

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>1170300702</u>
班 級	<u>1703007</u>
學 生	<u>張志豪</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.c 文件(hello.c)是如何被计算机执行的。其通过预处理,编译,汇编,链接等步骤,最后加载到内存中去形成一个进程的过程。

其中 Editor+Cpp+Compiler+AS+LD + OS + CPU/RAM/IO。

和 from program to process 就形象的描述出了一个程序是如何被运行的。

他是由操作系统各个部分和硬件层面的相互支持才让 hello 完整的走完他的一生。

关键词: 关键词 1: p2p

关键词 2: 020

关键词 3: 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	- 5 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 7 -
2.4 本章小结	- 7 -
第 3 章 编译	- 8 -
3.1 编译的概念与作用	- 8 -
3.2 在 UBUNTU 下编译的命令	- 8 -
3.3 HELLO 的编译结果解析	- 8 -
3.4 本章小结	- 13 -
第 4 章 汇编	- 14 -
4.1 汇编的概念与作用	- 14 -
4.2 在 UBUNTU 下汇编的命令	- 14 -
4.3 可重定位目标 ELF 格式	- 14 -
4.4 HELLO.O 的结果解析	- 16 -
4.5 本章小结	- 17 -
第 5 章 链接	- 18 -
5.1 链接的概念与作用	- 18 -
5.2 在 UBUNTU 下链接的命令	- 18 -
5.3 可执行目标文件 HELLO 的格式	- 18 -
5.4 HELLO 的虚拟地址空间	- 19 -
5.5 链接的重定位过程分析	- 20 -
5.6 HELLO 的执行流程	- 22 -
5.7 HELLO 的动态链接分析	- 22 -
5.8 本章小结	- 23 -
第 6 章 HELLO 进程管理	- 24 -
6.1 进程的概念与作用	- 24 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 24 -
6.3 HELLO 的 FORK 进程创建过程	- 24 -
6.4 HELLO 的 EXECVE 过程	- 25 -
6.5 HELLO 的进程执行.....	- 25 -
6.6 HELLO 的异常与信号处理	- 26 -
6.7 本章小结	- 30 -
第 7 章 HELLO 的存储管理.....	- 31 -
7.1 HELLO 的存储器地址空间	- 31 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 31 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 32 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 33 -
7.5 三级 CACHE 支持下的物理内存访问	- 35 -
7.6 HELLO 进程 FORK 时的内存映射	- 36 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 37 -
7.8 缺页故障与缺页中断处理.....	- 38 -
7.9 动态存储分配管理	- 38 -
7.10 本章小结	- 40 -
第 8 章 HELLO 的 IO 管理	- 41 -
8.1 LINUX 的 IO 设备管理方法	- 41 -
8.2 简述 UNIX IO 接口及其函数	- 41 -
8.3 PRINTF 的实现分析.....	- 41 -
8.4 GETCHAR 的实现分析.....	- 42 -
8.5 本章小结	- 42 -
结论	- 42 -
附件	- 44 -
参考文献.....	- 45 -

第 1 章 概述

1.1 Hello 简介

P2p:即 7 从 program 到 process, 首先我们写好代码得到没有语法错误的 hello.c。然后经过预处理会得到 Hello.i 文件, 然后经过编译得到 hello.s 文件, 这是一个由汇编指令得到的文件, 然后在经过链接得到生成可重定向的 elf 格式的文件为.o 文件。随后如果有其他的文件就和其他的文件进行连接可以得到可执行文件。

随后可执行问价就诞生了, 启动这个可执行文件, 操作系统会 fork 一个子进程然后这个子进程会 execve, 将你需要的进程替换掉这个子进程。这就是 p2p 的过程。

020: Editor+Cpp+Compiler+AS+LD + OS + CPU/RAM/IO。在编译链接, 生成可执行文件, 并且加载之后。此时操作系统就需要内存, cpu 等硬件的支持了。由于 elf 格式是映射到内存中去的, 当执行 elf 格式的可执行目标文件的时候, 操作系统通过虚拟内存技术将 elf 各个段都映射到内存的空间中去。然后 cpu 会执行 elf 格式中文件的操作。通过取值, 译码, 访存, 计算, 写回, 更新 pc 等操作一步一步的完成指令, 再通过 IO 端口进行硬件的输出等操作, 完成 020 操作。

