

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</u>
學	號	<u>1170300815</u>
班	級	<u>1703008</u>
學	生	<u>范天祥</u>
指	導	教
師		<u>鄭貴濱</u>

計算機科學與技術學院

2018 年 12 月

摘 要

本文描述了一个高级语言程序从编写到执行的全过程，详细地介绍了高级语言程序如何通过预处理、编译、汇编和链接生成完全链接的可执行文件。然后介绍了进程与存储管理，描述了一个可执行文件如何装载到内存，如何通过 shell（linux 下）运行程序，以及对故障的处理。最后还介绍了 I/O 管理，描述了如何调用函数利用硬件设计，实现输出到屏幕。全文对一个简单的 C 语言程序生命周期内操作系统与硬件实现机制进行了粗略的介绍，将计算机系统各个组成部分有机地结合起来，搭建出一个较为完整的知识体系。

关键词：编译；汇编；链接；进程存储管理；虚拟内存；操作系统

（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	- 4 -
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	- 8 -
2.3 HELLO 的预处理结果解析	- 8 -
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	- 15 -
第 4 章 汇编	- 16 -
4.1 汇编的概念与作用	- 16 -
4.2 在 UBUNTU 下汇编的命令	- 16 -
4.3 可重定位目标 ELF 格式	- 16 -
4.4 HELLO.O 的结果解析	- 19 -
4.5 本章小结	- 22 -
第 5 章 链接	- 23 -
5.1 链接的概念与作用	- 23 -
5.2 在 UBUNTU 下链接的命令	- 23 -
5.3 可执行目标文件 HELLO 的格式	- 23 -
5.4 HELLO 的虚拟地址空间	- 26 -
5.5 链接的重定位过程分析	- 29 -
5.6 HELLO 的执行流程	- 32 -
5.7 HELLO 的动态链接分析	- 32 -
5.8 本章小结	- 33 -
第 6 章 HELLO 进程管理	- 34 -
6.1 进程的概念与作用	- 34 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 34 -
6.3 HELLO 的 FORK 进程创建过程	- 34 -
6.4 HELLO 的 EXECVE 过程	- 35 -
6.5 HELLO 的进程执行.....	- 35 -
6.6 HELLO 的异常与信号处理	- 37 -
6.7 本章小结	- 39 -
第 7 章 HELLO 的存储管理.....	- 40 -
7.1 HELLO 的存储器地址空间	- 40 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 40 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 41 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 42 -
7.5 三级 CACHE 支持下的物理内存访问	- 43 -
7.6 HELLO 进程 FORK 时的内存映射	- 44 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 45 -
7.8 缺页故障与缺页中断处理.....	- 45 -
7.9 动态存储分配管理	- 46 -
7.10 本章小结	- 47 -
第 8 章 HELLO 的 IO 管理	- 48 -
8.1 LINUX 的 IO 设备管理方法	- 48 -
8.2 简述 UNIX IO 接口及其函数	- 48 -
8.3 PRINTF 的实现分析.....	- 48 -
8.4 GETCHAR 的实现分析.....	- 49 -
8.5 本章小结	- 50 -
结论	- 50 -
附件	- 53 -
参考文献.....	- 54 -

第 1 章 概述

1.1 Hello 简介

P2P: From Program to Process

在 linux 环境下, `hello.c` 文件经过预处理器 `cpp` 编译器 `ccl` 汇编器 `as` 链接器 `ld` 处理之后, 采用虚拟地址形式装载进入内存。然后我们要将它的文件名输入到称为外壳 (`shell`) 的应用程序中, 外壳是一个命令行解释器, 它输出一个提示符, 等待你输入一个命令, 然后执行这个命令。OS(进程管理)为 `hello` 进行 `fork(Process)`, 为其 `execve`, 为其 `mmap`, 分其时间片, 让程序得以在 Hardware(CPU/RAM/IO)上驰骋 (取指译码执行/流水线等)

O2O: From Zero-0 to Zero-0

当程序运行结束后, `shell` 父进程会负责回收 `hello` 子进程, 内核删除相关数据结构, 完美谢幕。

1.2 环境与工具

硬件环境: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70GHz 8G RAM
256G SSD x64 处理器

软件环境: VMware 14 player、ubuntu-18.04.1、Windows10

开发与调试工具: gcc、vim、edb、hexedit、readelf、objdump

1.3 中间结果

文件名称	文件作用
<code>hello.c</code>	提供源程序
<code>hello.i</code>	<code>cpp</code> 预处理之后的文件
<code>hello.s</code>	<code>cc1</code> 之后的汇编语言格式文件
<code>hello.o</code>	<code>as</code> 之后的可重定位目标文件
<code>hello</code>	<code>ld</code> 之后的可执行目标文件
<code>helloftx.elf</code>	<code>hello.o</code> 的 ELF 格式文件
<code>hello.objdump</code>	Objdump 得到的 <code>hello</code> 的反汇编代码

1.4 本章小结

本章介绍了 hello 程序的 P2P，020 的整个过程。
提供了完成本次实验的硬件软件环境和所使用的开发与调试工具
粗略介绍了研究过程中的中间结果，以及其用途和对研究过程产生的作用。
(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

概念：

预处理一般是在编译之前进行的处理。C 语言的预处理主要有三个方面的内容： 1.宏定义； 2.文件包含； 3.条件编译。 预处理命令以符号“#”开头。

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。

作用：

经过预编译处理后，得到的是预处理文件（如：**hello.i**），它还是一个可读的文本文件，但不包含任何宏定义。

1. 宏定义

`#define` 标识符 字符串：将宏名替换为字符串。

`#define` 宏名（参数表） 字符串：除了一般的字符串替换，还要做参数代换。

2. 文件包含

编辑：一个文件包含另一个文件的内容

格式：`#include "文件名"` 或 `#include <文件名>`

编译时以包含处理以后的文件为编译单位，被包含的文件是源文件的一部分。

编译以后只得到一个目标文件.obj

被包含的文件又被称为“标题文件”或“头部文件”、“头文件”，并且常用.h 作扩展名。修改头文件后所有包含该文件的文件都要重新编译。头文件的内容除了函数原型和宏定义外，还可以有结构体定义，全局变量定义。

3. 条件编译

A.双分支条件编译

形式：

`#if` 条件表达式

代码段 1

`#else`

代码段 2

`#endif`

作用：

//endif 结束条件编译

//#if,#else 和 C 语言里的 if else 功能一样，但是时间开销不一样

//if else 会编译所有的代码，源码会较长，编译时间会较长

//程序体积大，占用更多内存，运行时间长

//#if,#else 只编译符合条件的语句，有效减少被编译的语句，

//缩短源码长度，缩短程序执行时间

B.多分支条件编译

条件编译的概括作用：

使用条件编译可以使目标程序变小，运行时间变短。

预编译使问题或算法的解决方案增多，有助于我们选择合适的解决方案。

PS: 此外，还有布局控制：**#pragma**，这也是我们应用预处理的一个重要方面，主要功能是为编译程序提供非常规的控制流信息。

具体形式、作用：

#if 条件表达式 1

代码段 1

#elif 条件表达式 2

代码段 2

#elif 条件表达式 3

代码段 3

#elif 条件表达式 4

代码段 4

#else

代码段 5

#endif

检测宏是否定义

#ifdef 宏定义

代码段 1

#endif

#ifdef M 检测 M 这个宏是否定义，定义了就执行代码段 1，没有定义就不执行任何操作

#ifdef 一般用于开启某个功能

检测宏是否未定义

#ifndef 宏定义

代码段 1

#endif

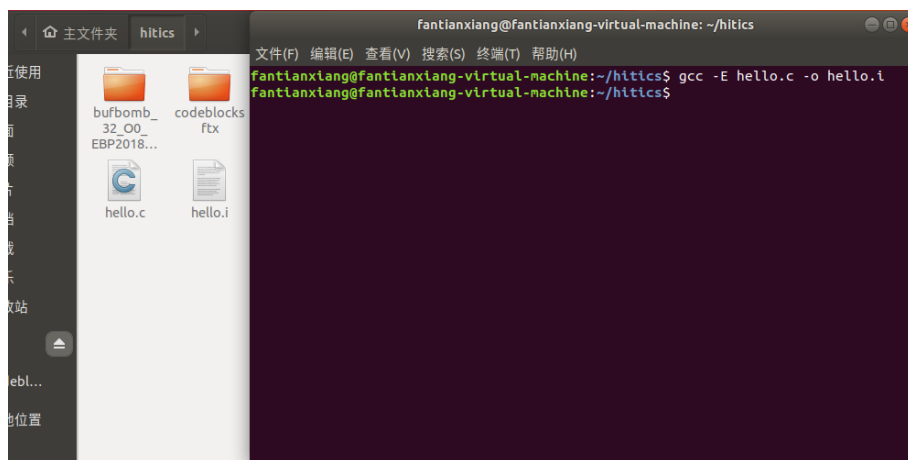
#ifndef M 检测 M 这个宏是否未定义，没有定义就执行代码段 1，定义就不执行任何操作

#ifndef 一般用于开启某个功能或者 include 重包含排错

2.2 在 Ubuntu 下预处理的命令

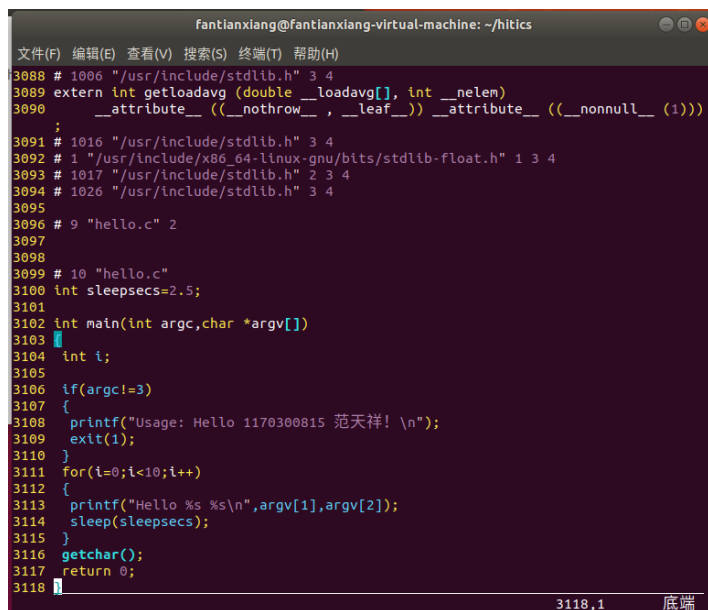
命令：linux> gcc -E hello.c -o hello.i

预处理过程截图：



2.3 Hello 的预处理结果解析

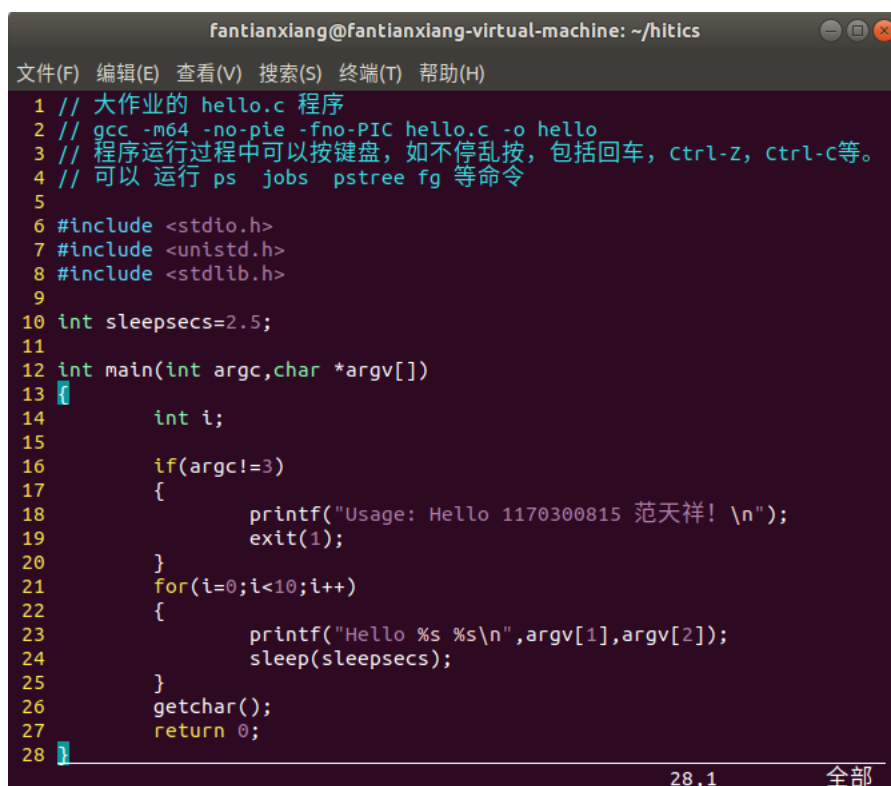
1.使用命令 linux> vim hello.i 查看预处理结果的内容：



可以发现程序已经拓展至 3000 多行，main 函数的代码在 hello.i 的最下面。表

示这个程序已经预处理完毕。

2.再来查看原来的 hello.c 文件，如下：

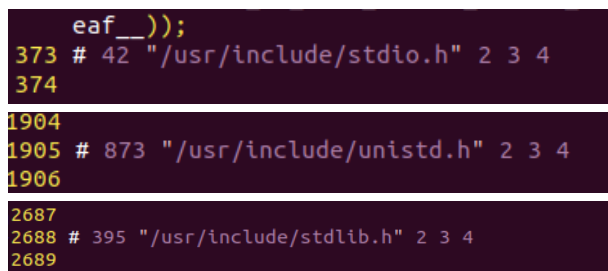


```

fantianxiang@fantianxiang-virtual-machine: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
1 // 大作业的 hello.c 程序
2 // gcc -m64 -no-pie -fno-PIC hello.c -o hello
3 // 程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z, Ctrl-C等。
4 // 可以运行 ps jobs pstree fg 等命令
5
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9
10 int sleepsecs=2.5;
11
12 int main(int argc,char *argv[])
13 {
14     int i;
15
16     if(argc!=3)
17     {
18         printf("Usage: Hello 1170300815 范天祥! \n");
19         exit(1);
20     }
21     for(i=0;i<10;i++)
22     {
23         printf("Hello %s %s\n",argv[1],argv[2]);
24         sleep(sleepsecs);
25     }
26     getchar();
27     return 0;
28 }
28,1 全部

```

可以看见这个程序里面包含了 3 个头文件，故前面的部分应该为 3 个头文件的依次包含（多次出现，只展示部分截图），



```

    eof__));
373 # 42 "/usr/include/stdio.h" 2 3 4
374
1904
1905 # 873 "/usr/include/unistd.h" 2 3 4
1906
2687
2688 # 395 "/usr/include/stdlib.h" 2 3 4
2689

```

当然这 3 个.h 文件里面也包含了很多其它的.h 文件和有很多#define 语句 #if,#else 语句等，gcc 会对此进行递归展开，上面的介绍里面已经描述过，在此不再赘述。

2.4 本章小结

经过预编译处理后，得到的是一个完整的预处理文件（如：hello.i），它还是一个可读的文本文件，但不包含任何宏定义。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

