

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>計算學部</u>
學 號	<u>1190200708</u>
班 級	<u>1903008</u>
學 生	<u>熊峰</u>
指 導 教 師	<u>吳銳</u>

計算機科學與技術學院
2021 年 5 月

摘 要

hello 程序作为最简单的、最经典的程序，在实现上非常简单，但即使是最简单的 hello.c 程序，也经历了复杂的一生，它需要计算机上的几乎所有的硬件设备与操作系统等协同工作，才能运行起来。

本文将结合《深入理解计算机系统》，深入分析 Linux 环境下，hello 的 P2P, 020 的整个过程，进而将理论知识与实践紧密结合。

关键词：预处理、编译、汇编、链接、进程管理、存储管理、IO 管理

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	- 8 -
第 3 章 编译	- 9 -
3.1 编译的概念与作用	- 9 -
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	- 22 -
第 4 章 汇编	- 23 -
4.1 汇编的概念与作用	- 23 -
4.2 在 UBUNTU 下汇编的命令	- 23 -
4.3 可重定位目标 ELF 格式	- 23 -
4.4 HELLO.O 的结果解析	- 28 -
4.5 本章小结	- 33 -
第 5 章 链接	- 34 -
5.1 链接的概念与作用	- 34 -
5.2 在 UBUNTU 下链接的命令	- 34 -
5.3 可执行目标文件 HELLO 的格式	- 35 -
5.4 HELLO 的虚拟地址空间	- 40 -
5.5 链接的重定位过程分析	- 43 -
5.6 HELLO 的执行流程	- 45 -
5.7 HELLO 的动态链接分析	- 48 -
5.8 本章小结	- 50 -
第 6 章 HELLO 进程管理	- 51 -
6.1 进程的概念与作用	- 51 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 51 -
6.3 HELLO 的 FORK 进程创建过程	- 52 -
6.4 HELLO 的 EXECVE 过程	- 52 -
6.5 HELLO 的进程执行.....	- 54 -
6.6 HELLO 的异常与信号处理	- 55 -
6.7 本章小结	- 60 -
第 7 章 HELLO 的存储管理.....	- 61 -
7.1 HELLO 的存储器地址空间	- 61 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 61 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 63 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 65 -
7.5 三级 CACHE 支持下的物理内存访问	- 66 -
7.6 HELLO 进程 FORK 时的内存映射	- 67 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 67 -
7.8 缺页故障与缺页中断处理.....	- 69 -
7.9 动态存储分配管理	- 70 -
7.10 本章小结	- 71 -
第 8 章 HELLO 的 IO 管理	- 72 -
8.1 LINUX 的 IO 设备管理方法	- 72 -
8.2 简述 UNIX IO 接口及其函数	- 72 -
8.3 PRINTF 的实现分析.....	- 73 -
8.4 GETCHAR 的实现分析.....	- 76 -
8.5 本章小结	- 76 -
结论	- 77 -
附件	- 78 -
参考文献.....	- 79 -

第 1 章 概述

1.1 Hello 简介

hello 程序的生命周期是从一个高级 C 语言程序开始的，然而为了在系统上运行 hello.c 程序，每条 C 语句都必须被其他程序转化为一系列的低级机器语言指令。然后这些指令按照一种成为可执行目标程序的格式打好包，并以二进制磁盘文件的形式存放起来。

hello 程序的 P2P 过程：

hello 程序经过预处理、编译、汇编、链接，生成可执行文件，在 shell 中运行 hello 程序时，shell 通过调用系统调用来执行我们的请求，系统调用会将控制权传递给操作系统，操作系统保存 shell 进程的上下文，使用 fork 函数，创建一个新的 hello 进程及其上下文，完成程序到进程的转变 P2P(From Program to Process)。

hello 程序的 O2O 过程：

shell 执行可执行目标文件，使用 execve 函数加载进程，映射虚拟内存，进入程序入口后程序载入物理内存。进入 main 函数执行目标代码，CPU 为运行的 hello 分配时间片执行逻辑控制流。当程序运行结束后，shell 父进程负责回收 hello 进程，内核删除相关数据结构。

1.2 环境与工具

硬件环境：Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz;RAM 16.0 GB (15.8 GB 可用)

软件环境：Win 10 20H2;Ubuntu 20.04.2 LTS;WSL2;VMware workstation

开发与调试工具：vim; gcc; VSCode; readelf; ld; as; edb

1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名称，文件的作用等。

文件名	文件作用
hello.c	源程序
hello.i	预处理文件
hello.s	编译后的汇编文件
hello.o	汇编后的可重定位目标程序
hello	链接后的可执行目标程序
hello_asm.s	hello 可执行文件的反汇编文件
hello_elf	hello 可执行文件的 elf 的格式
hello_o.elf	hello.o 可重定位文件的 elf 的格式
hello_o.s	hello.o 可重定位文件的反汇编文件

1.4 本章小结

本章简要介绍了 hello 程序的 P2P, 020 的过程、本次实验的环境工具, 以及实验过程中, 生成的中间结果文件。

(第 1 章 0.5 分)

第 2 章 预处理

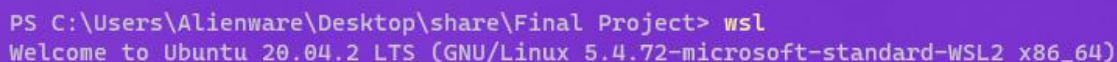
2.1 预处理的概念与作用

预处理器(cpp)根据以#开头的命令, 修改原始的 C 程序。比如 `hello.c` 中第一行的`#include <stdio.h>`命令告诉预处理器读取系统头文件 `stdio.h` 中的内容, 并把它直接插入到程序文本中。结果就得到了另一个 C 程序, 通常是以*.i* 作为文件的扩展名。

预处理过程是整个编译过程的第一步。预处理是指在进行编译的第一遍扫描之前所做的工作。预处理指令指示在程序正式编译前就由编译器进行的操作, 可放在程序中任何位置。C 语言提供多种预处理功能, 预处理器会分析预处理指令: 包含文件、条件编译、宏定义与扩展、特殊宏与指令、Token 字符串化、Token 连接、用户定义的编译错误与警告以及编译器相关的预处理特性等, 并去除源代码中的注释。

2.2 在 Ubuntu 下预处理的命令

首先启用 WSL, 启动 Ubuntu 20.04.2 LTS, 如图 2-1 所示。



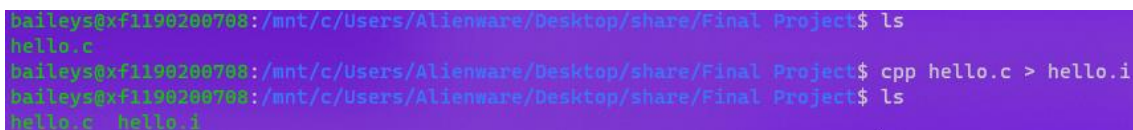
```
PS C:\Users\Alienware\Desktop\share\Final Project> wsl
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.72-microsoft-standard-WSL2 x86_64)
```

图 2 - 1

在 Ubuntu 下对 `hello.c` 进行预处理, 应使用如下命令:

`gcc -E hello.c -o hello.i` 或 `cpp hello.c > hello.i`

在 wsl 中键入命令, 得到经过预处理后的文件, 如图 2-2 所示:



```
baileys@xf1190200703:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello.c
baileys@xf1190200703:/mnt/c/Users/Alienware/Desktop/share/Final Project$ cpp hello.c > hello.i
baileys@xf1190200703:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello.c  hello.i
```

图 2 - 2

2.3 Hello 的预处理结果解析

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ vim hello.i
```

图 2 - 3

```
extern int getsubopt (char **__restrict __optionp,
                    char *const *__restrict __tokens,
                    char **__restrict __valuep)
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1, 2, 3))) ;
# 1801 /usr/include/stdlib.h 1 4
extern int getloadavg (double __loadavg[], int __nelem)
    __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1))) ;
# 1811 /usr/include/stdlib.h 3 4
# 1 /usr/include/x86_64-linux-gnu/bits/stdlib-float.h 1 3 4
# 1818 /usr/include/stdlib.h 2 3 4
# 1823 /usr/include/stdlib.h 3 4

# 0 hello.c 2

# 18 hello.c
int main(int argc, char *argv[]){
    int i;

    if(argc==4){
        printf("用法: Hello 学号 姓名 秒数! \n");
        exit(1);
    }
    for(i=0; i<8; i++){
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(atoi(argv[3]));
    }
    getchar();
    return 0;
}
"hello.i" 3060L, 64732C                                     3048,1      Bot
```

图 2 - 4

如图 2-4 所示，使用 Vim 查看得到的与处理文件，可以发现 hello.i 已经被扩展到了 3060 行。

```
# 18 hello.c
int main(int argc, char *argv[]){
    int i;

    if(argc==4){
        printf("用法: Hello 学号 姓名 秒数! \n");
        exit(1);
    }
    for(i=0; i<8; i++){
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(atoi(argv[3]));
    }
    getchar();
    return 0;
}
search hit BOTTOM, continuing at TOP                                     3047,1      Bot
```

图 2 - 5

如图 2-5 所示，通过 Vim 中的搜索功能，可以看到此时原来的 main 函数位于 3047 行。此时，hello.c 程序中的头文件 stdio.h、unistd.h、stdlib.h 中的内容插入到了 hello.i 文件中，内容包括函数声明、结构体定义、变量定义等，头文件的内容被递归展开进 hello.i 文件中。

2.4 本章小结

本章主要介绍了预处理的概念与作用、在 Ubuntu 下预处理的命令以及 Hello 程序经过预处理后的简要分析。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

编译的概念：

编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文件。经历如下过程：

- 1) 词法分析：词法分析使用程序实现词法扫描，它会按照用户之前描述好的词法规则将输入的字符串分割成一个个记号。产生的记号一般分为：关键字、标识符、字面量（包含数字、字符串等）和特殊符号（运算符、等号等），然后他们放到对应的表中。
- 2) 语法分析：语法分析器根据用户给定的语法规则，将词法分析产生的记号序列进行解析，然后将它们构成一棵语法树。对于不同的语言，只是其语法规则不一样。
- 3) 语义分析：语法完成了对表达式语法层面的分析，但是它不了解这个语句是否真正有意义。有的语句在语法上是合法的，但是却是没有实际的意义，比如说两个指针的做乘法运算，这个时候就需要进行语义分析，但是编译器能分析的语义也只有静态语义。
- 4) 中间代码生成：对于一些在编译期间就能确定的值，编译器可以将他优化。优化器会先将语法树转成中间代码。中间代码一般与目标机器和运行环境无关。
- 5) 目标代码生成与优化：代码生成器将中间代码转成机器代码，这个过程是依赖于目标机器的，因为不同的机器有着不同的字长、寄存器、数据类型等。最后目标代码优化器对目标代码进行优化，比如选择合适的寻址方式、使用唯一来代替乘除法、删除出多余的指令等。

编译作用：编译过程是整个程序构建的核心部分，编译成功，编译器会将源代码由文本形式转换成机器语言。

在本例中，编译器将 `hello.i` 编译成 `hello.s`。

3.2 在 Ubuntu 下编译的命令

在 Ubuntu 下对 hello.i 进行编译，应使用如下命令，如图 3-1 所示：

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



```
kali@kali:~/Final Project$ gcc -S hello.i -o hello.s
kali@kali:~/Final Project$ ls
hello.c  hello.i  hello.s
```

图 3 - 1

3.3 Hello 的编译结果解析

3.3.1 汇编文件声明解析

声明	含义
.file	声明源文件
.text	代码段
.section .rodata	只读代码段
.align	对齐的格式
.global	全局变量
.string	字符串类型数据
.long	long 类型数据
.type	符号类型
.size	声明大小

3.3.2 整型数据类型

源码中出现多种数据类型，首先分析整型，在源代码中出现的整型有 `argc` 参数、`i` 计数参数。

```

main:
.LFB6:
.cfi_startproc
endbr64
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    $4, -20(%rbp)
je      .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi
call    exit@PLT

```

图 3 - 2

首先分析 argc，根据 x86-64 的寄存器使用，argc 应该被保存在 %edi 中，通过查看 hello.s，如图 3-2，我们可以看到此时 %edi 的值被保存在 -20(%rbp) 中，并与 4 做比较，用于条件跳转，若相等则跳转到 .L2 处，否则执行 leaq 及接下来的指令。

```

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

```

图 3 - 3

```

.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
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi
call    atoi@PLT
movl    %eax, %edi
call    sleep@PLT
addl    $1, -4(%rbp)
.L3:
cmpl    $7, -4(%rbp)
jle     .L4
call    getchar@PLT
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc

```

图 3 - 4

接下来分析计数参数 i ，如图 3-3 所示， i 首先出现在.L2 处，将 0 保存在 $-4(\%rbp)$ 中，在跳转到.L3 处，如图 3-4 所示，用 `cmpl` 语句，使 $-4(\%rbp)$ 处的值与 7 相比，若大于 7 则跳出循环，作为循环的出口。并在.L4 的结尾处，将 $-4(\%rbp)$ 进行加一的操作，使其能够正常进行循环。

实际代码中，还存在着一些立即数，如常数 1 用于控制循环加减等操作。

3.3.3 字符串数据类型

```
.LC0:  
    .string "\347\224\250\346\263\225: Hello \345\255\246\345\217\267"  
.LC1:  
    .string "Hello %s %s\n"
```

图 3 - 5

查看 `hello.s`，如图 3-5，可以发现在 `hello.s` 中，字符串类型数据共出现两次，保存在程序的 `.rodata` 段中。

```
    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
```

图 3 - 6

如图 3-6 所示，在 `hello.s` 中，两个字符串被调用的地方。

3.3.4 数组数据类型

```

main:
.LFB6:
.cfi_startproc
endbr64
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    $4, -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
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi
call    atoi@PLT
movl    %eax, %edi
call    sleep@PLT
addl    $1, -4(%rbp)

```

图 3 - 7

如图 3-7，查看 hello.s 源代码，根据 x86-64 使用寄存器保存参数的规则，此时 argv 应该保存在 %rsi 中，如图，发现 %rsi 被调用的地方。根据 .L4 中对 addq \$8, %rax 的操作，可得 argv 中元素为八个字节(详细分析具体在 3.3.8 中)。此处分别调用了 argv[1] 和 argv[2]，可以看到 %rax 又执行了加 24 的操作，可以看出此时取 argv[3] 中的元素，并将其取到 %rax 中，并放入 %rdi 中作为参数，调用 atoi 函数。并将返回结果 %eax 放入 %edi 中，调用 sleep 函数。

3.3.5 赋值操作

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

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

    if(argc!=4){
        printf("用法: Hello 学号 姓名 秒数! \n");
        exit(1);
    }
    for(i=0;i<8;i++){
        printf("Hello %s %s\n",argv[1],argv[2]);
        sleep(atoi(argv[3]));
    }
    getchar();
    return 0;
}
```

图 3 - 8

如图 3-8，首先查看源代码，发现存在对 i 赋值为 0 和 $i++$ 的操作(具体操作在 3.3.6 处)。因此查看 `hello.s` 中，循环部分的操作。

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

图 3 - 9

通过对 `hello.s` 分析，我们可知，当 `argv` 为 4 时，程序会跳转到 `.L2` 处，如图所示，我们可以发现将 0 赋值给 `-4(%rbp)`。

3.3.6 算术操作

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

图 3 - 10

如图 3-10, 此时用 $-4(\%rbp)$ 的值与 7 作比较, 控制循环。并在小于等于的时候, 跳转到 .L4 处。

```
.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
    movq    -32(%rbp), %rax
    addq    $24, %rax
    movq    (%rax), %rax
    movq    %rax, %rdi
    call    atoi@PLT
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)
```

图 3 - 11

如图 3-11, 我们可以发现, 此时 $-4(\%rbp)$ 执行加一的操作, 即对 i 执行 $i++$ 的操作。

3.3.7 关系操作

```
if(argc!=4){  
    printf("用法: Hello 学号 姓名 秒数! \n");  
    exit(1);  
}  
for(i=0;i<8;i++){  
    printf("Hello %s %s\n",argv[1],argv[2]);  
    sleep(atoi(argv[3]));  
}
```

图 3 - 12

如图 3-12，通过对 hello.c 分析，一共出现两处关系操作。

```
main:  
.LFB6:  
.cfi_startproc  
endbr64  
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    $4, -20(%rbp)  
je      .L2  
leaq    .LC0(%rip), %rdi  
call    puts@PLT  
movl    $1, %edi  
call    exit@PLT  
.L2:  

```

图 3 - 13

如图 3-13，此处为第一处关系操作，将 argc 的值与 4 相比。若相等则跳转到.L2 处，否则继续执行。

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

图 3 - 14

如图 3-14，此处为第二处关系操作，利用 -4(%rbp) 处的值与 7 相比，控制循环的出口。

3.3.8 数组/指针/结构操作

```
for(i=0;i<8;i++){  
    printf("Hello %s %s\n",argv[1],argv[2]);  
    sleep(atoi(argv[3]));  
}  
getchar();  
return 0;
```

图 3 - 15

分析源码，此处 `argv[1]`、`argv[2]`和 `argv[3]`存在数组的操作，分别用作 `printf`和 `sleep` 函数的参数。

```
.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  
movq    -32(%rbp), %rax  
addq    $24, %rax  
movq    (%rax), %rax  
movq    %rax, %rdi  
call    atoi@PLT  
movl    %eax, %edi  
call    sleep@PLT  
addl    $1, -4(%rbp)
```

图 3 - 16

hello.s 源代码如图 3-17 所示.

```

main:
.LFB6:
.cfi_startproc
endbr64
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    $4, -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
movq    -32(%rbp), %rax
addq    $24, %rax
movq    (%rax), %rax
movq    %rax, %rdi
call    atoi@PLT
movl    %eax, %edi
call    sleep@PLT
addl    $1, -4(%rbp)

```

图 3 - 17

如图 3-17，查看 hello.s 源代码，根据 x86-64 使用寄存器保存参数的规则，此时 argv 应该保存在 %rsi 中，如图，发现 %rsi 被调用的地方。根据 .L4 中对 addq \$8, %rax 的操作，可得 argv 中元素为八个字节。分析：发现，首先将 %rsi 的值赋给 -32(%rbp)，再将 -32(%rbp) 的值赋给 %rax，再使用 addq \$16, %rax 的指令，将 %rax 的指针移到 argv 偏移 16 字节的位置，再将它的值保存到 %rdx，再将 -32(%rbp) 中的值加 8，并将对应的值保存到 %rsi。此处分别对应 argv[1]、argv[2]。