1.2 环境与工具

1.2.1 硬件环境

X64 CPU; 2GHz; 2G RAM; 256GHD Disk

1.2.2 软件环境

Windows10 64 位; VirtualBox/Vmware 11; Ubuntu 16.04 LTS 64 位/优麒麟 64 位;
10

1.2.3 开发工具

Visual Studio 2010 64 位以上; CodeBlocks; vi/vim/gpedit+gcc

1.3 中间结果

Hello	hello.o 链接之后生成的可执行文件
Hello.c	hello 程序的 C 代码
Helloo.elf	临时文件，用来观察 hello 的 readelf 和 hello.o 的 readelf 形式
Hello.i	预处理之后的文件
Hello.o	汇编之后的文件
Hello.s	编译之后的文件
Helloelf.txt	临时文件 hello 的反汇编代码
Hello.objdump.txt	临时文件 hello.o 的反汇编代码

1.4 本章小结

本章基本描述了 hello 的诞生，p2p,020 的过程，同时也为接下来的论述奠定了基础。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

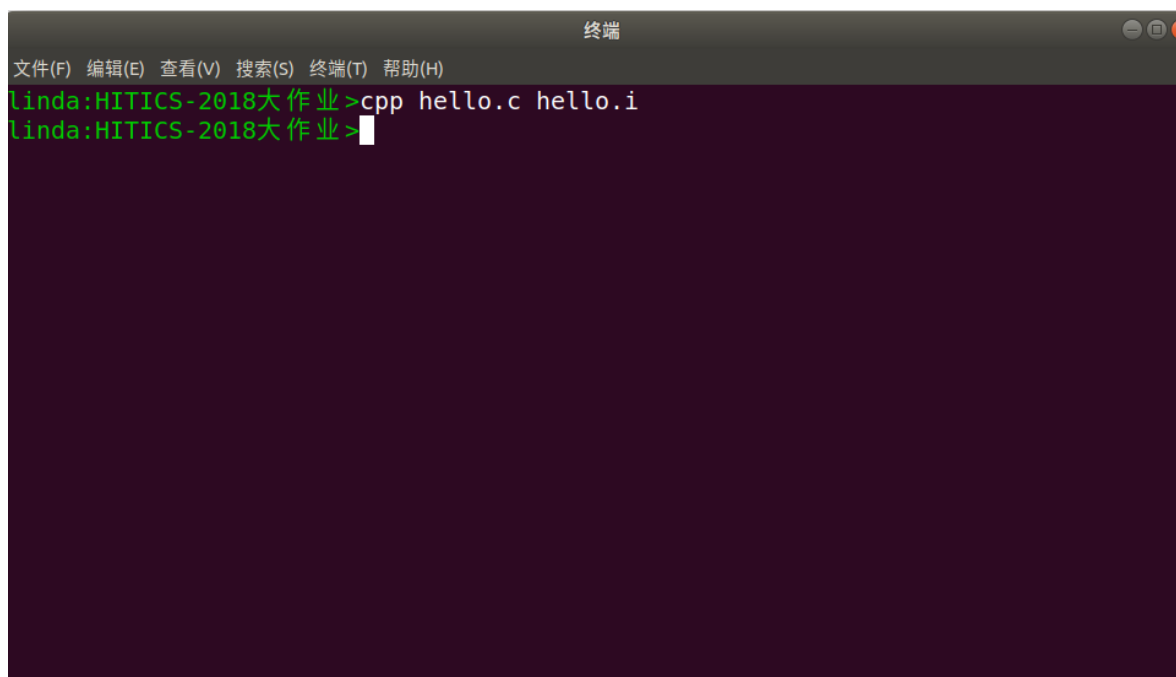
预处理主要是将文件的宏信息都改掉，使其变成一个相对而言易于计算机读取的文件。其操作处理相关 c 代码如下：

```
#if  #ifdef  #ifndef      #else  #elif
#endif      #define      #undef   #line      #error      #pragma
#include
```