概念：

编译过程就是将预处理后得到的预处理文件（如 `hello.i`）进行词法分析、语法分析、语义分析、优化后，生成汇编代码文件。

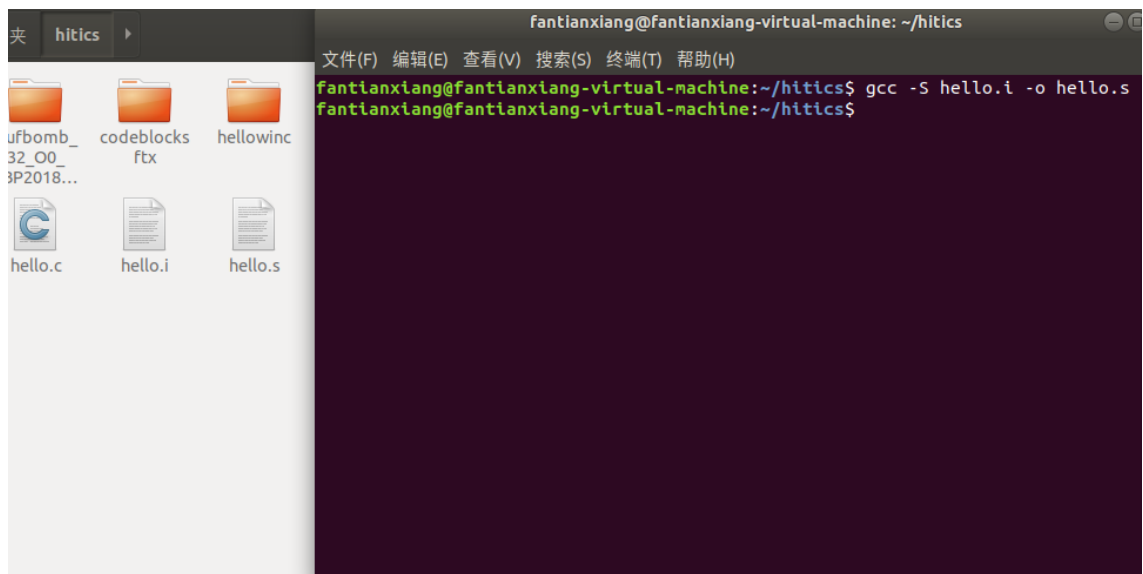
作用：

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。

3.2 在 Ubuntu 下编译的命令

命令行：`linux> gcc -S hello.i -o hello.s`

预处理过程截图：



3.3 Hello 的编译结果解析

编译器是怎么处理 C 语言的各个数据类型以及各类操作
首先查看 `hello.s` 文件内容：

```

                                                                    fantianxiang@fan
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
1      .file    "hello.c"
2      .text
3      .globl   sleepsecs
4      .data
5      .align   4
6      .type    sleepsecs, @object
7      .size    sleepsecs, 4
8  sleepsecs:
9      .long    2
10     .section      .rodata
11     .align 8
12  .LC0:
13     .string "Usage: Hello 1170300815 \350\214\203\345\244\251\347\245\245\357\274\201"
14  .LC1:
15     .string "Hello %s %s\n"
16     .text
17     .globl   main
18     .type    main, @function
19  main:
20  .LFBS:
21     .cfi_startproc
22     pushq    %rbp
23     .cfi_def_cfa_offset 16
24     .cfi_offset 6, -16
25     movq     %rsp, %rbp
26     .cfi_def_cfa_register 6
27     subq     $32, %rsp
28     movl     %edi, -20(%rbp)
29     movq     %rsi, -32(%rbp)
30     cmpl     $3, -20(%rbp)
31     je       .L2
32     leaq     .LC0(%rip), %rdi
33     call     puts@PLT
34     movl     $1, %edi
35     call     exit@PLT
36  .L2:
37     movl     $0, -4(%rbp)
38     jmp      .L3
39  .L4:
40     movq     -32(%rbp), %rax
41     addq     $16, %rax
42     movq     (%rax), %rdx
43     movq     -32(%rbp), %rax
44     addq     $8, %rax
45     movq     (%rax), %rax
46     movq     %rax, %rsi
47     leaq     .LC1(%rip), %rdi
48     movl     $0, %eax
49     call     printf@PLT
50     movl     sleepsecs(%rip), %eax
51     movl     %eax, %edi
52     call     sleep@PLT
53     addl     $1, -4(%rbp)
54  .L3:
55     cmpl     $9, -4(%rbp)
56     jle     .L4
57     call     getchar@PLT
58     movl     $0, %eax
59     leave
60     .cfi_def_cfa 7, 8
61     ret
62     .cfi_endproc
63  .LFES:
64     .size    main, .-main
65     .ident   "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
66     .section .note.GNU-stack,"",@progbits

```

3.3.1 数据类型（整数、字符串、数组）

整数：可以在 C 文件里面看见，有 `int sleepsecs=2.5; int argc; int i;` 这 3 个整型变量和很多立即数，编译器处理分析如下：

a. `int 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：编译器首先将 `sleepsecs` 在 `.text` 代码段中声明为全局变量，然后在 `.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）。

```
1  .file "hello.c"
2  .text
3  .globl sleepsecs
4  .data
5  .align 4
6  .type sleepsecs, @object
7  .size sleepsecs, 4
8  sleepsecs:
9  .long 2
10 .section .rodata
11 .align 8
```

b. `int argc`：声明参变量个数。

c. `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。

```
36 .L2:
37     movl    $0, -4(%rbp)
38     jmp     .L3
39 .L4:
```

d. 立即数：其他整形数据的出现都是以立即数的形式出现的，直接出现在汇编代码中。

字符串：

声明在 `.LC0` 和 `.LC1` 段中的字符串（`.rodata` 节）

```
12 .LC0:
13     .string "Usage: Hello 1170300815 \350\214\203\345\244\251\347\245\245\357\274\201"
14 .LC1:
15     .string "Hello %s %s\n"
```

数组：

数组是一段数据类型相同的物理位置相邻的变量集合，对数组的索引实际上就是在第一个元素地址的基础上通过加索引值乘以数据大小来实现

`argv` 数组作为一个 `char*` 类型的数组，`char*` 的大小是 8 个字节，所以 `argv[1]` 加 `0x8`，而 `argv[2]` 加 `0x10`。

对应的汇编位置为：

```
.L4:
    movq     -32(%rbp), %rax
    addq     $16, %rax
    movq     (%rax), %rdx
    movq     -32(%rbp), %rax
    addq     $8, %rax
    movq     (%rax), %rax
    movq     %rax, %rsi
```

3.3.2 控制转移：

a. `if` 语句跳转实现：

```

30      cmpl    $3, -20(%rbp)
31      je      .L2
32      leaq    .LC0(%rip), %rdi
33      call    puts@PLT
34      movl    $1, %edi
35      call    exit@PLT

```

对应的 C 代码:

```

16      if(argc!=3)
17      {
18          printf("Usage: Hello 1170300815 范天祥! \n");
19          exit(1);
20      }

```

b. for 循环跳转实现:

使用了一个局部变量 i, 该变量存放在栈中 rbp 寄存器指向地址-4 处。

```

36 .L2:
37      movl    $0, -4(%rbp)
38      jmp     .L3

```

首先对其置零进行初始化, 接着使用 cmpl 进行比较, 如果 $i \leq 9$, 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。

```

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

```

对应的 C 代码实现:

```

21      for(i=0;i<10;i++)
22      {
23          printf("Hello %s %s\n",argv[1],argv[2]);
24          sleep(sleepsecs);
25      }

```

3.3.3 函数操作:

函数调用的基本过程。首先将 %rbp 通过 push 指令存储在栈中, 并将 %rsp 栈指针赋值给 %rbp。通过 sub 指令减小栈指针地址从而为局部变量分配空间。然后将 %edi 和 %rsi 寄存器中的值存储在栈中。具体函数分析如下:

main 函数本身的参数读取

参数部分给出了 int argc, char *argv[] 两个参数, 其中 %edi 代表 argc, %rsi 代表 argv[]。

```

26      .cfi_def_cfa_register 6
27      subq    $32, %rsp
28      movl    %edi, -20(%rbp)
29      movq    %rsi, -32(%rbp)

```

然后分配和释放内存

printf 的调用

先传递数据后控制转移

第一次实际上调用的是 puts (只有一个字符串参数)

```

2      leaq    .LC0(%rip), %rdi
3      call    puts@PLT
4      movl    $1, %edi

```

第二次为 printf

```

48      movl    $0, %eax
49      call    printf@PLT

```

exit 函数的调用

exit 状态为 1

```

    exit(1);

```

所以 movl \$1,%edi

```

34    movl    $1, %edi
35    call    exit@PLT

```

sleep 函数的调用

先将 sleepsecs 的值传给%edi, 再调用 sleep@PLT (PLT 后面 PIC 函数动态链接时使用)

```

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

```

getchar 函数的调用:

%eax 置零, call getchar@PLT

```

57    call    getchar@PLT
58    movl    $0, %eax

```

3.3.4 赋值操作

根据不同大小的数据类型有 movb、movw、movl、movq、movabsq

```

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

```

3.3.5 算数操作

1. 算术和逻辑操作类指令分四类: 加载有效地址, 一元操作, 二元操作和移位, 如下:

指令	效果	描述
leaq S, D	$D \leftarrow \&S$	加载有效地址
INC D	$D \leftarrow D + 1$	加1
DEC D	$D \leftarrow D - 1$	减1
NEG D	$D \leftarrow -D$	取负
NOT D	$D \leftarrow \sim D$	取补
ADD S, D	$D \leftarrow D + S$	加
SUB S, D	$D \leftarrow D - S$	减
IMUL S, D	$D \leftarrow D * S$	乘
XOR S, D	$D \leftarrow D \wedge S$	异或
OR S, D	$D \leftarrow D \vee S$	或
AND S, D	$D \leftarrow D \& S$	与
SAL k, D	$D \leftarrow D \ll k$	左移
SHL k, D	$D \leftarrow D \ll k$	左移 (等同于SAL)
SAR k, D	$D \leftarrow D \gg_A k$	算术右移
SHR k, D	$D \leftarrow D \gg_L k$	逻辑右移

2. leaq 指令, 类似 mov 指令, 它左侧的数看似是给出一个地址, 在内存中从给定的地址取操作数, 传给右边的目的地。但其实没有取, 而是直接将左侧的数

对应的地址传给了右侧的目的地。

本代码里面涉及的操作为：

`i++`，对计数器 `i` 自增

`leaq` 操作：

```
46      movq    %rax, %rsi
47      leaq    .LC1(%rip), %rdi
48      movl    %eax, %eax
```

3.3.6 关系操作（比较测试指令）

常见关系操作如下：

`cmp S1,S2` 比较设置条件码

`test S1,S2` 测试设置条件码

`jne/ja/j...` 按照条件码来跳转

此次程序涉及的关系运算为 `cmpl`，分别在判断 `argc` 是否等于 3 和循环变量是否小于 10 时使用，都是利用条件码来跳转。

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

```
56      jle     .L4
```

3.3.7 类型转换

可以看见在声明全局变量的时候有一个类型转换

```
10 int sleepsecs=2.5;
```

当 `int`、`float` 和 `double` 格式之间进行强制类型转换时，程序改变数值和位模式的原则如下：

- (1)从 `int` 转换成 `float`，数字不会溢出，但是可能被舍入。
- (2)从 `int` 或者 `float` 转换成 `double`，因为 `double` 有更大的范围和精度，所以能够保留精确的数值。
- (3)从 `double` 转换成 `float`，因为范围要小一些，所以值可能溢出为 $-\infty$ 或者 $+\infty$ ，另外，由于精度较小，它还可能被舍入。
- (4)从 `float` 或者 `double` 转换成 `int`，值将会向零舍入。

3.4 本章小结

经过编译后，得到的汇编代码文件（如 `hello.s`）还是可读的文本文件，但是 CPU 无法理解和执行它。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

概念：

汇编指令和机器指令一一对应，前者是后者的符号表示，它们都属于机器级指令，所构成的程序称为机器级代码。

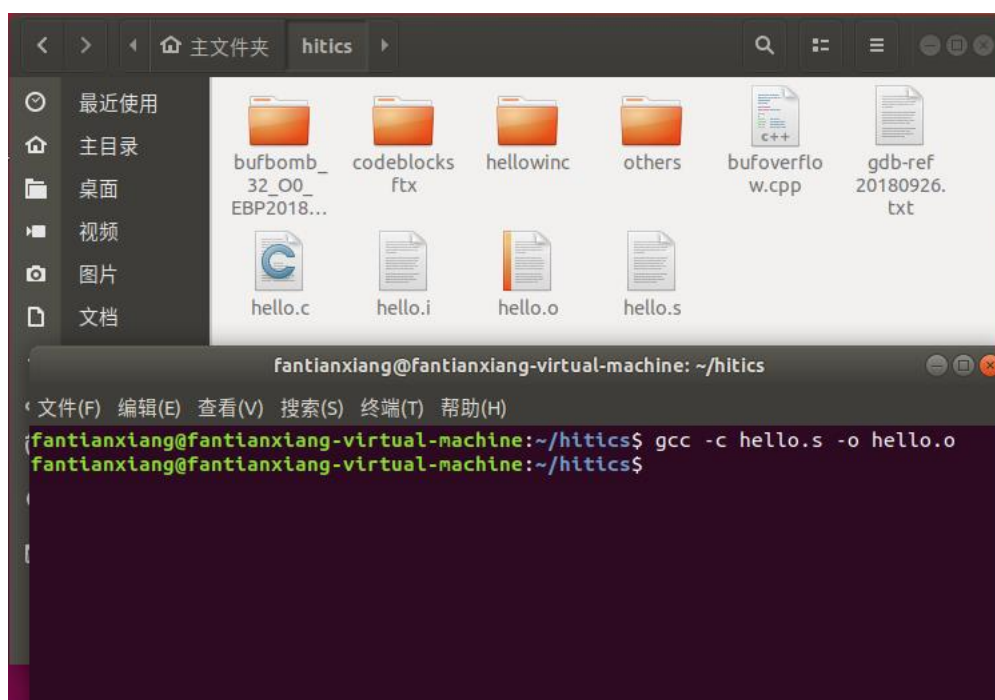
作用：

汇编程序（汇编器 `as`）用来将汇编语言源程序转换为机器指令序列（机器语言程序）。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。