接下来将 %rax 执行加 24 的操作，将此时的偏移移动到 argv[3] 处，并将取出的结果保存到 %rax 处，再将 %rax 的值赋给 %rdi，此时 %rdi 作为 atoi 函数的参数，调用 atoi 函数，再将返回结果 %rax 赋给 %rdi 作为 sleep 函数的参数，再调用 sleep 函数。

3.3.9 控制转移

源代码中存在两次控制转移：

```
if(argc!=4){
    printf("用法: Hello 学号 姓名 秒数! \n");
    exit(1);
}
for(i=0;i<8;i++){
    printf("Hello %s %s\n",argv[1],argv[2]);
    sleep(atoi(argv[3]));
}
```

图 3 - 18

如图 3-18，首先 if 及 for 实现控制转移。

```
cmpl    $4, -20(%rbp)
je      .L2
leaq    .LC0(%rip), %rdi
```

图 3 - 19

如图 3-19，图中显示的是 if 控制转移。若 -20(%rbp)处的值等于 4，则跳转到.L2，否则继续执行 leaq 等操作。

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

图 3 - 20

如图 3-20，图中显示的是 for 控制转移。若 -4(%rbp)处的值大于 4，则调用 getchar 函数，否则跳转到.L4 处。

3.3.10 函数调用

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

    if(argc!=4){
        printf("用法: Hello 学号 姓名 秒数! \n");
        exit(1);
    }
    for(i=0; i<8; i++){
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(atoi(argv[3]));
    }
    getchar();
    return 0;
}
```

图 3 - 21

如图 3-21, hello.c 文件中, 共存在七处函数调用。

printf、exit:

```
main:
.LFB6:
    .cfi_startproc
    endbr64
    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    $4, -20(%rbp)
    je      .L2
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT
```

图 3 - 22

如图 3-22, 若 $argc \neq 4$, 则此时会调用 puts 函数, 输出“用法: Hello 学号 姓名 秒数! \n”, 再将 1 赋给 %edi 中, 此时 %edi 作为参数调用 exit 函数, 完成两次函数调用。

Printf、atoi、sleep:

```

.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
    movq    -32(%rbp), %rax
    addq    $24, %rax
    movq    (%rax), %rax
    movq    %rax, %rdi
    call    atoi@PLT
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)

```

图 3 - 23

如图 3-23，若此时 argc 为 4，则在经过赋值等操作后，将跳转到.L4 中，经过前面的分析，我们有此时将会调用 printf 函数，输出 argv[1]和 argv[2]，并根据将 argv[3]的值到存在%rdi 中，调用 atoi 函数，在将结果保存到%edi 中，%edi 作为参数，调用 sleep 函数，此时完成了三次函数调用。

getchar:

```

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

```

图 3 - 24

如图 3-24，若此时跳出循环，则会调用 getchar 函数。

main:

```

main:
.LFB6:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16

```

图 3 - 25

如图 3-25，存在对 main 函数的调用。

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

图 3 - 26

如图 3-26，将%rax 赋值为 0，作为 main 函数的返回值，并通过调用 leave、ret 指令结束 main 函数。故共计存在七次函数调用。

3.4 本章小结

本章详细介绍了编译的概念与作用、在 Ubuntu 下编译的指令，以及对 hello.c 的详细分析，及其所对应的汇编代码 hello.s 的详细分析，对其各项数据类型、赋值操作、算术操作、关系操作、数组/指针/结构操作、控制转移操作、函数调用等结合汇编代码深入分析和细致探讨。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

汇编的概念：

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。这样的过程称之为汇编。

汇编的作用：

将汇编文本程序翻译成机器语言指令并打包成可重定位目标程序。可重定位目标程序文件是一个二进制文件，他包含的字节是函数 `main` 的指令编码。

4.2 在 Ubuntu 下汇编的命令

在 Ubuntu 下对 `hello.s` 进行汇编，应使用如下命令：

`as hello.s -o hello.o` 或者 `gcc -c hello.s -o hello.o`

在 WSL 中，执行该命令，如图 4-1 所示：



```
bailey@xf1198240708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ as hello.s -o hello.o
bailey@xf1198240708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello.o  hello.i  hello.s  hello.s~
```

图 4 - 1

4.3 可重定位目标 elf 格式

分析 `hello.o` 的 ELF 格式，用 `readelf` 等列出其各节的基本信息，特别是重定位项目分析。

首先回忆一下 elf 文件的结构，如图 4-2 所示：

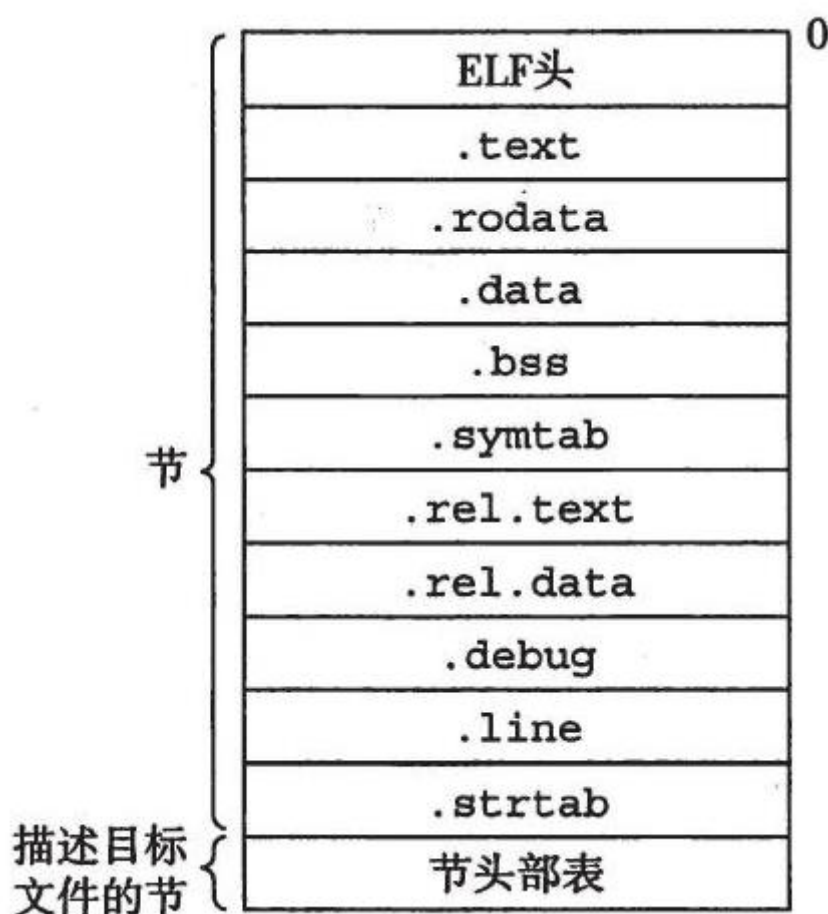


图 4 - 2

在 Ubuntu 下读取分析 hello.o 文件使用如下命令，将 hello.o 文件的基本信息读入到 hello_o.elf 文件中：

```
readelf -a hello.o > hello_o.elf
```

```
hello@ubuntu:~/Final Project$ readelf -a hello.o > hello_o.elf
hello@ubuntu:~/Final Project$
```

图 4 - 3

通过查看 hello_o.elf 文件，我们可以获得如下信息：

- 1) **ELF Header:** 以 16 字节的序列 Magic 开始，这个序列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连接器语法分析和解释目标文件的信息。其中包含 ELF 头的大小、目标文件的类型（如可重定位、可执行或者共享的）、机器类型（如 x86-64）、节头部表的文件偏移，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的条目（entry）。

```

baileys@xf158356768:/mnt/c/Users/Alienware/Desktop/share/Final Project$ readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                  2's complement, little endian
  Version:                              1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  REL (Relocatable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:              1240 (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:             14
  Section header string table index: 13

```

图 4 - 4

- 2) **Section Headers:** 如图 4-5, 节头表包括节名称, 节的类型, 节的属性 (读写权限), 节在 ELF 文件中所占的长度以及节的对齐方式和偏移量。我们可以使用终端指令 `readelf -S hello.o` 来查看节头表。

.text: 已编译程序的机器代码。

.rodata: 只读数据, 比如 `printf` 语句中的格式串和开关语句的跳转表。

.data: 已初始化的全局和静态 C 变量。局部 C 变量在运行时被保存在栈中, 既不出现在 `.data` 节中, 也不出现在 `.bss` 中。

.bss: 未初始化的全局和静态 C 变量, 以及所有被初始化为 0 的全局或静态 C 变量。在目标文件中这个节不占据实际的空间, 它仅仅是一个占位符。目标文件格式区分已初始化和未初始化变量是为了空间效率: 在目标文件中, 未初始化变量不需要占据任何实际的磁盘空间。运行时, 在内存中分配这些变量, 初始值为 0。

.symtab: 一个符号表, 他存放在程序中定义和引用的函数和全局变量的信息。每个可重定位目标文件在 `.symtab` 中都有一张符号表 (除非程序员特意用 `STRIP` 命令去掉它)。然而, 和编译器中的符号表不同, `.symtab` 符号表不包含局部变量的条目。

.rel.text: 一个 `.text` 节中位置的列表, 当连接器把这个目标文件和其他文件组合时, 需要修改这些位置。一般而言, 任何调用外部函数或者引用全局变量的指令都需要修改, 另一方面, 调用本地函数的指令则不需要修改。

.rel.data: 被模块引用或定义的所有全局变量的重定位信息。一般而言, 任何已初始化的全局变量, 如果它的初始值是一个全局变量地址或者外部定义的函数的地址, 都需要被修改。

.debug: 一个调试符号表, 其条目是程序中定义的局部变量和类型定义, 程序中定义和引用的全局变量, 以及原始的 C 源文件。只有以 `-g` 选项调用

编译器驱动程序时，才会得到这张表。

.line: 原始 C 源程序中的行号和.text 节中机器指令之间的映射。只有以-g 选项调用编译器驱动程序时，才会得到这张表。

.strtab: 一个字符串表，其内容包括.symtab 和.debug 节中的符号表，以及节头部中的节名字。字符串表就是以 null 结尾的字符串的序列。

```
baileys@xf1196200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ readelf -S hello.o
There are 14 section headers, starting at offset 0x4d8:
```

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	
	0000000000000092	0000000000000000	AX 0 0 1		
[2]	.rela.text	RELA	0000000000000000	00000388	
	00000000000000c0	0000000000000018	I 11 1 8		
[3]	.data	PROGBITS	0000000000000000	000000d2	
	0000000000000000	0000000000000000	WA 0 0 1		
[4]	.bss	NOBITS	0000000000000000	000000d2	
	0000000000000000	0000000000000000	WA 0 0 1		
[5]	.rodata	PROGBITS	0000000000000000	000000d8	
	0000000000000033	0000000000000000	A 0 0 8		
[6]	.comment	PROGBITS	0000000000000000	0000010b	
	000000000000002b	0000000000000001	MS 0 0 1		
[7]	.note.GNU-stack	PROGBITS	0000000000000000	00000136	
	0000000000000000	0000000000000000	0 0 1		
[8]	.note.gnu.property	NOTE	0000000000000000	00000138	
	0000000000000020	0000000000000000	A 0 0 8		
[9]	.eh_frame	PROGBITS	0000000000000000	00000158	
	0000000000000038	0000000000000000	A 0 0 8		
[10]	.rela.eh_frame	RELA	0000000000000000	00000448	
	0000000000000018	0000000000000018	I 11 9 8		
[11]	.symtab	SYMTAB	0000000000000000	00000190	
	000000000000001b0	0000000000000018	12 10 8		
[12]	.strtab	STRTAB	0000000000000000	00000340	
	0000000000000048	0000000000000000	0 0 1		
[13]	.shstrtab	STRTAB	0000000000000000	00000460	
	0000000000000074	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 4 - 5

除 ELF 头之外，节头表是 ELF 可重定位目标文件中最重要的一部分内容。描述每个节的节名、在文件中的偏移、大小、访问属性、对齐方式等。

3) 符号表：如图 4-6 所示，符号表存放在程序中定义和引用的函数和全局变量的信息。


```

baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ readelf -s hello.o

Symbol table '.symtab' contains 18 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    9
    9: 0000000000000000          0 SECTION LOCAL  DEFAULT    6
   10: 0000000000000000       146 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 atoi
   16: 0000000000000000          0 NOTYPE  GLOBAL  DEFAULT  UND sleep
   17: 0000000000000000          0 NOTYPE  GLOBAL  DEFAULT  UND getchar

```

图 4 - 6

- 4) 重定位表：重定位条目包含了链接时重定位所需的全部信息：需要被重定位的代码在其段中的偏移、该段代码所对应的符号在符号表中的索引以及重定位类型、重定位时被使用到的加数。如图 4-7 所示，为 hello.o 对应的重定位表。

```

Relocation section '.rela.text' at offset 0x388 contains 8 entries:
   Offset          Info          Type           Sym. Value      Sym. Name + Addend
   ---:          -
000000000001c    000500000002  R_X86_64_PC32  0000000000000000 .rodata - 4
0000000000021    000c00000004  R_X86_64_PLT32 0000000000000000 puts - 4
000000000002b    000d00000004  R_X86_64_PLT32 0000000000000000 exit - 4
0000000000054    000500000002  R_X86_64_PC32 0000000000000000 .rodata + 22
000000000005e    000e00000004  R_X86_64_PLT32 0000000000000000 printf - 4
0000000000071    000f00000004  R_X86_64_PLT32 0000000000000000 atoi - 4
0000000000078    001000000004  R_X86_64_PLT32 0000000000000000 sleep - 4
0000000000087    001100000004  R_X86_64_PLT32 0000000000000000 getchar - 4

Relocation section '.rela.eh_frame' at offset 0x448 contains 1 entry:
   Offset          Info          Type           Sym. Value      Sym. Name + Addend
   ---:          -
0000000000020    000200000002  R_X86_64_PC32 0000000000000000 .text + 0

```

图 4 - 7

重定位文件必须知道如何修改其所包含的"节"的内容,在构建可执行文件或共享目标文件时,把节中的符号引用转换成这些符号在进程地址空间中的虚拟地址;包含这些转换信息数据的就是"重定位项"。

带加数的重定位项的格式使用结构体 Elf64_Rela 描述:

```

typedef struct {
    Elf64_Addr    r_offset; /* Address */
    Elf64_Xword   r_info;   /* Relocation type and symbol index */
    Elf64_Sxword  r_addend; /* Addend */
} Elf64_Rela;

```

对以上字段解释:

a) r_offset

此成员指定应用重定位操作的位置。不同的目标文件对于此成员的解释会稍有不同。但对于可重定位文件，该值表示节偏移。重定位节说明如何修改文件中的其他节。重定位偏移会在第二节中指定一个存储单元。

b) `r_info`

此成员指定必须对其进行重定位的符号表索引以及要应用的重定位类型。重定位类型特定于处理器。重定位项的重定位类型或符号表索引是 `ELF32_R_TYPE` 或 `ELF32_R_SYM` 分别应用于项的 `r_info` 成员所得的结果。对于 64 位 SPARC `Elf64_Rela` 结构，`r_info` 字段可进一步细分为 8 位类型标识符和 24 位类型相关数据字段。对于现有的重定位类型，数据字段为零。但是，新的重定位类型可能会使用数据位。

c) `r_addend`

此成员指定常量加数，用于计算将存储在可重定位字段中的值，`Rela` 项包含显式加数。64 位 x86 仅使用 `Elf64_Rela` 重定位项。因此，`r_addend` 成员用作重定位加数。

4.4 `hello.o` 的结果解析

在 Ubuntu 下对 `hello.o` 进行反汇编，应使用如下命令：

```
objdump -d -r hello.o
```

并通过 `objdump -d -r hello.o > hello_o.s` 指令，将反汇编的代码写入 `hello_o.s` 中，如图 4-8 所示。

```

baileys@F1190200780:/mnt/c/Users/Alienware/Desktop/share/Final Project$ objdump -d -r hello.o
hello.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0: f3 0f 1e fa          endbr64
 4: 55                  push  %rbp
 5: 48 89 e5            mov   %rsp,%rbp
 8: 48 83 ec 20         sub   $0x20,%rsp
 c: 89 7d ec           mov   %edi,-0x14(%rbp)
 f: 48 89 75 e0         mov   %rsi,-0x20(%rbp)
13: 83 7d ec 04         cmpl  $0x4,-0x14(%rbp)
17: 74 16              je    2f <main+0x2f>
19: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 20 <main+0x20>
                    1c: R_X86_64_PC32      .rodata+0x4
20: e8 00 00 00 00      callq 25 <main+0x25>
                    21: R_X86_64_PLT32      puts+0x4
25: bf 01 00 00 00      mov   $0x1,%edi
2a: e8 00 00 00 00      callq 2f <main+0x2f>
                    2b: R_X86_64_PLT32      exit+0x4
2f: c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%rbp)
36: eb 48              jmp   80 <main+0x80>
38: 48 8b 45 e0         mov   -0x20(%rbp),%rax
3c: 48 83 c0 10         add   $0x10,%rax
40: 48 8b 10           mov   (%rax),%rdx
43: 48 8b 45 e0         mov   -0x20(%rbp),%rax
47: 48 83 c0 08         add   $0x8,%rax
4b: 48 8b 00           mov   (%rax),%rax
4e: 48 89 c6           mov   %rax,%rsi
51: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi    # 58 <main+0x58>
                    54: R_X86_64_PC32      .rodata+0x22
58: b8 00 00 00 00      mov   $0x0,%eax
5d: e8 00 00 00 00      callq 62 <main+0x62>
                    5e: R_X86_64_PLT32      printf+0x4
62: 48 8b 45 e0         mov   -0x20(%rbp),%rax
66: 48 83 c0 18         add   $0x18,%rax
6a: 48 8b 00           mov   (%rax),%rax
6d: 48 89 c7           mov   %rax,%rdi
70: e8 00 00 00 00      callq 75 <main+0x75>
                    71: R_X86_64_PLT32      atoi+0x4
75: 89 c7              mov   %eax,%edi
77: e8 00 00 00 00      callq 7c <main+0x7c>
                    78: R_X86_64_PLT32      sleep+0x4
7c: 83 45 fc 01         addl  $0x1,-0x4(%rbp)
80: 83 7d fc 07         cmpl  $0x7,-0x4(%rbp)
84: 7e b2              jle   38 <main+0x38>
86: e8 00 00 00 00      callq 8b <main+0x8b>
                    87: R_X86_64_PLT32      getchar+0x4
8b: b8 00 00 00 00      mov   $0x0,%eax
90: c9                  leaveq
91: c3                  retq

baileys@F1190200780:/mnt/c/Users/Alienware/Desktop/share/Final Project$ objdump -d -r hello.o > hello_o.s
baileys@F1190200780:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello.o  hello.s  hello.o.s  hello.o.s.s  hello.o.s.s.s

```

图 4 - 8

通过 VSCode 打开 hello.s 与 hello_o.s，并仔细对比，如图 4-9 所示。

