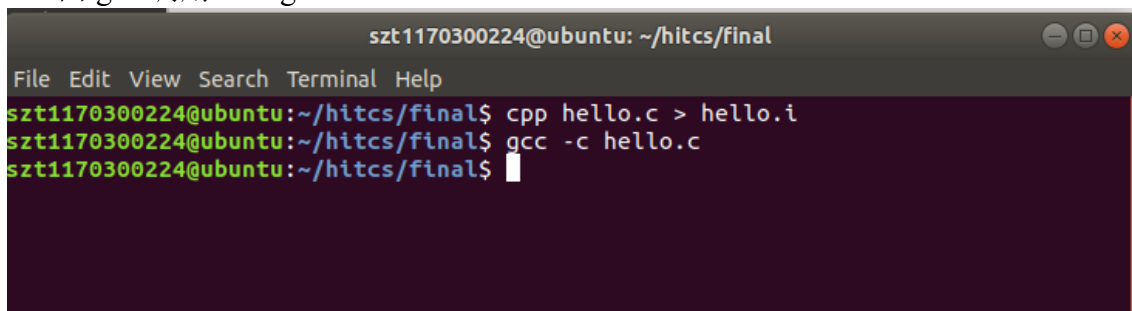


图 2.1 预编译命令

2.3 Hello 的预处理结果解析

源文件 `hello.c` 仅 28 行而预处理后的 `hello.i` 增至 3118 行。

查看 `Hello.i`：

`Hello.i` 头部是一些以 “#” 开头的文件信息：


```

typedef unsigned long int __dev_t;
typedef unsigned int __uid_t;
typedef unsigned int __gid_t;
typedef unsigned long int __ino_t;
typedef unsigned long int __ino64_t;
typedef unsigned int __mode_t;
typedef unsigned long int __nlink_t;
typedef long int __off_t;
typedef long int __off64_t;
typedef int __pid_t;
typedef struct { int __val[2]; } __fsid_t;
typedef long int __clock_t;
typedef unsigned long int __rlim_t;
typedef unsigned long int __rlim64_t;
typedef unsigned int __id_t;
typedef long int __time_t;
typedef unsigned int __useconds_t;
typedef long int __suseconds_t;

typedef int __daddr_t;
typedef int __key_t;

typedef int __clockid_t;

```

图 2.4 hello.i (typedef)

```

typedef struct
{
    int __count;
    union
    {
        unsigned int __wch;
        char __wchb[4];
    } __value;
} __mbstate_t;
# 22 "/usr/include/x86_64-linux-gnu/bits/_G_config.h" 2 3 4

```

图 2.5 hello.i (struct)

```

char* _IO_read_ptr;
char* _IO_read_end;
char* _IO_read_base;
char* _IO_write_base;
char* _IO_write_ptr;
char* _IO_write_end;
char* _IO_buf_base;
char* _IO_buf_end;

```

图 2.6 hello.i (全局变量)

如上所示，有全局变量定义，typedef，结构体定义等内容。

查看源文件 `hello.c` 的内容

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

图 2.7 `hello.c` 头文件包含
可以看到以上三行宏包含指令，
可见新增添的文本是以上库的内容。

2.4 本章小结

预处理执行简单的文本替换，预处理器修改 `hello.c` 得到 `hello.i`。`hello.c` 在预处理之后会增添大量源自于库的内容，文本量激增。

(第 2 章 0.5 分)

第 3 章 编译

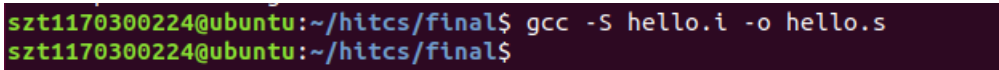
3.1 编译的概念与作用

编译器将预处理后的源程序文本(.i)翻译成汇编语言源程序(.s)。

编译阶段包括词法分析，语法分析。同时也进行代码优化。

3.2 在 Ubuntu 下编译的命令

```
gcc -S hello.i -o hello.s
```



```
szt1170300224@ubuntu:~/hitcs/final$ gcc -S hello.i -o hello.s
szt1170300224@ubuntu:~/hitcs/final$
```

图 3.1 gcc 编译指令

3.3 Hello 的编译结果解析

3.3.1 数据类型常量

代码中的幻数

```
if(argc!=3)
```

图 3.2 源代码

常量 3 被直接编码到代码段中。

```
cmpl $3, -20(%rbp)
```

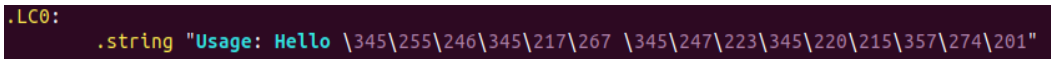
图 3.3 汇编代码

3.3.2. 字符串常量

Printf()的格式化串，例如：

```
printf("Usage: Hello 学号 姓名!\n");
```

图 3.4 源代码



```
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
```

图 3.5 汇编代码

两个格式化串：

```

.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"

```

图 3.6 格式化串汇编代码

可以看到，格式化串被编码到.rodata 段

3.3.3 全局变量

```
int sleepsecs=2.5;
```

图 3.7 源代码

```

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

```

图 3.8 汇编代码

全局变量 sleepsecs 在.text 段中被声明为 全局变量。

定义在.data 段，按 4 字节对齐，大小 4 字节，截断后，被赋值为 2。(可见 long 与 int 在 gcc 64 位下有相同大小，编码位 long 应为编译器偏好)

3.3.4 局部变量

```
int i;
for(i=0;i<10;i++)
{

```

图 3.9 源代码

```

addl    $1, -4(%rbp)
cmpl    $9, -4(%rbp)

```

图 3.10 汇编代码

可见，编译器将 i 存储在了栈上，-4 (%rbp) 处。

3.3.5 整型参数

```

(int argc,
if(argc!=3)

```

图 3.11 源代码

```

movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $3, -20(%rbp)

```

图 3.12 汇编代码

分析如上代码可知，第一个参数 `argc` 被存储在寄存器 `%edi` 内，后又移动到栈上 `-20 (%rbp)` 处。

3.3.6 数组

源代码：

```

char *argv[])

printf("Hello %s %s\n",argv[1],argv[2]);

```

图 3.13 源代码

这是作为第二个参数出现的，元素为 `char *` 的数组。

`Argv` 数组的值应该由调用者准备好，放置在栈上。元素大小为 1byte, 8bit
以下代码为 `printf()` 准备参数：

```

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 3.14 汇编代码

分析代码可知，`argv[1]` 被存储在 `-32 (%rbp) + 16` 处，
`argv[2]` 被存储在 `-32 (%rbp) + 8` 处。

3.3.7 赋值

```

int i;
for(i=0;i<10;i++)

```

图 3.15 源代码

```

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

```

图 3.16 汇编代码

局部变量 `i` 被存储在栈上，赋值由 `mov` 指令实现。

3.3.8 类型转换

```
int sleepsecs=2.5;
```

图 3.17 源代码

```
.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2
```

图 3.18 汇编代码

这里有一个隐式类型转换，`double` 常量 2.5 被截断为 `int` 常量 2；

3.3.9 算术运算

```
for(i=0;i<10;i++)
```

图 3.19 源代码

```
addl    $1, -4(%rbp)
```

图 3.20 汇编代码

这里有一个对局部变量 `i` 的递增操作 `i++`。
编译器用 `add` 指令实现。

3.3.10 关系操作

```
(argc!=3)
```

图 3.21 源代码

```
cmpl    $3, -20(%rbp)
```

图 3.22 汇编代码

编译器用一个 `cmpl` 指令实现 `!=` 操作。

```
for(i=0;i<10;i++)
```

图 3.23 源代码

```
cmpl    $9, -4(%rbp)
jle     .L4
```

图 3.24 汇编代码

编译器将 $i < 10$ 用 `cmpl` 指令配合 `jle` 实现为 $i \leq 9$

3.3.11 控制转移指令

```
if(argc!=3)
```

图 3.25 源代码

```
cmpl    $3, -20(%rbp)
je      .L2
```

图 3.26 汇编代码

编译器用 `cmpl` 指令设置标志寄存器，
再根据标志寄存器，由 `je` 指令实现跳转

```
for(i=0;i<10;i++)
```

图 3.27 源代码

```
addl    $1, -4(%rbp)
.L3:    cmpl    $9, -4(%rbp)
jle     .L4
```

图 3.28 汇编代码

这里的第一行是 `for` 循环的计数器更新指令，
第二行是条件判断指令，
第三行根据 `cmpl` 设置的标志寄存器执行跳转，决定循环是否继续。

3.3.12 函数操作

1)

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

图 3.29 main 原型

这是源代码中定义的唯一函数，也是程序的主函数。

调用者：系统函数 (`__libc_start_main`)

参数传递：调用者将两个参数依次存放到 `%edi,%rsi`

`Main()`函数在使用前将之放到栈上如下位置：

```
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
```

图 3.30 参数传递汇编代码

```
return 0;
```

图 3.31 返回值传递源代码

正常返回的返回值为 0，代码如下：

```
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
```

图 3.32 返回值传递汇编代码

函数栈帧：

帧指针：%rbp 指向栈底

栈指针：%rsp 指向栈顶

分配帧指针：

```
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
```

图 3.33 帧指针分配汇编代码

先保存调用者帧指针，再将当前过程栈作为帧指针。

分配栈帧空间：

```
subq    $32, %rsp
```

图 3.34 栈帧分配汇编代码

在栈上分配了 32 字节的空间

收尾，释放栈空间：

```
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

图 3.35 栈释放汇编代码

Leave 指令将帧指针值移入栈指针，并弹出栈上的调用者帧，以完成扫尾工作。

2) printf

第一个 printf:


```
printf("Usage: Hello 学号 姓名!\n");
```

```
leaq    .LC0(%rip), %rdi
call    puts@PLT
```

图 3.36 printf 代码一

Gcc 将之优化为 puts 函数:

参数放置在%rdi 中传递，是一个全局变量（格式化字符串）的地址。

@PLT 表明这是一个要通过 PLT 间接访问的动态重定位函数

第二个 printf:

```
printf("Hello %s %s\n",argv[1],argv[2]);
```

```
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 3.37 printf 代码二

参数准备:

Leaq 指令载入全局变量（格式化串）的地址到 %rdi

从栈上取出数组 argv 两个元素，存入%rdx,%rsi。

用 call printf@PLT 调用

3) Exit()

源代码:

```
exit(1);
```

汇编代码:

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

图 3.38 exit 代码

4) getchar()

源代码:

```
getchar();
```

汇编代码:

```
call    getchar@PLT
```

图 3.39 getchar 调用

5)sleep()

源代码:

```
sleep(sleepsecs);
```

汇编代码:

```
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep@PLT
```

图 3.40 sleep 调用

3.4 本章小结

编译器 `cpp` 将修改了的源程序文本 `hello.i` 翻译为汇编语言程序文本 `hello.s` 除简单翻译外, 还进行了一些优化工作。编译器只处理当前文本, 只有正在翻译的文件的上下文, 没有统筹兼顾(查看其他文件代码, 例如查看库函数代码)的能力, 对于库函数标记为 `@PLT`, 交由链接阶段处理; 同时也没有预测运行时信息的能力, 不了解数据, 代码的运行时地址, 通过标号, 相对位移访问一些全局信息。

(第3章2分)

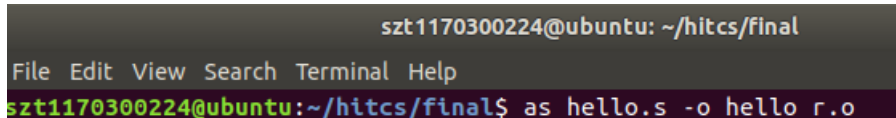
第 4 章 汇编

4.1 汇编的概念与作用

汇编器 `as` 将 汇编语言程序文本（.s） 翻译成 二进制的机器语言代码（.o） - 可重定位目标文件。

4.2 在 Ubuntu 下汇编的命令

指令：`as hello.s`