4.2 在 Ubuntu 下汇编的命令

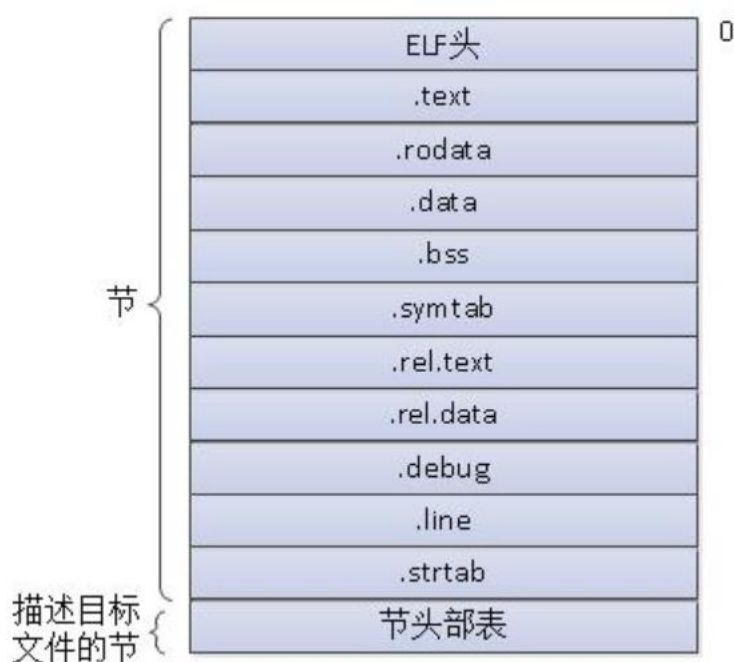
命令行：`linux> gcc -c hello.s -o hello.o`

预处理过程截图：



4.3 可重定位目标 elf 格式

A.很有必要先看一下典型的 ELF 可定位目标文件的格式：



B.分析 hello.o 的 ELF 格式（各节的基本信息）：

1.ELF 头

以 16 字节的序列开始，对应于图中的 magic 部分，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。

```

fantianxiang@fantianxiang-virtual-machine: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
1 ELF 头:
2 Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 类别: ELF64
4 数据: 2 补码, 小端序 (little endian)
5 版本: 1 (current)
6 OS/ABI: UNIX - System V
7 ABI 版本: 0
8 类型: REL (可重定位文件)
9 系统架构: Advanced Micro Devices X86-64
10 版本: 0x1
11 入口点地址: 0x0
12 程序头起点: 0 (bytes into file)
13 Start of section headers: 1160 (bytes into file)
14 标志: 0x0
15 本头的大小: 64 (字节)
16 程序头大小: 0 (字节)
17 Number of program headers: 0
18 节头大小: 64 (字节)
19 节头数量: 13
20 字符串表索引节头: 12
21

```

2.节头表部分：

节头表中条目的大小和数量描述了各个节的含义：节类型，节大小，节地址和偏移量。

```

fantianxiang@fantianxiang-virtual-machine: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
21 节头:
22 [号] 名称      类型      地址      偏移量
23 [号] 大小      全体大小  标志      链接      信息      对齐
24 [ 0]          NULL      0000000000000000 0 0 0
25 [ 1] .text      PROGBITS 0000000000000000 00000040
26 [ 2] .rela.text  RELA      0000000000000000 00000348
27 [ 3] .data      PROGBITS 0000000000000000 000000c4
28 [ 4] .bss       NOBITS    0000000000000000 000000c8
29 [ 5] .rodata     PROGBITS 0000000000000000 000000c8
30 [ 6] .comment   PROGBITS 0000000000000000 000000fa
31 [ 7] .note.GNU-stack  PROGBITS 0000000000000000 00000125
32 [ 8] .eh_frame   PROGBITS 0000000000000000 00000128
33 [ 9] .rela.eh_frame  RELA      0000000000000000 00000408
34 [10] .syntab     SYMTAB    0000000000000000 00000160
35 [11] .strtab     STRTAB    0000000000000000 000002f8
36 [12] .shstrtab   STRTAB    0000000000000000 00000420
37 Key to Flags:
38 W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
39 L (link order), O (extra OS processing required), G (group), T (TLS),
40 C (compressed), x (unknown), o (OS specific), E (exclude),
41 l (large), p (processor specific)
42 There are no section groups in this file.
43 本文件中没有程序头。
44
26.1 33%

```

3.重定位节:

代码的重定位条目，即为图中的.rela.text 节，.rela.text 节包含.text 节的重定位信息，用于重新修改代码段的指令中的地址信息。

首先是各个函数(.rodata-4（第一个 printf 中的字符串）、puts 函数、exit 函数、.rodata+21（第二个 printf 中的字符串）、printf 函数、sleepsecs 函数、sleep 函数、getchar 函数)的:

重定位的偏移量（在.text 节或者是.data 节的偏移位置）;

重定位的信息;

重定位到目标的类型;

重定位符号值和重定位符号。

```

fantianxiang@fantianxiang-virtual-machine: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
61 There is no dynamic section in this file.
62
63 重定位节 '.rela.text' at offset 0x348 contains 8 entries:
64 偏移量      信息      类型      符号值      符号名称 + 加数
65 000000000018 000500000002 R_X86_64_PLT32 0000000000000000 .rodata - 4
66 00000000001d 000c00000004 R_X86_64_PLT32 0000000000000000 puts - 4
67 000000000027 000d00000004 R_X86_64_PLT32 0000000000000000 exit - 4
68 000000000050 000500000002 R_X86_64_PLT32 0000000000000000 .rodata + 21
69 00000000005a 000e00000004 R_X86_64_PLT32 0000000000000000 printf - 4
70 000000000060 000900000002 R_X86_64_PLT32 0000000000000000 sleepsecs - 4
71 000000000067 000f00000004 R_X86_64_PLT32 0000000000000000 sleep - 4
72 000000000076 001000000004 R_X86_64_PLT32 0000000000000000 getchar - 4
73
74 重定位节 '.rela.eh_frame' at offset 0x408 contains 1 entry:
75 偏移量      信息      类型      符号值      符号名称 + 加数
76 000000000020 000200000002 R_X86_64_PLT32 0000000000000000 .text + 0
77
78 The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not currently supported.
79

```

然后就是符号表（此处包含 17 个符号（入口项））:

符号表给出了：

符号的序号 Num、符号的地址（偏移量）Value、符号的大小 Size、符号的类型 Type、符号的名称 Name；

还有：Bind = GLOBAL 绑定意味着该符号对外部文件的可见。LOCAL 绑定意味着该符号只在这个文件中可见。WEAK 有点像 GLOBAL，该符号可以被重载；

（Vis）符号可以是默认的、受保护的、隐藏的或者内部的；

（Ndx）符号所在的区号。ABS 意味着绝对：不调整到任何区段地址的迁移。

```

79
80 Symbol table '.syntab' contains 17 entries:
81   Num:   Value                Size Type   Bind   Vis   Ndx Name
82   0: 0000000000000000      0 NOTYPE LOCAL DEFAULT UND
83   1: 0000000000000000      0 FILE  LOCAL DEFAULT ABS hello.c
84   2: 0000000000000000      0 SECTION LOCAL DEFAULT 1
85   3: 0000000000000000      0 SECTION LOCAL DEFAULT 3
86   4: 0000000000000000      0 SECTION LOCAL DEFAULT 4
87   5: 0000000000000000      0 SECTION LOCAL DEFAULT 5
88   6: 0000000000000000      0 SECTION LOCAL DEFAULT 7
89   7: 0000000000000000      0 SECTION LOCAL DEFAULT 8
90   8: 0000000000000000      0 SECTION LOCAL DEFAULT 6
91   9: 0000000000000000      4 OBJECT GLOBAL DEFAULT 3 sleepsecs
92  10: 0000000000000000    129 FUNC  GLOBAL DEFAULT 1 main
93  11: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
94  12: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND puts
95  13: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND exit
96  14: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND printf
97  15: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND sleep
98  16: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND getchar
99
100 No version information found in this file.

```

4.4 Hello.o 的结果解析

命令行：linux> objdump -d -r hello.o 查看 hello.o 的反汇编
与第 3 章的 hello.s 进行对照分析，如下所示：

```

.file "hello.c"
.text
.globl sleepsecs
.data
.align 4
.type sleepsecs, @object
.size sleepsecs, 4
sleepsecs:
.long 2
.section .rodata
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
.string "Hello %s %s\n"
.text
.globl main
.type main, @function
main:
.LFB5:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movl %edi, -20(%rbp)
movq %rsi, -32(%rbp)
cmpl $3, -20(%rbp)
je .L2
leaq .LC0(%rip), %rdi
call puts@PLT
movl $1, %edi
call exit@PLT
.L2:
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 $0, -4(%rbp)
jne .L4
cmpl $9, -4(%rbp)
jle .L4
call getchar@PLT
movl $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE5:
.size main, .-main
.ident "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
.section .note.GNU-stack,"",@progbits

```

```

fantianxiang@fantianxiang-virtual-machine: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
fantianxiang@fantianxiang-virtual-machine:~/hitics$ objdump -d -r hello.o

hello.o:          文件格式 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>
34: 48 8b 45 e0             mov     -0x20(%rbp),%rax
38: 48 83 c0 10             add     $0x10,%rax
3c: 48 8b 10                mov     (%rax),%rdx
3f: 48 8b 45 e0             mov     -0x20(%rbp),%rax
43: 48 83 c0 08             add     $0x8,%rax
47: 48 8b 00                mov     (%rax),%rax
4a: 48 89 c6                mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 00     lea     0x0(%rip),%rdi        # 54 <main+0x54>
                        50: R_X86_64_PC32      .rodata+0x21
54: b8 00 00 00 00         mov     $0x0,%eax
59: e8 00 00 00 00         callq   5e <main+0x5e>
                        5a: R_X86_64_PLT32      printf-0x4
5e: 8b 05 00 00 00 00       mov     0x0(%rip),%eax        # 64 <main+0x64>
                        60: R_X86_64_PC32      sleepsecs-0x4
64: 89 c7                   mov     %eax,%edi
66: e8 00 00 00 00         callq   6b <main+0x6b>
                        67: R_X86_64_PLT32      sleep-0x4
6b: 83 45 fc 01             addl     $0x1,-0x4(%rbp)
6f: 83 7d fc 09            cmpl     $0x9,-0x4(%rbp)
73: 7e bf                   jle     34 <main+0x34>
75: e8 00 00 00 00         callq   7a <main+0x7a>
                        76: R_X86_64_PLT32      getchar-0x4
7a: b8 00 00 00 00         mov     $0x0,%eax
7f: c9                      leaveq   %rax
80: c3                      retq

fantianxiang@fantianxiang-virtual-machine:~/hitics$ 

```

4.4.1 机器语言的构成与汇编语言的映射关系:

这个就是一些汇编指令对应着相应的机器编码（由于操作数类型、寻址方式等的不同，同一个汇编语言符号可能对应着不同的机器语言操作码。），以及立即数和地址等组成的机器语言。当然要注意在看反汇编时注意小端序。

4.4.2 分支转移:

反汇编和 gcc 编译得到的汇编语言代码都有具体详尽的程序实现基本代码，反汇编代码通过地址表示跳转的目标，而 gcc 得到的是通过抽象的形式，例如.L2 来表示跳转的目标。这里的抽象形式.L2 是一种段名称，作为助记符便于编写，在编译为机器语言后当然要转化为地址模式，未链接之前一般是 00 00 00 00 来填充。

4.4.3 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。需要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。

反汇编的结果中包含了供对照的来自可重定位目标文件中的机器语言代码及相关注释，包括一些相对寻址的信息与重定位信息

4.5 本章小结

汇编结果是一个可重定位目标文件（如：hello.o）（其中包含的是不可读的二进制代码，必须用相应的工具软件来查看其内容）。

当然，生成最终的可执行文件还需要经过链接这一步。

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

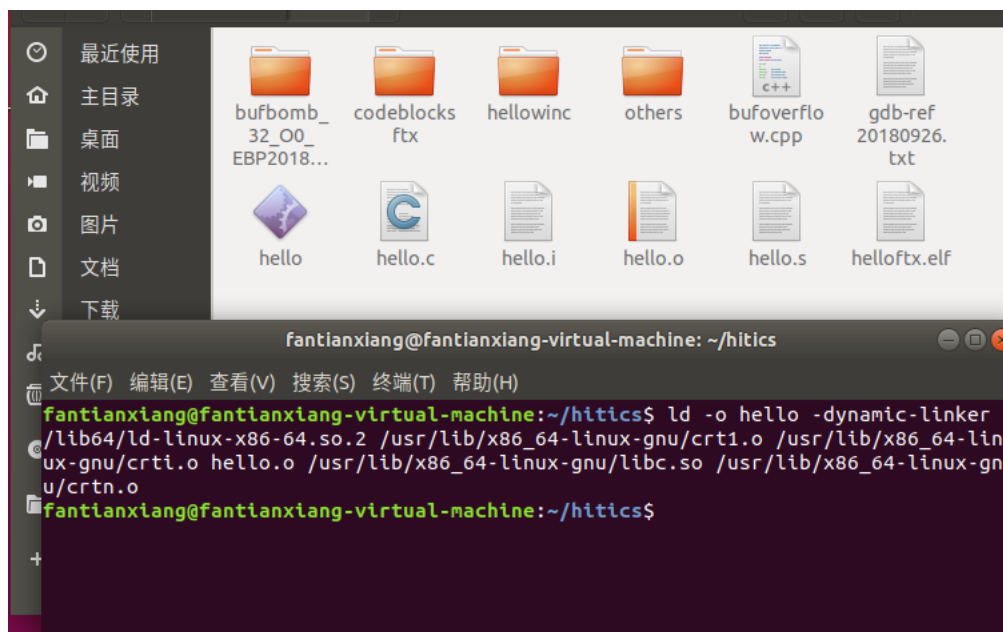
概念：预处理、编译和汇编三个阶段针对一个模块（一个*.c 文件）进行处理，得到对应的一个可重定位目标文件（一个*.o 文件）。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。

作用：链接过程将多个可重定位目标文件合并以生成可执行目标文件

5.2 在 Ubuntu 下链接的命令

命令行 `linux> 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`

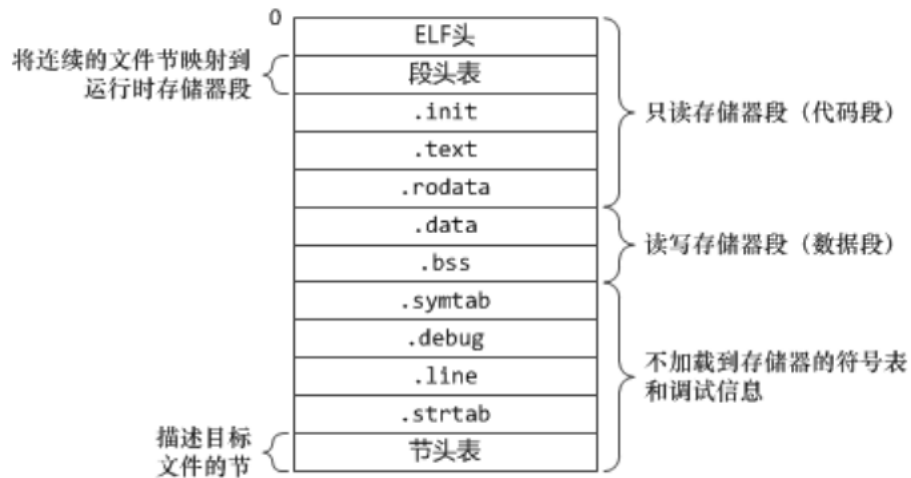
使用 `ld` 的链接命令，如图所示：



5.3 可执行目标文件 hello 的格式

命令行: `linux> readelf -a hello`

首先看一下可执行目标文件的格式：



再看 hello 的 ELF 具体的形式:

a.hello.out 文件的文件头

基本信息:有头的大小、程序头的大小、节头的大小、节头的数量、字符串索引节头等信息。当然也显示这是一个可执行文件。

```

fantianxiang@fantianxiang-virtual-machine:~/hitics$ readelf -a hello
ELF 头:
  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  类别:      ELF64
  数据:      2 补码, 小端序 (little endian)
  版本:      1 (current)
  OS/ABI:    UNIX - System V
  ABI 版本:  0
  类型:      EXEC (可执行文件)
  系统架构:  Advanced Micro Devices X86-64
  版本:      0x1
  入口点地址: 0x400500
  程序头起点: 64 (bytes into file)
  Start of section headers: 5928 (bytes into file)
  标志:      0x0
  本头的大小: 64 (字节)
  程序头大小: 56 (字节)
  Number of program headers: 8
  节头大小:  64 (字节)
  节头数量:  25
  字符串表索引节头: 24
  