```

2  hello.o:      file format elf64-x86-64
3
4
5  Disassembly of section .text:
6
7  0000000000000000 <main>:
8      0: f3 0f 1e fa      endbr64
9      4: 55                push    %rbp
10     5: 48 89 e5          mov     %rsp,%rbp
11     8: 48 83 ec 20       sub     $0x20,%rsp
12     c: 89 7d ec          mov     %edi,-0x14(%rbp)
13     f: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
14    13: 83 7d ec 04       cmpl    $0x4,-0x14(%rbp)
15    17: 74 16             je      2f <main+0x2f>
16    19: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi
17    1c: R_X86_64_PC32 .rodata-0x4
18    20: e8 00 00 00 00    callq   25 <main+0x25>
19    21: R_X86_64_PLT32 puts-0x4
20    25: bf 01 00 00 00    mov     $0x1,%edi
21    2a: e8 00 00 00 00    callq   2f <main+0x2f>
22    2b: R_X86_64_PLT32 exit-0x4
23    2f: c7 45 fc 00 00 00 movl    $0x0,-0x4(%rbp)
24    36: eb 48             jmp     80 <main+0x80>
25    38: 48 8b 45 e0       mov     -0x20(%rbp),%rax
26    3c: 48 83 c0 10       add     $0x10,%rax
27    40: 48 8b 10          mov     (%rax),%rdx
28    43: 48 8b 45 e0       mov     -0x20(%rbp),%rax
29    47: 48 83 c0 08       add     $0x8,%rax
30    4b: 48 8b 00          mov     (%rax),%rax
31    4e: 48 89 c6          mov     %rax,%rsi
32    51: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi
33    54: R_X86_64_PC32 .rodata+0x22
34    58: b8 00 00 00 00    mov     $0x0,%eax
35    5d: e8 00 00 00 00    callq   62 <main+0x62>
36    5e: R_X86_64_PLT32 printf-0x4
37    62: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38    66: 48 83 c0 18       add     $0x18,%rax
39    6a: 48 8b 00          mov     (%rax),%rax
40    6d: 48 89 c7          mov     %rax,%rdi
41    70: e8 00 00 00 00    callq   75 <main+0x75>
42    71: R_X86_64_PLT32 atoi-0x4
43    75: 89 c7             mov     %eax,%edi
44    77: e8 00 00 00 00    callq   7c <main+0x7c>
45    78: R_X86_64_PLT32 sleep-0x4
46    7c: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
47    80: 83 7d fc 07       cmpl    $0x7,-0x4(%rbp)
48    84: 7e b2             jle     38 <main+0x38>
49    86: e8 00 00 00 00    callq   8b <main+0x8b>
50    87: R_X86_64_PLT32 getchar-0x4
51    8b: b8 00 00 00 00    mov     $0x0,%eax
52    90: c9               leaveq  %eax
53    91: c3               retq
54

```

```

.type    %a Abi
main:
.LFB6:
.cfi_startproc
endbr64
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     $4, -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
movq     -32(%rbp), %rax
addq     $24, %rax
movq     (%rax), %rax
movq     %rax, %rdi
call     atoi@PLT
movl     %eax, %edi
call     sleep@PLT
addl     $1, -4(%rbp)
.L3:
cmpl     $7, -4(%rbp)
jle      .L4
call     getchar@PLT
movl     $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE6:
.size    main, .-main
.ident   "GCC: (Ubuntu 9.3.0-
.section .note.GNU-stack,
.section .note.gnu.proper
.align   8
.long    1f - 0f

```

图 4 - 9

从图中可以看出，hello.o 的反汇编代码和 hello.s 大致相同，在某些地方存在小部分地区别。

区别如下：

- 1) 分支转移：hello.s 文件中分支通过使用段名称跳转，而在 hello.o 的反汇编代码中，通过地址跳转。

如图 4-10，我们可以发现在 hello.s 中，分支转移通过.L2、.L3 进行跳转。如图 4-11，我们可以发现，在 hello.o 的反汇编代码中，分支转移通过计算地址，进行跳转。

```

    cmpb    $4, -20(%rbp)
    je      .L2
    leaq     .LC0(%rip),
    call     puts@PLT
    movl     $1, %edi
    call     exit@PLT
.L2:
    movl     $0, -4(%rbp)
    jmp      .L3
.L4:

```

图 4 - 10

```

1c: R_X86_64_PC32 .rodata-0x4
20: e8 00 00 00 00    callq 25 <main+0x25>
21: R_X86_64_PLT32 puts-0x4
25: bf 01 00 00 00    mov    $0x1,%edi
2a: e8 00 00 00 00    callq 2f <main+0x2f>
2b: R_X86_64_PLT32 exit-0x4

```

图 4 - 11

- 2) 函数调用：如图 4-12，在 hello.s 文件中，函数调用 call 只需要加函数名称，在 hello.o 反汇编代码中，如图 4-13，call 则是使用了当前指令的下一个字节。原因是因为该函数迟绑定，该函数为共享库中函数，只有运行时，动态链接器作用后才能确定相应的 PLT 条目地址。

```

call     printf@PLT
movq     -32(%rbp), %rax
addq     $24, %rax
movq     (%rax), %rax
movq     %rax, %rdi
call     atoi@PLT

```

图 4 - 12


```

5d: e8 00 00 00 00    callq 62 <main+0x62>
| | 5e: R_X86_64_PLT32 printf-0x4
62: 48 8b 45 e0       mov    -0x20(%rbp),%rax
66: 48 83 c0 18       add    $0x18,%rax
6a: 48 8b 00          mov    (%rax),%rax
6d: 48 89 c7          mov    %rax,%rdi
70: e8 00 00 00 00    callq 75 <main+0x75>
| | 71: R_X86_64_PLT32 atoi-0x4
75: 89 c7             mov    %eax,%edi
77: e8 00 00 00 00    callq 7c <main+0x7c>

```

图 4 - 13

- 3) 操作数：如图 4-14，在 hello.s 中，操作数为十进制数，而在 hello.o 的反汇编代码中，如图 4-15，操作数为十六进制数。

```

.cfi_def_cfa_register 6
subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $4, -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

```

图 4 - 14

```

endbr64
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
cmpl    $0x4,-0x14(%rbp)
je      2f <main+0x2f>
leaq    0x0(%rip),%rdi
.rodata-0x4
callq   25 <main+0x25>
2: puts-0x4
mov     $0x1,%edi
callq   2f <main+0x2f>
2: exit-0x4
movl    $0x0,-0x4(%rbp)
jmp     80 <main+0x80>

```

图 4 - 15

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

说明：

机器语言：二进制的机器指令的集合；

机器指令：由操作码和操作数构成的；

机器语言：灵活、直接执行和速度快。

汇编语言：主体是汇编指令，是机器指令便于记忆并表示形式，为了方便程序员读懂和记忆的语言指令。

汇编指令和机器指令在指令的表示方法上有所不同。

4.5 本章小结

本章对汇编做了详尽的介绍，详细介绍了汇编的概念、作用，实践了在 Ubuntu 下汇编的命令，以及详细解读了可重定位目录 elf 目标格式，并对 hello.o 的结果作出了详细的分析，将 hello.s 和 hello.o 的反汇编代码对比，并讨论不同之处，有利于我更深入的理解汇编。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

链接的概念：链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。链接可以执行于编译时，也就是在源代码被翻译成机器代码时，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；甚至执行于运行时，也就是由应用程序来执行。在现代系统中，链接是由叫做连接器的程序自动执行的。

链接的作用：链接使分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。当我们该边这些模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其他文件。

5.2 在 Ubuntu 下链接的命令

在 Ubuntu 下链接，应使用如下命令：

```
ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o
/usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
```

或 `gcc hello.o -o hello`

如图 5-1 使用方法一链接，5-2 使用方法二链接。(最终保留通过 `ld` 链接生成的可执行文件)

```
baileys@xf1198288708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ 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
baileys@xf1198288708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello hello.o hello.o hello.o hello.o hello.o hello.o elf hello.o
```

图 5 - 1

```
baileys@xf1198288708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello.c hello.i hello.o hello.s hello.o.elf hello.o.s
baileys@xf1198288708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ gcc hello.o -o hello
baileys@xf1198288708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello hello.c hello.i hello.o hello.s hello.o.elf hello.o.s
baileys@xf1198288708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ |
```

图 5 - 2

5.3 可执行目标文件 hello 的格式

首先获取 hello 文件的 elf 格式文件，通过如下命令：

```
Readelf -a hello > hello_elf
```

```
baileys@xf1188x00700:/mnt/c/Users/Alienware/Desktop/share/Final Project$ readelf -a hello > hello_elf
```

图 5 - 3

执行上述命令，如图 5-3 所示。

1) 读取 hello 的 ELF 头，如图 5-4 所示。

```
baileys@xf1188x00700:/mnt/c/Users/Alienware/Desktop/share/Final Project$ readelf -h hello
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:                               EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:                0x4010f0
  Start of program headers:           64 (bytes into file)
  Start of section headers:          14208 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           12
  Size of section headers:            64 (bytes)
  Number of section headers:           27
  Section header string table index: 26
```

图 5 - 4

以 16 字节的序列 **Magic** 开始，这个序列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连接器语法分析和解释目标文件的信息。其中包含 ELF 头的大小、目标文件的类型（如可重定位、可执行或者共享的）、机器类型（如 x86-64）、节头部表的文件偏移，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的条目（entry），与链接前的 ELF header 相比，除了系统决定的基本信息不变，section header 和 program header 都增加，并且增加了入口处的地址。

2) 读取 hello 的节头表，如图 5-5 所示。

Section Headers 对 hello 中所有节的信息进行了声明，其中包括大小 Size 以及在程序中的偏移量 Offset，因此根据 Section Headers 中的信息，我们可以用 Hexedit 定位到各个节所在的空间。Address 为程序被载入到虚拟地址的起始空间。

```

baileys@xf1198288788:/mnt/c/Users/Alienware/Desktop/share/Final Project$ readelf -S hello
There are 27 section headers, starting at offset 0x3780:

Section Headers:
[Nr] Name              Type              Address            Offset
     Size              EntSize           Flags Link Info  Align
[ 0]                      NULL              0000000000000000  0 0 0
[ 1] .interp              PROGBITS          0000000004002e0  000002e0
     000000000000001c  0000000000000000  A 0 0 1
[ 2] .note.gnu.property  NOTE              000000000400300  00000300
     0000000000000020  0000000000000000  A 0 0 8
[ 3] .note.ABI-tag        NOTE              000000000400320  00000320
     0000000000000020  0000000000000000  A 0 0 4
[ 4] .hash                HASH              000000000400340  00000340
     0000000000000038  0000000000000004  A 6 0 8
[ 5] .gnu.hash            GNU_HASH          000000000400378  00000378
     000000000000001c  0000000000000000  A 6 0 8
[ 6] .dynsym              DYNSYM            000000000400398  00000398
     00000000000000d8  0000000000000018  A 7 1 8
[ 7] .dynstr              STRTAB            000000000400470  00000470
     000000000000005c  0000000000000000  A 0 0 1
[ 8] .gnu.version         VERSYM            0000000004004cc  000004cc
     0000000000000012  0000000000000002  A 6 0 2
[ 9] .gnu.version_r       VERNEED           0000000004004e0  000004e0
     0000000000000020  0000000000000000  A 7 1 8
[10] .rela.dyn            RELA              000000000400500  00000500
     0000000000000030  0000000000000018  A 6 0 8
[11] .rela.plt            RELA              000000000400530  00000530
     0000000000000090  0000000000000018  AI 6 21 8
[12] .init                PROGBITS          000000000401000  00001000
     000000000000001b  0000000000000000  AX 0 0 4
[13] .plt                 PROGBITS          000000000401020  00001020
     0000000000000070  0000000000000010  AX 0 0 16
[14] .plt.sec             PROGBITS          000000000401090  00001090
     0000000000000060  0000000000000010  AX 0 0 16
[15] .text                PROGBITS          0000000004010f0  000010f0
     0000000000000145  0000000000000000  AX 0 0 16
[16] .fini                PROGBITS          000000000401238  00001238
     000000000000000d  0000000000000000  AX 0 0 4
[17] .rodata              PROGBITS          000000000402000  00002000
     000000000000003b  0000000000000000  A 0 0 8
[18] .eh_frame            PROGBITS          000000000402040  00002040
     00000000000000fc  0000000000000000  A 0 0 8
[19] .dynamic              DYNAMIC           000000000403e50  00002e50
     00000000000001a0  0000000000000010  WA 7 0 8
[20] .got                  PROGBITS          000000000403ff0  00002ff0
     0000000000000010  0000000000000008  WA 0 0 8
[21] .got.plt              PROGBITS          000000000404000  00003000
     0000000000000048  0000000000000008  WA 0 0 8
[22] .data                 PROGBITS          000000000404048  00003048
     0000000000000004  0000000000000000  WA 0 0 1
[23] .comment              PROGBITS          000000000000000  0000304c
     000000000000002a  0000000000000001  MS 0 0 1
[24] .symtab               SYMTAB            000000000000000  00003078
     000000000000004c8  0000000000000018  25 30 8
[25] .strtab               STRTAB            000000000000000  00003540
     0000000000000158  0000000000000000  0 0 1
[26] .shstrtab             STRTAB            000000000000000  00003698
     00000000000000e1  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 - 5

- 3) 程序头部表，ELF 文件头结构就像是一个总览图，描述了整个文件的布局情况，因此在 ELF 文件头结构允许的数值范围，整个文件的大小是可以动态增减的，告诉系统如何创建进程映像。如图 5-6 所示，即为 hello 可执行文件的程序头部表。

Program Headers:				
Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align	
PHDR	0x0000000000000040 0x00000000000002a0	0x0000000000400040 0x000000000002a0	0x0000000000400040 R	0x8
INTERP	0x00000000000002e0 0x00000000000001c	0x00000000004002e0 0x00000000000001c	0x00000000004002e0 R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]				
LOAD	0x0000000000000000 0x00000000000005c0	0x0000000000400000 0x00000000000005c0	0x0000000000400000 R	0x1000
LOAD	0x0000000000001000 0x0000000000000245	0x0000000000401000 0x0000000000000245	0x0000000000401000 R E	0x1000
LOAD	0x0000000000002000 0x000000000000013c	0x0000000000402000 0x000000000000013c	0x0000000000402000 R	0x1000
LOAD	0x0000000000002e50 0x00000000000001fc	0x0000000000403e50 0x00000000000001fc	0x0000000000403e50 RW	0x1000
DYNAMIC	0x0000000000002e50 0x00000000000001a0	0x0000000000403e50 0x00000000000001a0	0x0000000000403e50 RW	0x8
NOTE	0x0000000000000300 0x0000000000000020	0x0000000000400300 0x0000000000000020	0x0000000000400300 R	0x8
NOTE	0x0000000000000320 0x0000000000000020	0x0000000000400320 0x0000000000000020	0x0000000000400320 R	0x4
GNU_PROPERTY	0x0000000000000300 0x0000000000000020	0x0000000000400300 0x0000000000000020	0x0000000000400300 R	0x8
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW	0x10
GNU_RELRO	0x0000000000002e50 0x00000000000001b0	0x0000000000403e50 0x00000000000001b0	0x0000000000403e50 R	0x1

图 5 - 6

- 4) 动态偏移表：若目标文件参与动态链接，则其程序头表将包含一个类型为 PT_DYNAMIC 的元素，此段包含 .dynamic 节。特殊符号 _DYNAMIC 用于标记包含以下结构的数组的节，如图 5-7 所示。

```
Dynamic section at offset 0x2e50 contains 21 entries:
  Tag          Type          Name/Value
0x0000000000000001 (NEEDED)      Shared library: [libc.so.6]
0x000000000000000c (INIT)        0x401000
0x000000000000000d (FINI)        0x401238
0x0000000000000004 (HASH)        0x400340
0x0000000006ffffef5 (GNU_HASH)       0x400378
0x0000000000000005 (STRTAB)       0x400470
0x0000000000000006 (SYMTAB)       0x400398
0x000000000000000a (STRSZ)        92 (bytes)
0x000000000000000b (SYMENT)       24 (bytes)
0x0000000000000015 (DEBUG)        0x0
0x0000000000000003 (PLTGOT)       0x404000
0x0000000000000002 (PLTRELSZ)     144 (bytes)
0x0000000000000014 (PLTREL)       RELA
0x0000000000000017 (JMPREL)       0x400530
0x0000000000000007 (RELA)        0x400500
0x0000000000000008 (RELASZ)      48 (bytes)
0x0000000000000009 (RELAENT)     24 (bytes)
0x0000000006ffffffe (VERNEED)     0x4004e0
0x0000000006fffffff (VERNEEDNUM) 1
0x0000000006ffffff0 (VERSYM)     0x4004cc
0x0000000000000000 (NULL)        0x0
```

图 5 - 7

- 5) 重定位节`.rela.text`，一个`.text`节中位置的列表，包含`.text`节中需要进行重定位的信息，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。如图 5-8 所示，分别描述了 `main`、`puts`、`printf`、`getchar`、`atoi`、`exit`、`sleep` 函数的重定位声明。

```
Relocation section '.rela.dyn' at offset 0x500 contains 2 entries:
  Offset      Info      Type           Sym. Value  Sym. Name + Addend
000000403ff0  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000403ff8  000500000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x530 contains 6 entries:
  Offset      Info      Type           Sym. Value  Sym. Name + Addend
000000404018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000404020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000404028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 getchar@GLIBC_2.2.5 + 0
000000404030  000600000007 R_X86_64_JUMP_SLO 0000000000000000 atoi@GLIBC_2.2.5 + 0
000000404038  000700000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
000000404040  000800000007 R_X86_64_JUMP_SLO 0000000000000000 sleep@GLIBC_2.2.5 + 0
```

图 5 - 8

- 6) 符号表节：目标文件的符号表包含定位和重定位程序的符号定义和符号引用所需的信息。符号表索引是此数组的下标。索引 0 指定表中第一项用作未定义的符号索引。如图 5-9 所示。


```

Symbol table '.symtab' contains 51 entries:

```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	000000000004002e0	0	SECTION	LOCAL	DEFAULT	1	
2:	00000000000400300	0	SECTION	LOCAL	DEFAULT	2	
3:	00000000000400320	0	SECTION	LOCAL	DEFAULT	3	
4:	00000000000400340	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000000400378	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000000400398	0	SECTION	LOCAL	DEFAULT	6	
7:	00000000000400470	0	SECTION	LOCAL	DEFAULT	7	
8:	000000000004004cc	0	SECTION	LOCAL	DEFAULT	8	
9:	000000000004004e0	0	SECTION	LOCAL	DEFAULT	9	
10:	00000000000400500	0	SECTION	LOCAL	DEFAULT	10	
11:	00000000000400530	0	SECTION	LOCAL	DEFAULT	11	
12:	00000000000401000	0	SECTION	LOCAL	DEFAULT	12	
13:	00000000000401020	0	SECTION	LOCAL	DEFAULT	13	
14:	00000000000401090	0	SECTION	LOCAL	DEFAULT	14	
15:	000000000004010f0	0	SECTION	LOCAL	DEFAULT	15	
16:	00000000000401238	0	SECTION	LOCAL	DEFAULT	16	
17:	00000000000402000	0	SECTION	LOCAL	DEFAULT	17	
18:	00000000000402040	0	SECTION	LOCAL	DEFAULT	18	
19:	00000000000403e50	0	SECTION	LOCAL	DEFAULT	19	
20:	00000000000403ff0	0	SECTION	LOCAL	DEFAULT	20	
21:	00000000000404000	0	SECTION	LOCAL	DEFAULT	21	
22:	00000000000404048	0	SECTION	LOCAL	DEFAULT	22	
23:	00000000000000000	0	SECTION	LOCAL	DEFAULT	23	
24:	00000000000000000	0	FILE	LOCAL	DEFAULT	ABS	hello.c
25:	00000000000000000	0	FILE	LOCAL	DEFAULT	ABS	
26:	00000000000403e50	0	NOTYPE	LOCAL	DEFAULT	19	__init_array_end
27:	00000000000403e50	0	OBJECT	LOCAL	DEFAULT	19	__DYNAMIC
28:	00000000000403e50	0	NOTYPE	LOCAL	DEFAULT	19	__init_array_start
29:	00000000000404000	0	OBJECT	LOCAL	DEFAULT	21	__GLOBAL_OFFSET_TABLE__
30:	00000000000401230	5	FUNC	GLOBAL	DEFAULT	15	__libc_csu_fini
31:	00000000000404048	0	NOTYPE	WEAK	DEFAULT	22	data_start
32:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@@GLIBC_2.2.5
33:	0000000000040404c	0	NOTYPE	GLOBAL	DEFAULT	22	_edata
34:	00000000000401238	0	FUNC	GLOBAL	HIDDEN	16	_fini
35:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@@GLIBC_2.2.5
36:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@@GLIBC_2.2.5
37:	00000000000404048	0	NOTYPE	GLOBAL	DEFAULT	22	__data_start
38:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	getchar@@GLIBC_2.2.5
39:	00000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
40:	00000000000402000	4	OBJECT	GLOBAL	DEFAULT	17	_IO_stdin_used
41:	000000000004011c0	101	FUNC	GLOBAL	DEFAULT	15	__libc_csu_init
42:	00000000000404050	0	NOTYPE	GLOBAL	DEFAULT	22	_end
43:	00000000000401120	5	FUNC	GLOBAL	HIDDEN	15	_dl_relocate_static_pie
44:	000000000004010f0	47	FUNC	GLOBAL	DEFAULT	15	_start
45:	0000000000040404c	0	NOTYPE	GLOBAL	DEFAULT	22	__bss_start
46:	00000000000401125	146	FUNC	GLOBAL	DEFAULT	15	main
47:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	atoi@@GLIBC_2.2.5
48:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	exit@@GLIBC_2.2.5
49:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	sleep@@GLIBC_2.2.5
50:	00000000000401000	0	FUNC	GLOBAL	HIDDEN	12	_init

图 5 - 9

- 7) 动态符号表用来保存于动态链接相关的导入导出符号，不包括模块内部的符号。如图 5-10 所示。

Symbol table '.dynsym' contains 9 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	atoi@GLIBC_2.2.5 (2)
7:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	exit@GLIBC_2.2.5 (2)
8:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	sleep@GLIBC_2.2.5 (2)

图 5 - 10

5.4 hello 的虚拟地址空间

- 1) 打开 edb

```
xf1190200708@1190200708:~/桌面/edb-debugger/build$ ./edb
Starting edb version: 1.3.0
Please Report Bugs & Requests At: https://github.com/eteran/edb-debugger/issues
```

图 5 - 11

- 2) 加载 hello 可执行文件

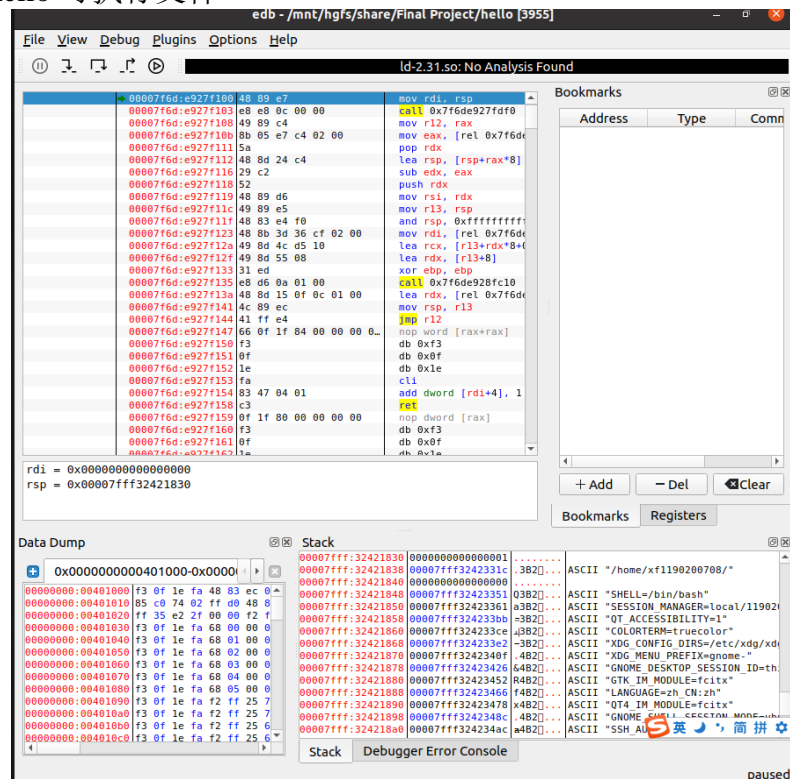


图 5 - 12

- 3) 观察 edb 的 Data Dump 窗口。窗口显示虚拟地址由 0x401000 开始，到 0x401ff0 结束。之间对应 5.3 中的节头表的声明。根据截图 5-5，可得起始位置为 0x401000。Data Dump 窗口如图 5-13 所示：

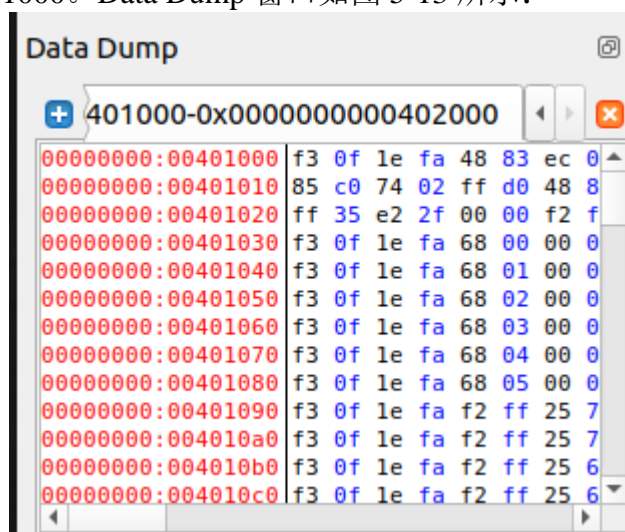


图 5 - 13

- 4) 观察 edb 的 Symbols 的窗口，如图 5-14 所示，与 5-5 中各节所示一致。

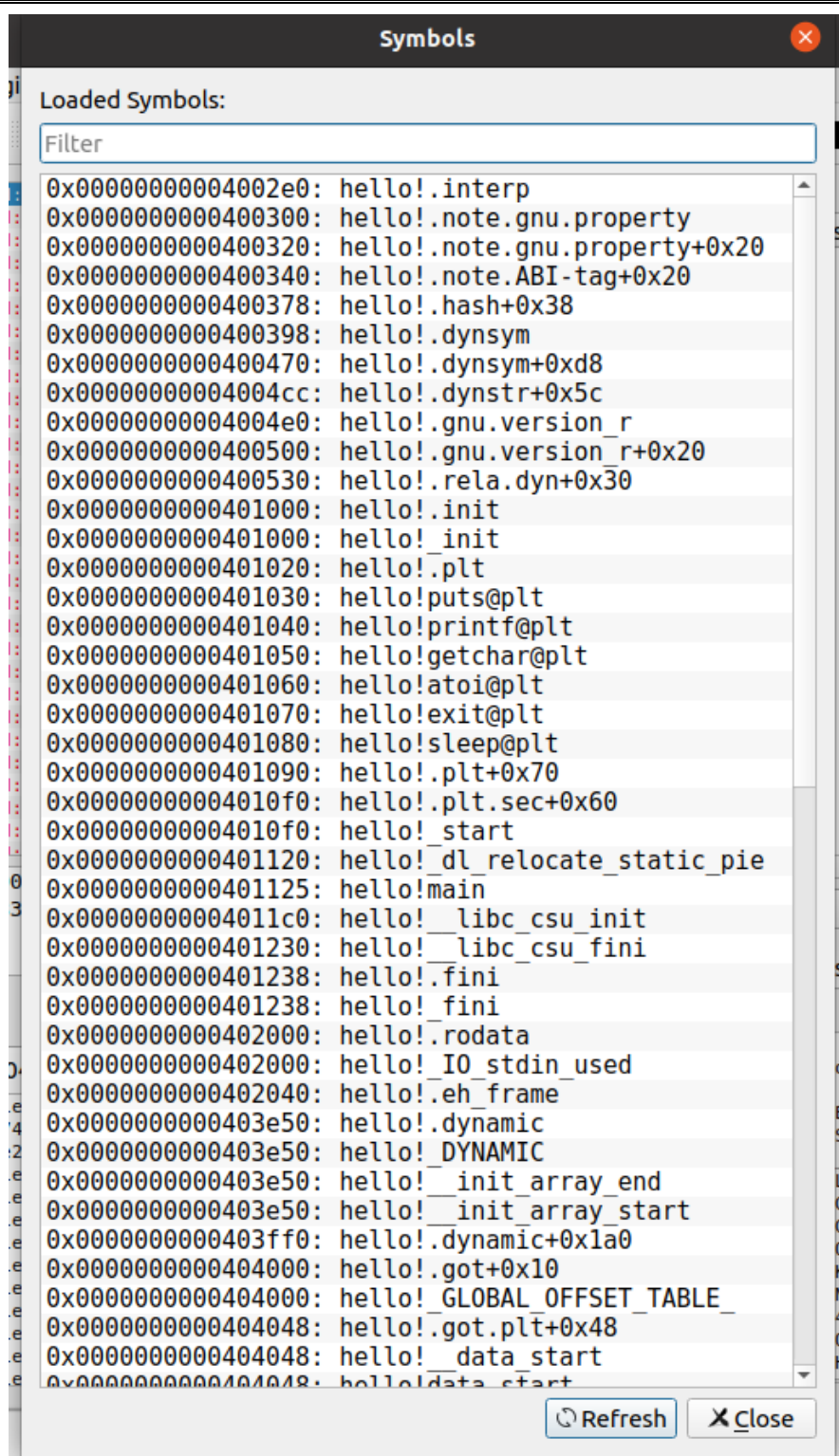


图 5 - 14

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息，并与 5.3 对照分析说明。

5.5 链接的重定位过程分析

使用 `objdump -d -r hello` 指令对 `hello` 反汇编，如图 5-15 所示：

```
hailuys@f1190200700:/mnt/c/Users/Alienware/Desktop/share/Final Project$ objdump -d -r hello > hello_asm.s
hailuys@f1190200700:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ls
hello hello.c hello.i hello.o hello.s hello.wasm.s hello.wif hello.w.wif hello.w.s
```

图 5 - 15

打开 VSCode 检查 `hello` 与 `hello.o` 的反汇编文件的不同。如图 5-16 所示。

<pre>0000000000000000 <main>: 0: f3 0f 1e fa endbr64 4: 55 push %rbp 5: 48 89 e5 mov %rsp,%rbp 8: 48 83 ec 20 sub \$0x20,%rsp c: 89 7d ec mov %edi,-0x14(%rbp) f: 48 89 75 e0 mov %rsi,-0x20(%rbp) 13: 83 7d ec 04 cmpl \$0x4,-0x14(%rbp) 17: 74 16 je 2f <main+0x2f> 19: 48 8d 3d 00 00 00 lea 0x0(%rip),%rdi 1c: R_X86_64_PC32 .rodata-0x4 20: e8 00 00 00 00 callq 25 <main+0x25> 21: R_X86_64_PLT32 puts-0x4 25: bf 01 00 00 00 mov \$0x1,%edi 2a: e8 00 00 00 00 callq 2f <main+0x2f> 2b: R_X86_64_PLT32 exit-0x4 2f: c7 45 fc 00 00 00 movl \$0x0,-0x4(%rbp) 36: eb 48 jmp 80 <main+0x80> 38: 48 8b 45 e0 mov -0x20(%rbp),%rax 3c: 48 83 c0 10 add \$0x10,%rax 40: 48 8b 10 mov (%rax),%rdx 43: 48 8b 45 e0 mov -0x20(%rbp),%rax 47: 48 83 c0 08 add \$0x8,%rax 4b: 48 8b 00 mov (%rax),%rax 4e: 48 89 c6 mov %rax,%rsi 51: 48 8d 3d 00 00 00 lea 0x0(%rip),%rdi 54: R_X86_64_PC32 .rodata+0x22 58: b8 00 00 00 00 mov \$0x0,%eax 5d: e8 00 00 00 00 callq 62 <main+0x62> 5e: R_X86_64_PLT32 printf-0x4 62: 48 8b 45 e0 mov -0x20(%rbp),%rax 66: 48 83 c0 18 add \$0x18,%rax 6a: 48 8b 00 mov (%rax),%rax 6d: 48 89 c7 mov %rax,%rdi 70: e8 00 00 00 00 callq 75 <main+0x75> 71: R_X86_64_PLT32 atoi-0x4 75: 89 c7 mov %eax,%edi 77: e8 00 00 00 00 callq 7c <main+0x7c> 78: R_X86_64_PLT32 sleep-0x4 7c: 83 45 fc 01 addl \$0x1,-0x4(%rbp) 80: 83 7d fc 07 cmpl \$0x7,-0x4(%rbp) 84: 7e b2 jle 38 <main+0x38> 86: e8 00 00 00 00 callq 8b <main+0x8b> 87: R_X86_64_PLT32 getchar-0x4 8b: b8 00 00 00 00 mov \$0x0,%eax 90: c9 leaveq %rax,%rsi 91: c3 retq</pre>	<pre>101 102 0000000000401125 <main>: 103 401125: f3 0f 1e fa endbr64 104 401129: 55 push %rbp 105 40112a: 48 89 e5 mov %rsp,%rbp 106 40112d: 48 83 ec 20 sub \$0x20,%rsp 107 401131: 89 7d ec mov %edi,-0x14(%rbp) 108 401134: 48 89 75 e0 mov %rsi,-0x20(%rbp) 109 401138: 83 7d ec 04 cmpl \$0x4,-0x14(%rbp) 110 40113c: 74 16 je 401154 <main+0x2f> 111 40113e: 48 8d 3d c3 0e 00 lea 0xec3(%rip),%rdi 112 401145: e8 46 ff ff ff callq 401090 <puts@plt> 113 40114a: bf 01 00 00 00 mov \$0x1,%edi 114 40114f: e8 7c ff ff ff callq 4010d0 <exit@plt> 115 401154: c7 45 fc 00 00 00 movl \$0x0,-0x4(%rbp) 116 40115b: eb 48 jmp 4011a5 <main+0x80> 117 40115d: 48 8b 45 e0 mov -0x20(%rbp),%rax 118 401161: 48 83 c0 10 add \$0x10,%rax 119 401165: 48 8b 10 mov (%rax),%rdx 120 401168: 48 8b 45 e0 mov -0x20(%rbp),%rax 121 40116c: 48 83 c0 08 add \$0x8,%rax 122 401170: 48 8b 00 mov (%rax),%rax 123 401173: 48 89 c6 mov %rax,%rsi 124 401176: 48 8d 3d b1 0e 00 lea 0xeb1(%rip),%rdi 125 40117d: b8 00 00 00 00 mov \$0x0,%eax 126 401182: e8 19 ff ff ff callq 4010a0 <printf@plt> 127 401187: 48 8b 45 e0 mov -0x20(%rbp),%rax 128 40118b: 48 83 c0 18 add \$0x18,%rax 129 40118f: 48 8b 00 mov (%rax),%rax 130 401192: 48 89 c7 mov %rax,%rdi 131 401195: e8 26 ff ff ff callq 4010c0 <atoi@plt> 132 40119a: 89 c7 mov %eax,%edi 133 40119c: e8 3f ff ff ff callq 4010e0 <sleep@plt> 134 4011a1: 83 45 fc 01 addl \$0x1,-0x4(%rbp) 135 4011a5: 83 7d fc 07 cmpl \$0x7,-0x4(%rbp) 136 4011a9: 7e b2 jle 40115d <main+0x38> 137 4011ab: e8 00 ff ff ff callq 4010b0 <getchar@plt> 138 4011b0: b8 00 00 00 00 mov \$0x0,%eax 139 4011b5: c9 leaveq %rax,%rsi 140 4011b6: c3 retq 141 4011b7: 66 0f 1f 84 00 00 nopw 0x0(%rax,%rax,1) 142 4011be: 00 00 143 144 00000000004011c0 <__libc_csu_init>: 145 4011c0: f3 0f 1e fa endbr64 146 4011c4: 41 57 push %r15 147 4011c6: 4c 8d 3d 83 2c 00 lea 0x2c83(%rip),%r15 148 4011cd: 41 56 push %r14 149 4011cf: 49 89 d6 mov %rdx,%r14 150 4011d2: 41 55 push %r13</pre>
--	---