```
sztt1170300224@ubuntu: ~/hitcs/final
File Edit View Search Terminal Help
sztt1170300224@ubuntu:~/hitcs/final$ as hello.s -o hello_r.o
```

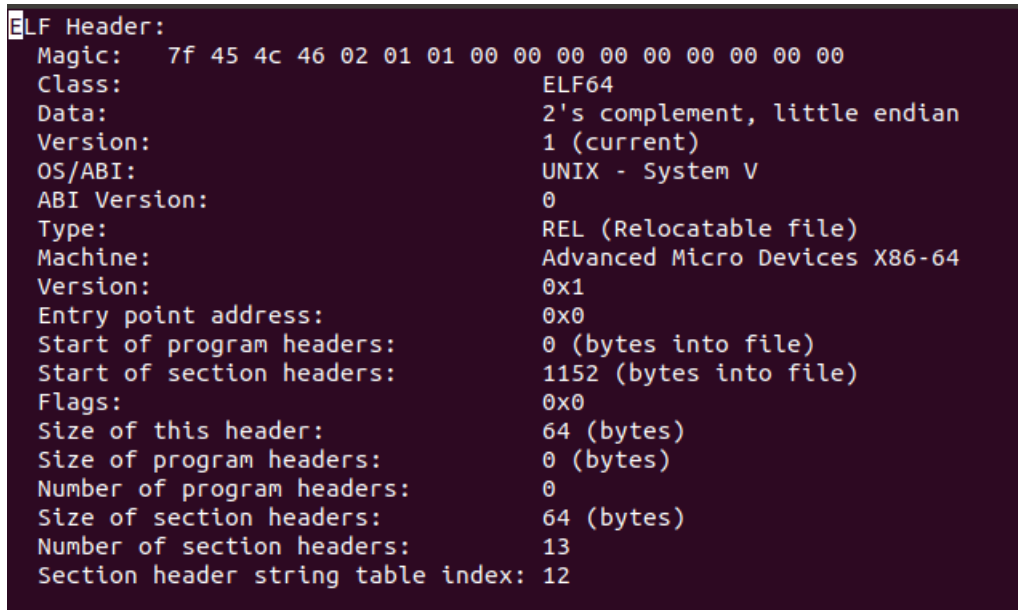
图 4.1 `as` 指令

4.3 可重定位目标 `elf` 格式

```
sztt1170300224@ubuntu:~/hitcs/final$ readelf -a hello_r.o > hello.elf
```

图 4.2 `readelf` 指令

1. Elf 头:



```
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 `elf` 头

以 16byte 的魔数开始，`elf` 头包含了 `elf` 文件的各种信息

如小端法，补码表示，64 位代码，基于 UNIX SystemV 平台 AMDx86-64 处理器等信息。

同时，还有节头部表的偏移量等信息。

2.节头部表:

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		
	0000000000000081	0000000000000000	AX	0	0	
[2]	.rela.text	RELA	0000000000000000	00000340		
	00000000000000c0	0000000000000018	I	10	1	
[3]	.data	PROGBITS	0000000000000000	000000c4		
	0000000000000004	0000000000000000	WA	0	0	
[4]	.bss	NOBITS	0000000000000000	000000c8		
	0000000000000000	0000000000000000	WA	0	0	
[5]	.rodata	PROGBITS	0000000000000000	000000c8		
	000000000000002b	0000000000000000	A	0	0	
[6]	.comment	PROGBITS	0000000000000000	000000f3		
	000000000000002b	0000000000000001	MS	0	0	
[7]	.note.GNU-stack	PROGBITS	0000000000000000	0000011e		
	0000000000000000	0000000000000000	0	0	0	
[8]	.eh_frame	PROGBITS	0000000000000000	00000120		
	0000000000000038	0000000000000000	A	0	0	
[9]	.rela.eh_frame	RELA	0000000000000000	00000400		
	0000000000000018	0000000000000018	I	10	8	
[10]	.symtab	SYMTAB	0000000000000000	00000158		
	00000000000000198	0000000000000018		11	9	
[11]	.strtab	STRTAB	0000000000000000	000002f0		
	000000000000004d	0000000000000000		0	0	
[12]	.shstrtab	STRTAB	0000000000000000	00000418		
	0000000000000061	0000000000000000		0	0	

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

图 4.4elf 节头部表

节头部表列出了 elf 文件各个节的信息，包括节名称在符号表中的偏移，节的类型，节的读写执行权限，以及对齐信息等。

3.代码段重定位节：.rela.text

```
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
```

图 4.5 重定位节

这是一个条目的列表，每个条目对应代码段中一个需要重定位的符号。

当链接器链接本文件和其他文件时，需要修改这些条目。

可以看到，源代码中调用的库函数 `puts`, `exit`, `printf`, `sleep`, `getchar` 都是 `R_X86_64_PLT32` 型的待重定位数据。

全局变量标号 `sleepsecs` 也需要重定位。此外，`printf` 的两个格式化串（字符串常量）也需要 `pc32` 相对寻址重定位。

重定位条目格式：

Offset 字段： 该符号在代码段中的偏移

Info 字段： 前 4 字节：重定位目标在符号表中的偏移量，

后 4 字节：重定位类型

Type 字段： 根据 info 字段的后 4 字节解析得到的重定位类型

Name 字段： info 字段前 4 字节和符号表解析得到的重定位符号名称

Addend 字段： 重定位算法需要的补充信息

4. eh_frame 段重定位节：

```
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 4.6 eh_frame 重定位节

5. 符号表

Symbol table '.symtab' contains 17 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	hello.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
9:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	sleepsecs
10:	0000000000000000	129	FUNC	GLOBAL	DEFAULT	1	main
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_GLOBAL_OFFSET_TABLE_
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	puts
13:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	exit
14:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf
15:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sleep
16:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	getchar

图 4.7 符号表

存放程序定义，引用的全局符号信息，0~8 号条目应当是给编译系统内部使用的条目（elf 节）。

4.4 Hello.o 的结果解析

汇编代码中 main 函数的定义部分与机器代码中 main 首地址以后的部分在反汇编以后非常相似：

```

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

```

图 4.8 汇编代码中 main 的头几行

```

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 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 00    movl    $0x0,-0x4(%rbp)
32: eb 3b                  jmp     6f <main+0x6f>
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

```

图 4.9 反汇编代码中 main 的头几行

可以看到，对大部分指令行，机器代码都是简单的将之翻译为了二进制串。指令的二进制编码长度不等，上图代码中长度跨度从 1 字节到 7 字节。

原汇编代码和反汇编代码的差异：

汇编代码中的标号被替换成了相对偏移。

1. 条件转移指令：

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

图 4.10 汇编代码:

```
13: 74 16          je     2b <main+0x2b>
```

图 4.11 反汇编代码

可以看到, 标号.L2 被编码成了相对下一条指令 PC 值的偏移, $0x15+0x16=0x2b$;

2. 全局变量访问

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

图 4.12 汇编代码

```
15: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 1c <main+0x1c>
                        18: R_X86_64_PC32      .rodata-0x4
```

图 4.13 反汇编代码

汇编代码中通过标号访问的全局变量, 在可重定位目标文件中用占位符 0x0000 暂时替代, 同时, 生成了一个 PC32 相对偏移寻址的到只读数据段的重定位条目。

3. 函数调用:

```
call puts@PLT
```

图 4.14 汇编代码

```
1c: e8 00 00 00 00      callq 21 <main+0x21>
                        1d: R_X86_64_PLT32      puts-0x4
```

图 4.15 反汇编代码

如上图, 汇编代码用函数名表示函数调用, 而在可重定位目标文件中, 库函数的运行时地址, 相对地址都不可知, 只能用占位符 0x0000 占位, 并生成一个 PLT 类型的重定位条目。

此外, 一些编译器生成的伪指令如.cfi (calling frame info) 指令在 -d, -r 的 objdump 反汇编代码中没有出现:

```
.cfi_startproc
pushq    %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq     %rsp, %rbp
.cfi_def_cfa_register 6
subq     $32, %rsp

push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
```

图 4.16 反汇编代码

4.5 本章小结

本章讨论了从汇编代码(.s)到可重定位目标文件(.o)的转换，借助 `readelf` 和 `objdump` 两工具，查看了 `elf` 文件的格式，对比了可重定位文件的反汇编代码和源汇编代码的差异，借此研究了汇编代码到机器代码的映射。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

链接是将多个程序模块组合成一个可执行模块的过程。

链接可分为编译时，加载时，运行时。

编译时链接是由链接器将多个可重定位目标文件链接成一个可执行目标文件的过程。

5.2 在 Ubuntu 下链接的命令

```
ld -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 /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o  
/usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o hello.o -lc  
-z relro -o hello
```

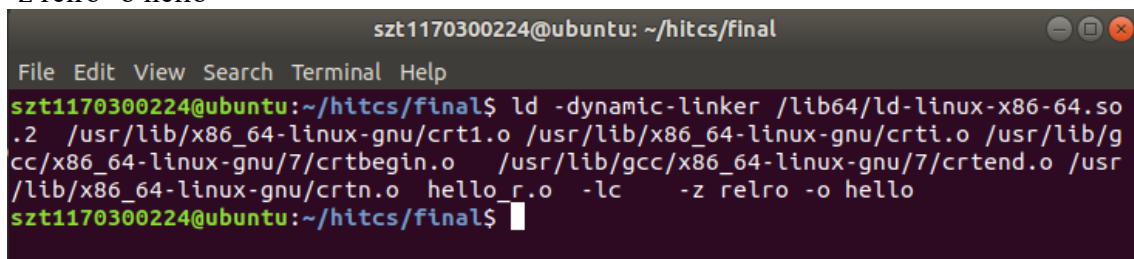


图 5.1 链接指令

5.3 可执行目标文件 hello 的格式

用 `readelf` 察看之：

```
szt1170300224@ubuntu:~/hitcs/final$ readelf -a hello > hello.elf  
szt1170300224@ubuntu:~/hitcs/final$
```

图 5.2 `readelf` 指令

Elf 文件头：