```

b.段头表(there are 25 section headers , starting at offset 0x1728):

```
fantianxiang@fantianxiang-virtual-machine:~/hitics$ readelf -s hello
There are 25 section headers, starting at offset 0x1728:
```

节头:

[号]	名称 大小	类型 全体大小	地址 旗标	链接 链接	信息 信息	偏移量 对齐
[0]	0000000000000000	NULL	0000000000000000	0	0	0
[1]	.interp 000000000000001c	PROGBITS 0000000000000000	000000000400200 A	0	0	1
[2]	.note.ABI-tag 0000000000000020	NOTE 0000000000000000	00000000040021c A	0	0	4
[3]	.hash 0000000000000034	HASH 0000000000000004	000000000400240 A	5	0	8
[4]	.gnu.hash 000000000000001c	GNU_HASH 0000000000000000	000000000400278 A	5	0	8
[5]	.dynsym 00000000000000c0	DYNSYM 0000000000000018	000000000400298 A	6	1	8
[6]	.dynstr 0000000000000057	STRTAB 0000000000000000	000000000400358 A	0	0	1
[7]	.gnu.version 0000000000000010	VERSYM 0000000000000002	0000000004003b0 A	5	0	2
[8]	.gnu.version_r 0000000000000020	VERNEED 0000000000000000	0000000004003c0 A	6	1	8
[9]	.rela.dyn 0000000000000030	RELA 0000000000000018	0000000004003e0 A	5	0	8
[10]	.rela.plt 0000000000000078	RELA 0000000000000018	000000000400410 AI	5	19	8
[11]	.init 0000000000000017	PROGBITS 0000000000000000	000000000400488 AX	0	0	4
[12]	.plt 0000000000000060	PROGBITS 0000000000000010	0000000004004a0 AX	0	0	16
[13]	.text 0000000000000132	PROGBITS 0000000000000000	000000000400500 AX	0	0	16
[14]	.fini 0000000000000009	PROGBITS 0000000000000000	000000000400634 AX	0	0	4
[15]	.rodata 000000000000003a	PROGBITS 0000000000000000	000000000400640 A	0	0	8
[16]	.eh_frame 00000000000000fc	PROGBITS 0000000000000000	000000000400680 A	0	0	8
[17]	.dynamic 00000000000001a0	DYNAMIC 0000000000000010	000000000600e50 WA	6	0	8
[18]	.got 0000000000000010	PROGBITS 0000000000000008	000000000600ff0 WA	0	0	8
[19]	.got.plt 0000000000000040	PROGBITS 0000000000000008	000000000601000 WA	0	0	8
[20]	.data 0000000000000008	PROGBITS 0000000000000000	000000000601040 WA	0	0	4
[21]	.comment 000000000000002a	PROGBITS 0000000000000001	MS	0	0	1

c.符号表（符号表的详细信息在之前已经解释过了）：

```
fantianxiang@fantianxiang-virtual-machine:~/hitics$ readelf -s hello
Symbol table '.dynsym' contains 8 entries:
Num:  Value      Size Type Bind  Vis      Ndx Name
 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
 1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.2.5 (2)
 2: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.2.5 (2)
 3: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
 4: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getchar@GLIBC_2.2.5 (2)
 5: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
 6: 0000000000000000 0 FUNC GLOBAL DEFAULT UND exit@GLIBC_2.2.5 (2)
 7: 0000000000000000 0 FUNC GLOBAL DEFAULT UND sleep@GLIBC_2.2.5 (2)
```

包含 49 个入口：

```
Symbol table '.symtab' contains 49 entries:
Num:      Value              Size Type      Bind      Vis      Ndx Name
0: 0000000000000000      0 NOTYPE   LOCAL     DEFAULT   UND
1: 0000000000400200      0 SECTION LOCAL     DEFAULT    1
2: 000000000040021c      0 SECTION LOCAL     DEFAULT    2
3: 0000000000400240      0 SECTION LOCAL     DEFAULT    3
4: 0000000000400278      0 SECTION LOCAL     DEFAULT    4
5: 0000000000400298      0 SECTION LOCAL     DEFAULT    5
6: 0000000000400358      0 SECTION LOCAL     DEFAULT    6
7: 00000000004003b0      0 SECTION LOCAL     DEFAULT    7
8: 00000000004003c0      0 SECTION LOCAL     DEFAULT    8
9: 00000000004003e0      0 SECTION LOCAL     DEFAULT    9
10: 0000000000400410      0 SECTION LOCAL     DEFAULT   10
11: 0000000000400488      0 SECTION LOCAL     DEFAULT   11
12: 00000000004004a0      0 SECTION LOCAL     DEFAULT   12
13: 0000000000400500      0 SECTION LOCAL     DEFAULT   13
14: 0000000000400634      0 SECTION LOCAL     DEFAULT   14
15: 0000000000400640      0 SECTION LOCAL     DEFAULT   15
16: 0000000000400680      0 SECTION LOCAL     DEFAULT   16
17: 0000000000600e50      0 SECTION LOCAL     DEFAULT   17
18: 0000000000600ff0      0 SECTION LOCAL     DEFAULT   18
19: 0000000000601000      0 SECTION LOCAL     DEFAULT   19
```

5.4 hello 的虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息如图所示：

section.interp 内容的虚拟地址：（详细各段信息见本节最后面的终端截图，有各节的名称和对应的详细的虚拟地址空间的信息）

section.text 内容的虚拟地址，开始于 0x400500 这个地方：

对照 5.3 来分析，可执行目标文件各节的基本信息都描述了各节在具体虚拟地址空间的相对位置关系。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。在形成的虚拟地址空间里，我们可以根据这一点来理解各个地址的含义。

```

Contents of section .interp:
400200 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-
400210 7838362d 36342e73 6f2e3200 x86-64.so.2.
Contents of section .note.ABI-tag:
40021c 04000000 10000000 01000000 474e5500 .....GNU.
40022c 00000000 03000000 02000000 00000000 .....
Contents of section .hash:
400240 03000000 08000000 07000000 06000000 .....
400250 04000000 00000000 00000000 00000000 .....
400260 01000000 02000000 00000000 03000000 .....
400270 05000000 ....
Contents of section .gnu.hash:
400278 01000000 01000000 01000000 00000000 .....
400288 00000000 00000000 00000000 .....
Contents of section .dynsym:
400298 00000000 00000000 00000000 00000000 .....
4002a8 00000000 00000000 10000000 12000000 .....
4002b8 00000000 00000000 00000000 00000000 .....
4002c8 15000000 12000000 00000000 00000000 .....
4002d8 00000000 00000000 2a000000 12000000 .....*.....
4002e8 00000000 00000000 00000000 00000000 .....
4002f8 1c000000 12000000 00000000 00000000 .....
400308 00000000 00000000 48000000 20000000 .....H...
400318 00000000 00000000 00000000 00000000 .....
400328 0b000000 12000000 00000000 00000000 .....
400338 00000000 00000000 24000000 12000000 .....$.
400348 00000000 00000000 00000000 00000000 .....
Contents of section .dynstr:
400358 006c6962 632e736f 2e360065 78697400 .libc.so.6.exit.
400368 70757473 00707269 6e746600 67657463 puts.printf.getc
400378 68617200 736c6565 70005f5f 6c696263 har.sleep.__libc
400388 5f737461 72745f6d 61696e00 474c4942 _start_main.GLIB
400398 435f322e 322e3500 5f5f676d 6f6e5f73 C_2.2.5.__gmon_s
4003a8 74617274 5f5f00 tart__.
Contents of section .gnu.version:
4003b0 00000200 02000200 02000000 02000200 .....
Contents of section .gnu.version_r:

```

```

400380 74017274 513100 Contents of section .gnu.version:
4003b0 00000200 02000200 02000000 02000200 .....
Contents of section .gnu.version_r:
4003c0 01000100 01000000 10000000 00000000 .....
4003d0 751a6909 00000200 3c000000 00000000 u.i.....<.....
Contents of section .rela.dyn:
4003e0 f00f6000 00000000 06000000 03000000 ..`.....
4003f0 00000000 00000000 f80f6000 00000000 .....`.....
400400 06000000 05000000 00000000 00000000 .....
Contents of section .rela.plt:
400410 18106000 00000000 07000000 01000000 ..`.....
400420 00000000 00000000 20106000 00000000 .....`.....
400430 07000000 02000000 00000000 00000000 .....
400440 28106000 00000000 07000000 04000000 (.`.....
400450 00000000 00000000 30106000 00000000 .....0.`.....
400460 07000000 06000000 00000000 00000000 .....
400470 38106000 00000000 07000000 07000000 8.`.....
400480 00000000 00000000 .....
Contents of section .init:
400488 4883ec08 488b0565 0b200048 85c07402 H...H..e..H..t.
400498 ffd04883 c408c3 ..H....
Contents of section .plt:
4004a0 ff35620b 2000ff25 640b2000 0f1f4000 .5b. ..%d. ...@.
4004b0 ff25620b 20006800 000000e9 e0ffffff .%b. .h.....
4004c0 ff255a0b 20006801 000000e9 d0ffffff .%Z. .h.....
4004d0 ff25520b 20006802 000000e9 c0ffffff .%R. .h.....
4004e0 ff254a0b 20006803 000000e9 b0ffffff .%J. .h.....
4004f0 ff25420b 20006804 000000e9 a0ffffff .%B. .h.....
Contents of section .text:
400500 31ed4989 d15e4889 e24883e4 f0505449 1.I..^H..H...PTI
400510 c7c03006 400048c7 c1c00540 0048c7c7 ..0.@.H....@.H..
400520 32054000 ff15c60a 2000f40f 1f440000 2.@.....D..
400530 f3c35548 89e54883 ec20897d ec488975 ..UH..H..}.H.u
400540 e0837dec 03741648 8d3dfa00 0000e85d ..}.t.H.=....]
400550 ffffffff 01000000 e883ffff ffc745fc .....E.
400560 00000000 eb3b488b 45e04883 c010488b .....;H.E.H...H.
400570 10488b45 e04883c0 08488b00 4889c648 .H.E.H...H..H..H
400580 8d3de700 0000b800 000000e8 30ffffff .=.....0...
400590 8b05ae0a 200089c7 e853ffff ff8345fc ....S...E.

```

```

Contents of section .text:
400634 4883ec08 4883c408 c3 H...H...
Contents of section .rodata:
400640 01000200 00000000 55736167 653a2048 .....Usage: H
400650 656c6c6f 20313137 30333030 38313520 ello 1170300815
400660 e88c83e5 a4a9e7a5 a5efbc81 0048656c .....Hel
400670 6c6f2025 73202573 0a00 lo %s %s..
Contents of section .eh_frame:
400680 14000000 00000000 017a5200 01781001 .....zR..x..
400690 1b0c0708 90010710 10000000 1c000000 .....
4006a0 60feffff 2b000000 00000000 14000000 `...+.....
4006b0 00000000 017a5200 01781001 1b0c0708 .....zR..x.....
4006c0 90010000 10000000 1c000000 64feffff .....d...
4006d0 02000000 00000000 24000000 30000000 .....$.0...
4006e0 c0fdffff 60000000 000e1046 0e184a0f ....`.....F..J.
4006f0 0b770880 003f1a3b 2a332422 00000000 .w...?.;*3$"....
400700 1c000000 58000000 2afeffff 81000000 ....X...*.....
400710 00410e10 8602430d 06027c0c 07080000 .A....C...|....
400720 44000000 78000000 98feffff 65000000 D...x.....e...
400730 00420e10 8f02420e 188e0345 0e208d04 .B....B....E. ..
400740 420e288c 05480e30 8606480e 3883074d B(..H.0..H.8..M
400750 0e40720e 38410e30 410e2842 0e20420e .@r.8A.0A.(B. B.
400760 18420e10 420e0800 10000000 c0000000 .B..B.....
400770 c0feffff 02000000 00000000 .....
Contents of section .dynamic:
600e50 01000000 00000000 01000000 .....
600e60 0c000000 00000000 88044000 .....@.....
600e70 0d000000 00000000 34064000 .....4.@.....
600e80 04000000 00000000 40024000 .....@.@.....
600e90 f5feff6f 00000000 78024000 .....O....x.@.....
600ea0 05000000 00000000 58034000 .....X.@.....
600eb0 06000000 00000000 98024000 .....@.....
600ec0 0a000000 00000000 57000000 .....W.....
600ed0 0b000000 00000000 18000000 .....
600ee0 15000000 00000000 00000000 .....
600ef0 03000000 00000000 00106000 .....
600f00 02000000 00000000 78000000 .....x.....
600f10 14000000 00000000 07000000 .....
600f20 17000000 00000000 10044000 .....@.....
600f30 07000000 00000000 e0034000 .....@.....
600f40 08000000 00000000 30000000 .....0.....
600f50 09000000 00000000 18000000 .....
600f60 f0ffff6f 00000000 c0024000 .....