图 5 - 16

分析 `hello` 反汇编文件与 `hello.o` 反汇编文件的区别：

- 1) `hello` 的反汇编代码中，比 `hello.o` 的反汇编代码中多了 `.init` 节，`.plt` 节，`.fini` 节，`.plt.sec` 等。如图 5-17 所示。

```

Disassembly of section .init:

0000000000401000 <_init>:
 401000: f3 0f 1e fa      endbr64
 401004: 48 83 ec 08      sub    $0x8,%rsp
 401008: 48 8b 05 e9 2f 00 00 mov    0x2fe9(%rip),%rax
 40100f: 48 85 c0         test   %rax,%rax
 401012: 74 02           je     401016 <_init+0x16>
 401014: ff d0           callq  *%rax
 401016: 48 83 c4 08      add    $0x8,%rsp
 40101a: c3             retq

Disassembly of section .plt:

0000000000401020 <.plt>:
 401020: ff 35 e2 2f 00 00 pushq  0x2fe2(%rip)      #
 401026: f2 ff 25 e3 2f 00 00 bnd jmpq *0x2fe3(%rip)
 40102d: 0f 1f 00        nopl   (%rax)
 401030: f3 0f 1e fa      endbr64
 401034: 68 00 00 00 00 00 pushq  $0x0

```

图 5 - 17

- 2) hello的反汇编代码中增加了外部链接的共享库函数。例如在hello的反汇编代码中可以看到puts@plt等，如图5-18、图5-19、图5-20所示atoi函数所示。

```

6d: 48 89 c7      mov    %rax,%rdi
70: e8 00 00 00 00 callq  75 <main+0x75>
71: R_X86_64_PLT32 atoi-0x4

```

图 5 - 18

```

401192: 48 89 c7      mov    %rax,%rdi
401195: e8 26 ff ff ff callq  4010c0 <atoi@plt>

```

图 5 - 19

```

00000000004010c0 <atoi@plt>:
 4010c0: f3 0f 1e fa      endbr64
 4010c4: f2 ff 25 65 2f 00 00 bnd jmpq *0x2f65(%rip)      # 404030 <atoi@GLIBC_2.2.5>
 4010cb: 0f 1f 44 00 00    nopl   0x0(%rax,%rax,1)

```

图 5 - 20

- 3) hello的反汇编代码相比hello.o的反汇编代码，跳转地址修改为了虚拟内存地址。如图5-18、图5-19所示。此时callq后跟的指令为0xff ff ff 26 = -0xda，且此时执行到的地址为0x401195，0x401195+0x5-0xda=0x4010c0。此时0x4010c0为atoi@plt函数的地址。
- 4) hello中节的起始位置修改为了虚拟地址，hello.o反汇编代码中为相对偏移地址。如图5-21所示。


```

Disassembly of section .text:
0000000000000000 <main>:
0: f3 0f 1e fa      endbr64
4: 55               push  %rbp
5: 48 89 e5         mov   %rsp,%rbp
8: 48 83 ec 20      sub   $0x20,%rsp
c: 89 7d ec         mov   %edi,-0x14(%rbp)
f: 48 89 75 e0      mov   %rsi,-0x20(%rbp)
13: 83 7d ec 04      cmpl  $0x4,-0x14(%rbp)
17: 74 16           je    2f <main+0x2f>
19: 48 8d 3d 00 00 00 lea    0x0(%rip),%rdi
      1c: R_X86_64_PC32 .rodata-0x4
20: e8 00 00 00 00   callq 25 <main+0x25>
      21: R_X86_64_PLT32 puts-0x4
25: bf 01 00 00 00   mov   $0x1,%edi
2a: e8 00 00 00 00   callq 2f <main+0x2f>
      2b: R_X86_64_PLT32 exit-0x4
2f: c7 45 fc 00 00 00 movl   $0x0,-0x4(%rbp)
36: eb 48           jmp    80 <main+0x80>
38: 48 8b 45 e0      mov   -0x20(%rbp),%rax
3c: 48 83 c0 10      add   $0x10,%rax
40: 48 8b 10        mov   (%rax),%rdx
43: 48 8b 45 e0      mov   -0x20(%rbp),%rax
47: 48 83 c0 08      add   $0x8,%rax
4b: 48 8b 00        mov   (%rax),%rax

Disassembly of section .init:
0000000000000000 <_init>:
8: 401000: f3 0f 1e fa      endbr64
9: 401004: 48 83 ec 08      sub   $0x8,%rsp
10: 401008: 48 8b 05 e9 2f 00 00 mov   0x2fe9(%rip),%rax
11: 40100f: 48 85 c0         test  %rax,%rax
12: 401012: 74 02           je    401016 <_init+0x16>
13: 401014: ff d0           callq *%rax
14: 401016: 48 83 c4 08      add   $0x8,%rsp
15: 40101a: c3              retq

Disassembly of section .plt:
0000000000000000 <.plt>:
20: 401020: ff 35 e2 2f 00 00 pushq 0x2fe2(%rip)
21: 401026: f2 ff 25 e3 2f 00 00 bnd jmpq *0x2fe3(%rip)
22: 40102d: 0f 1f 00        nopl   (%rax)
23: 401030: f3 0f 1e fa      endbr64
24: 401034: 68 00 00 00 00   pushq $0x0
25: 401039: f2 e9 e1 ff ff ff bnd jmpq 401020 <.plt>
26: 40103f: 90              nop
27: 401040: f3 0f 1e fa      endbr64
28: 401044: 68 01 00 00 00   pushq $0x1
29: 401049: f2 e9 d1 ff ff ff bnd jmpq 401020 <.plt>
30: 40104f: 90              nop

```

图 5 - 21

hello如何重定位的:

重定位节和符号定义: 在这一步, 链接器将所有相同类型的节合并为同一类型的新的聚合节。

重定位节中的符号引用: 在这一步中, 连接器修改代码节和数据节中对每个符号的引用, 使得它们指向正确运行时的地址, 要执行这一步, 链接器依赖于可重定位目标模块中成为重定位条目的数据结构。

5.6 hello 的执行流程

本处使用 gdb 进行追踪, 首先使用 objdump 指令, 获得 hello 中的函数, 如图 5-22 所示。

函数	地址
<_init>	0x0000000000401000
<puts@plt>	0x0000000000401090
<printf@plt>	0x00000000004010a0
<getchar@plt>	0x00000000004010b0
<atoi@plt>	0x00000000004010c0
<exit@plt>	0x00000000004010d0
<sleep@plt>	0x00000000004010e0
<_start>	0x00000000004010f0
<_dl_relocate_static_pie>	0x0000000000401120
<main>	0x0000000000401125
<__libc_csu_init>	0x00000000004011c0
<__libc_csu_fini>	0x0000000000401230
<_fini>	0x0000000000401238

```

kali@kali:~/Final Project$ objdump -d hello | grep "<.*>:"
0000000000401000 <_init>:
0000000000401020 <_plt>:
0000000000401090 <puts@plt>:
00000000004010a0 <printf@plt>:
00000000004010b0 <getchar@plt>:
00000000004010c0 <atoi@plt>:
00000000004010d0 <exit@plt>:
00000000004010e0 <sleep@plt>:
00000000004010f0 <_start>:
0000000000401120 <_dl_relocate_static_pie>:
0000000000401125 <main>:
00000000004011c0 <__libc_csu_init>:
0000000000401230 <__libc_csu_fini>:
0000000000401238 <_fini>:

```

图 5 - 22

并在使用 gdb 调试 hello 的时候，对每个函数加上断点，如图 5-23 所示：

```

(gdb) b _init
Breakpoint 1 at 0x401000
(gdb) b puts@plt
Breakpoint 2 at 0x401090
(gdb) b printf@plt
Breakpoint 3 at 0x4010a0
(gdb) b getchar@plt
Breakpoint 4 at 0x4010b0
(gdb) b atoi@plt
Breakpoint 5 at 0x4010c0
(gdb) b exit@plt
Breakpoint 6 at 0x4010d0
(gdb) b sleep@plt
Breakpoint 7 at 0x4010e0
(gdb) b _start
Breakpoint 8 at 0x4010f0
(gdb) b _dl_relocate_static_pie
Breakpoint 9 at 0x401120
(gdb) b main
Breakpoint 10 at 0x401125
(gdb) b __libc_csu_init
Breakpoint 11 at 0x4011c0
(gdb) b __libc_csu_fini
Breakpoint 12 at 0x401230
(gdb) b _fini
Breakpoint 13 at 0x401238

```

图 5 - 23

使用 r 1190200708 熊峰 2 开始调试，如图 5-24 所示：

```
(gdb) r 1190200708 熊峰 2
Starting program: /mnt/c/Users/Alienware/Desktop/share/Final Project/hello 1190200708 熊峰 2

Breakpoint 1, _init (argc=4, argv=0x7fffffffdd28, envp=0x7fffffffdd50) at init-first.c:52
52      init-first.c: No such file or directory.
(gdb) c
Continuing.
```

图 5 - 24

运行流程如下，如图 5-25 所示：首先调用 0x401000 处的 `_init` 函数，然后调用 0x4010f0 处的 `_start` 函数，然后调用 0x4011c8 处的 `__libc_csu_init` 函数，然后再调用 `_init` 函数，之后就进入了 0x401125 处的 `main` 函数。

```
Breakpoint 1, _init (argc=4, argv=0x7fffffffdd28, envp=0x7fffffffdd50) at init-first.c:52
52      init-first.c: No such file or directory.
(gdb) c
Continuing.

Breakpoint 8, 0x000000004010f0 in _start ()
(gdb) c
Continuing.

Breakpoint 11, 0x000000004011c8 in __libc_csu_init ()
(gdb) c
Continuing.

Breakpoint 1, 0x00000000401000 in _init ()
(gdb) c
Continuing.

Breakpoint 10, 0x00000000401125 in main ()
(gdb) c
Continuing.
```

图 5 - 25

如图 5-26，在进入 `main` 函数后，首先调用位于 0x4010a0 处的 `printf@plt` 函数。然后调用 0x4010c0 处的 `atoi@plt` 函数，然后调用位于 0x4010e0 处的 `sleep@plt` 函数，循环八次。

```
Breakpoint 3, 0x000000004010a0 in printf@plt ()
(gdb) c
Continuing.
Hello 1190200708 熊峰

Breakpoint 5, 0x000000004010c0 in atoi@plt ()
(gdb) c
Continuing.

Breakpoint 7, 0x000000004010e0 in sleep@plt ()
(gdb) c
Continuing.

Breakpoint 3, 0x000000004010a0 in printf@plt ()
(gdb) c
Continuing.
Hello 1190200708 熊峰

Breakpoint 5, 0x000000004010c0 in atoi@plt ()
(gdb) c
Continuing.

Breakpoint 7, 0x000000004010e0 in sleep@plt ()
(gdb) c
Continuing.

Breakpoint 3, 0x000000004010a0 in printf@plt ()
(gdb) |
```

图 5 - 26

如图 5-27, 在最后一次调用 `sleep@plt` 函数后, 调用位于 `0x4010b0` 处的 `getchar` 函数@plt 函数。

```
Breakpoint 7, 0x00000000004010e0 in sleep@plt ()
(gdb) c
Continuing.

Breakpoint 4, 0x00000000004010b0 in getchar@plt ()
(gdb) c
Continuing.
```

图 5 - 27

如图 5-28, 最后调用 `_fini` 函数, 程序调试结束。

```
Breakpoint 13, 0x0000000000401238 in _fini ()
(gdb) c
Continuing.
[Inferior 1 (process 32) exited normally]
```

图 5 - 28

5.7 Hello 的动态链接分析

动态链接说明: 共享库是一个目标模块, 在运行或加载时, 可以加载到任意的内存地址, 并和一个在内存中的程序链接起来, 这个过程称为动态链接, 由动态连接器执行。共享库是以两种不同的方式来“共享”的。首先, 在任何给定的文件系统中, 对于一个库只有一个 .so 文件。所有引用该库的可执行目标文件共享这个 .so 文件中的代码和数据, 而不是像静态库的内容那样被复制和嵌入到引用它们的可执行的文件中。其次, 在内存中, 一个共享库的 .text 节的一个副本可以被不同的正在运行的进程共享。

对于动态链接库中的函数, 编译器无法预测函数运行时的地址, 因此动态链接库的函数在程序执行的时候才会确定地址, 所以需要为其添加重定位记录, 并等待动态连接器处理。为避免运行时候修改调用模块的代码段, 链接器采用延迟绑定的策略。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。前者是数据段的一部分, 后者是代码段的一部分。GOT 中存放函数目标地址, PLT 使用 GOT 中地址跳转到目标函数。PLT 是一个数组, 其中每个条目是 16 字节代码。每个库函数都有自己的 PLT 条目, PLT[0] 是一个特殊的条目, 跳转到动态链接器中。从 PLT[2] 开始的条目调用用户代码调用的函数。

GOT 同样是一个数组, 每个条目是 8 字节的地址, 和 PLT 联合使用时, GOT[2]

是动态链接在 ld-linux.so 模块的入口点，其余条目对应于被调用的函数，在运行时被解析。每个条目都有匹配的 PLT 条目。

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。

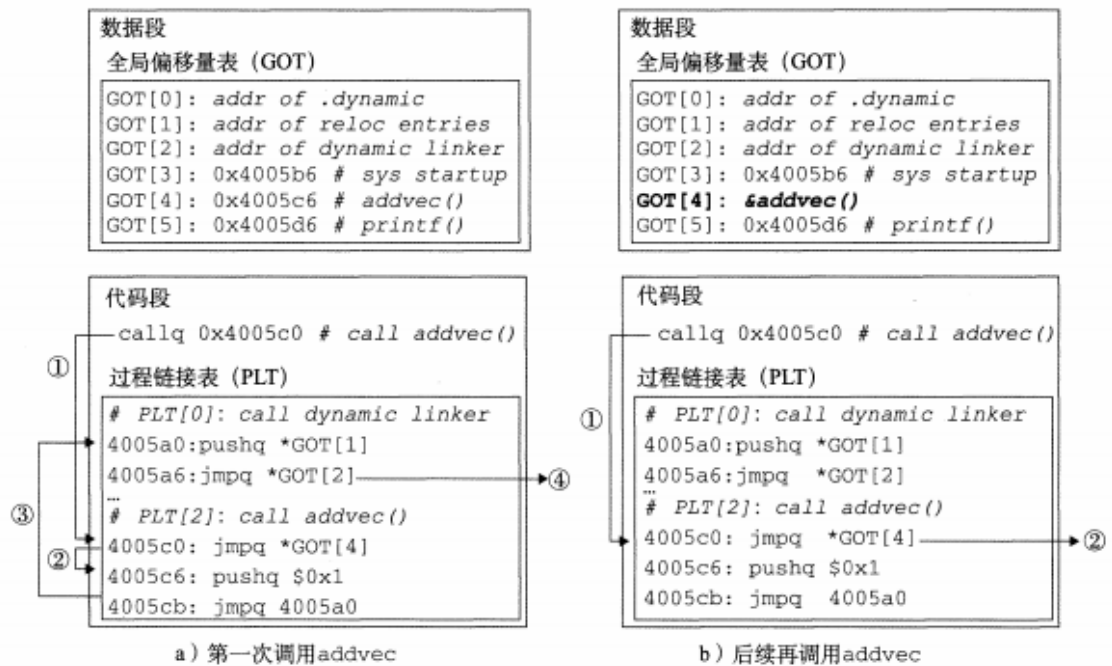


图 5 - 29

通过 `readelf` 工具，在 `hello` 的节头表中可以发现 GOT 表，如图 5-30 所示。

	00000000000000fc	0000000000000000	A	0	0	8
[19]	.dynamic	DYNAMIC	0000000000403e50	00002e50		
	00000000000001a0	0000000000000010	WA	7	0	8
[20]	.got	PROGBITS	0000000000403ff0	00002ff0		
	0000000000000010	0000000000000008	WA	0	0	8
[21]	.got.plt	PROGBITS	0000000000404000	00003000		
	0000000000000048	0000000000000008	WA	0	0	8
[22]	.data	PROGBITS	0000000000404048	00003048		
	0000000000000004	0000000000000000	WA	0	0	1
[23]	comment	PROGBITS	0000000000000000	0000304c		

图 5 - 30

可以看到 `.got.plt` 位于 `0x404000` 处，如图 5-31 所示。

```

Contents of section .got.plt:
 404000 503e4000 00000000 00000000 00000000  P>@.....
 404010 00000000 00000000 30104000 00000000  .....0.@.....
 404020 40104000 00000000 50104000 00000000  @.@.....P.@.....
 404030 60104000 00000000 70104000 00000000  `.@.....p.@.....
 404040 80104000 00000000  ..@.....