```
ELF Header:
Magic:   7f 45 4c 46 02 01 01 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:                               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:           6512 (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:           28
Section header string table index: 27
```

图 5.3elf 头

Elf 文件头列出了该文件的基本信息

节头目表:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info	Align
[0]	0000000000000000	NULL	0000000000000000	00000000
[1]	.interp	PROGBITS	0000000000400200	00000200
	0000000000000001c	0000000000000000	A 0 0	1
[2]	.note.ABI-tag	NOTE	000000000040021c	0000021c
	00000000000000020	0000000000000000	A 0 0	4
[3]	.hash	HASH	0000000000400240	00000240
	00000000000000034	0000000000000004	A 5 0	8
[4]	.gnu.hash	GNU_HASH	0000000000400278	00000278
	0000000000000001c	0000000000000000	A 5 0	8
[5]	.dynsym	DYNSYM	0000000000400298	00000298
	000000000000000c0	0000000000000018	A 6 1	8
[6]	.dynstr	STRTAB	0000000000400358	00000358
	00000000000000057	0000000000000000	A 0 0	1
[7]	.gnu.version	VERSYM	00000000004003b0	000003b0
	00000000000000010	0000000000000002	A 5 0	2
[8]	.gnu.version_r	VERNEED	00000000004003c0	000003c0
	00000000000000020	0000000000000000	A 6 1	8
[9]	.rela.dyn	RELA	00000000004003e0	000003e0
	00000000000000030	0000000000000018	A 5 0	8
[10]	.rela.plt	RELA	0000000000400410	00000410
	00000000000000078	0000000000000018	AI 5 21	8
[11]	.init	PROGBITS	0000000000400488	00000488
	00000000000000017	0000000000000000	AX 0 0	4
[12]	.plt	PROGBITS	00000000004004a0	000004a0
	00000000000000060	0000000000000010	AX 0 0	16
[13]	.text	PROGBITS	0000000000400500	00000500
	000000000000001e2	0000000000000000	AX 0 0	16
[14]	.fini	PROGBITS	00000000004006e4	000006e4
	00000000000000009	0000000000000000	AX 0 0	4
[15]	.rodata	PROGBITS	00000000004006f0	000006f0
	0000000000000002f	0000000000000000	A 0 0	4
[16]	.eh_frame	PROGBITS	0000000000400720	00000720
	000000000000000fc	0000000000000000	A 0 0	8
[17]	.init array	INIT ARRAY	0000000000600e00	00000e00

[17]	.init_array	INIT_ARRAY	0000000000600e00	00000e00
	0000000000000008	0000000000000008	WA 0 0	8
[18]	.fini_array	FINI_ARRAY	0000000000600e08	00000e08
	0000000000000008	0000000000000008	WA 0 0	8
[19]	.dynamic	DYNAMIC	0000000000600e10	00000e10
	00000000000001e0	0000000000000010	WA 6 0	8
[20]	.got	PROGBITS	0000000000600ff0	00000ff0
	0000000000000010	0000000000000008	WA 0 0	8
[21]	.got.plt	PROGBITS	0000000000601000	00001000
	0000000000000040	0000000000000008	WA 0 0	8
[22]	.data	PROGBITS	0000000000601040	00001040
	0000000000000014	0000000000000000	WA 0 0	8
[23]	.bss	NOBITS	0000000000601054	00001054
	0000000000000004	0000000000000000	WA 0 0	1
[24]	.comment	PROGBITS	0000000000000000	00001054
	000000000000002a	0000000000000001	MS 0 0	1
[25]	.symtab	SYMTAB	0000000000000000	00001080
	0000000000000600	0000000000000018	26 41	8
[26]	.strtab	STRTAB	0000000000000000	00001680
	000000000000020e	0000000000000000	0 0	1
[27]	.shstrtab	STRTAB	0000000000000000	0000188e
	00000000000000e2	0000000000000000	0 0	1

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

图 5.4 节头目表

节头目表列出了 elf 文件中各个节的基本信息。包括节在 elf 文件中的偏移量，节的大小等。

段头部表：

Program Headers:					
Type	Offset	VirtAddr	PhysAddr	Flags	Align
	FileSiz	MemSiz			
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040	R	0x8
	0x00000000000001c0	0x00000000000001c0			
INTERP	0x0000000000000200	0x0000000000400200	0x0000000000400200	R	0x1
	0x000000000000001c	0x000000000000001c			
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000	R E	0x200000
	0x000000000000081c	0x000000000000081c			
LOAD	0x0000000000000e00	0x0000000000600e00	0x0000000000600e00	RW	0x200000
	0x0000000000000254	0x0000000000000258			
DYNAMIC	0x0000000000000e10	0x0000000000600e10	0x0000000000600e10	RW	0x8
	0x00000000000001e0	0x00000000000001e0			
NOTE	0x000000000000021c	0x000000000040021c	0x000000000040021c	R	0x4
	0x0000000000000020	0x0000000000000020			
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000	RW	0x10
	0x0000000000000000	0x0000000000000000			
GNU_RELRO	0x0000000000000e00	0x0000000000600e00	0x0000000000600e00	R	0x1
	0x0000000000000200	0x0000000000000200			

Section to Segment mapping:
Segment Sections...

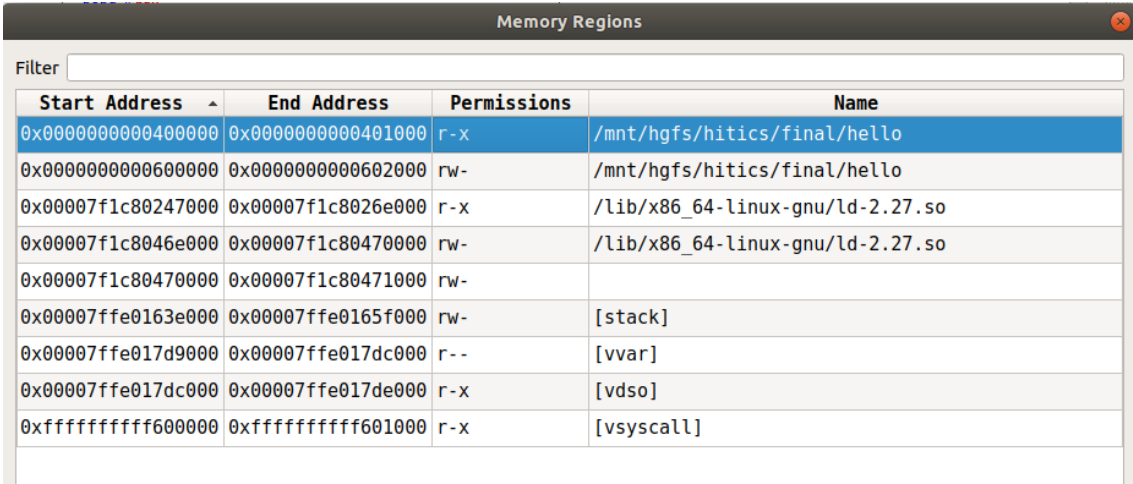
00	
01	.interp
02	.interp.note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame
03	.init_array .fini_array .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag
06	
07	.init_array .fini_array .dynamic .got

图 5.5 段头部表

段头部表描述可执行文件段与内存段的映射。

5.4 hello 的虚拟地址空间

在 edb 下观察程序地址空间情况：

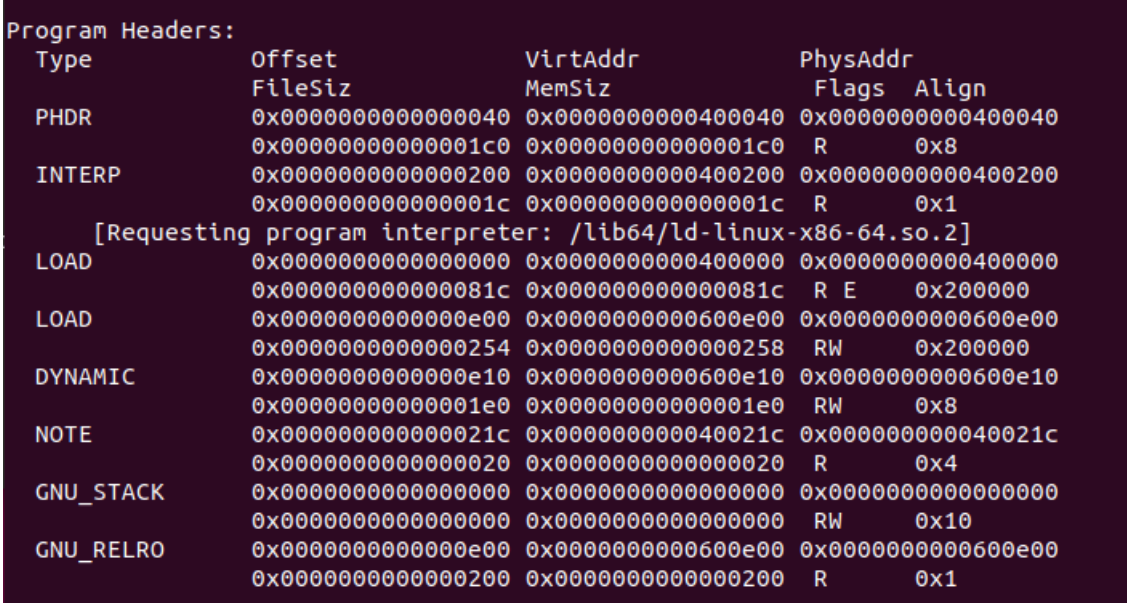


Start Address	End Address	Permissions	Name
0x0000000000400000	0x0000000000401000	r-x	/mnt/hgfs/hitcs/final/hello
0x0000000000600000	0x0000000000602000	rw-	/mnt/hgfs/hitcs/final/hello
0x00007f1c80247000	0x00007f1c8026e000	r-x	/lib/x86_64-linux-gnu/ld-2.27.so
0x00007f1c8046e000	0x00007f1c80470000	rw-	/lib/x86_64-linux-gnu/ld-2.27.so
0x00007f1c80470000	0x00007f1c80471000	rw-	
0x00007ffe0163e000	0x00007ffe0165f000	rw-	[stack]
0x00007ffe017d9000	0x00007ffe017dc000	r--	[vvar]
0x00007ffe017dc000	0x00007ffe017de000	r-x	[vdso]
0xfffffffff6000000	0xfffffffff6010000	r-x	[vsyscall]

图 5.6 地址空间分布

地址空间被分作多个区域，表的前两个字段指出了区域的边界，第三个字段是读写执行权限，最后一个字段是该区域的信息。

对照 elf 文件中的段头部表：



Type	Offset	FileSiz	VirtAddr	MemSiz	PhysAddr	Flags	Align
PHDR	0x0000000000000040	0x00000000000001c0	0x0000000000400040	0x00000000000001c0	0x0000000000400040	R	0x8
INTERP	0x0000000000000200	0x000000000000001c	0x0000000000400200	0x000000000000001c	0x0000000000400200	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x0000000000000000	0x000000000000081c	0x0000000000400000	0x000000000000081c	0x0000000000400000	R E	0x200000
LOAD	0x0000000000000e00	0x0000000000000258	0x0000000000600e00	0x0000000000000258	0x0000000000600e00	RW	0x200000
DYNAMIC	0x0000000000000e10	0x00000000000001e0	0x0000000000600e10	0x00000000000001e0	0x0000000000600e10	RW	0x8
NOTE	0x000000000000021c	0x0000000000000020	0x000000000040021c	0x0000000000000020	0x000000000040021c	R	0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	RW	0x10
GNU_RELRO	0x0000000000000e00	0x0000000000000200	0x0000000000600e00	0x0000000000000200	0x0000000000600e00	R	0x1

图 5.7 段头部表

Program headers 描述了可执行文件段与运行时内存段的映射关系。每个条目 7 个字段，分布描述对应字段在 elf 文件中的偏移，虚拟/物理内存地址，文件中的段大小，内存中应分配的段大小，读写执行权限，内存对齐要求。

64 位的 elf 条目格式：

```
typedef struct {
    Elf64_Word    p_type;
    Elf64_Word    p_flags;
    Elf64_Off     p_offset;
    Elf64_Addr    p_vaddr;
    Elf64_Addr    p_paddr;
    Elf64_Xword   p_filesz;
    Elf64_Xword   p_memsz;
    Elf64_Xword   p_align;
} Elf64_Phdr;
```

图 5.8 elf 格式

条目类型：

PHDR

PT_PHDR

Specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type cannot occur more than once in a file. Moreover, it can occur only if the program header table is part of the memory image of the program. This type, if present, must precede any loadable segment entry. See "[Program Interpreter](#)" for further information.

图 5.9 PHDR 文档

这是描述段头部表本身的条目。

INTERP

PT_INTERP

Specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is mandatory for dynamic executable files and can occur in shared objects. It cannot occur more than once in a file. This type, if present, it must precede any loadable segment entry. See "[Program Interpreter](#)" for further information.

图 5.10 INTERP 文档

这是和解释器（interpreter）相关的条目。

LOAD

PT_LOAD

Specifies a loadable segment, described by `p_filesz` and `p_memsz`. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (`p_memsz`) is larger than the file size (`p_filesz`), the extra bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size can not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the `p_vaddr` member.

图 5.11 LOAD 文档

此条目描述一个可载入内存的段。

DYNAMIC

PT_DYNAMIC

Specifies dynamic linking information. See "[Dynamic Section](#)".

图 5.12 DYNAMIC 文档

描述动态链接信息。

NOTE:

PT_NOTE

Specifies the location and size of auxiliary information. See "[Note Section](#)" for details.

图 5.13 NOTE 文档

此条目和辅助信息描述相关。

GNU_STACK

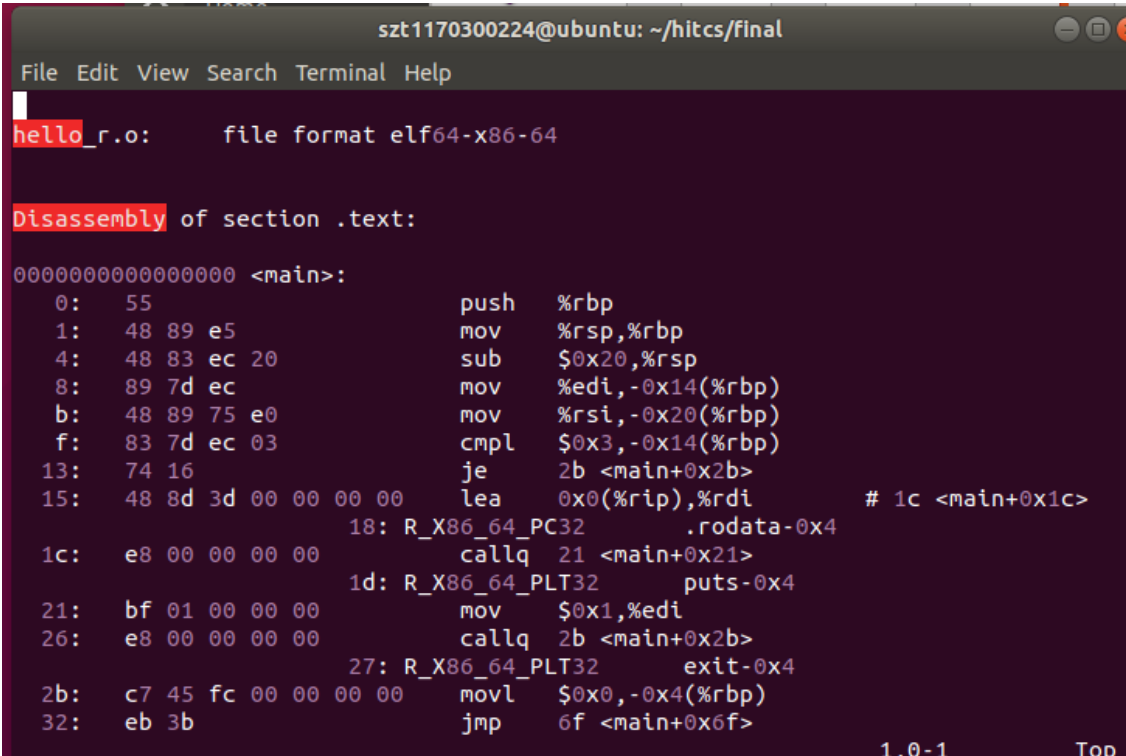
描述栈的信息：读写执行权限，对齐要求。

GNU_RELRO:

描述重定位相关的只读内存区域信息。

5.5 链接的重定位过程分析

以 `objdump -d -r` 选项查看可重定位目标文件 `hello_r.o` 和可执行目标文件 `hello`,