```

5.5 链接的重定位过程分析

命令行 `linux> objdump -d -r hello`

如图所示：


```

fantianxiang@fantianxiang-virtual-machine: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
fantianxiang@fantianxiang-virtual-machine:~/hitics$ objdump -d -r hello

hello:      文件格式 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      # 600ff8 <__gmon_start__>
400493:  48 85 c0             test   %rax,%rax
400496:  74 02              je     40049a <_init+0x12>
400498:  ff d0             callq  *%rax
40049a:  48 83 c4 08          add    $0x8,%rsp
40049e:  c3                retq

Disassembly of section .plt:

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)

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>
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>

Disassembly of section .text:

```

```

Disassembly of section .text:

000000000400500 <_start>:
400500: 31 ed                xor    %ebp,%ebp
400502: 49 89 d1             mov    %rdx,%r9
400505: 5e                  pop    %rsi
400506: 48 89 e2             mov    %rsp,%rdx
400509: 48 83 e4 f0          and    $0xfffffffffffffff0,%rsp
40050d: 50                  push   %rax
40050e: 54                  push   %rsp
40050f: 49 c7 c0 30 06 40 00 mov    $0x400630,%r8
400516: 48 c7 c1 c0 05 40 00 mov    $0x4005c0,%rcx
40051d: 48 c7 c7 32 05 40 00 mov    $0x400532,%rdi
400524: ff 15 c6 0a 20 00    callq *0x200ac6(%rip)          # 600ff0 <__libc_start_main@GLIBC_2.
2.5>
40052a: f4                  hlt
40052b: 0f 1f 44 00 00      nopl   0x0(%rax,%rax,1)

000000000400530 <_dl_relocate_static_pie>:
400530: f3 c3              repz retq

000000000400532 <main>:
400532: 55                  push   %rbp
400533: 48 89 e5             mov    %rsp,%rbp
400536: 48 83 ec 20          sub    $0x20,%rsp
40053a: 89 7d ec             mov    %edi,-0x14(%rbp)
40053d: 48 89 75 e0          mov    %rsi,-0x20(%rbp)
400541: 83 7d ec 03          cmpl   $0x3,-0x14(%rbp)
400545: 74 16               je     40055d <main+0x2b>
400547: 48 8d 3d fa 00 00 00 lea     0xfa(%rip),%rdi          # 400648 <_IO_stdin_used+0x8>
40054e: e8 5d ff ff ff      callq 4004b0 <puts@plt>
400553: bf 01 00 00 00      mov    $0x1,%edi
400558: e8 83 ff ff ff      callq 4004e0 <exit@plt>
40055d: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
400564: eb 3b               jmp     4005a1 <main+0x6f>
400566: 48 8b 45 e0          mov    -0x20(%rbp),%rax
40056a: 48 83 c0 10          add     $0x10,%rax
40056e: 48 8b 10             mov    (%rax),%rdx
400571: 48 8b 45 e0          mov    -0x20(%rbp),%rax
400575: 48 83 c0 08          add     $0x8,%rax
400579: 48 8b 00             mov    (%rax),%rax
40057c: 48 89 c6             mov    %rax,%rsi
40057f: 48 8d 3d e7 00 00 00 lea     0xe7(%rip),%rdi          # 40066d <_IO_stdin_used+0x2d>
400586: b8 00 00 00 00      mov    $0x0,%eax
40058b: e8 30 ff ff ff      callq 4004c0 <printf@plt>
400590: 8b 05 ae 0a 20 00    mov    0x200aae(%rip),%eax          # 601044 <sleepsecs>
400596: 89 c7               mov    %eax,%edi
400598: e8 53 ff ff ff      callq 4004f0 <sleep@plt>
40059d: 83 45 fc 01          addl    $0x1,-0x4(%rbp)
4005a1: 83 7d fc 09          cmpl    $0x9,-0x4(%rbp)

```

hello 与 hello.o 的不同:

hello.o 没有经过链接, 所以 main 的地址从 0 开始, 并且不存在调用的如 printf 这样函数的代码。另外, 很多地方都有重定位标记, 用于后续的链接过程。hello.o 反汇编代码的相对寻址部分的地址也不具有参考性, 没有经过链接并不准确。而对于 hello 这个已经链接好的可执行目标文件来说, 库函数的代码都已经链接到了程序中, 程序各个节变得更加完整, 跳转的地址也具有参考性。

结合 hello.o 的重定位项目, 分析 hello 中对其怎么重定位的:

使用 ld 命令链接的时候, 将 hello.c 中用到的函数都加入链接器, 当链接器解析重定条目时发现对外部函数调用, 比如说:

```

5e: 8b 05 00 00 00 00    mov    0x0(%rip),%eax          # 64 <main+0x64>
60: R_X86_64_PC32      sleepsecs-0x4
64: 89 c7               mov    %eax,%edi

```

这个是一个使用 32 位 PC 相对地址的引用, 未链接之前是用 00 00 00 00 填充的, 下面及时给出了具体重定位的信息: 符号为 sleepsecs, 偏移量为-0x4, 即加数 raddend 为 0x-4。

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。使用书上的公式计算出结果 00 20 0a ae，观察链接后的填充值，得到验证。（小端序）

```

40058b: e8 30 ff ff ff callq 4004c0 <printf@plt>
400590: 8b 05 ae 0a 20 00 mov     0x200aae(%rip),%eax    # 601044 <sleepsecs>
400596: 89 c7          mov     %eax,%edi

```

5.6 hello 的执行流程

edb 执行 hello 结果如图所示：



加载 **hello**:

ld-2.23.so!_dl_start/ld-2.23.so!_dl_init/LinkAddress!_start/libc-2.23.so!_libc_start_main libc-2.23.so!_cxa_atexit LinkAddress!_libc_csu.init libc-2.23.so!_setjmp

call main : 使用 gdb 单步运行，可以找出.text 节中 main 函数前后执行的函数名称。在 main 函数之前执行的程序有：_start、__libc_start_main@plt、__libc_csu_init、_init、frame_dummy、register_tm_clones。在 main 函数之后执行的程序有：exit、_cxa_thread_atexit_impl、fini。

程序终止：libc-2.23.so!exit

5.7 Hello 的动态链接分析

动态链接时：函数在程序执行的时候才会确定地址。GNU 编译系统采用延迟绑定技术来解决动态库函数模块调用的问题，它将过程地址的绑定推迟到了第一次调用该过程时。延迟绑定通过全局偏移量表（GOT）和过程链接表（PLT）实现。如果一个目标模块调用定义在共享库中的任何函数，那么就有自己的 GOT 和 PLT。当某个动态链接函数第一次被调用时先进入对应的 PLT 条目例如 PLT[2]，然后 PLT 指令跳转到对应的 GOT 条目中例如 GOT[4]，其内容是 PLT[2]的下一条指令。然后将函数的 ID 压入栈中后跳转到 PLT[0]。PLT[0]通过 GOT[1]将动态链接库的一个参数压入栈中，再通过 GOT[2]间接跳转进动态链接器中。动态链接器使用两个栈条目来确定函数的运行时位置，用这个地址重写 GOT[4]，然后再次调用函数。经过上述操作，再次调用时 PLT[2]会直接跳转通过 GOT[4]跳转到函数而不是 PLT[2]的下一条地址，具体见书上的图例。

先观察调用 **dl_init** 前，动态库函数指向的地址。

```

00:00601000 50 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00 P...
00:00601010 00 00 00 00 00 00 00 00 b6 04 40 00 00 00 00 00 .....
00:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00 |.@...
00:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 |h.@...
00:00601040 00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62 .....
00:00601050 75 6e 74 75 20 37 2e 33 2e 30 2d 32 37 75 62 75 |untu

```

调用 **dl_init** 后再次查看，经过 **dl_init** 的调用，这里已经有了一段地址：

```

00:00601000 50 0e 60 00 00 00 00 00 70 a1 89 48 ac 7f 00 00 P...
00:00601010 50 87 68 48 ac 7f 00 00 b6 04 40 00 00 00 00 00 P.hHa...
00:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00 |.@...
00:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 |h.@...
00:00601040 00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62 .....
00:00601050 75 6e 74 75 20 37 2e 33 2e 30 2d 32 37 75 62 75 |untu

```

5.8 本章小结

链接可以在编译时由静态编译器来完成，也可以在加载时和运行时由动态链接器来完成。

链接器处理称为目标文件的二进制文件，它有三种不同的形式：可重定位的、可撕的和共享的：

可重定位的目标文件由静态链接器合并成一个可执行的目标文件，它可以加载到存储器中并执行。

共享目标文件（共享库）是在运行时由动态链接器链接和加载的，或者隐含地在调用程序被加载和开始执行时或者根据需要在程序调用 `dlopen` 库的函数时。

链接器的两个主要任务是符号解析和重定位，符号解析将目标文件中的每个全局符号都绑定到一个唯一的定义，而重定位确定每个符号的最终存储器地址，并修改对那些目标的引用。

多个目标文件可以定义相同的符号，而链接器用来悄悄地解析这些多重定义的规则可能在用户程序中引入的微妙错误。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：

进程的经典定义就是：一个执行中程序的实例。

一也就是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，是系统进行资源分配和调度运行的基本单位。

系统中的每个程序都运行在某个进程的上下文中，一般情况下，包括文本区域（执行代码）、数据区域、和堆栈等。

作用：

为用户提供了假象：我们的程序好像是系统唯一运行的程序，单独占用内存和处理器。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：代表用户运行其他程序。

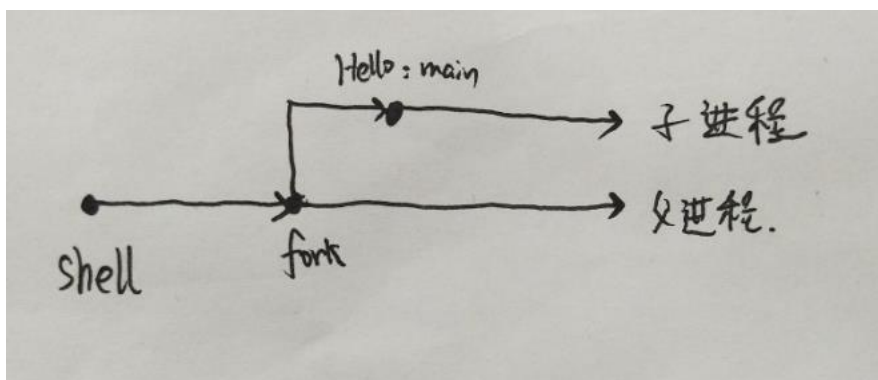
处理流程：

- 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

Shell 通过调用 fork 函数创建一个新的运行的子进程，也就是 Hello 程序，Hello 进程几乎但不完全与 Shell 相同。Hello 进程得到与 Shell 用户级虚拟地址空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。Hello 进程还获得与 Shell 任何打开文件描述符相同的副本，这就意味着当 Shell 调用 fork 时，Hello 可以读写 Shell 中打开的任何文件。Shell 和 Hello 进程之间最大的区别在于它们有不同的 PID。

如图所示：

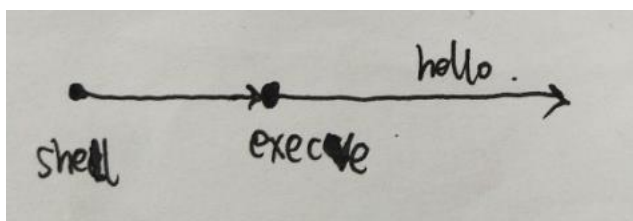


6.4 Hello 的 execve 过程

创建进程后，在子进程中通过判断 `pid` 即 `fork()` 函数的返回值，判断处于子进程，则会通过 `execve` 函数在当前进程的上下文中加载并运行一个新程序。`execve` 函数加载并运行可执行目标文件 `filename`，且带参数列表 `argv` 和环境变量列表 `envp`。只有当出现错误时，例如找不到 `filename`，`execve` 才会返回到调用程序。所以，与 `fork` 一次调用返回两次不同，`execve` 调用一次并不返回。

在 `execve` 加载了可执行程序之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，即可执行程序的主函数。此时用户栈已经包含了命令行参数与环境变量，进入 `main` 函数后便开始逐步运行程序。

具体如图所示：



6.5 Hello 的进程执行

上下文信息：

进程的上下文切换由如下组成：

1) 决定是否作上下文切换以及是否允许作上下文切换。包括对进程调度原因的检查分析，以及当前执行进程的资格和 CPU 执行方式的检查等。在操作系统中，上下文切换程序并不是每时每刻都在检查和分析是否可作上下文切换，它们设置有适当的时机。(2) 保存当前执行进程的上下文。这里所说的当前执行进程，实际上是指调用上下文切换程序之前的执行进程。如果上下文切换不是被那个当前执行进程所调用，且不属于该进程，则所保存的上下文应是先前执行进程的上下文，或称为“老”进程上下文。显然，上下文切换程序不能破坏“老”进程的上下

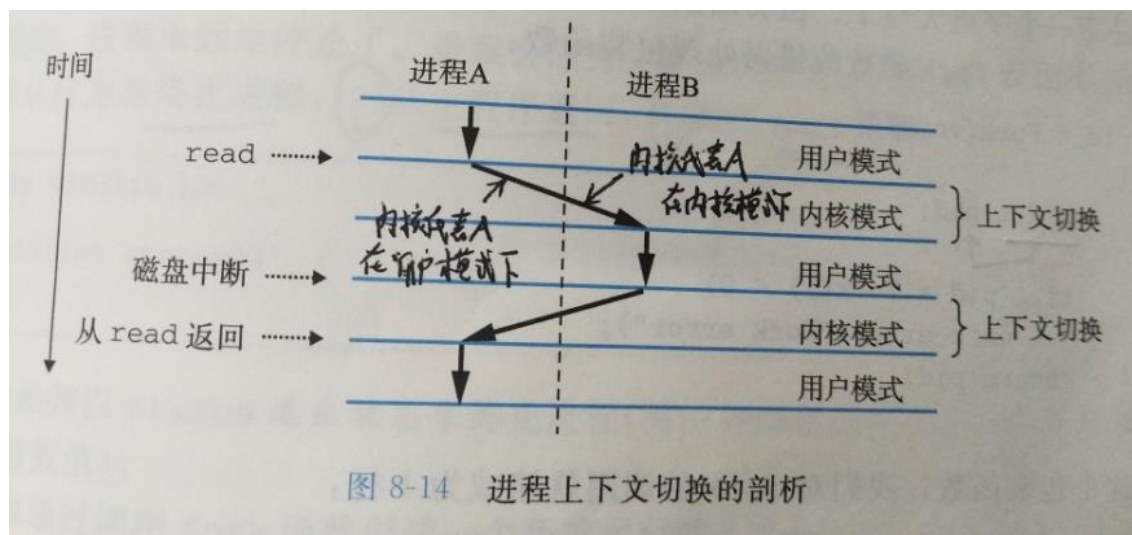
下文结构。(3) 使用进程调度算法, 选择一处于就绪状态的进程。(4) 恢复或装配所选进程的上下文, 将 CPU 控制权交到所选进程手中。

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。

用户态与内核态转换:

首先看一下书上的用户态与内核态转换的解释(中断处理):



在处理操作系统陷入时,

- 1, 内核根据系统调用号查系统调用入口表, 找到相应的内核子程序的地址。
- 2, 内核还要确定该系统调用所要求的参数个数。
- 3, 从用户地址空间拷贝参数到 U 区 (Unix V)。
- 4, 保存当前上下文, 执行系统调用代码。

核心态: 当 CPU 正在运行内核代码时 (内核代码是共享的)。

用户态: 当 CPU 正在运行用户代码时。

用户模式: 不可以访问内核空间 ($\geq 0x80000000$)

内核模式: 可以访问任何有效虚拟地址, 包括内核空间。一个线程可以访问其他任何线程地址空间。

对于 hello 程序来说:

子进程通过 `execve` 系统调用启动加载器。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(chunk), 新的代码和数据段被初始化为可执行文件的内容。最后, 加载器跳转到 `_start` 地址, 它最终会调用应用程序的 `main` 函数。

再看 hello 程序执行时对 `sleep` 函数的分析:

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。

6.6 hello 的异常与信号处理

6.6.1 hello 执行过程中会出现哪几类异常：

Hello 在执行的过程中，可能会出现处理器外部 I/O 设备引起的异常，执行指令导致的陷阱、故障和终止。第一种被称为外部异常，常见的有时钟中断、外部设备的 I/O 中断等。第二种被称为同步异常。陷阱指的是有意的执行指令的结果，故障是非有意的可能被修复的结果，而终止是非故意的不可修复的致命错误。

6.6.2 会产生哪些信号，又怎么处理的：

在发生异常时会产生信号。例如缺页故障会导致 OS 发生 SIGSEGV 信号给用户进程，而用户进程以段错误退出。

中断：SIGSTP:挂起程序

终止：SIGINT:终止程序

6.6.3 各命令及运行结截屏：

Ctrl-C:直接终止进程：

```
fantianxiang@fantianxiang-virtual-machine:~/hitics$ ./hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
^C
fantianxiang@fantianxiang-virtual-machine:~/hitics$ ps
  PID TTY          TIME CMD
  7476 pts/0        00:00:00 bash
  7891 pts/0        00:00:00 ps
fantianxiang@fantianxiang-virtual-machine:~/hitics$ fg
bash: fg: 当前: 无此任务
fantianxiang@fantianxiang-virtual-machine:~/hitics$
```