```

图 5 - 31

我们可以看到在执行 `_init` 之后，发生了变化。如图 5-30 及 5-31 所示。

```

(gdb) b _init
Breakpoint 1 at 0x401000
(gdb) r 1190200708 熊峰 2
Starting program: /mnt/c/Users/Alienware/Desktop/share/Final Project/hello 1190200708 熊峰 2

Breakpoint 1, _init (argc=4, argv=0x7fffffffdd28, envp=0x7fffffffdd50) at init-first.c:52
52      init-first.c: No such file or directory.
(gdb) p/40xb
Size letters are meaningless in "print" command.
(gdb) x/40xb 0x404000
0x404000:      0x50      0x3e      0x40      0x00      0x00      0x00      0x00      0x00
0x404008:      0x90      0xe1      0xff      0xf7      0xff      0x7f      0x00      0x00
0x404010:      0xb0      0x7b      0xfe      0xf7      0xff      0x7f      0x00      0x00
0x404018: <puts@got.plt>:      0x30      0x10      0x40      0x00      0x00      0x00      0x00      0x00
0x404020: <printf@got.plt>:      0x40      0x10      0x40      0x00      0x00      0x00      0x00      0x00

```

图 5 - 32

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

5.8 本章小结

本章详细讨论了链接的概念与作用、在 `Ubuntu` 下链接的命令，以及可执行目标文件 `hello` 的格式，详细分析可执行目标文件和可重定位目标文件的区别。详细解释了 `hello` 虚拟地址空间。细致分析了链接的重定位过程。模拟 `hello` 的执行流程，并深入探讨了 `hello` 的动态链接。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

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

进程的作用：

在现代系统上运行一个程序时，进程为用户提供一个假象：就好像我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

每次用户通过向 `shell` 输入一个可执行目标文件的名字，运行程序时，`shell` 就会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

进程提供给应用程序两个关键抽象：一个独立的逻辑控制流；一个私有的地址空间。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell-bash 的作用：Shell 是一个命令行解释器，它为用户提供了一个向 Linux 内核发送请求以便运行程序的界面系统级程序，用户可以用 Shell 来启动、挂起、停止甚至编写一些程序。Shell 还是一个功能相当强大的编程语言，易编写，易调试，灵活性较强。Shell 是解释执行的脚本语言，在 Shell 中可以直接调用 Linux 系统命令。bash 提供了一个图形化界面，提升交互速度。

Shell-bash 的处理流程：

- 1) 从终端或控制台获取用户输入命令
- 2) 将用户输入命令进行解析，判断输入命令是否为内置命令
- 3) 若是内置命令，则直接执行；若不是内置命令，则 `bash` 在初始子进程上下

文中加载和运行它

- 4) 判断程序的执行状态，若为前台进程则等待进程结束；否则直接将进程放在后台执行，继续等待用户下一次输入。

6.3 Hello 的 fork 进程创建过程

在终端中输入 `./hello 1190200708 熊峰 5`，shell 首先判断它不是内置命令，于是 shell 查找当前目录下的可执行文件 `hello`，并将其调入内存，shell 将它解释为系统功能函数并交给内核执行。Shell 通过 `pid_t fork(void)` 函数创建一个子进程，子进程会获得与父进程虚拟地址空间相同的一段数据结构的副本。父进程与子进程最大的不同在于他们分别拥有不同的 PID。

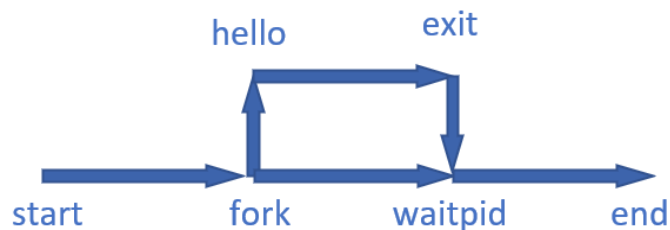


图 6 - 1

6.4 Hello 的 execve 过程

`execve` 函数加载并运行可执行目标文件 `hello`，且包含相对应的一个带参数的列表 `argv` 和环境变量 `exenvp`，只有当出现错误时，例如找不到 `hello` 文件等，`execve` 才会返回 -1 到调用程序，`execve` 调用成功则不会产生。函数原型为：`int exeve(const char *filename, const char *argv[], const char *envp[])`。

在 shell 调用 `fork` 函数之后，`execve` 调用驻留在内存中的被称为启动加载器的操作系统代码来执行程序，使用启动加载器，子进程调用 `execve` 函数，在当前进程即子进程的上下文中加载新程序 `hello`，这个程序覆盖当前正在执行的进程的所有的地址空间，但是这样的操作并没有创建一个新的进程，新的进程有和原先进程相同的 PID，并且他还继承了打开 `hello` 调用 `execve` 函数之前所有已经打开的文件描述符。新的栈和堆段都会被初始化为零，新的代码和数据段被初始化为可执

行文件中的内容。只有一些调用程序头部的信息才可能会在加载的过程中被从可执行磁盘复制到对应的可执行区域的内存。

如图 6-2 所示，当 main 开始执行时，一个典型的用户栈组织结构如下：

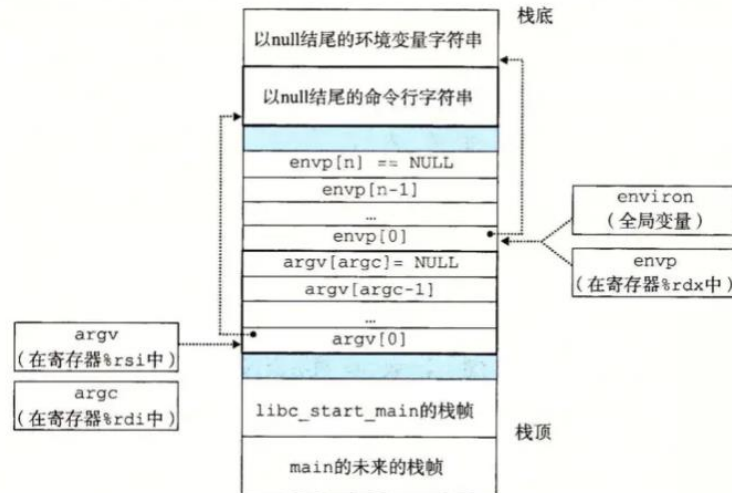


图 6 - 2

如图 6-3 所示，linux x86-64 运行的内存映像。

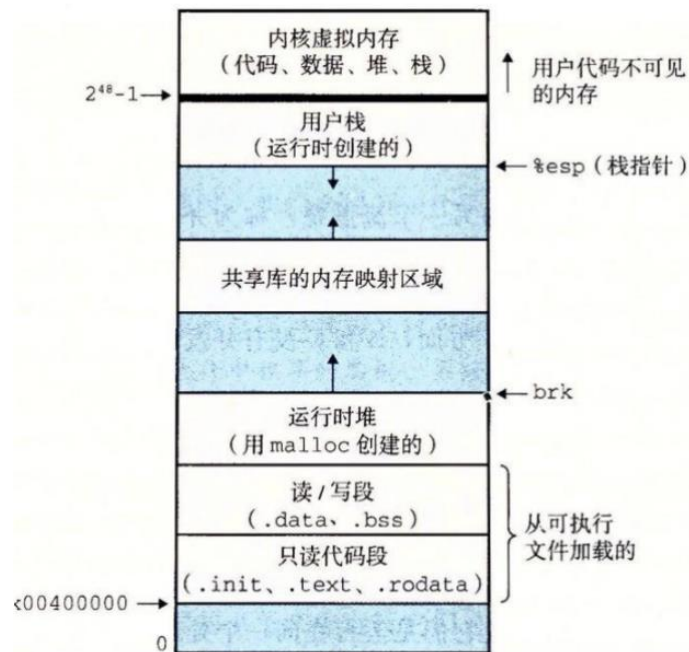


图 6 - 3

6.5 Hello 的进程执行

在现代系统上运行一个程序时，进程为用户提供一个假象：就好像我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。这种处理效果的具体实现效果本身就是一个逻辑控制流，它指的是一系列可执行程序计数器 `pc` 的值，这些计数值唯一的定义对应于那些包含在程序的可执行文件目标对象中的可执行指令，或者说它指的是那些包含在程序运行时可以动态通过链接触到可执行程序的共享文件对象的可执行指令。

进程的时间片是指一个进程在执行控制流时候所处的每一个时间段。

处理器通过设置在某个控制寄存器中的一个模式位来限制一个程序可以执行的指令以及它可以访问的地址空间。没有设置模式位时，进程就运行在用户模式中。用户模式下不允许执行特权指令，不允许使用或者访问内核区的代码或者数据。设置模式位时，进程处于内核模式，该进程可以访问系统中的任何内存位置，可以执行指令集中的任何命令。进程从用户模式变为内核模式的唯一方式是使用诸如终端、故障或陷入系统调用这样的异常。异常发生时，控制传递到异常处理程序，处理器从用户模式转到内核模式。

上下文在运行时候的状态这也就是一个进程内核重新开始启动一个被其他进程或对象库所抢占的网络服务器时，该进程所可能需要的一个下文状态。它由通用寄存器、浮点数据寄存器、程序执行计数器、用户栈、状态数据寄存器、内部多核栈和各种应用内核数据结构等各种应用对象的最大值数据寄存器组成。

在调用进程发送 `sleep` 之前，`hello` 在当前的用户内核模式下进程继续运行，在内核中进程再次调用当前的 `sleep` 之后进程转入用户内核等待休眠模式，内核中所有正在处理等待休眠请求的应用程序主动请求释放当前正在发送处理 `sleep` 休眠请求的进程，将当前调用 `hello` 的进程自动加入正在执行等待的队列，移除或退出正在内核中执行的进程等待队列。

设置定时器，休眠的时间等于自己设置的时间，当定时器时间到时，发送一个中断信号。内核收到中断信号进行终端处理，`hello` 被重新加入运行队列，等待执行，这时候 `hello` 就可以运行在自己的逻辑控制流。

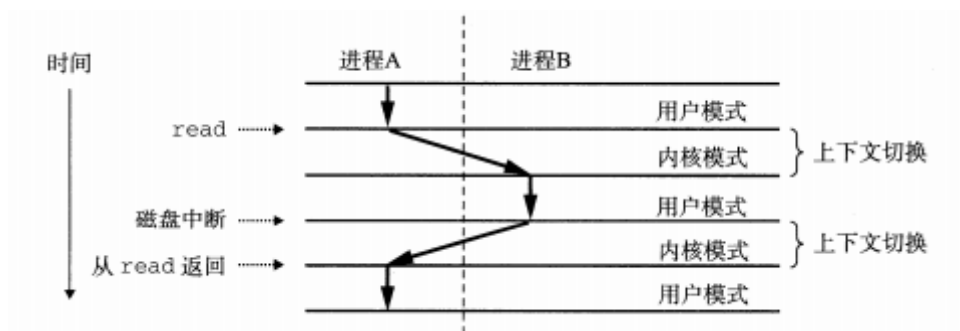


图 6 - 4

6.6 hello 的异常与信号处理

hello 的异常：在执行过程中，来自处理器外部 I/O 设备的信号的结果，例如 Ctrl-C、Ctrl-Z 等；或有意的执行指令的结果，如系统调用的等。

产生的信号：SIGINT, SIGSTP, SIGCONT, SIGWINCH

1) Ctrl-Z

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ./hello 1190200708 熊峰 2
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
^Z
[1]+  Stopped                  ./hello 1190200708 熊峰 2
```

图 6 - 5

图 6-5 为正常执行 hello 程序。

当按下 Ctrl-Z 的时候，将会给进程发出 SIGSTP 信号，hello 程序被挂起，使用 ps 查看，如图 6-6 所示，发现存在 hello，直到 SIGCONT 信号到来：

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ps
PID TTY          TIME CMD
309 pts/1        00:00:00 bash
322 pts/1        00:00:00 hello
323 pts/1        00:00:00 ps
```

图 6 - 6


```

baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ jobs
[1]+  Stopped                  ./hello 1190200708 熊峰 2
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ fg
./hello 1190200708 熊峰 2
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰

```

图 6 - 7

```

391 pts/1    00:00:00 hello
392 pts/1    00:00:00 ps
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ kill -9 391
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ps
  PID TTY          TIME CMD
  309 pts/1    00:00:00 bash
  393 pts/1    00:00:00 ps
[1]+  Killed                  ./hello 1190200708 熊峰 2
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ jobs
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ps
  PID TTY          TIME CMD
  309 pts/1    00:00:00 bash
  394 pts/1    00:00:00 ps

```

图 6 - 8

如图6-8，对挂起的程序，发送SIGKILL信号，强制终止进程。

2) Ctrl-C

```

baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ./hello 1190200708 熊峰 2
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
^C
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Final Project$ ps
  PID TTY          TIME CMD
  309 pts/1    00:00:00 bash
  331 pts/1    00:00:00 ps

```

图 6 - 9

当按下 ctrl+c 的时候，父进程会接收到 SIGINT 信号，使 hello 结束运行，用调用 wait 等函数回收 hello 进程，使用 ps 查看，如图 6-7 所示，当前后台中已没有 hello 进程。

3) 在运行过程中乱按键盘

```

xf1190200708@1190200708: /mnt/c/Users/Alienware/Desktop/share/Final Project$ ./hello 1190200708 熊峰 2
Hello 1190200708 熊峰
dasdasda
Hello 1190200708 熊峰
fdsfdf
Hello 1190200708 熊峰
weewfwef
Hello 1190200708 熊峰
qweqeqweq
Hello 1190200708 熊峰
dqwqq
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
xf1190200708@1190200708: /mnt/c/Users/Alienware/Desktop/share/Final Project$ fdsfdf
fdsfdf: command not found
xf1190200708@1190200708: /mnt/c/Users/Alienware/Desktop/share/Final Project$ weewfwef
weewfwef: command not found
xf1190200708@1190200708: /mnt/c/Users/Alienware/Desktop/share/Final Project$ qweqeqweq
qweqeqweq: command not found
xf1190200708@1190200708: /mnt/c/Users/Alienware/Desktop/share/Final Project$ dqwqq
dqwqq: command not found

```

图 6 - 10

乱按的内容会在屏幕上显示，但是不影响程序的继续执行，`getchar` 将读入一行输入，并且 shell 会将之后输入的字符串当作新的指令。

4) pstree

```

xf1190200708@1190200708: ~/桌面/hitcs/Final Project$ ./hello 1190200708 熊峰 3
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
Hello 1190200708 熊峰
^Z
[1]+  已停止                  ./hello 1190200708 熊峰 3
xf1190200708@1190200708: ~/桌面/hitcs/Final Project$ pstree
systemd└─ModemManager─2*[{ModemManager}]
          └─NetworkManager─2*[{NetworkManager}]

```

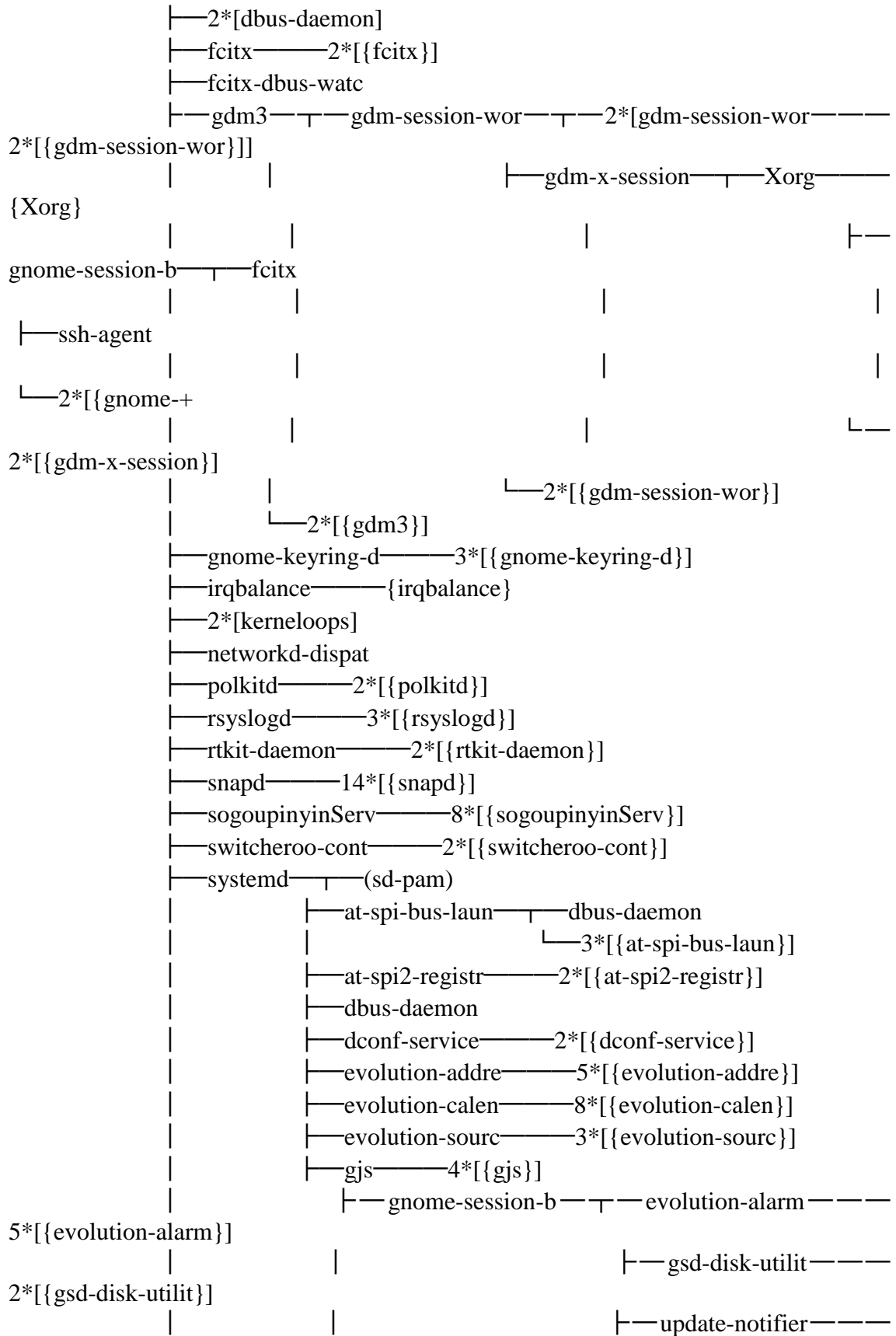
图 6 - 11

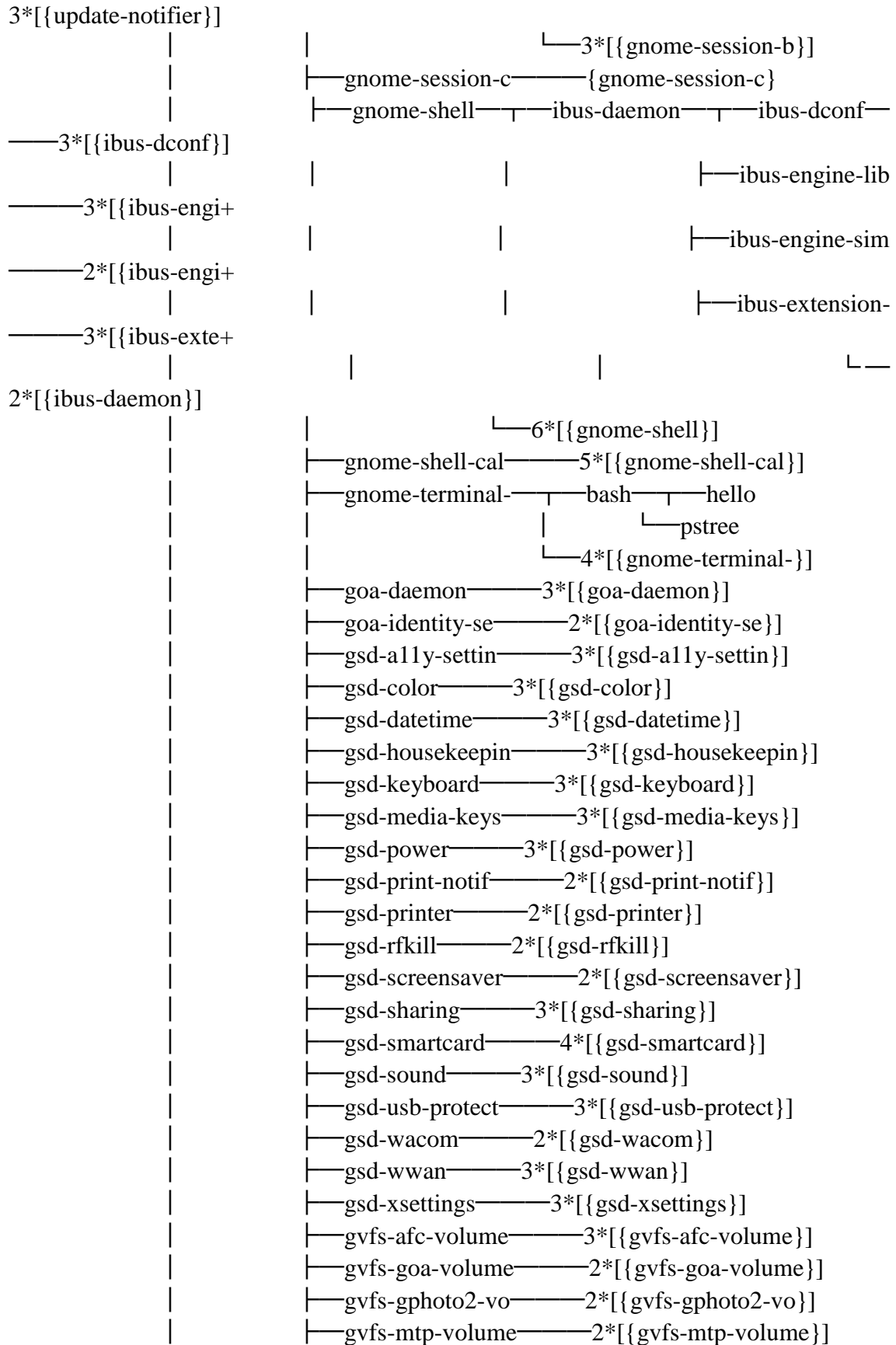
pstree 如下：

```

systemd└─ModemManager──2*[{ModemManager}]
          └─NetworkManager──2*[{NetworkManager}]
            └─VGAuthService
              └─accounts-daemon──2*[{accounts-daemon}]
                └─acpid
                  └─avahi-daemon──avahi-daemon
                    └─bluetoothd
                      └─colord──2*[{colord}]
                        └─cron
                          └─cups-browsed──2*[{cups-browsed}]
                            └─cupsd