```

sztt1170300224@ubuntu: ~/hits/final
File Edit View Search Terminal Help
hello_r.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 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 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
1,0-1          Top

```

图 5.14 可重定位目标文件（头几行）


```

hello:      file format elf64-x86-64

Disassembly of section .init:

0000000000400488 <_init>:
400488:  48 83 ec 08          sub    $0x8,%rsp
40048c:  48 8b 05 65 0b 20 00  mov    0x200b65(%rip),%rax      # 600f
f8 <__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:

00000000004004a0 <.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>

```

图 5.15 可执行目标文件（头几行）

对比两文件，有如下差异：

1. 地址

可重定位目标文件中，代码段的地址是相对于段起始位置的偏移。

而可执行目标文件中的地址则是实际的运行时地址，

例如：

可执行目标文件：

```
00000000004005e7 <main>:
```

图 5.16 main 地址

Edb 查看的实际运行地址：

```
● 00000000:004005e7 hello!main | 55
```

图 5.17 main 运行时地址

可执行目标文件：

```
00000000004005e0 <frame_dummy>:
```

图 5.18 frame_dummy 地址

Edb 查看的实际运行地址：

```
00000000:004005e0 hello!frame_dummy | 55
```

图 5.19 frame_dummy 运行时地址

2. 节

可重定位目标文件里只有一个.text 节，而可执行目标文件里有多节，且

代码节也增添了多个函数

可执行目标文件:

.init 初始化节

Disassembly of section .init:

图 5.20

含一个初始化函数:

0000000000400488 <_init>:

图 5.21

.plt 过程链接节

Disassembly of section .plt:

图 5.22

亦即 PLT (过程链接表)

每个表项大小都是 16bytes

PLT[0]是跳转到动态链接器的表项:

```
00000000004004a0 <.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)
```

图 5.23

可以看到,PLT 表是配合 GOT 表使用的。

接下来的表项是源代码中调用的库函数

```
00000000004004b0 <puts@plt>:
4004b0: ff 25 62 0b 20 00    jmpq  *0x200b62(%rip)    # 601018 <puts@GLIBC_2.2.5>
4004b6: 68 00 00 00 00 00    pushq $0x0
4004bb: e9 e0 ff ff ff      jmpq  4004a0 <_.plt>

00000000004004c0 <printf@plt>:
4004c0: ff 25 5a 0b 20 00    jmpq  *0x200b5a(%rip)    # 601020 <printf@GLIBC_2.2.5>
4004c6: 68 01 00 00 00 00    pushq $0x1
4004cb: e9 d0 ff ff ff      jmpq  4004a0 <_.plt>

00000000004004d0 <getchar@plt>:
4004d0: ff 25 52 0b 20 00    jmpq  *0x200b52(%rip)    # 601028 <getchar@GLIBC_2.2.5>
4004d6: 68 02 00 00 00 00    pushq $0x2
4004db: e9 c0 ff ff ff      jmpq  4004a0 <_.plt>

00000000004004e0 <exit@plt>:
4004e0: ff 25 4a 0b 20 00    jmpq  *0x200b4a(%rip)    # 601030 <exit@GLIBC_2.2.5>

00000000004004e0 <exit@plt>:
4004e0: ff 25 4a 0b 20 00    jmpq  *0x200b4a(%rip)    # 601030 <exit@GLIBC_2.2.5>
4004e6: 68 03 00 00 00 00    pushq $0x3
4004eb: e9 b0 ff ff ff      jmpq  4004a0 <_.plt>

00000000004004f0 <sleep@plt>:
4004f0: ff 25 42 0b 20 00    jmpq  *0x200b42(%rip)    # 601038 <sleep@GLIBC_2.2.5>
4004f6: 68 04 00 00 00 00    pushq $0x4
4004fb: e9 a0 ff ff ff      jmpq  4004a0 <_.plt>
```

图 5.24 PLT 表

.text 代码节

Disassembly of section .text:

图 5.25

函数:

```
0000000000400500 <_start>:
-----
0000000000400530 <_dl_relocate_static_pie>:
-----
0000000000400540 <deregister_tm_clones>:
0000000000400570 <register_tm_clones>:
00000000004005b0 <__do_global_ctors_aux>:
00000000004005e0 <frame_dummy>:
00000000004005e7 <main>:
0000000000400670 <__libc_csu_init>:
-----
00000000004006e0 <__libc_csu_fini>:
```

图 5.26

.fini 收尾节

Disassembly of section .fini:

图 5.27

函数:

```
00000000004006e4 <_fini>:
```

图 5.28

由此可见，除程序员直接编辑的 main 函数外，还有大量的系统函数做了许多预备，收尾的工作。

当前，还有一些“例行公事”，打打酱油的函数：

```
00000000004006e4 <_fini>:
4006e4: 48 83 ec 08      sub    $0x8,%rsp
4006e8: 48 83 c4 08      add    $0x8,%rsp
4007ec: c3              retq
```

图 5.29 酱油君

23333

3. 重定位符号

可重定位目标文件中的一些符号（全局变量，库函数调用等）被设置位占位符 0x0000，而链接后的可执行目标文件修正了这些符号。

3.1 函数调用:

动态链接的库函数在.plt 节中, 通过 PLT 表访问:

如 puts():

```
0000000004004b0 <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>
```

图 5.30

Main()中调用 Puts 的代码:

```
4005fc: 48 8d 3d f1 00 00 00    lea     0xf1(%rip),%rdi    # 4006f4 <_IO_stdin_used+0x4>
400603: e8 a8 fe ff ff      callq   4004b0 <puts@plt>
```

图 5.31

可以看到, main()中并没有直接跳转到 puts 的实际地址(因为做不到, 动态链接的过程地址要到加接后才能确定), 而是跳转到了 PLT 表的第二项 PLT[1](注意 call 指令的操作数是相对寻址), 再间接跳转到 puts()函数。

3.2 全局变量引用

字符串常量引用:

```
4005fc: 48 8d 3d f1 00 00 00    lea     0xf1(%rip),%rdi    # 4006f4 <_IO_stdin_used+0x4>
400634: 48 8d 3d d7 00 00 00    lea     0xd7(%rip),%rdi    # 400712 <_IO_stdin_used+0x22>
```

图 5.32 printf 的两个格式化串

如上图, 这两个符号是通过相对 PC 的偏移访问的, 因.rodata 段相对于.text 段有确定的偏移, 故可如此处理。

全局整型变量引用:

源代码:

```
int sleepsecs=2.5;
sleep(sleepsecs);
```

图 5.33

可执行代码:

```
400645: 8b 05 05 0a 20 00      mov     0x200a05(%rip),%eax    # 601050 <sleepsecs>
```

图 5.34

也是通过 PC 相对偏移访问的。

5.6 hello 的执行流程

edb 下观察的程序主要控制流如下;

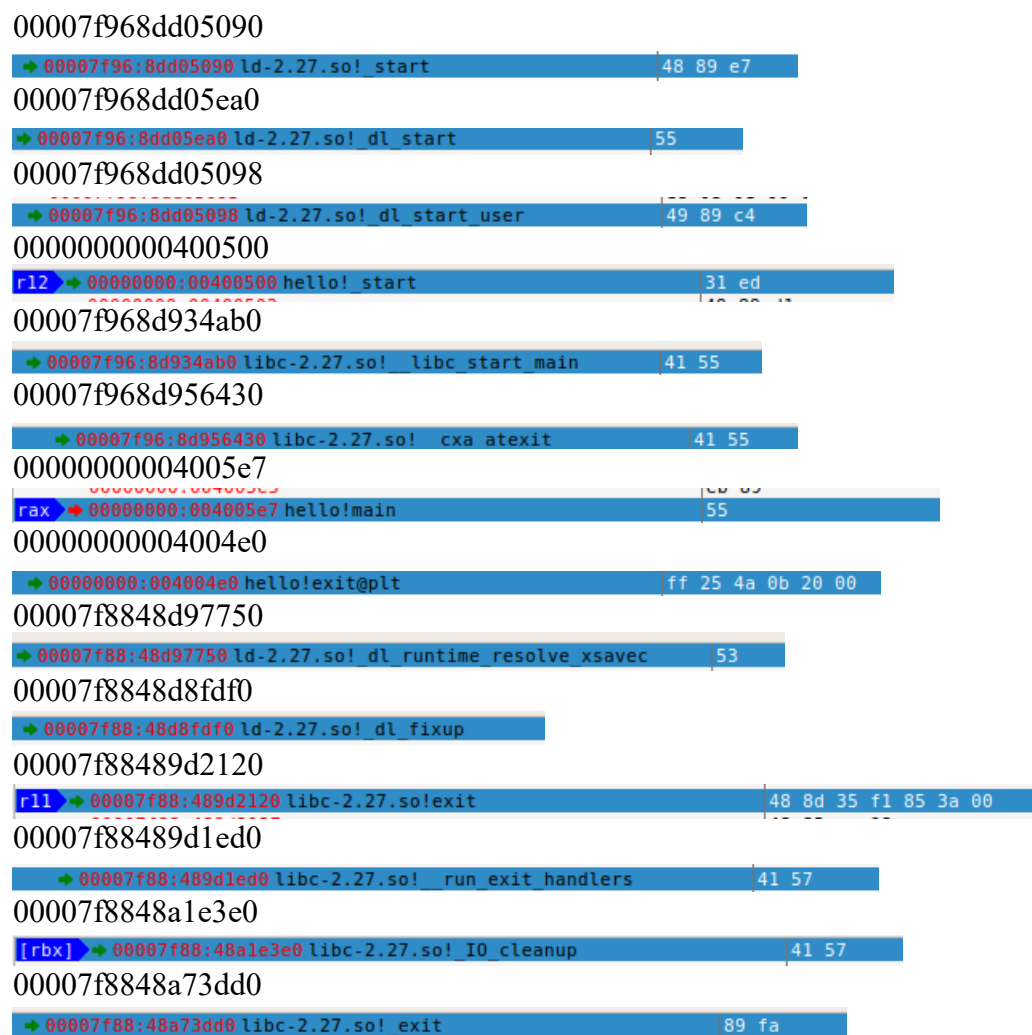


图 5.35 hello 执行过程

5.7 Hello 的动态链接分析

对动态链接库函数，编译系统采取延迟绑定策略，使用 PLT（过程链接表）配合 GOT（全局偏移表）间接访问。GOT 中存放目标函数地址，PLT 使用 GOT 中地址跳转到目标函数

延迟绑定

为避免加载时，重定位共享库中大量不会被实际使用的函数，GNU 编译系统将过程地址的绑定推迟到首次调用该过程时。

过程链接表 PLT

PLT 是代码段的一部分，可以看作是条目的数组，每个条目 16 byte, PLT[0]跳转到动态链接器，PLT[1]跳转到系统启动函数，`__libc_start_main`，PLT[2]即以后的条目调用用户代码调用的函数，每个被可执行程序调用的函数都有自己的 PLT 条目。

全局变量偏移表 GOT

存储于数据段开始处，是条目的数组，每个条目 8 byte。每个被本目标模块引用的全局符号（全局变量，过程）对应一个条目。编译器为每个条目生成一个重定位条目。动态链接器在目标模块加载时为每个条目生成指向目标的绝对地址。代码段的代码通过 `GOT[i]` 间接访问全局变量。每个引用全局变量的目标模块都有自己的 GOT。

联用 PLT, GOT

GOT[0], GOT[1] 含动态链接器所需的信息，GOT[2] 对应动态链接器在 `ld-linux.so` 模块的入口点。其余每条目对应一个被调函数，其地址在运行时解析。初始时，每个 GOT 条目都指向 PLT 条目的第二条指令。

`dl_init` 由 `dl_start_user` 调用，运行时地址如下：