Ctrl-Z:如果在程序运行过程中输入 Ctrl-Z，那么会发送一个 SIGTSTP 信号给前台进程组中的进程，从而将其挂起：

```
fantianxiang@fantianxiang-virtual-machine:~/hitics$ ./hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
^Z
[1]+  已停止                  ./hello 1170300815 范天祥
fantianxiang@fantianxiang-virtual-machine:~/hitics$
```

回车程序会忽略：

```

fantianxiang@fantianxiang-virtual-machine:~/hitics$ ./hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥

Hello 1170300815 范天祥

Hello 1170300815 范天祥

Hello 1170300815 范天祥

```

随机按键盘：（不响应）

```

fantianxiang@fantianxiang-virtual-machine:~/hitics$ fg
./hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
sdaHello 1170300815 范天祥
dasHello 1170300815 范天祥
fhdgfg Hello 1170300815 范天祥
dfgds v Hello 1170300815 范天祥

```

ps: 查看进程及其运行时间

```

fantianxiang@fantianxiang-virtual-machine:~/hitics$ ps
  PID TTY          TIME CMD
  7476 pts/0        00:00:00 bash
  7634 pts/0        00:00:00 hello
  7719 pts/0        00:00:00 ps
fantianxiang@fantianxiang-virtual-machine:~/hitics$

```

jobs: 查看当前暂停的进程

```

fantianxiang@fantianxiang-virtual-machine:~/hitics$ jobs
[1]+  已停止                  ./hello 1170300815 范天祥
fantianxiang@fantianxiang-virtual-machine:~/hitics$

```

pstree（部分截图）：

```

fantianxiang@fantianxiang-virtual-machine:~/hitics$ pstree
systemd--ManagementAgent--6*[{ManagementAgent}]
      |--ModemManager--2*[{ModemManager}]
      |--NetworkManager--dhclient
      |                   |
      |                   +--2*[{NetworkManager}]
      |--VGAAuthService
      |--accounts-daemon--2*[{accounts-daemon}]
      |--acpid
      |--avahi-daemon--avahi-daemon
      |--bluetoothd
      |--boltd--2*[{boltd}]
      |--colord--2*[{colord}]
      |--cron
      |--cups-browsed--2*[{cups-browsed}]
      |--cupsd--dbus
      |--2*[dbus-daemon]
      |--fcitx--{fcitx}
      |--fcitx-dbus-watc
      |--fwupd--4*[{fwupd}]
      |--gdm3
      |--gdm-session-wor
      |--gdm-wayland-ses
      |--gnome-session-b
      |--gnome-sh+
      |--gsd-a11y+
      |--gsd-clip+
      |--gsd-colo+
      |--gsd-date+
      |--gsd-hous+
      |--gsd-keyb+
      |--gsd-medi+
      |--gsd-mous+
      |--gsd-powe+
      |--gsd-prin+
      |--gsd-rfki+
      |--gsd-scre+
      |--gsd-shar+

```


fg: 输入 fg 使进程重新在前台执行

```
wpd_supplicanc
fantianxiang@fantianxiang-virtual-machine:~/hitics$ fg
./hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
Hello 1170300815 范天祥
```

kill: 使用 kill 杀死特定进程

```
fantianxiang@fantianxiang-virtual-machine:~/hitics$ kill -9 8024
[1]+  已杀死                  ./hello 1170300815 范天祥
fantianxiang@fantianxiang-virtual-machine:~/hitics$ ps
  PID TTY          TIME CMD
 7965 pts/0    00:00:00 bash
 8087 pts/0    00:00:00 ps
fantianxiang@fantianxiang-virtual-machine:~/hitics$
```

6.7 本章小结

shell 中执行是通过 fork 函数及 execve 创建新的进程并执行程序。进程拥有与父进程相同却又独立的环境，与其他系统进程并发执行，拥有各自的时间片，在内核的调度下有序进行程序的执行。

在应用层，一个进程可以发送信号到另一个进程，而接收者会将控制突然转移到它的一个信号处理程序。一个程序可以通过回避通常的栈规则，并执行到其他函数中任意位置的非本地跳转来对错误做出反应。

异常分为中断、陷阱、故障和终止四类，均有对应的处理方法。操作系统提供了信号这一机制，实现了异常的反馈。这样，程序能够对不同的信号调用信号处理子程序进行处理。

(第 6 章 1 分)

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址概念：

逻辑地址是指由程序产生的与段相关的偏移地址部分。

对应于 hello.o 里面的相对偏移地址。

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。是逻辑地址到物理地址变换之间的中间层。程式代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为 4G（2 的 32 次方即 32 根地址总线寻址）。

虚拟地址：

现代操作系统都提供了一种内存管理的抽象，即虚拟内存。进程使用虚拟内存中的地址，即虚拟地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。hello.s 中使用的就是虚拟空间的虚拟地址。

物理地址：

在实地址方式下，物理地址是通过段地址乘以 16 加上偏移地址得到的。而 16 位的段地址乘以 16 等同于左移 4 位二进制位，这样变成 20 位的段基地址，最后段基地址加上段内偏移地址即可得到物理地址。

20 位物理地址计算方法：物理地址=段地址*16d+偏移地址。

是指出目前 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

机器语言指令中出现的内存地址，都是逻辑地址，需要转换成线性地址，再经过 MMU(CPU 中的内存管理单元)转换成物理地址才能够被访问到。

在 x86 保护模式下，段的信息（段基线性地址、长度、权限等）即段描述符占 8 个字节，段信息无法直接存放在段寄存器中（段寄存器只有 2 字节）。Intel 的设计是段描述符集中存放在 GDT 或 LDT 中，而段寄存器存放的是段描述符在 GDT 或 LDT 内的索引值(index)。

逻辑地址，必须加上隐含的 DS 数据段的基地址，才能构成线性地址。具体如下所示：

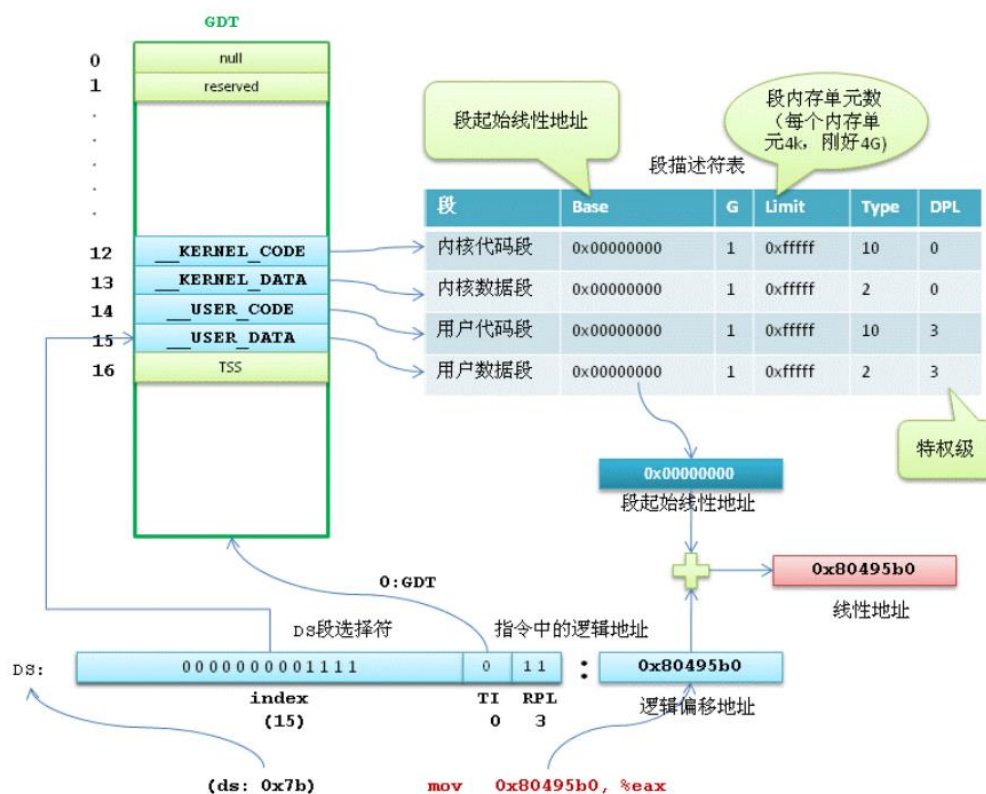


图1 逻辑地址转线性地址

7.3 Hello 的线性地址到物理地址的变换-页式管理

在保护模式下,控制寄存器CR0的最高位PG位控制着分页管理机制是否生效,如果PG=1,分页机制生效,需通过页表查找才能把线性地址转换物理地址。如果PG=0,则分页机制无效,线性地址就直接做为物理地址。

分页的基本原理是把内存划分成大小固定的若干单元,每个单元称为一页(page),每页包含4k字节的地址空间(为简化分析,我们不考虑扩展分页的情况)。这样每一页的起始地址都是4k字节对齐的。为了能转换成物理地址,我们需要给CPU提供当前任务的线性地址转物理地址的查找表,即页表(page table)。注意,为了实现每个任务的平坦的虚拟内存,每个任务都有自己的页目录表和页表。

为了节约页表占用的内存空间,x86将线性地址通过页目录表和页表两级查找转换成物理地址。

32位的线性地址被分成3个部分:

最高10位Directory是页目录表偏移量,中间10位Table是页表偏移量,最低12位Offset是物理页内的字节偏移量。

页目录表的大小为4k(刚好是一个页的大小),包含1024项,每个项4字节(32

位)，项目里存储的内容就是页表的物理地址。如果页目录表中的页表尚未分配，则物理地址填 0。

页表的大小也是 4k，同样包含 1024 项，每个项 4 字节，内容为最终物理页的物理内存起始地址。

具体实现如图所示：

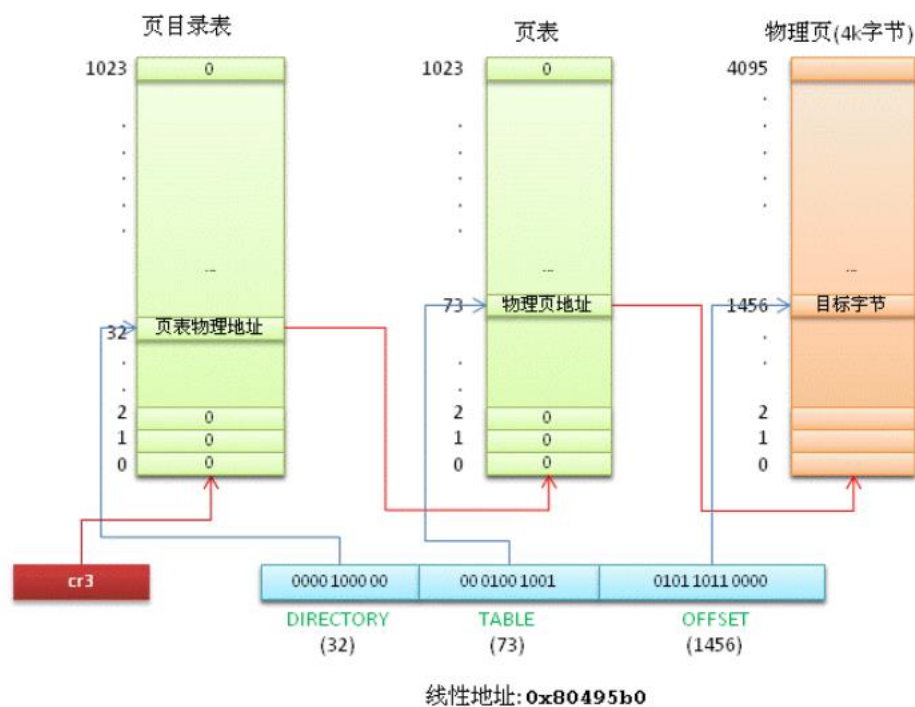


图2 线性地址转物理地址 <http://blog.csdn.net/Aryang>

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

页表一般都很大，并且存放在内存中，所以处理器引入 MMU 后，读取指令、数据需要访问两次内存：首先通过查询页表得到物理地址，然后访问该物理地址读取指令、数据。为了减少因为 MMU 导致的处理器性能下降，引入了 TLB，TLB 是 Translation Lookaside Buffer 的简称，可翻译为“地址转换后援缓冲器”，也可简称为“快表”。简单地说，TLB 就是页表的 Cache，其中存储了当前最可能被访问到的页表项，其内容是部分页表项的一个副本。只有在 TLB 无法完成地址翻译任务时，才会到内存中查询页表，这样就减少了页表查询导致的处理器性能下降。

TLB 中的项由两部分组成：标识和数据。标识中存放的是虚地址的一部分，而数据部分中存放物理页号、存储保护信息以及其他一些辅助信息。虚地址与 TLB 中项的映射方式有三种：全关联方式、直接映射方式、分组关联方式。

Core i7 MMU 使用四级的页表将虚拟地址翻译成物理地址。36 位 VPN 被划分成四个 9 位 VPN，分别用于一个页表的偏移量。具体结构如图所示。

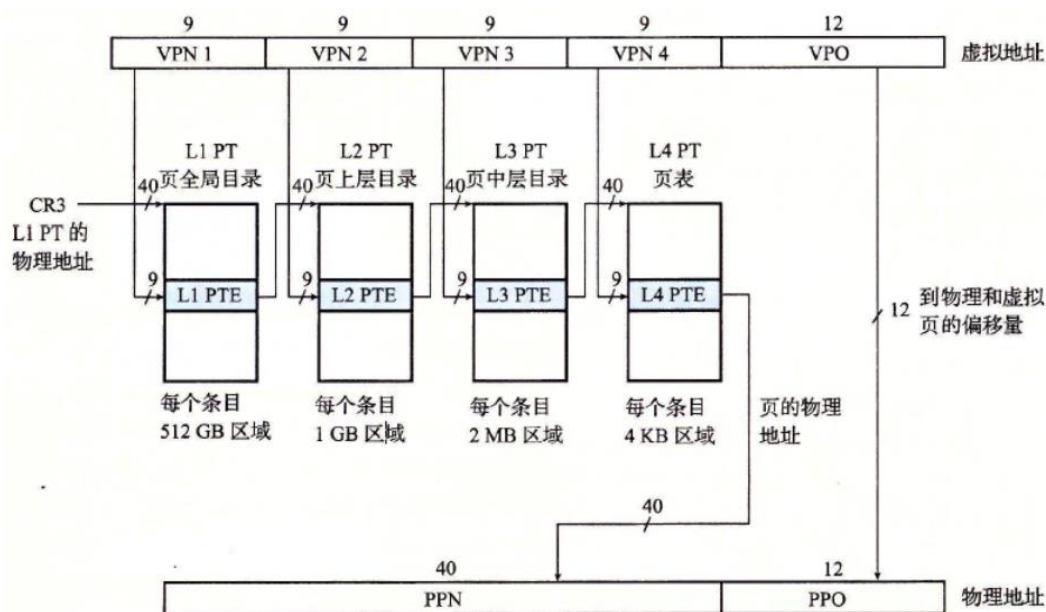


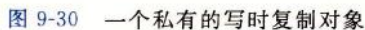
图 9-25 Core i7 页表翻译 (PT: 页表, PTE: 页表条目, VPN: 虚拟页号, VPO: 虚拟页偏移, PPN: 物理页号, PPO: 物理页偏移量。图中还给出了这四级页表的 Linux 名字)