```





6.7 本章小结

(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

1) 逻辑地址:

逻辑地址 (Logical Address) 是指由程序 hello 产生的与段相关的偏移地址部分 (hello.o)，它由选择符和偏移量组成。

2) 线性地址:

线性地址 (Linear Address) 是逻辑地址到物理地址变换之间的中间层。具体的格式可以表示为: 虚拟地址描述符: 偏移量程序。它作为一个逻辑分页地址被输入到一个物理地址变换之间的一个中间层, 在分页地址变换机制中需要使用一个线性分页地址描述符作为输入, 线性地址可以再经过物理地址的变换以产生一个新的物理分页地址。再不同时启用一个分页地址机制的情况下, 线性地址本身就是与虚拟地址同意义的。hello 的代码会产生逻辑地址, 或者说是 (即 hello 程序) 段中的偏移地址, 它加上相应段的基地址就生成了一个线性地址。

3) 虚拟地址:

有时我们也把逻辑地址称为虚拟地址。因为与虚拟内存空间的概念类似, 逻辑地址也是与实际物理内存容量无关的, 是 hello 中的虚拟地址。

4) 物理地址:

物理地址 (Physical Address) 是指出现在 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 hello 的线性地址会使用页目录和页表中的项转换成 hello 的物理地址; 如果没有启用分页机制, 那么 hello 的线性地址就是物理地址。

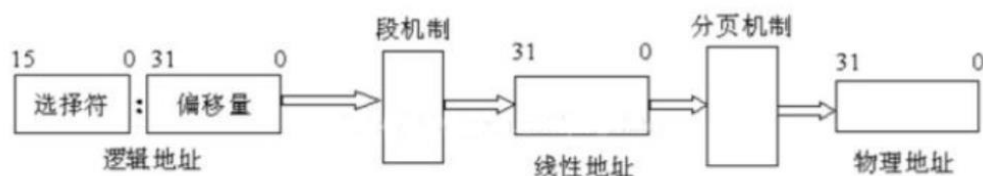


图 7 - 1

如图 7-1 所示。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

1) 将内存分为不同的段，段有段寄存器对应，段寄存器：**CS**（代码段）：程序代码所在段；**SS**（段栈）：栈区所在段；**DS**（数据段）：全局静态数据区所在段，以及**ES**、**GS**、**FS**等备用寄存器。分段功能在实模式和保护模式下有所不同：

- a) 实模式：逻辑地址 **CS:EA** 到物理地址 $CS \times 16 + EA$
- b) 保护模式：以段描述符作为下标，到 **GDT/LDT** 表查表获得段地址，
段地址+偏移地址=线性地址

段选择符的组成和定义如下，分别指的是索引为位（**index**），**TI**，**RPL**，当索引位 **TI=0** 时，段描述符在 **rpgdt** 中，**TI=1** 时，段描述表在 **rpldt** 中。而索引位 **index** 就类似于一个数组，每个元素内都存放一个段的描述符，索引位首地址就是我们在查找段描述符时在这个元素数组当中的索引。一个段描述符的首地址是指含有 8 个元素的字节，我们通常可以查找到段描述符之后获取段的首地址，再把它与线性逻辑地址的偏移量进行相加就可以得到段所需要的一个线性逻辑地址。

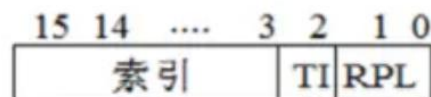


图 7 - 2

如图7-3所示，其中**RPL**提供权限管理，**TI**提供描述表选择。高13位则作为地址，给出当前段在段描述符表中的位置。即在整个段表中的偏移。

其后，即可按照一定的流程进行逻辑地址到线性地址转化：

在段选择符中，用**TI**全段选择附表的类型，在使用高13位的地址，找到段描述符表，获得32位的段基址，再将段内偏移与段基址相加，获得32位线性地址，但是在现代**Linux**系统中，不适用分段机制，将前16位全设置为0，逻辑地址即线性地址。

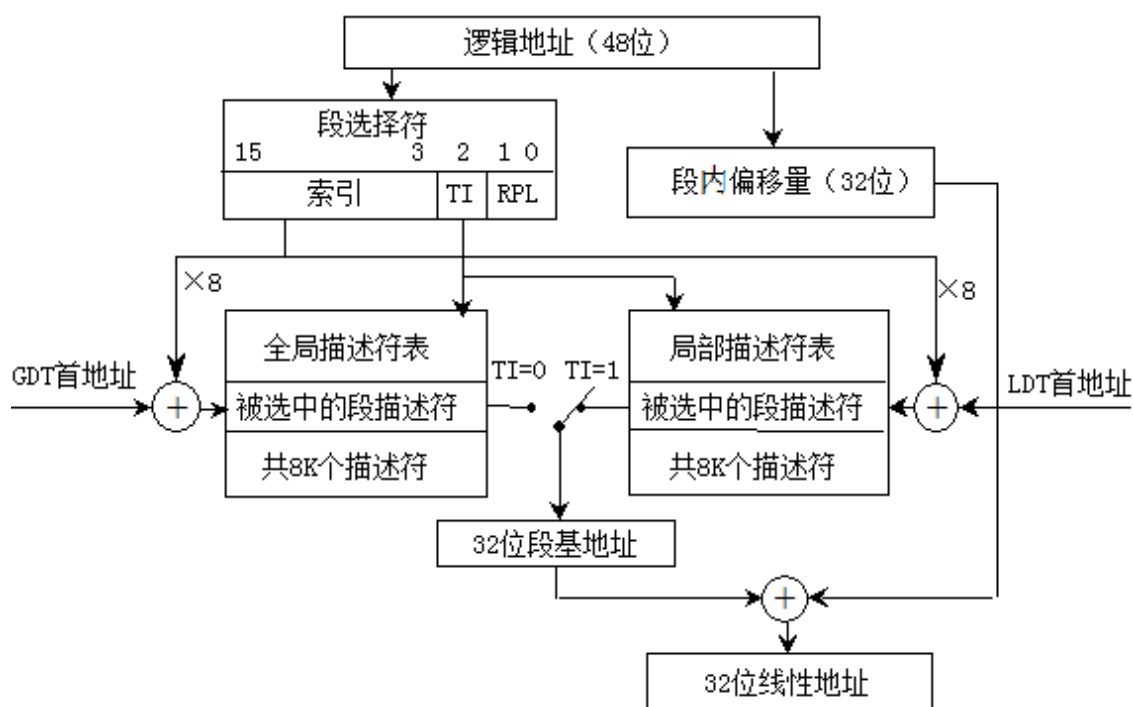


图 7 - 3

7.3 Hello 的线性地址到物理地址的变换-页式管理

通过分页机制实现线性地址（虚拟地址 VA）到物理地址（PA）之间的转换，分页机制是指对虚拟地址内存空间进行分页。

- 1) 首先 Linux 系统有自己的虚拟内存系统，Linux 将虚拟内存组织成一些段的集合，段之外的虚拟内存不存在因此不需要记录。
- 2) 内核为 hello 进程维护了一个段的任务结构即图中的 `task_struct`，其中条目 `mm->mm_struct`（描述了虚拟内存的当前状态），`pgd->`第一级页表的基址（结合一个进程一串页表），`mmap->vm_area_struct` 的链表。一个链表条目对应一个段，链表相连指出了 hello 进程虚拟内存中的所有段。

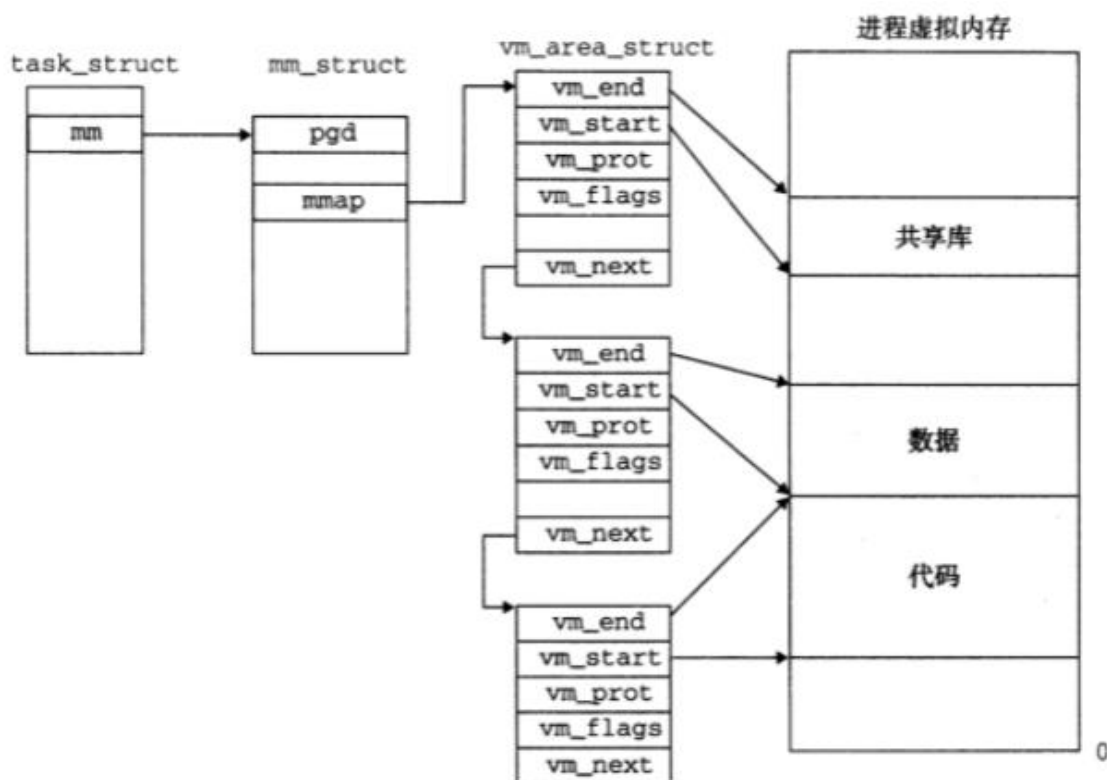


图 7 - 4

- 3) 虚拟页 (VP)：系统将每个段分割为大小固定的块，来作为进行数据传输的单元。对于 Linux，每个虚拟页大小为 4KB.
- 4) 物理页 (PP/页帧)：类似于虚拟页，虚拟内存也被分割。虚拟内存系统中 MMU 负责地址翻译，MMU 使用页表，即存一种放在物理内存中的数据结构，将虚拟页到物理页映射，即虚拟地址到物理地址的映射。
- 5) 通过页表机制寄存器 PTBR+VPN 在页表中获得 PTE，PTE 中包含有效位、权限信息、物理页号。
 - a) 如果有效位是 0+NULL 则代表没有在虚拟内存空间中分配该内存；
 - b) 如果有效位是 0+非 NULL，则代表在虚拟内存空间中分配了但是没有被缓存到物理内存中；
 - c) 如果有效位是 1，则代表该内存已经缓存在了物理内存中，可以得到其物理页号 PPN，与虚拟页偏移量共同构成物理地址 PA。

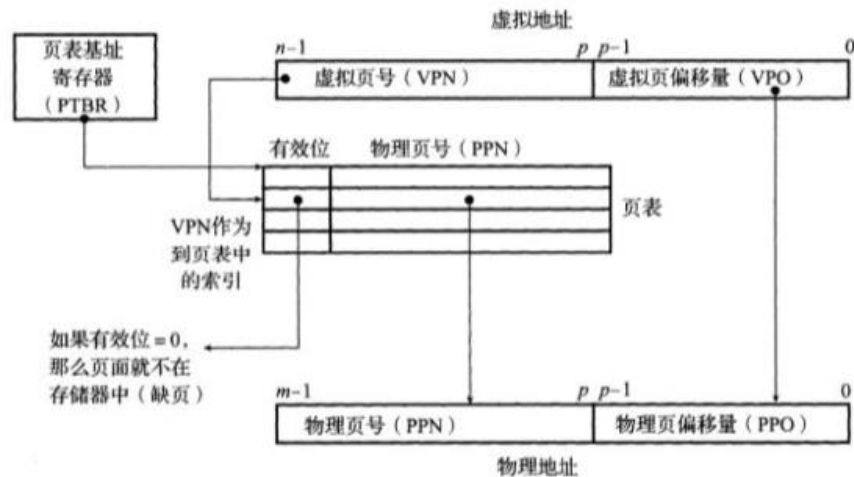


图 7 - 5

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

如果按照上述模式，每次 CPU 产生一个虚拟地址并且发送给地址管理单元，MMU 就必须查找一个 PTE 行来用将虚拟地址翻译成物理地址。为了消除这种操作带来的大量时间开销，MMU 中设计了一个关于 PTE 的小的缓存，称为翻译后备缓冲器 (TLB) 也叫快表。例如当每次 CPU 发现需要重新翻译一个虚拟地址是，它就必须发送一个 VPN 得到虚拟地址 MMU，发送一个 VPO 位得到一个 L1 高速缓存，例如当我们使用 MMU 向一个 TLB 的组请求一个页表中的条目时，告诉缓存通过一个 VPO 位在页表中查找一个相应的数据标记组，并在页表中读出这个组里的个数据标记和相应的数据关键字，当 MMU 从一个 TLB 的组得到一个 PPN 时，代表缓存的工作在这个组的请求之前就已经完全准备好，这个组的 PPN 与就已经可以与这些数据标记文件中的一个虚拟地址进行很好的匹配。

Core i7 采用四级页表层次结构，每个四级页表进程都有它自己的私有的页表层次结构，这种设计方法从两个基本方面就是减少了对内存的需求，如果一级页表的 PTE 全部为空，那么二级页表就不会继续存在，从而为进程节省了大量的内存，而且也只有一级页表才会有需要总是在一个内存中。四级页表的层次结构操作流程如下:36 位虚拟地址被寄存器划分出来组成四个 9 位的片，每个片被寄存器用作到一个页表的偏移量。CR3 寄存器内储存了一个 L1 页表的一个物理起始基地址，指向第一级页表的一个起始和最终位置，这个地址是页表上下文的一部分信息。VPN1 提供了到一个 L1PTE 的偏移量，这个 PTE 寄存器包含一个 L2 页表的起始基地址。VPN2 提供了到一个 L2PTE 的偏移量，一共四级，逐级以此层次类