```
00007f3a:d60f38c5 | e8 66 f5 00 00 | callq ld-2.27.so!_dl_init
```

图 5.36

在可执行目标文件代码中找到 PLT 地址：

00000000004004a0

Disassembly of section .plt:

00000000004004a0 <.plt>:

图 5.37

找到 GOT 地址：

0000000000601000

```
4004a6: ff 25 64 0b 20 00 jmpq *0x200b64(%rip) # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
```

图 5.38 GOT 地址

运行时 PLT:

```
00000000:00400490 0b 20 00 48 85 c0 74 02 ff d0 48 83 c4 08 c3 00 | .H.t.H.t.t.
00000000:004004a0 ff 35 62 0b 20 00 ff 25 64 0b 20 00 0f 1f 40 00 | 5b .d .@.
00000000:004004b0 ff 25 62 0b 20 00 68 00 00 00 00 e9 e0 ff ff ff | %b .h...+d
00000000:004004c0 ff 25 5a 0b 20 00 68 01 00 00 00 e9 d0 ff ff ff | %Z .h...+
00000000:004004d0 ff 25 52 0b 20 00 68 02 00 00 00 e9 c0 ff ff ff | %R .h...+
00000000:004004e0 ff 25 4a 0b 20 00 68 03 00 00 00 e9 b0 ff ff ff | %J .h...+
00000000:004004f0 ff 25 42 0b 20 00 68 04 00 00 00 e9 a0 ff ff ff | %B .h...+
00000000:00400500 31 ed 49 89 d1 5e 48 89 e2 48 83 e4 f0 50 54 49 | I.I.H.-H. | PTI
00000000:00400510 c7 c0 e0 06 40 00 48 c7 c1 70 06 40 00 48 c7 c7 | d.@.H.p.@.H
00000000:00400520 e7 05 40 00 ff 15 c6 0a 20 00 f4 0f 1f 44 00 00 | .@. . .D..
```

图 5.39 PLT:

运行时 GOT(dl_start 前):

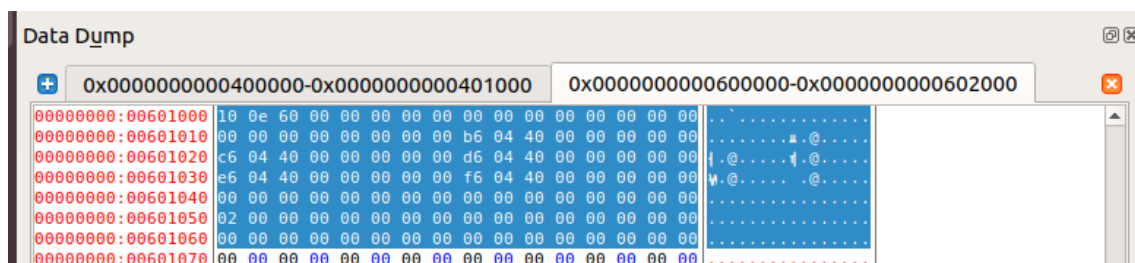


图 5.40 GOT

运行时 GOT(dl_start 后):

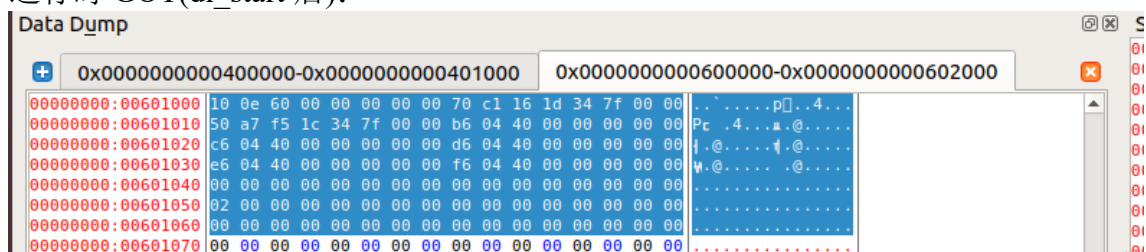


图 5.41 GOT

可以看到 GOT[1],GOT[2]发生了变化:

```
00000000:00601000| 70 c1 16 1d 34 7f 00 00
00000000:00601010| 50 a7 f5 1c 34 7f 00 00
```

GOT[1] = 0x00007f341d16c170

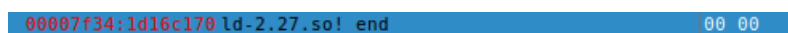


图 5.42 GOT[1]

指向重定位表。

GOT[2] = 0x00007f341cf5a750

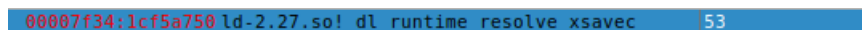


图 5.43 GOT[2]

指向动态链接的系统函数

5.8 本章小结

本章讨论了编译系统的最后一个环节——链接。通过观察 hello 程序的运行，分析了重定位，链接，虚拟地址空间等内容。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程是执行中程序的一个实例。在资源管理上，是一组资源的容器，拥有独立的地址空间，寄存器文件，打开的文件描述符等进程上下文。在控制逻辑上，进程提供了一个独占计算机资源的控制抽象。

早期面向进程设计的计算机结构中，进程是程序的基本执行实体；

当代面向线程设计的计算机结构中，进程是线程的容器；

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 是提供系统内核与用户间交互接口的应用程序。

处理流程：

在终端读入用户命令；

解析用户命令；

若是内部命令，立即执行；

若是外部命令，fork 子进程，并由子进程执行命令；

等待子进程结束或直接等待下一个命令（&参数）

6.3 Hello 的 fork 进程创建过程

1.用户在 shell 终端输入 “./hello 1170300224 szt” 并发送。

2.shell 接受输入行，并解析参数。

3.shell 分析命令，判定不是内部命令，于是假定这是一个外部命令。

4.shell 发起调用 fork,陷入内核

5.内核创建 shell 的一个几乎完全相同的副本（pid 不同），父子进程地址空间相同但独立（写时复制）。

6.父子进程并发运行，父进程默认（无&参数）等待子进程结束。

6.4 Hello 的 execve 过程

1.子进程在被 fork 调用生成以后，做一些预备工作。

2.子进程调用 execve，陷入内核

3.内核代码在子进程的上下文中，调用加载器，替换进程的整个核心映像，加

载 hello 程序的代码和数据。

4. 内核返回控制权给 hello 程序。

具体地，加载器删除子进程现有的虚拟内存段，创建一组新的代码，数据，堆栈段，新的栈和堆初始化为零。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上，初始化新代码段，数据段为 hello 程序的内容。最终，加载器跳转到 `_start` 程序，由他调用 hello 的 `main` 函数。

加载器创建的内存映像如下：

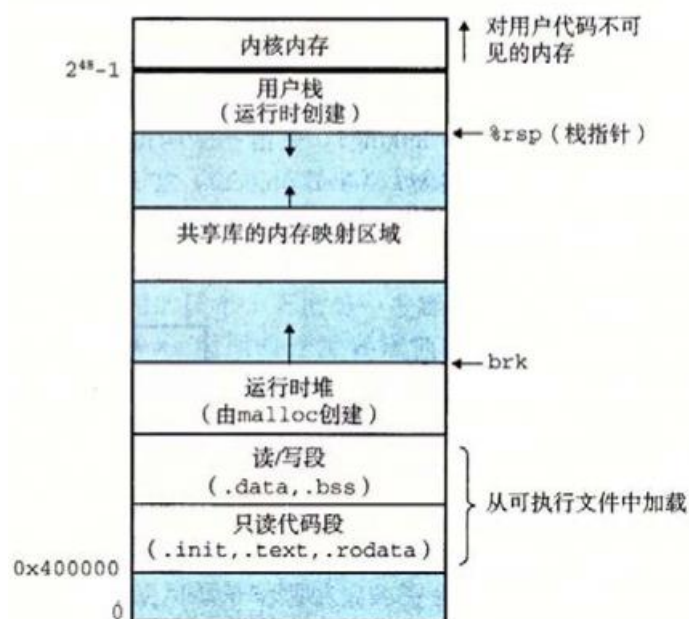


图 6.1 内存映像

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。在抢占式系统中，进程轮流使用处理器。每个进程占有 `cpu` 一段时间后被抢占（挂起），然后进程调度程序分配时间片给其他进程。

时间片：进程调度程序分配给进程的，允许该进程一次占用 `cpu` 的最大时间。

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。

进程上下文：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。

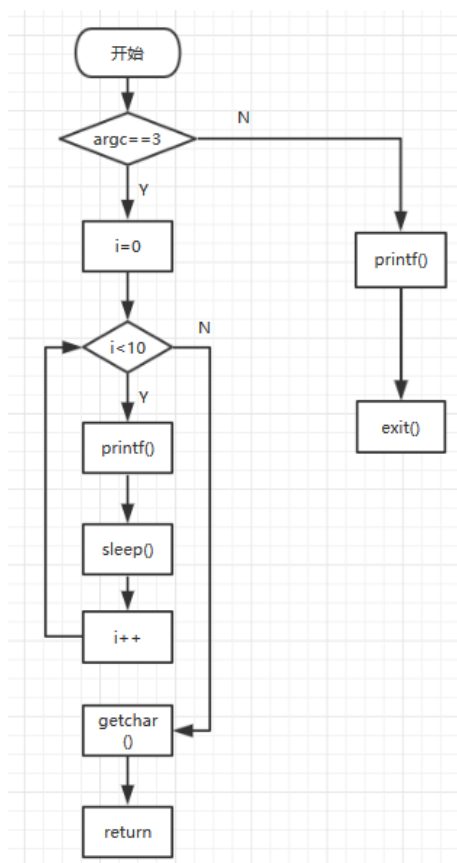


图 6.2 Hello 程序逻辑

假定 **hello** 程序正常执行，到某时刻，**hello** 的时间片用尽，进程调度程序决定调用另一个就绪的进程。于是发生一次上下文切换。内核需要保存 **hello** 的寄存器，页表指针，已打开文件的描述符等所有为将来恢复 **hello** 所需的上下文信息，并载入目标进程的上下文信息。假定稍后进程调度程序又决定调用 **hello**，于是内核恢复 **hello** 的上下文，将控制权转交给 **hello** 用户程序。稍后，**hello** 发起了 **sleep** 系统调用，该调用导致进程陷入内核态，相应的系统代码释放了 **hello** 对 **cpu** 的控制，并通知系统在 2 秒以后给 **hello** 进程一个信号。进程调度程序调度了另一个进程，在该进程运行过程中，定时器中断了该进程，异常处理程序给 **hello** 一个信号并将控制权转交给 **hello**，**hello** 继续运行。

系统接收到键盘中断，发送给 shell 一个 SIGSTP 信号，shell 的信号处理程序在屏幕上打印信息，暂停子进程 hello。

```
sztt1170300224@ubuntu:~/hitcs/final$ ps
  PID TTY          TIME CMD
 25514 pts/0        00:00:00 bash
 25573 pts/0        00:00:00 hello
 25576 pts/0        00:00:00 ps
```

图 6.6 Ps 命令

可见 hello 没有被回收，只是被挂起了。

```
sztt1170300224@ubuntu:~/hitcs/final$ jobs
[1]+  Stopped                  ./hello 1170300224 sztt
sztt1170300224@ubuntu:~/hitcs/final$
```

图 6.7 Jobs 命令

可以看到只有一个 job: hello,且它被挂起了

```
sztt1170300224@ubuntu:~/hitcs/final$ fg
./hello 1170300224 sztt
Hello 1170300224 sztt
Hello 1170300224 sztt
Hello 1170300224 sztt
Hello 1170300224 sztt
Hello 1170300224 sztt
Hello 1170300224 sztt
```

图 6.8 Fg 命令

Fg 命令重启了 hello 程序。