7.5 三级 Cache 支持下的物理内存访问

首先 CPU 发出一个虚拟地址，在 TLB 里面寻找。如果命中，那么将 PTE 发送给 L1Cache，否则先在页表中更新 PTE。然后再进行 L1 根据 PTE 寻找物理地址，检测是否命中的工作。这样就能完成 Cache 和 TLB 的配合工作。具体流程如图所示：

shell 通过 `fork` 为 `hello` 创建新进程。当 `fork` 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给 `hello` 进程唯一的 `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`、区域结构和样表的原样副本。它将两个进程中的每个页面都标记为只读，并将每个进程中的每个区域结构都标记为写时复制。在新进程中返回时，新进程拥有与调用 `fork` 进程相同的虚拟内存，随后的写操作通过写时复制机制创建新页面。

第 1 次 11 月 7 日 第 2 次 11 月 28 日 第 3 次 12 月 12 日 第 4 次 12 月 26 日 第 5 次 1 月 9 日 第 6 次 1 月 23 日 第 7 次 2 月 6 日 第 8 次 2 月 20 日 第 9 次 3 月 6 日 第 10 次 3 月 20 日 第 11 次 4 月 3 日 第 12 次 4 月 17 日 第 13 次 4 月 30 日 第 14 次 5 月 14 日 第 15 次 5 月 28 日 第 16 次 6 月 11 日 第 17 次 6 月 25 日 第 18 次 7 月 9 日 第 19 次 7 月 23 日 第 20 次 8 月 6 日 第 21 次 8 月 20 日 第 22 次 9 月 3 日 第 23 次 9 月 17 日 第 24 次 9 月 30 日 第 25 次 10 月 14 日 第 26 次 10 月 28 日 第 27 次 11 月 11 日 第 28 次 11 月 25 日 第 29 次 12 月 9 日 第 30 次 12 月 23 日 第 31 次 1 月 6 日 第 32 次 1 月 20 日 第 33 次 2 月 3 日 第 34 次 2 月 17 日 第 35 次 2 月 27 日 第 36 次 3 月 13 日 第 37 次 3 月 27 日 第 38 次 4 月 10 日 第 39 次 4 月 24 日 第 40 次 5 月 8 日 第 41 次 5 月 22 日 第 42 次 6 月 5 日 第 43 次 6 月 19 日 第 44 次 7 月 3 日 第 45 次 7 月 17 日 第 46 次 7 月 31 日 第 47 次 8 月 14 日 第 48 次 8 月 28 日 第 49 次 9 月 11 日 第 50 次 9 月 25 日 第 51 次 10 月 9 日 第 52 次 10 月 23 日 第 53 次 11 月 6 日 第 54 次 11 月 20 日 第 55 次 12 月 4 日 第 56 次 12 月 18 日 第 57 次 12 月 31 日 第 58 次 1 月 14 日 第 59 次 1 月 28 日 第 60 次 2 月 11 日 第 61 次 2 月 25 日 第 62 次 3 月 11 日 第 63 次 3 月 25 日 第 64 次 4 月 8 日 第 65 次 4 月 22 日 第 66 次 4 月 29 日 第 67 次 5 月 13 日 第 68 次 5 月 27 日 第 69 次 6 月 10 日 第 70 次 6 月 24 日 第 71 次 7 月 8 日 第 72 次 7 月 22 日 第 73 次 7 月 29 日 第 74 次 8 月 12 日 第 75 次 8 月 26 日 第 76 次 9 月 9 日 第 77 次 9 月 23 日 第 78 次 9 月 30 日 第 79 次 10 月 14 日 第 80 次 10 月 28 日 第 81 次 11 月 11 日 第 82 次 11 月 25 日 第 83 次 12 月 9 日 第 84 次 12 月 23 日 第 85 次 1 月 6 日 第 86 次 1 月 20 日 第 87 次 2 月 3 日 第 88 次 2 月 17 日 第 89 次 2 月 27 日 第 90 次 3 月 13 日 第 91 次 3 月 27 日 第 92 次 4 月 10 日 第 93 次 4 月 24 日 第 94 次 5 月 8 日 第 95 次 5 月 22 日 第 96 次 6 月 5 日 第 97 次 6 月 19 日 第 98 次 7 月 3 日 第 99 次 7 月 17 日 第 100 次 7 月 31 日 第 101 次 8 月 14 日 第 102 次 8 月 28 日 第 103 次 9 月 11 日 第 104 次 9 月 25 日 第 105 次 10 月 9 日 第 106 次 10 月 23 日 第 107 次 11 月 6 日 第 108 次 11 月 20 日 第 109 次 12 月 4 日 第 110 次 12 月 18 日 第 111 次 12 月 31 日 第 112 次 1 月 14 日 第 113 次 1 月 28 日 第 114 次 2 月 11 日 第 115 次 2 月 25 日 第 116 次 3 月 11 日 第 117 次 3 月 25 日 第 118 次 4 月 8 日 第 119 次 4 月 22 日 第 120 次 4 月 29 日 第 121 次 5 月 13 日 第 122 次 5 月 27 日 第 123 次 6 月 10 日 第 124 次 6 月 24 日 第 125 次 7 月 8 日 第 126 次 7 月 22 日 第 127 次 7 月 29 日 第 128 次 8 月 12 日 第 129 次 8 月 26 日 第 130 次 9 月 9 日 第 131 次 9 月 23 日 第 132 次 9 月 30 日 第 133 次 10 月 14 日 第 134 次 10 月 28 日 第 135 次 11 月 11 日 第 136 次 11 月 25 日 第 137 次 12 月 9 日 第 138 次 12 月 23 日 第 139 次 1 月 6 日 第 140 次 1 月 20 日 第 141 次 2 月 3 日 第 142 次 2 月 17 日 第 143 次 2 月 27 日 第 144 次 3 月 13 日 第 145 次 3 月 27 日 第 146 次 4 月 10 日 第 147 次 4 月 24 日 第 148 次 5 月 8 日 第 149 次 5 月 22 日 第 150 次 6 月 5 日 第 151 次 6 月 19 日 第 152 次 7 月 3 日 第 153 次 7 月 17 日 第 154 次 7 月 31 日 第 155 次 8 月 14 日 第 156 次 8 月 28 日 第 157 次 9 月 11 日 第 158 次 9 月 25 日 第 159 次 10 月 9 日 第 160 次 10 月 23 日 第 161 次 11 月 6 日 第 162 次 11 月 20 日 第 163 次 12 月 4 日 第 164 次 12 月 18 日 第 165 次 12 月 31 日 第 166 次 1 月 14 日 第 167 次 1 月 28 日 第 168 次 2 月 11 日 第 169 次 2 月 25 日 第 170 次 3 月 11 日 第 171 次 3 月 25 日 第 172 次 4 月 8 日 第 173 次 4 月 22 日 第 174 次 4 月 29 日 第 175 次 5 月 13 日 第 176 次 5 月 27 日 第 177 次 6 月 10 日 第 178 次 6 月 24 日 第 179 次 7 月 8 日 第 180 次 7 月 22 日 第 181 次 7 月 29 日 第 182 次 8 月 12 日 第 183 次 8 月 26 日 第 184 次 9 月 9 日 第 185 次 9 月 23 日 第 186 次 9 月 30 日 第 187 次 10 月 14 日 第 188 次 10 月 28 日 第 189 次 11 月 11 日 第 190 次 11 月 25 日 第 191 次 12 月 9 日 第 192 次 12 月 23 日 第 193 次 1 月 6 日 第 194 次 1 月 20 日 第 195 次 2 月 3 日 第 196 次 2 月 17 日 第 197 次 2 月 27 日 第 198 次 3 月 13 日 第 199 次 3 月 27 日 第 200 次 4 月 10 日 第 201 次 4 月 24 日 第 202 次 5 月 8 日 第 203 次 5 月 22 日 第 204 次 6 月 5 日 第 205 次 6 月 19 日 第 206 次 7 月 3 日 第 207 次 7 月 17 日 第 208 次 7 月 31 日 第 209 次 8 月 14 日 第 210 次 8 月 28 日 第 211 次 9 月 11 日 第 212 次 9 月 25 日 第 213 次 10 月 9 日 第 214 次 10 月 23 日 第 215 次 11 月 6 日 第 216 次 11 月 20 日 第 217 次 12 月 4 日 第 218 次 12 月 18 日 第 219 次 12 月 31 日 第 220 次 1 月 14 日 第 221 次 1 月 28 日 第 222 次 2 月 11 日 第 223 次 2 月 25 日 第 224 次 3 月 11 日 第 225 次 3 月 25 日 第 226 次 4 月 8 日 第 227 次 4 月 22 日 第 228 次 4 月 29 日 第 229 次 5 月 13 日 第 230 次 5 月 27 日 第 231 次 6 月 10 日 第 232 次 6 月 24 日 第 233 次 7 月 8 日 第 234 次 7 月 22 日 第 235 次 7 月 29 日 第 236 次 8 月 12 日 第 237 次 8 月 26 日 第 238 次 9 月 9 日 第 239 次 9 月 23 日 第 240 次 9 月 30 日 第 241 次 10 月 14 日 第 242 次 10 月 28 日 第 243 次 11 月 11 日 第 244 次 11 月 25 日 第 245 次 12 月 9 日 第 246 次 12 月 23 日 第 247 次 1 月 6 日



7.7 hello 进程 execve 时的内存映射

首先删除已存在的用户区域，创建新的区域结构，代码和初始化数据映射到.text 和.data 区（目标文件提供），.bss 和栈映射到匿名文件，设置 PC，指向代码区域的入口点。Linux 根据需要换入代码和数据页面。下一步是映射共享区域，将一些动态链接库映射到 hello 的虚拟地址空间，最后设置程序计数器，使之指向 hello 程序的代码入口。

7.8 缺页故障与缺页中断处理

缺页故障：进程线性地址空间里的页面不必常驻内存，在执行一条指令时，如果发现他要访问的页没有在内存中（即存在位为 0），那么停止该指令的执行，并产生一个页不存在的异常，对应的故障处理程序可通过从外存加载该页的方法来排除故障，之后，原先引起的异常的指令就可以继续执行，而不再产生异常。

产生缺页中断的几种情况：

- 1、当内存管理单元（MMU）中确实没有创建虚拟物理页映射关系，并且在该虚拟地址之后再没有当前进程的线性区（vma）的时候，可以肯定这是一个编码错误，这将杀掉该进程；
- 2、当 MMU 中确实没有创建虚拟页物理页映射关系，并且在该虚拟地址之后存在当前进程的线性区 vma 的时候，这很可能是缺页中断，并且可能是栈溢出导致的缺页中断；
- 3、当使用 malloc/mmap 等希望访问物理空间的库函数/系统调用后，由于 linux 并未真正给新创建的 vma 映射物理页，此时若先进行写操作，将和 2 产生缺页中断的情况一样；若先进行读操作虽然也会产生缺页异常，将被映射给默认的零页，等再进行写操作时，仍会产生缺页中断，这次必须分配 1 物理页了，进入写时复制的流程；
- 4、当使用 fork 等系统调用创建子进程时，子进程不论有无自己的 vma，它的 vma 都有对于物理页的映射，但它们共同映射的这些物理页属性为只读，即 linux 并未给予进程真正分配物理页，当父子进程任何一方要写相应物理页时，导致缺页中断的写时复制；

缺页中断处理：

缺页中断处理一般流程：

- 1.硬件陷入内核，在堆栈中保存程序计数器，大多数当前指令的各种状态信息保存在特殊的 cpu 寄存器中。
- 2.启动一个汇编例程保存通用寄存器和其他易丢失信息，以免被操作系统破坏。
- 3.当操作系统发现缺页中断时，尝试发现需要哪个虚拟页面。通常一个硬件寄存器包含了这些信息，如果没有的话操作系统必须检索程序计数器，取出当前指令，分析当前指令正在做什么。
- 4.一旦知道了发生缺页中断的虚拟地址，操作系统会检查地址是否有效，并检查读写是否与保护权限一致，不过不一致，则向进程发一个信号或者杀死该进程。如

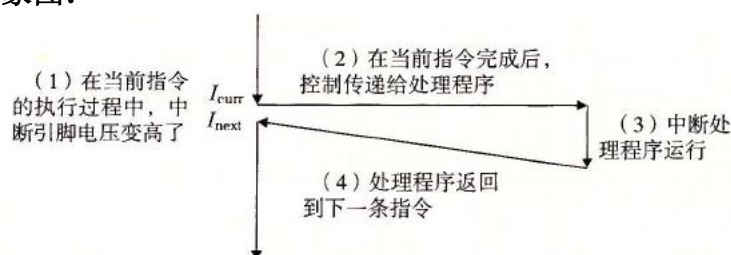
果是有效地址并且没有保护错误发生则系统检查是否有空闲页框。如果没有，则执行页面置换算法淘汰页面。

5.调度引发缺页中断的进程，操作系统返回调用他的汇编例程

6.该例程恢复寄存器和其他状态信息，返回到用户空间继续执行，就好像缺页中断没有发生过。

等等。

抽象图：



8-5 中断处理。中断处理程序将控制返回给应用程序控制流中的下一条指令

7.9 动态存储分配管理

概念：在进程运行的时候，动态存储器分配器维护着一个进程的虚拟存储器区域，称为堆。分配器将对视为一组大小不同的块（block）的集合来分配。每个块就是一个连续的虚拟存储器片（chunk），要么是已经分配的，要么是还没有分配的。已经分配的，显示的保留给应用程序使用（malloc 得到一段内存）。空闲的，则是可以继续分配。一个已经分配的块保持已经分配的转态，直到被 free 掉，得以被堆重用。当已经分配的块一直没有被 free，那么一直保持分配转态，这个就是内存泄漏了。

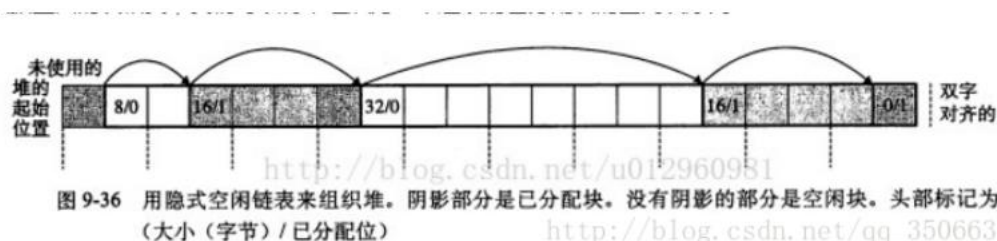
分配器有两种基本风格：

显示分配器：要求应用显示的释放已经分配的块，像 C，C++语言采用的机制。

隐式分配器：分配器检测一个已经分配的块何时不再被程序所使用时，自动 free 掉。这个就是垃圾收集。

记录空闲块：

隐式：



显示空闲链表：

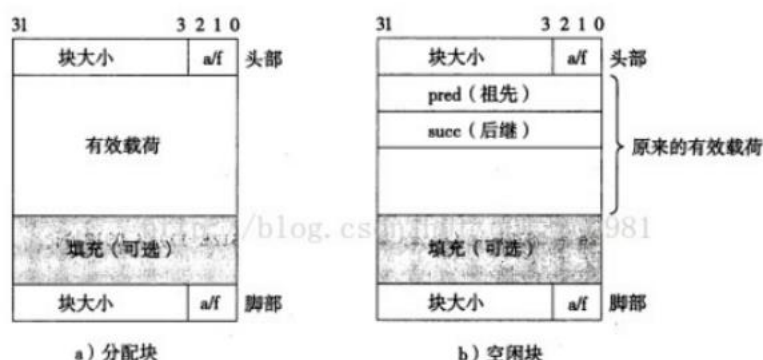


图 9-48 使用双向空闲链表的堆块的格式

分离空闲链表：简单分离存储、分离适配、伙伴系统

Printf 会调用 malloc： printf 会引用全局变量 stdout，malloc。

7.10 本章小结

1、存储器管理的主要任务

存储器管理的主要任务是为多道程序的并发运行提供良好的存储器环境。它包括以下内容：

- (1)能让每道程序“各得其所”，并在不受干扰的环境中运行，还可以使用户从存储空间分配、保护等烦琐事务中解脱出来；
- (2)向用户提供更大的存储空间，使更多的作业能同时投入运行或使更大的作业能在较小的内存空间中运行；
- (3)为用户对信息的访问、保护、共享以及动态链接等方面提供方便；
- (4)能使存储器有较高的利用率。