推。

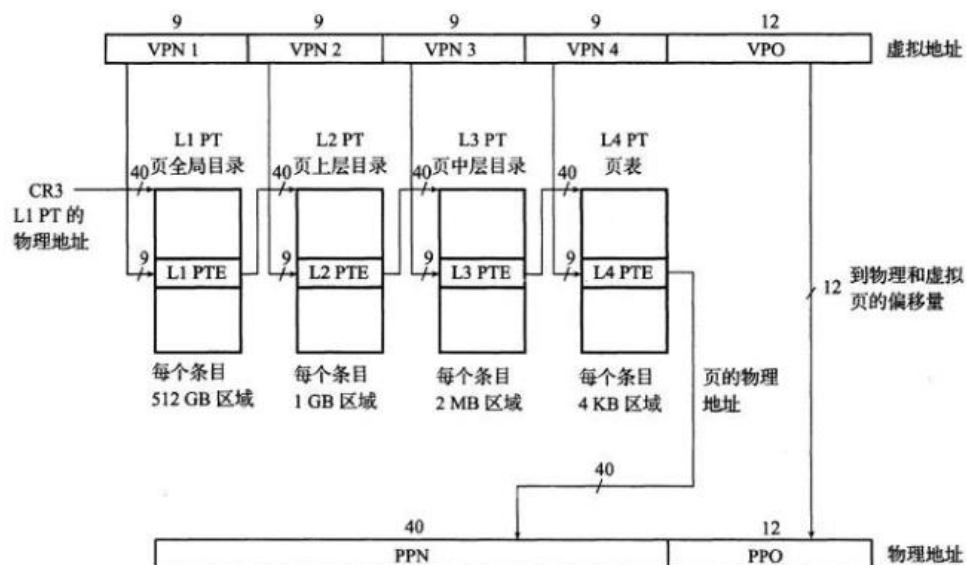


图 7 - 6

7.5 三级 Cache 支持下的物理内存访问

三级 cache 的缓存层次的结构如图 7-7 所示：

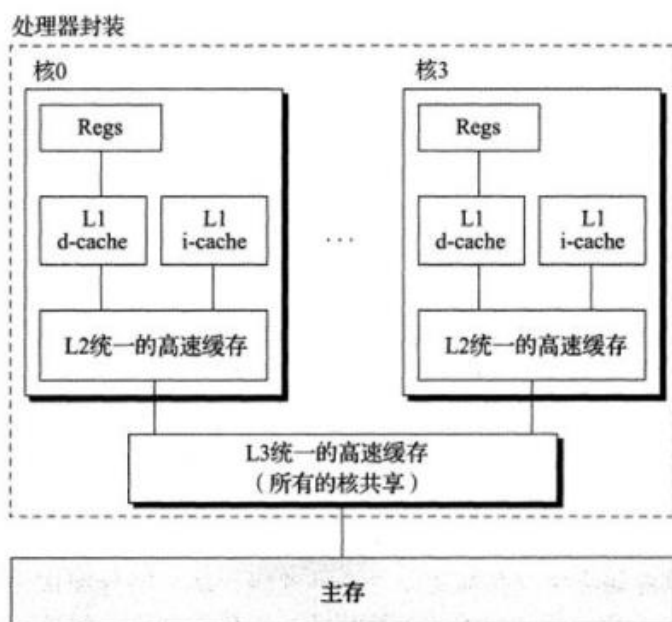


图 7 - 7

物理地址的结构包括组索引位 CI（倒数 7-12 位），使用它进行组索引，使用

组索引位找到对应的组后，假设我们的 cache 采用 8 路的块，匹配标记位 CT（前 40 位）如果匹配成功且寻找到的块的有效位 valid 上的标志的值为 1，则命中，根据数据偏移量 CO（后 6 位）取出需要的数据然后进行返回。

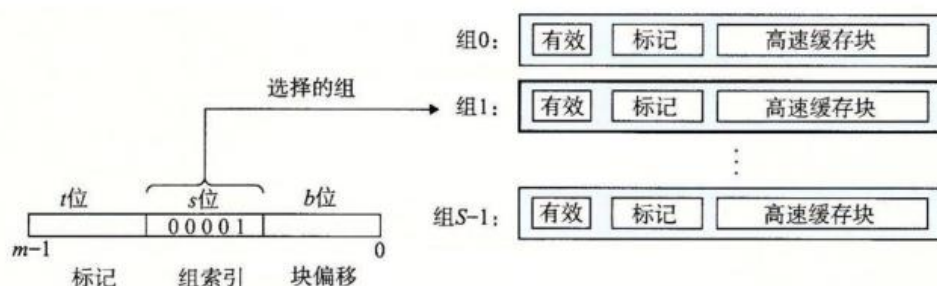


图 7 - 8

如果没有数据被匹配成功，或者匹配成功但是标记位是 1，这些都是未命中即 miss 的情况，此时向下一级缓存中查询数据（L2 Cache->L3 Cache->主存），并且将查找到的数据加载到 Cache 里，我们通常采用 LRU 策略，确定哪一个块作为牺牲快。

7.6 hello 进程 fork 时的内存映射

Shell 通过调用 fork 的函数可以让进程内核自动创建 yige 新的进程，这个新的进程拥有各自新的数据结构，并且被内核分配了唯一的 pid。它有着自己独立的虚拟内存空间。

虚拟内存和内存映射解释了 fork 函数如何为 hello 进程提供私有的虚拟地址空间。fork 为 hello 的进程创建虚拟内存，创建当前进程的 mm_struct, vm_area_struct 和页表的原样副本；两个进程中的每个页面都标记为只读；两个进程中的每个区域结构都标记为私有的写时复制，在 hello 进程中返回时，hello 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。

并且它还拥有自己独立的逻辑控制流，它同样可以拥有当前已经可以打开的各类文件信息和页表的原始数据和样本，为了有效保护进程的私有数据和信息，同时为了节省对内存的消耗，进程的每个数据区域都被内核标记起来作为写时复制。

7.7 hello 进程 execve 时的内存映射

在 `bash` 中的进程中执行 `execve` 的函数调用，`execve` 函数在当前进程中加载并运行包含在可执行文件 `hello` 中的程序，用 `hello` 替代了当前 `bash` 中的程序。

下面是加载并运行 `hello` 的几个步骤

删除已存在的用户区域：删除当前进程虚拟地址的用户部分的已存在的区域结构。

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

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

设置程序计数器（PC）：`execve` 会设置当前进程上下文的程序计数器，使之指向代码区域的入口点后，对该进程的调度就将重代码区域入口点开始，并根据需要通过缺页故障将磁盘中的数据与代码载入物理内存。

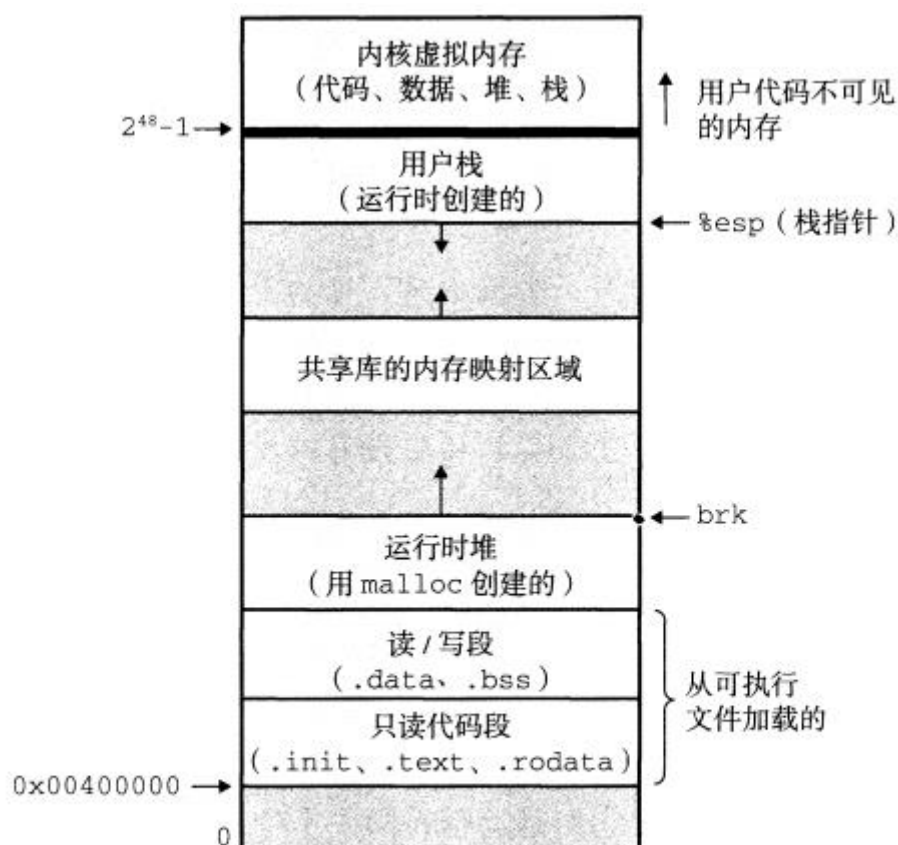


图 8-13 进程地址空间

7.8 缺页故障与缺页中断处理

如图 7-10，处理缺页是由硬件和操作系统内核协作完成的。

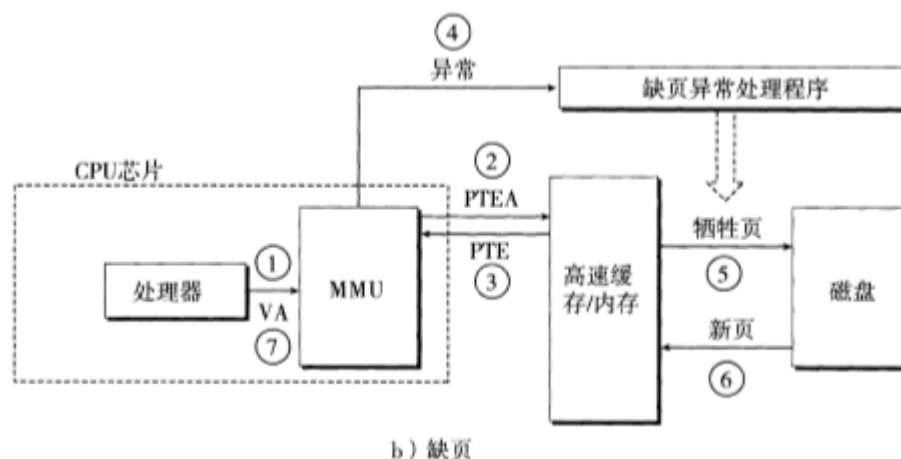


图 7 - 10

具体处理流程如下：

- 1) 处理器生成一个虚拟地址，并将它传送给 MMU
- 2) MMU 生成 PTE 地址，并从高速缓存/主存请求得到它
- 3) 高速缓存/主存向 MMU 返回 PTE
- 4) PTE 中的有效位是 0，所以 MMU 出发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序
- 5) 缺页处理程序确认出物理内存中的牺牲页，如果这个页已经被修改了，则把它换到磁盘
- 6) 缺页处理程序页面调入新的页面，并更新内存中的 PTE
- 7) 缺页处理程序返回到原来的进程，再次执行导致缺页的命令。CPU 将引起缺页的虚拟地址重新发送给 MMU。因为虚拟页面已经换存在物理内存中，所以就会命中。

7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，内核维护着一个变量 `brk`，它指向堆的顶部。

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格：显式分配器，隐式分配器。两种分隔偶要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

显式分配器：要求应用显式地释放任何已分配的块。例如，c 标准库提供一种叫做 `malloc` 程序包的显式分配器。c 程序通过调用 `malloc` 函数来分配一个块，并通过调用 `free` 函数来释放一个块。c++ 中的 `new` 和 `delete` 操作符与 c 中的 `malloc` 和 `free` 相当。

显示空闲链表：将空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 `pred`（前驱）和 `succ`（后继）指针。

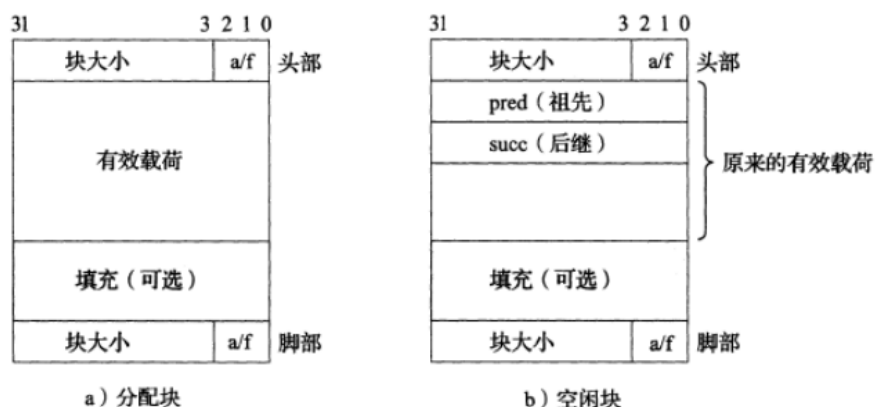


图 7 - 11

隐式分配器：另一方面，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配

的块的过程叫做垃圾收集，例如，注入 Lisp、ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

分离适配：分配器维护着一个空闲链表的数组，每个空闲链表是和一个大小类相关联的，并且被组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块。malloc 采用的就是分离适配的方法。

7.10 本章小结

本章详细说明了 hello 的存储器地址空间；深入探讨了 Intel 逻辑地址到线性地址的变换；并对 hello 的线性地址到物理地址的变换进行了深入的解析；同时还细致分析了 TLB 与四级页表支持下的 VA 到 PA 的变换；三级 Cache 支持下的物理内存访问；hello 在进程 fork 和 execve 的内存映射；以及缺页故障与缺页中断处理；最后详细分析了动态存储分配管理原理及方案。

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

8.1.1 设备的模型化：文件

所有的 I/O 设备都被模型化为文件，甚至内核也被映射为文件。

一个 Linux 文件就是一个 m 字节的序列。所有的输入、输出都被认为是对相应文件的读和写来执行。

文件的类型有：

- 1) 普通文件：包含任何数据，分成文本文件、二进制文件。
- 2) 目录：包含一组链接的文件。每个连接都将一个文件名映射到一个文件
- 3) 套接字：用于与另一个进程进行跨网络通信的文件

8.1.2 设备管理：unix io 接口

将 I/O 设备模型化为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O，这使所有的输入、输出都能以一种统一且一致的方式来执行。

我们可以对文件的操作有：打开关闭操作 `open` 和 `close`；读写操作 `read` 和 `write`；改变当前文件位置 `lseek` 等

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的文件，来宣告它想要访问一个 I/O 设备，内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息，应用程序只需要记住这个描述符。

2. linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）。头文件 `<unistd.h>` 定义了常量 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，它们用来代替显式的描述符值。

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 字节的文件，当 $k\sim m$ 时执行读操作会触发一个称为 `end-of-file(EOF)` 的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF 符号”。类似地，写操作就是从内存复制 $n>0$ 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。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，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。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

函数：

1.打开和关闭文件。

打开文件函数原型： `int open(char* filename,int flags,mode_t mode)`

返回值：若成功则为新文件描述符，否则返回-1；

`flags`： `O_RDONLY`（只读）， `O_WRONLY`（只写）， `O_RDWR`（可读写）

`mode`：指定新文件的访问权限位。

关闭文件函数原型： `int close(fd)`

返回值：成功返回 0，否则为-1

2，读和写文件

读文件函数原型： `ssize_t read(int fd,void *buf,size_t n)`

返回值：成功则返回读的字节数，若 EOF 则为 0，出错为-1

描述：从描述符为 `fd` 的当前文件位置复制最多 n 个字节到内存位置 `buf`

写文件函数原型： `ssize_t write(int fd,const void *buf,size_t n)`

返回值：成功则返回写的字节数，出错则为-1

描述：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

1) printf 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;
}
```

va_list arg = (va_list)((char*)&fmt + 4);

va_list 的定义: typedef char *va_list

这说明它是一个字符指针。其中的: (char*)&fmt + 4 表示的是第一个参数。

C 语言中, 参数压栈的方向是从右往左。当调用 printf 函数的适合, 先是最右边的参数入栈。fmt 是一个指针, 这个指针指向第一个 const 参数 (const char *fmt) 中的第一个元素。fmt 也是个变量, 它的位置, 是在栈上分配的, 它也有地址。 对于一个 char *类型的变量, 它入栈的是指针, 而不是这个 char *型变量。

2) vsprintf 函数

```
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);
}
```

`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。

3) write

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

发现反汇编语句中的 `int INT_VECTOR_SYS_CALL`，它表示要通过系统来调用 `sys_call` 这个函数。

4) sys_call

```
sys_call:
    call save
    push dword [p_proc_ready]
    sti
    push ecx
    push ebx
    call [sys_call_table + eax * 4]
    add esp, 4 * 3
    mov [esi + EAXREG - P_STACKBASE], eax
    cli
    ret
```

显示格式化了了的字符串。

- 5) 字符显示驱动子程序：从 ASCII 到字模库到显示 `vram`（存储每一个点的 RGB 颜色信息）。
- 6) 显示芯片按照刷新频率逐行读取 `vram`，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：在用户按键盘时，键盘接口获得一个键盘扫描码，这时会产生一个中断的请求，键盘会中断处理子程序。接受按键扫描码转成 ASCII 码，保存到系统的键盘缓冲区。getchar 函数通过调用 read 系统函数，通过系统调用读取按键的 ASCII 码，直到接收到回车才返回这个字符串。getchar 函数读取一个字符。

8.5 本章小结

本章详细描述了 Linux 的 IO 设备的管理方法：设备的模型化和设备管理，进而分析了 Unix I/O 的接口机器函数，细致分析了 printf 和 getchar 函数的具体实现。到这，我们完成了 hello 的一生的分析。

(第 8 章 1 分)

结论

hello 程序是几乎所有程序员的第一个程序，它的代码实现非常简单，但实际在计算机中运行时，非常复杂，他几乎需要计算机上大部分的硬件和软件协同合作。

通过本次大作业，使我对 hello 的一生有了更深刻的了解：

首先经历 `cpp` 预处理器的预处理，将所有宏定义等递归展开到 `hello.i` 文件中，消除所有的注释；然后经历 `ccl` 编译器的编译，将 `hello.i` 编译为汇编代码；随后经过汇编器 `as` 的汇编，将 `hello.s` 会变为可重定位文件 `hello.o`，且此时是二进制机器指令；最后经过 `ld` 链接器与其他必要的可重定位文件、共享库链接，并留下动态链接接口，在加载时动态链接。生成了可执行文件 `hello`。

在 `shell` 中输入指令，`shell` 调用 `fork` 函数，生成子进程，并由 `execve` 函数在当前进程即子进程的上下文中加载新程序 `hello`。CPU 为 `hello` 分配一个时间片，在这个时间片中，`hello` 程序按顺序执行自己的控制逻辑流。当程序执行需要数据时，就会通过多级存储器层次结构层层访问。`hello` 中的访存操作需要经历逻辑地址到线性地址到物理地址的变换。若此时 `hello` 遇到了信号与异常的影响，会触发系统调用异常。

`hello` 在运行时会调用一些函数，比如 `printf` 等，这些函数与 `linux I/O` 的设备模型化密切相关。最后 `hello` 会被 `shell` 的父进程回收，内核会回收为其创建的信息。

附件

文件名	文件作用
hello.c	源程序
hello.i	预处理文件
hello.s	编译后的汇编文件
hello.o	汇编后的可重定位目标程序
hello	链接后的可执行目标程序
hello_asm.s	hello 可执行文件的反汇编文件
hello_elf	hello 可执行文件的 elf 的格式
hello_o.elf	hello.o 可重定位文件的 elf 的格式
hello_o.s	hello.o 可重定位文件的反汇编文件

参考文献

- [1] RANDELE.BRYANT, DAVIDR.O ‘HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.
- [2] printf 函数实现的深入剖析 <https://www.cnblogs.com/pianist/p/3315801.html>
- [3] GCC 学习 - 预处理 <https://blog.csdn.net/xiaosaizi/article/details/105669070>
- [4] 逻辑地址、线性地址和物理地址之间的转换
<https://blog.csdn.net/gdj0001/article/details/80135196>