```
sztt1170300224@ubuntu:~/hitcs/final$ ./hello 1170300224 sztt
Hello 1170300224 sztt
Hello 1170300224 sztt
^C
sztt1170300224@ubuntu:~/hitcs/final$
```

图 6.9 Ctrl+c

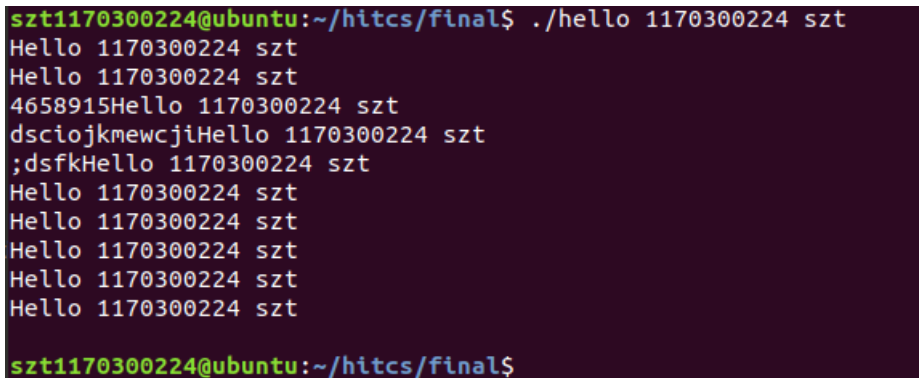
Ctrl+c 终止了前台任务。

Shell 进程收到 SIGINT 信号，信号处理程序分析命令后给予进程 hello 发送了 SIGKILL 信号，终止了程序。

```
sztt1170300224@ubuntu:~/hitcs/final$ ps
  PID TTY          TIME CMD
 25514 pts/0        00:00:00 bash
 25596 pts/0        00:00:00 ps
sztt1170300224@ubuntu:~/hitcs/final$ jobs
sztt1170300224@ubuntu:~/hitcs/final$
```

图 6.10 ps, jobs

执行 ps 命令和 jobs 命令，可见 hello 进程确实被终止了。