2、存储器管理的主要功能

为了实现存储器管理的主要任务，存储器管理应具有以下几个方面的功能。

(1)内存分配。根据分配策略，为多道程序分配内存，并实现共享。同时对程序释放的存储空间进行回收。

(2)地址映射。每道程序都有自己的逻辑地址，在多道程序环境中，内存空间被多道程序共享，这就必然导致程序的逻辑地址与在内存中的物理地址不一致。因此，存储器管理必须提供地址映射功能，用于逻辑地址和物理地址间的变换。地址映射通常在硬件支持下完成。

(3)内存保护。确保每道程序在自己的内存空间中运行，互不干扰。内存保护一般由硬件完成。

(4)内存扩充。利用虚拟存储技术从逻辑上扩充内存空间，为用户营造一个比实际物理内存更大的存储空间。程序运行过程中常常涉及到动态内存分配，动态内存分配通过动态内存分配器完成，能够对堆空间进行合理地分配与管理，分割与合并。现代使程序内存分配器采取多种策略来提高吞吐量以及内存占用率，从在灵活使用内存的基础上保证了效率。

(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：在设备模型中，所有的设备都通过总线相连。每一个设备都是一个文件。设备模型展示了总线和它们所控制的设备之间的实际连接。在最底层，Linux 系统中的每个设备由一个 struct device 代表，而 Linux 统一设备模型就是在 kobject kset ktype 的基础之上逐层封装起来的。

设备管理：是通过 unix io 接口实现的。

8.2 简述 Unix IO 接口及其函数

linux 提供如下 IO 接口：

read 和 write -- 最简单的读写函数

readn 和 writen -- 原子性读写操作

recvfrom 和 sendto -- 增加了目标地址和地址结构长度的参数

recv 和 send -- 允许从进程到内核传递标志

readv 和 writev -- 允许指定往其中输入数据或从其中输出数据的缓冲区

recvmsg 和 sendmsg -- 结合了其他 IO 函数的所有特性，并具备接受和发送辅助数据的能力

Unix I/O 接口提供了以下函数供应用程序调用：

打开文件：int open(char *filename, int flags, mode_t mode);

关闭文件：int close(int fd);

读文件：ssize_t read(int fd, void *buf, size_t n);

写文件：ssize_t write(int fd, const void *buf, size_t n);

8.3 printf 的实现分析

printf 函数的函数体

```
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;
```

```
}
```

vsprintf 的作用就是格式化。它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。

Write 系统函数：

write:

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

write 有个参数：1

1 表示的是 tty 所对应的一个文件句柄。

在 linux 里，所有设备都是被当作文件来看待的，这个 1 就是表示往当前显示器里写入数据

int INT_VECTOR_SYS_CALL 表示要通过系统来调用 **sys_call** 这个函数。

最后：printf()函数不能确定参数，只会根据 format 中的打印格式的数目依次打印堆栈中参数 format 后面地址的内容。这样就存在一个**可能的缓冲区溢出问题**。

syscall 将字符串中的字节“Hello 1170300815 范天祥”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）。

于是打印字符串“Hello 1170300825 范天祥”就显示在了屏幕上。

8.4 getchar 的实现分析

首先，getchar()是标准 I/O 标准库里的库函数，其原型是

```
int getchar(void)
```

它没有参数，原因是因为它是从 stdin 标准输入流中读入一个字符的函数，已经有了默认的流参数 stdin 了。其返回值是一个整型数，是用来表示字符用的。

在 C 语言中有个重要的库函数 getchar(), 可从终端获得一个字符的 ASCII 码值。在终端输入字符时并非输入一个字符就会返回，而是在遇到回车换行前，所有输入的在 C 语言中有个重要的库函数 getchar(), 可从终端获得一个字符的 ASCII 码值。在终端输入字符时并非输入一个字符就会返回，而是在遇到回车换行前，所有输入的字符都会缓冲在键盘缓冲器中，直到回车换行一次性将所有字符按序依次赋给相应的变量，在这里一定要注意最后一个字符即'\n'，该字符也会赋给一个相应的变量。

当用 getchar 进行输入时，如果输入的第一个字符为有效字符(即输入是文件结束符 EOF，Windows 下为组合键 Ctrl+Z， Unix/Linux 下为组合键 Ctrl+D)，那么只有当最后一个输入字符为换行符'\n'(也可以是文件结束符 EOF，EOF 将在后面讨论)时，getchar 才会停止执行，整个程序将会往下执行。

当终端有字符输入时，Ctrl+D 产生的 EOF 相当于结束本行的输入，将引起 getchar()新一轮的输入；当终端没有字符输入或者说当 getchar()读取新的一次

输入时，输入 Ctrl+D，此时产生的 EOF 相当于文件结束符，程序将结束 getchar() 的执行。字符都会缓冲在键盘缓冲器中，直到回车换行一次性将所有字符按序依次赋给相应的变量，在这里一定要注意最后一个字符即 '\n'，该字符也会赋给一个相应的变量。

getchar 函数通过调用 read 函数返回字符。read 函数的返回值是读入的字符数，如果为 1 说明读入成功，那么直接返回字符，否则说明读到了缓冲区的最后。

read 函数同样通过 sys_call 中断来调用内核中的系统函数。键盘中断处理子程序会接受按键扫描码并将其转换为 ASCII 码后保存在缓冲区。然后 read 函数调用的系统函数可以对缓冲区 ASCII 码进行读取，直到接受回车键返回。这样，getchar 函数通过 read 函数返回字符，实现了读取一个字符的功能。

8.5 本章小结

输入/输出 I/O 是主存和外部设备如终端、网络之间拷贝数据的过程。输入操作是从 I/O 设备拷贝数据到主存。输出操作是从主存拷贝数据到 I/O 设备。

在 UNIX 中所有的 I/O 设备都被模型化为文件。并提供 Unix I/O 接口。通过这个接口，程序能够进行输入与输出，只需要找到描述符，底层硬件实现操作系统就可以实现。

(第 8 章 1 分)

结论

(用计算机系统的语言，逐条总结 hello 所经历的过程。

你对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。)

总结：

1. 计算机系统

我们知道计算机系统是由硬件和软件组成的。它们共同工作来运行应用程序。虽然系统的实现方式随着时间不断变化，但是系统内在的概念却没有改变。所有计算机系统都有相似的硬件和软件组件，它们执行着相似的功能，我们只有深入了解这些组件是如何工作的，以及这些组件是如何影响程序的正确性和性能的，才能写出高质量的代码。

2. hello.c

ASCII 码构成就是用一个唯一的单字节大小的整数来表示每个字符，以字节序列的方式存储在文件中。

3. 程序的编译

hello 程序的生命周期是从一个高级 C 语言程序开始的，因为这种形式能被人读懂。然而，计算机系统是读不懂高级语言的。为了在系统上运行 hello.c 程序，每条 C 语句都必须要被其他程序转化为一系列的低级机器语言指令。

一般来说, 要将 `hello.c` 变成一个可执行的目标程序, 必须要经过预处理器、编译器、汇编器和链接器的处理。

预处理器、编译器、汇编器和链接器 一起构成了编译系统, 下面对每个步骤分别进行解析:

①预处理阶段: 预处理器 `cpp` 根据以字符 `#` 开头的命令, 修改原始的 C 程序, 比如 `Hello.c` 中第一行 `#include<stdio.h>` 命令告诉预处理器读取系统文件 `stdio.h` 的内容, 并把它直接插入到程序中。结果就得到另一个 C 程序, 通常是以 `.i` 作为文件扩展名。

②编译阶段: 编译器 `ccl` 将文本文件 `hello.i` 翻译成文本文件 `hello.s`, 它包含一个汇编语言程序, 汇编语言程序中的每条语句都以一种标准的文本格式确切的描述一条低级机器语言指令。汇编语言能为不同高级语言的不同编译器提供通用的输出语言。

③汇编阶段: 汇编器 `as` 将 `hello.s` 翻译成机器语言指令, 把这些指令打包成一种叫做可重定位目标程序的格式, 并将结果保存在目标文件 `hello.o` 中, `hello.o` 文件是一个二进制文件, 它的字节编码是机器预言指令而不是字符。如果我们用文本编辑器打开 `hello.o` 文件, 将会是一堆乱码。

④链接阶段: 在 `hello.c` 程序中, 我们看到程序调用了 `printf` 函数, 它是每个 C 编译器都会提供的标准 C 库中的一个函数。`printf` 函数存在于一个名为 `printf.o` 的单独的预编译好了的目标文件中, 而这个文件必须以某种方式合并到我们的 `hello.o` 程序中。链接器 `ld` 就是负责处理这种合并, 结果就得到一个 `hello` 文件, 它是一个可执行目标程序, 可以被加载到内存中, 由系统运行。

静态链接:

在 程序执行之前就完成链接工作。也就是等链接完成后文件才能执行。但是这有一个明显的缺点, 比如说库函数。如果文件 A 和文件 B 都需要用到某个库函数, 链接完成后他们连接后的文件中都有这个库函数。当 A 和 B 同时执行时, 内存中就存在该库函数的两份拷贝, 这无疑浪费了存储空间。当规模扩大的时候, 这种浪费尤为明显。静态链接还有不容易升级等缺点。为了解决这些问题, 现在的很多程序都用动态链接。

动态链接:

和静态链接不一样, 动态链接是在程序执行的时候才进行链接。也就是当程序加载执行的时候。还是上面的例子, 如果 A 和 B 都用到了库函数 `Fun()`, A 和 B 执行的时候内存中就只需要有 `Fun()` 的一个拷贝。

关于链接还有很多知识, 这里只涉及皮毛。

4.装载:

我们知道, 程序要运行是必然要把程序加载到内存中的。在过去的机器里都是把整个程序都加载进入物理内存中, 现在一般都采用了虚拟存储机制, 即每个进程都有完整的地址空间, 给人的感觉好像每个进程都能使用完成的内存。然后由一个内存管理器把虚拟地址映射到实际的物理内存地址。

按照上文的叙述, 程序的地址可以分为虚拟地址和实际地址。虚拟地址即虚拟内存空间中的地址, 物理地址就是被加载的实际地址。

加载的过程可以这样理解: 先为程序中的各部分分配好虚拟地址, 然后再建立

虚拟地址到物理地址的映射。其实关键的部分就是虚拟地址到物理地址的映射过程。程序装在完成之后，cpu 的程序计数器 pc 就指向文件中的代码起始位置，然后程序就按顺序执行。

5.程序的运行：

想要在 Linux 系统中运行该可执行程序，我们要将它的文件名输入到称为外壳（shell）的应用程序中，外壳是一个命令行解释器，它输出一个提示符，等待你输入一个命令，然后执行这个命令。如果该命令行的第一个单词不是一个内置的外壳命令，那么外壳就会假设这是一个可执行文件的名字，它将加载并运行这个文件。

初始时，外壳程序执行它的指令，等待我们输入一个命令。当我们在键盘上输入字符串"./hello"后，外壳程序将字符逐一读入到寄存器中，再把它放入到存储器中。

当我们在键盘上敲回车键的时候，外壳程序知道我们已经结束了命令的输入。然后外壳执行一系列指令来加载可执行的 hello 文件，将 hello 目标文件中的代码和数据从磁盘复制到主存。数据包括最终会被输出的字符串“Hello 1170300815 范天祥\n”，一旦目标文件中的代码和数据被加载到主存，处理器就开始执行 hello 程序的 main 程序中的机器语言指令。这些指令将“Hello 1170300815 范天祥\n”字符串中的字节从主存复制到寄存器文件，再从寄存器文件中复制到显示设备，最终显示在屏幕上。

感悟：

写这篇报告的目的在于梳理程序运行的机制，在一个可执行文件执行的背后都隐藏了什么。

从源代码到可执行文件通常要经历许多中间步骤，每一个中间步骤都生成一个中间文件。只是现在的集成开发环境都把这些步骤隐藏了，习惯于集成开发环境的我们也就逐渐地忽略了这些重要的技术内幕。这篇报告也只是介绍了一下这个过程的主线而已。其中的每一个细节展开来讲都可足已用一篇文章来论述。

这门课让我收获颇丰，她将会伴随我往后的程序生涯，让我受益匪浅。

（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。

文件名称	文件作用
hello.c	提供源程序
hello.i	cpp 预处理之后的文件
hello.s	cc1 之后的汇编语言格式文件
hello.o	as 之后的可重定位目标文件
hello	ld 之后的可执行目标文件
hellofix.elf	hello.o 的 ELF 格式文件
hello.objdump	Objdump 得到的 hello 的反汇编代码

(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] nm 和 readelf 命令的区别: <https://www.cnblogs.com/foohack/p/4103074.html>
- [2] Linux 下链接器 ld 链接顺序的总结:
<https://www.jb51.net/LINUXjishu/236871.html>
- [3] printf 函数实现的深入剖析 <https://www.cnblogs.com/pianist/p/3315801.html>
- [4] UNIX IO 操作函数 <http://www.techlog.cn/article/list/10182663>
- [5] 关于 C 语言中 getchar() 的详细使用 - Dormant 专栏 - CSDN 博客
<https://blog.csdn.net/meidong52617/article/details/44728517>
- [6] linux2.6 内存管理——逻辑地址转换为线性地址（逻辑地址、线性地址、物理地址、虚拟地址）<http://www.cnblogs.com/diaohaiwei/p/5094959.html>
- [7] Linux 内存管理：逻辑地址到线性地址和物理地址的转换 - pi9nc 的专栏 - CSDN 博客 <https://blog.csdn.net/pi9nc/article/details/21031651>
- [8] 详解缺页中断-----缺页中断处理（内核、用户）- 柯南的博客 - CSDN 博客
https://blog.csdn.net/m0_37962600/article/details/81448553
- [9] 九、完善堆内系统调用：Linux 系统调用、printf 的内部实现、malloc 的内部实现- zhangyang249 的博客 - CSDN 博客
<https://blog.csdn.net/zhangyang249/article/details/78582809>
- [10] 从 hello world 解析程序运行机制 - 云+社区 - 腾讯云
<https://cloud.tencent.com/developer/article/1021739>

（参考文献 0 分，缺失 -1 分）