```
szt1170300224@ubuntu:~/hitcs/final$ ./hello 1170300224 szt
Hello 1170300224 szt
Hello 1170300224 szt
4658915Hello 1170300224 szt
dsciojkmewcjiHello 1170300224 szt
;dsfkHello 1170300224 szt
Hello 1170300224 szt
Hello 1170300224 szt
Hello 1170300224 szt
Hello 1170300224 szt
Hello 1170300224 szt
szt1170300224@ubuntu:~/hitcs/final$
```

图 6.11 在 hello 执行过程中随意乱按

输入只是被简单回显在屏幕上，可见输入停留在输入缓冲区内，没有被发送给进程。当 hello 等待 `getchar()` 时，键入一个回车键，之前的这些输入被一并发送给 hello 程序，并不会被发送到 shell 进程。

6.7 本章小结

本章讨论了操作系统提供的一个重要抽象——进程。该抽象使得程序员得以在编程时脱离底层系统复杂丑陋的实现，避开大量涉及并发处理的底层工作。但对异常和信号的讨论，使我们不得不回到底层一探究竟，体验系统级编程时如履薄冰的危险。

(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

逻辑地址空间：段地址：偏移地址

线性地址空间：非负整数地址的有序集合： $\{0, 1, 2, 3 \dots\}$

虚拟地址空间： $N = 2^n$ 个虚拟地址的集合 \Rightarrow 线性地址空间 $\{0, 1, 2, 3, \dots, N-1\}$

物理地址空间： $M = 2^m$ 个物理地址的集合 $\{0, 1, 2, 3, \dots, M-1\}$

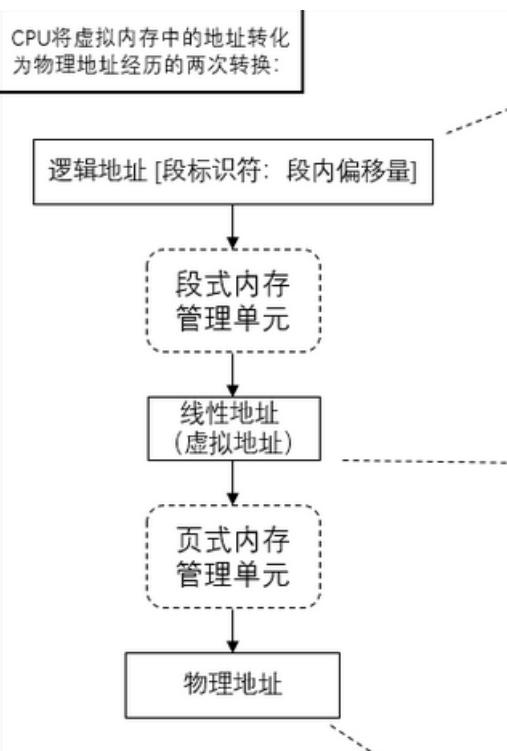


图 7.1 地址空间转换

7.2 Intel 逻辑地址到线性地址的变换-段式管理

段寄存器

16 位，用于存放段选择符

CS(代码段)：程序代码所在段

SS(栈段)：栈区所在段

DS(数据段)：全局静态数据区所在段

其他 3 个段寄存器 ES、GS 和 FS 可指向任意数据段

段描述符

段描述符是一种数据结构，实际上就是段表项

用户段描述符：代码段，数据段；

系统段描述符：

特殊系统控制段描述符：局部描述符表（LDT）描述符和任务状态段（TSS）描述符

控制转移类描述符：调用门描述符、任务门描述符、中断门描述符和陷阱门描述符

段描述符各字段：

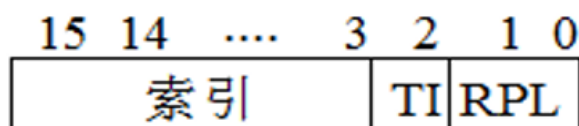


图 7.2 段描述符格式

RPL[1:0]: CS 寄存器中的 RPL 字段表示 CPU 的当前特权级 (Current Privilege Level, CPL)，特权级越小，权限越高，内核运行在 0 权级，用户程序 3 权级。

TI[2]: TI=0，选择全局描述符表(GDT)，TI=1，选择局部描述符表(LDT)

索引[15:3]: 确定当前使用的段描述符在描述符表中的位置

段描述符表

描述符表实际上就是段表，由段描述符(段表项)组成。

全局描述符表 GDT: 只有一个，用来存放系统内每个任务都可能访问的描述符。例如：内核代码段、内核数据段、用户代码段、用户数据段以及 TSS（任务状态段）等都属于 GDT 中描述的段

局部描述符表 LDT: 存放某任务（即用户进程）专用的描述符

中断描述符表 IDT: 包含 256 个中断门、陷阱门和任务门描述符

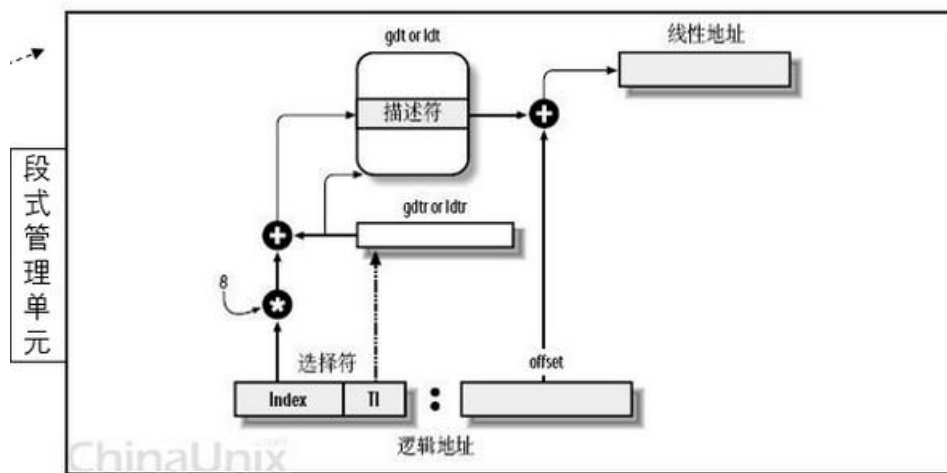


图 7.3 段式管理单元示意

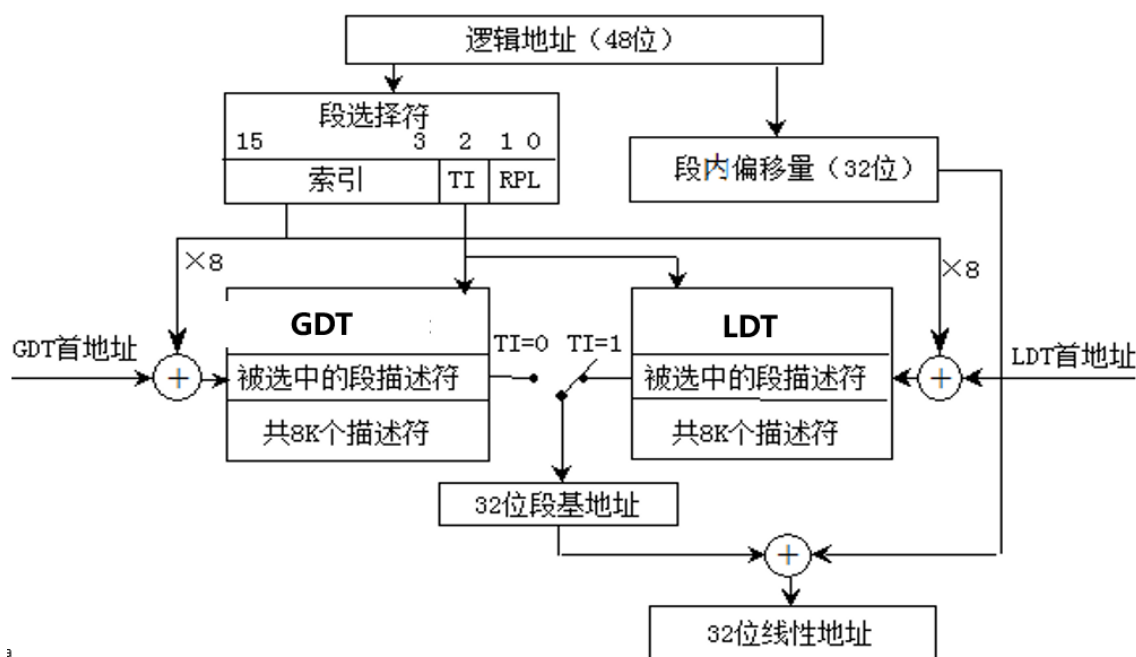


图 7.4 逻辑地址到线性地址转换

如图，由 48 位逻辑地址抽取 16 位段选择符和 32 位段内偏移。由段选择符访问段描述符表，获取段基址。段基址加上段内偏移即可获得 32 位的线性地址。

为使能移植到绝大多数流行处理器平台，Linux 简化了分段机制。RISC 对分段支持非常有限，因此 Linux 仅使用 IA-32 的分页机制，而对于分段，则通过在初始化时将所有段描述符的基址设为 0 来简化。所以分段机制事实上没有得到很好的使用。

7.3 Hello 的线性地址到物理地址的变换-页式管理

虚拟地址空间: Virtual Address Space $V = \{0, 1, \dots, N-1\}$

物理地址空间: Physical Address Space $P = \{0, 1, \dots, M-1\}$

地址翻译:

MAP: $V \rightarrow P \cup \{\emptyset\}$

For virtual address a :

- $MAP(a) = a'$ 如果虚拟地址 a 处的数据在 p 的物理地址 a' 处
- $MAP(a) = \emptyset$ 如果虚拟地址 a 处的数据不在物理内存中
 - 不论无效地址还是存储在磁盘上

图 7.5 地址翻译

不考虑多级页表时的情况:

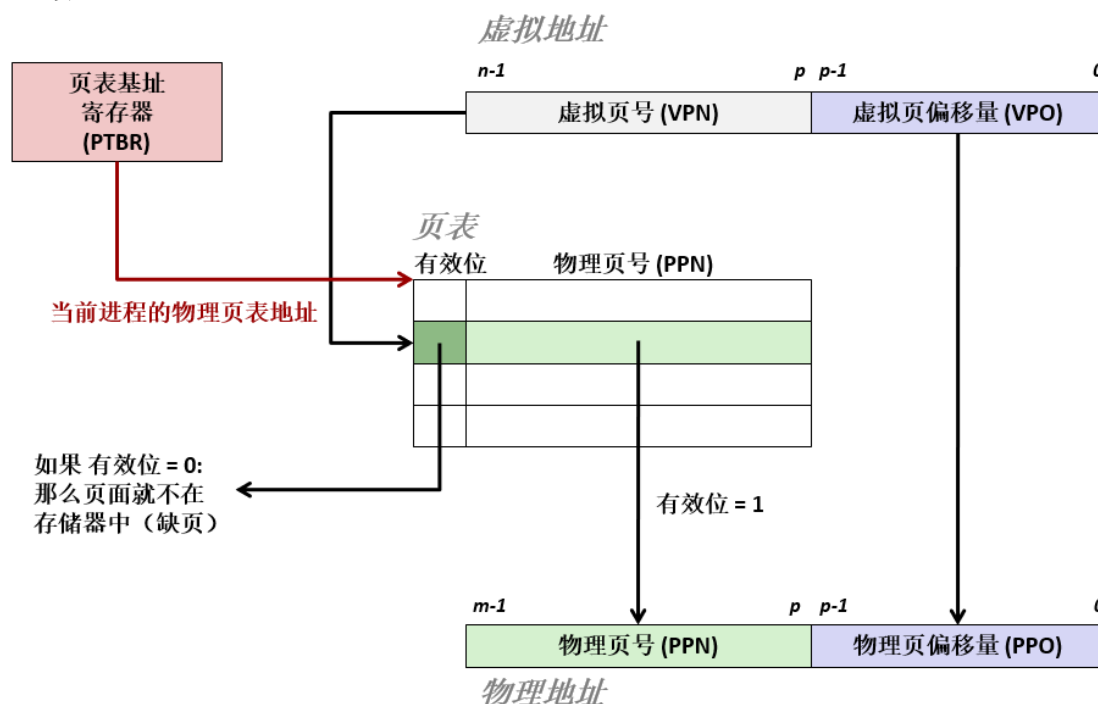


图 7.6 页表

从虚拟地址中抽取出虚拟页号 VPN 和虚拟页偏移 VPO;

由虚拟页号作为页表索引, 以通过页表基址寄存器访问相应页表项;

检查页表项的有效位:

若为 0 则触发缺页错误;

若为 1 则取出物理页号 PPN

虚拟页偏移 VPO 就是物理页偏移 PPO

由物理页号和物理页偏移组合得到物理地址。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

多级页表的必要性：

对 4KB (2^{12}) 页面, 48 位地址空间, 8 字节 页表项 (PTE)

$2^{48} / 2^{12} * 2^3 = 2^{39}$ bytes, 亦即需要 512 GB 的页表。

事实上, 地址空间中的大部分地址都不会被使用, 有大量页表项是不必要的, 故而引入多级页表以避免存储大量不必要的页表项。

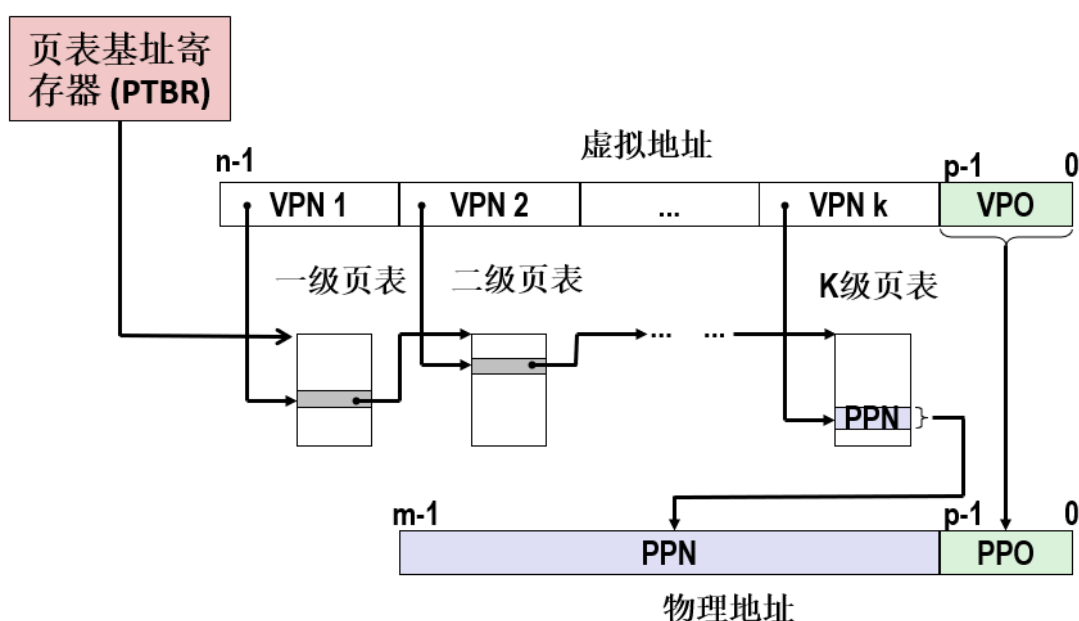


图 7.7 多级页表

如图, 有使用 K 级页表的系统

首先, 从虚拟地址中抽出一级页表的页表项索引 VPN_1 , 以页表基址寄存器为基址访问相应页表项, 获得相应二级页表基址。

。。。。

从虚拟地址中抽取 i 级页表的页表项索引 VPN_i , 以上一步获得的第 i 级页表基址值为基址访问相应页表项, 获得相应 $i+1$ 级页表基址。

。。。。

其中若有任何一步页表项的有效位为 0, 则触发缺页故障; 若许可位不匹配, 则触发段错误。

以次类推, 直至获得最终的物理地址基址, 再和虚拟地址中抽取出的页表内偏移量组合即可获得物理地址。

实际上, 现代系统大多使用 4 级页表, 原理同上。

7.5 三级 Cache 支持下的物理内存访问

以下主要讨论 L1 cache 在存储系统中的作用。当前的 cpu 大多有 3 级 cache, 但原理与一级类似。

大多系统选择以物理地址访问 SRAM 高速缓存, 亦即地址翻译发生在高速缓存查找之前。

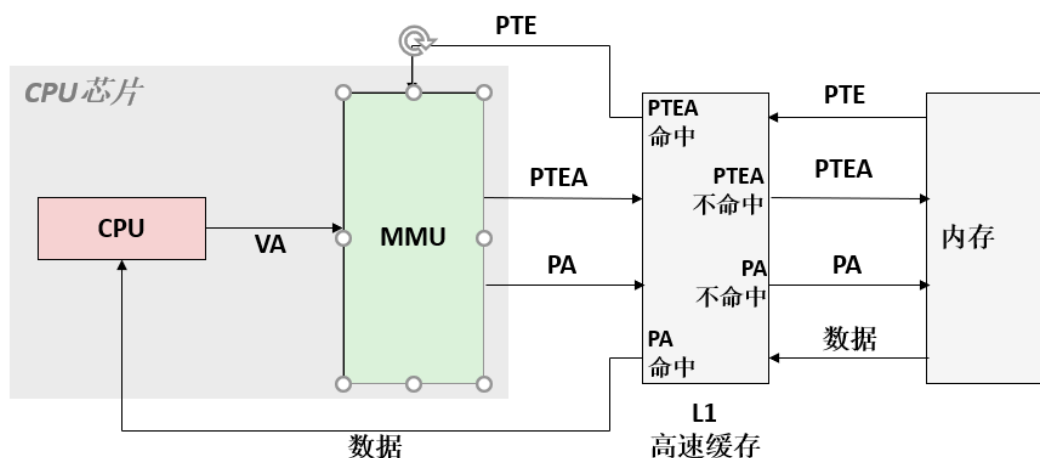


图 7.8 cache 下内存访问

大致流程如下:

cpu 向 MMU 请求一个虚拟地址 VA;

MMU 从 VA 中抽出 VPN, 结合页表基址计算出相应页表条目地址 PTEA, 发送给 L1 cache;

若 PTEA 命中, L1 cache 返回相应页表条目 PTE 给 MMU;

MMU 从 PTE 中解析出物理地址 PA (假定合法有效) 发送给 L1 cache;

L1 cache 根据地址 PA 返回相应数据(假定 PA 命中)

以上讨论假定了高速缓存命中, 没有缺页故障和段错误, 此时数据传输都发生在 CPU 和 L1 cache 之间, 有很高的效率。

下面讨论 PA 高速缓存未命中的情况:

L1 cache 收到 MMU 发送的 PA, 但是缓存未命中;

于是 L1 cache 将 PA 发送给下级存储器 (L2 cache, 上图中抽象为内存单元);

下级存储器的处理方式类似 L1 cache, 若 PA 命中则直接返回相应数据, 否则继续向更下级存储请求数据。

缺页故障的情况在 7.8 节讨论。

7.6 hello 进程 fork 时的内存映射

Fork 函数一次调用，两次返回（成功时）。它为当前进程创建一个几乎完全相同的副本，只是在子进程父进程中 `fork()` 的返回值不同。父进程中返回子进程的 `pid`，子进程中返回 0。

Shell 进程调用 `fork` 时，内核为新进程创建所需的各种数据结构，分配唯一的 `pid`。新进程拥有独立的地址空间，通过写时复制技术，新进程的页面被映射到与父进程一致的内存区域。内核将写时复制的页标记为只读，并标记写时复制位。若两进程都不写共享的内存区域，该区域将一直只有一份副本。否则，触发异常，内核复制一个同样内容的新页再执行写操作。

7.7 hello 进程 `execve` 时的内存映射

Shell 子进程调用 `execve` 陷入内核，调用加载器的代码，在当前进程上下文中加载并运行 `hello` 程序。

`Execve` 会删除当前进程地址空间中已有的用户区域。

创建新内存区域。将代码段映射到 `hello` 程序文件的 `.text` 段；数据段映射到 `hello` 文件的 `.data` 段；为 `.bss` 段请求二进制零。为堆栈请求初始长度为零的二进制零区域。

`Execve` 还会动态链接 `hello` 程序和共享库对象，将共享库代码映射到共享内存段。

最后，`execve` 设置程序计数器 `PC` 执行代码段的入口点。

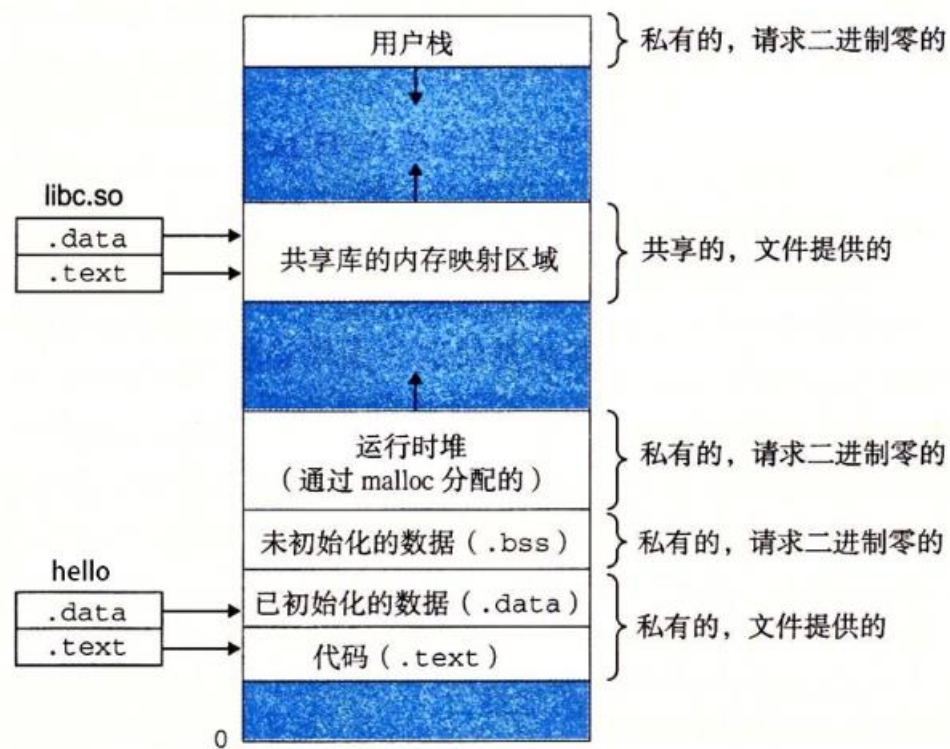


图 7.9 exeve 内存映像

7.8 缺页故障与缺页中断处理

当所请求地址 VA 所对应的物理页面不在主存中时，会触发缺页故障，大致流程如下：

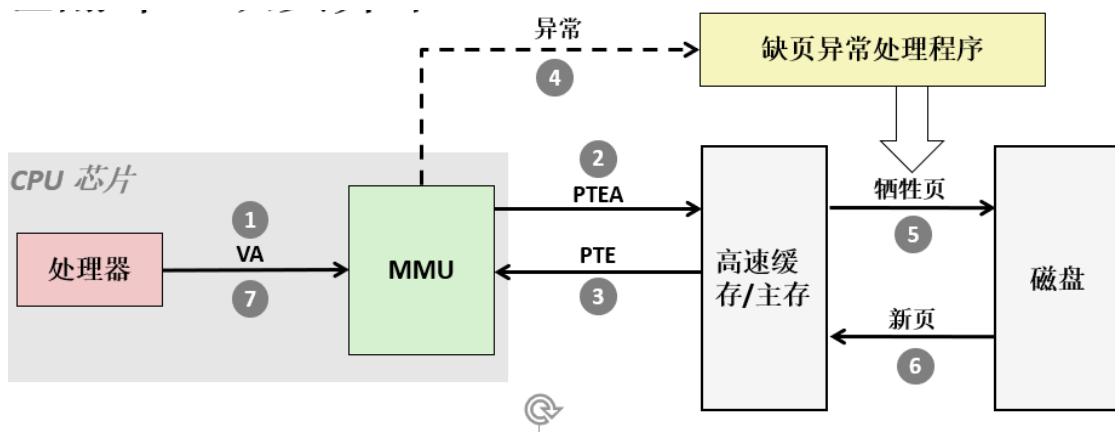


图 7.10 缺页中断

- 1) 处理器将虚拟地址发送给 MMU
- 2-3) MMU 使用内存中的页表生成 PTE 地址
- 4) 有效位为零, 因此 MMU 触发缺页异常
- 5) 缺页处理程序确定物理内存中牺牲页 (若页面被修改, 则换出到磁盘)
- 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来进程, 再次执行缺页的指令

7.9 动态存储分配管理

Printf 会调用 *malloc*, 以下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一个进程的虚拟内存区域, 称作堆。堆通常是请求二进制零的区域, 它紧接在未初始化的数据区域后面开始, 并向上 (高地址) 增长。对每个进程, 内核维护一个变量 *brk*, 指向堆顶。

分配器将堆视作一组块的集合。每个块是一个连续的虚拟内存片。块是已分配的或空闲的。

分配器有两种基本风格:

显式分配器: 要求应用显式释放任何已分配块。

隐式分配器: 自动检测不再使用的已分配块, 并释放之。

带边界标签的隐式空闲链表分配器:

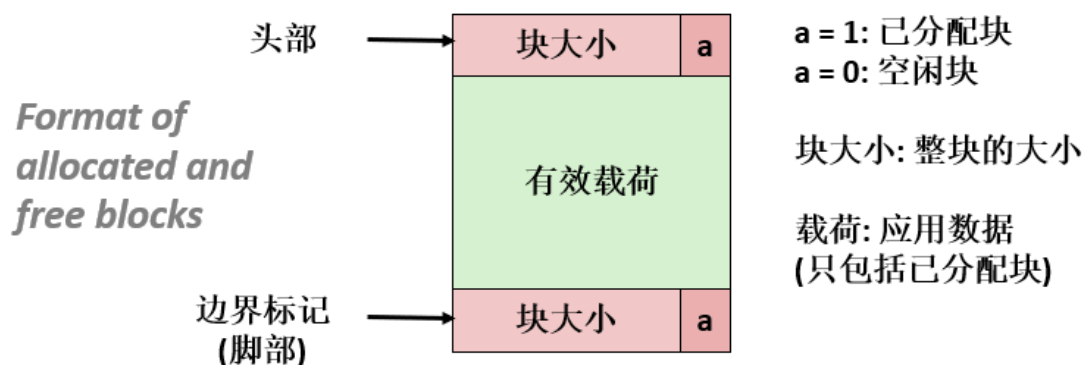


图 7.11 内存块结构

在空闲块的“底部”标记 大小/已分配

允许我们反查“链表”, 但这需要额外的空间

内存块组织——隐式空闲链表

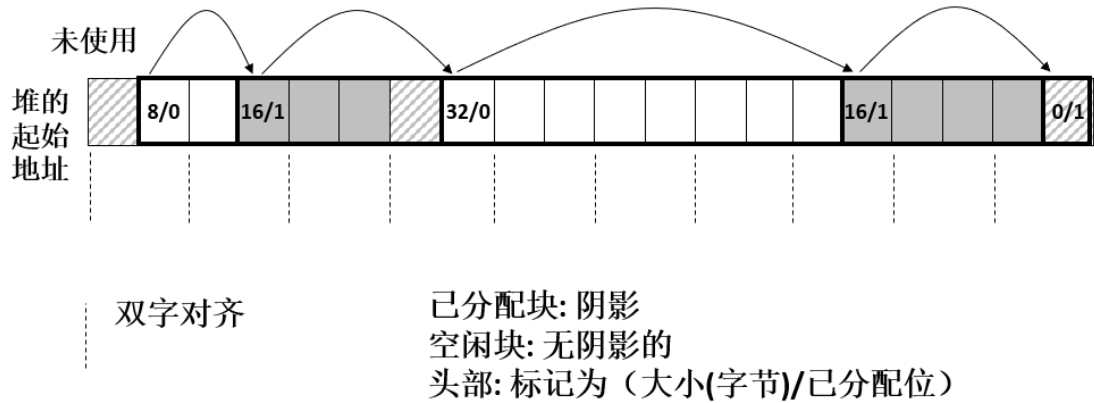


图 7.12 内存块组织结构

每个块内没有显式的指针域，通过头部和尾部的块大小信息隐式的将整个堆组织成一个大的线性表。

通过块头部尾部的已分配位，判断该块是否空闲，又可以间接地获得空闲块的链表。

显式空闲链表：

在空闲块中使用指针连接空闲块

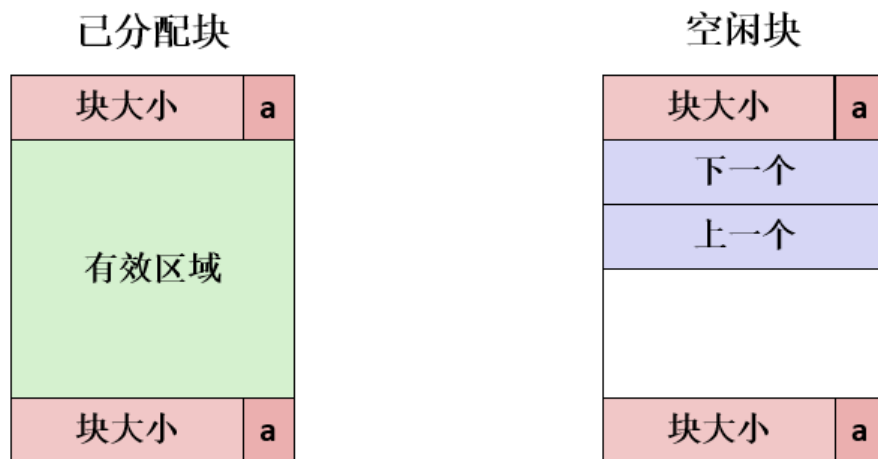


图 7.13 块格式

保留空闲块 链表，而不是所有块

“下一个” 空闲块可以在任何地方

因此我们需要存储前/后指针，而不仅仅是大小
还需要合并边界标记

幸运的是，我们只跟踪空闲块，所以我们可以使用有效区域。

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。

链表组织顺序：

LIFO (last-in-first-out) policy 后进先出法

将新释放的块放置在链表的开始处

Address-ordered policy 地址顺序法

按照地址顺序维护链表

此外，还有分离的空闲链表，伙伴空闲链表，按平衡树组织的显式空闲链表等其它实现方式。

7.10 本章小结

本章讨论了操作系统提供的又一大抽象——虚拟地址空间。虚拟地址空间是高速缓存，内存，硬盘等存储系统的抽象。它使得程序员可以在不涉及底层繁琐实现的情况下就能编写较好利用现代存储设备的代码。

具体的，本章讨论了地址空间概念，存储系统段式管理、页式管理，fork 和 exeve 的内存映射，动态内存分配等内容。

(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：文件

操作系统将所有 I/O 设备抽象为文件，输入操作抽象为读，输出操作抽象为写。这一简单而优雅的实现方式使得应用得以通过统一的接口——UNIX IO 接口，访问 I/O 设备。

8.2 简述 Unix IO 接口及其函数

Unix I/O 接口：

- 1) 打开文件：一个应用程序通过要求内核打开相应文件，宣告它想要访问该 I/O 设备。内核返回一个小非负整数，称作描述符。它在后续对此文件的所有操作中标识这个文件。内核维护关于这个文件的所有信息。
- 2) 默认打开的文件：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

首先查看 `printf` 的示意代码：

```
typedef char * va_list;

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

    //可变参数
    //(char*)&fmt + 4 表示 ... 中的第一个参数
    va_list arg = (va_list)((char*)&fmt + 4);

    i = vsprintf(buf, fmt, arg);
    write(buf, i);

    return i;
}
```

图 8.1 Printf 实现

`Printf` 首先获得可变参数 `arg`，而后调用 `vsprintf`

```

//格式化,接受确定输出格式的格式字符串fmt。用格式字符串对个数变化的参数进行格式化,产生格式化输出。
//返回最后生成的字符串的长度
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 != '%') {
            *p++ = *fmt;
            continue;
        }
        fmt++;

        switch (*fmt) {
            case 'x':
                itoa(tmp, *((int*)p_next_arg));
                strcpy(p, tmp);
                p_next_arg += 4;
                p += strlen(tmp);
                break;
            case 's':
                break;
            default:
                break;
        }
    }

    return (p - buf);
}

```

图 8.2 vsprintf 代码

这是 vsprintf 的一个简易实现,只实现了对 16 进制的格式化。

Vsprintf 产生格式化的输出字符串,并返回字符串长度。

而后,printf 调用系统函数 write()

```

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

```

图 8.3 Write 代码

write 函数将栈中参数放入寄存器

eax 是系统调用号

ecx 是字符个数

ebx 存放第一个字符地址

int INT_VECTOR_SYS_CALLA 是发起系统调用 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 + ecx * 4]
    add esp, 4 * 3

    mov [eax + EAXREG - P_STACKBASE], eax

    cli

    ret

```

图 8.4 syscall 实现

syscall 将字符串“hello 1170300224 szt”从寄存器中通过总线复制到显卡的显存中，显存中存储相应字符的 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。

显卡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）。从而在屏幕上显示目标字符串。

8.4 getchar 的实现分析

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。

具体地，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 等输入函数调用 read 系统函数，通过系统调用读取输入缓冲区内的按键 ascii 码，直到接受到回车键才返回。具体到 Getchar 函数，会读取返回的字符流中的第一个字符。

8.5 本章小结

本章介绍了操作系统提供的另一个重要抽象——文件。文件是所有 I/O 设备的抽象，这为程序员访问 I/O 设备提供了一个简洁而统一的接口。

具体介绍了 UNIX I/O 接口和函数，剖析了输出函数 `printf` 和输入函数 `getchar` 的实现。

(第 8 章 1 分)

结论

Hello 大事记：

- 1.编写：程序员编写源代码 `hello.c`
- 2.预处理：预处理器 `cpp` 预处理 `hello.c` 生成 `hello.i`
- 3.编译：编译器 `ccl` 将 `hello.i` 编译为 `hello.s`
- 4.汇编：汇编器 `as` 将 `hello.s` 翻译为机器代码 `hello.o`
- 5.链接：链接器 `ld` 将 `hello.o` 与库代码链接为 `hello` 可执行程序
- 6.执行：用户在 `shell` 下请求 “`./hello`”
- 7.创建进程：`shell` 为该任务 `fork` 了子进程
- 8.载入 `hello`：子进程执行 `exeve` 换入 `hello` 程序映像
- 9.运行：`hello` 程序在操作系统代码的执行下运行
- 10.回收：`shell` 父进程回收结束任务的 `hello` 进程

短小简练如 `hello` 的程序，也有着坎坷不凡的一生。它的出世，有程序猿击键不辍，有编译系统的齐心协力；它的运行，操作系统的配合无间；它的逝去，有父亲 `shell` 的送别收殓……

(结论 0 分，缺失 -1 分，根据内容酌情加分)

附件

列出所有的中间产物的文件名，并予以说明起作用。

Hello.c	源代码文件
Hello.i	预处理后的源代码文件
Hello.s	汇编后的源代码文件
Hello_r.o	可重定位目标文件
Hello	可执行目标文件
Hello_r.elf	hello_r.o 的 readelf 输出文件
Hello.elf	hello.o 的 readelf 输出文件
Hello_r.obj	hello_r.o 的 objdump 输出文件
Hello.obj	hello.o 的 objdump 输出文件

(附件 0 分，缺失 -1 分)

参考文献

为完成本次大作业你翻阅的书籍与网站等

[1]节头目表文档:

<https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html>

[2] LINUX 逻辑地址、线性地址、虚拟地址和物理地址:

https://blog.csdn.net/baidu_35679960/article/details/80463445

[3]printf()剖析:

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

[4]《深入理解计算机系统》机械工业出版社, (美) Bryant,R.E.

[5]《现代操作系统》机械工业出版社, (荷兰)AnderwS.Tanenbaum

[6] 哈尔滨工业大学计算机系统课程课件

(参考文献 0 分, 缺失 -1 分)