其主要处理的是带有#开头的相关代码

2.2 在 Ubuntu 下预处理的命令

使用 `cpp hello.c hello.i` 这个命令得到 `hello.i` 这个预处理文件。



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
linda:HITICS-2018大作业>cpp hello.c hello.i
linda:HITICS-2018大作业>
```

2.3 Hello 的预处理结果解析

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

# 9 "hello.c" 2

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

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

    if(argc!=3)
    {
        printf("Usage: Hello 1170300702 zhangzhihao! \n");
        exit(1);
    }
    for(i=0; i<10; i++)
    {
        printf("Hello %s %s\n", argv[1], argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
```

上图所示为.i 文件，通过读.i 文件可以得知.i 文件的内容相对于.c 文件多得多，因为.i 文件需要将.c 文件需要的库文件都包含进来。此外通过阅读.i 文件我们可以看到原来的注释部分已经完全被删除。而代码部分却没有被修改。如果有#define 等宏定义的话也会被修改。

2.4 本章小结

预处理过程是一个程序进行的前奏，通过预处理的过程，使得代码更加易于操作系统读取，（同时也变得越来越不方便人来读取）他删掉了很多的注释部分，让程序的宏定义全都改变，也让这个函数所包含的库都全部纳入进来。总之预处理在程序执行过程中是一个比较重要的一环。

（第2章 0.5分）

第 3 章 编译

3.1 编译的概念与作用

编译：机器将原来预处理的.i 文件变为.s 文件。即将原来的 c 代码变成汇编代码。

作用：通过编译，将 C 语言变为汇编语言，而汇编语言的每一条指令都对应着每一条机器指令，使得文件方便机器的下一步的操作。

在编译的过程中也会有对 C 语言的修改，使其变得更为简单，即实现代码优化。使得 cpu 在执行时更加具有效率。

3.2 在 Ubuntu 下编译的命令

使用命令 `gcc -S hello.i -o hello.s`



3.3 Hello 的编译结果解析

3.3.1 数据类型：

由于在汇编层面缺乏 C 语言那么多的数据类型，在汇编层面由所占字节大小来表示数据，

```

.file    "hello.c"
.text
.globl   sleepsecs
.data
.align   4
.type    sleepsecs, @object
.size    sleepsecs, 4

```

```
int sleepsecs=2.5;
```

这是由.s 文件中所定义的全局变量，在汇编中，全局变量在.data 节中，如上图所示：声明全局变量 sleepsecs，其为 int 类型，尽管在 c 语言写得是小数，但是在.s 文件中依然是作为整数来用，不会作为小数。

3.3.1 赋值操作：

1、全局变量赋值所示：

```

sleepsecs:
    .long    2
    .section .rodata
    .align   8

```

由此图可以看出 C 语言所定义的 int 类型的全局变量赋值为 2，分配的大小为 8 个字节。（没有赋值 2.5）

2、函数内部赋值所示：

```

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

```

在函数的内部，使用 mov 指令形如 mov S D 将 S 的值传送给 D。

如上图所示，第一条指令是将 %edi 的值传送给地址为 %rbp 的值减去 20 所指向的地址。下面的也仿照。Mov 指令后面可以带有表示传送指定位大小的字节。

如可以使用 b 来表示字节，使用 w 来表示字，使用 l 表示双字，使用 q 来表式四字。此外上述表示的是将寄存器的值赋值给某个地址所在的值，还可以

用来将寄存器的值赋值给另一个寄存器，以及将某个地址里的值赋值给某一个寄存器。但是不能从一个地址的值直接赋值给另一个地址中的值，必须要通过寄存器。

3.3.2 类型转换：

程序中含有隐式类型转换：int sleepsecs=2.5。

隐式地转换成整数类型。

3.3.3 算术计算以及逻辑比较操作：

1、算术操作

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

算术操作常用的有加法（add）减法（sub）乘法（multi）以及自增自减等。

上述图所示的是减法操作，是价格呢%rsp 的值减去常数 5

比如说：

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

这条汇编指令就对应着：

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

2、逻辑判断（比较）

如上图所示在 for 循环中的判断，判断 i 的值是否小于 10。在汇编中会做算术操作，从而来根据 ZF OF 等这些条件码寄存器的值来决定是否发生跳转。

比如说 C 语言中的循环判断条件判断 i 的值是否小于 10：

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

而在汇编中：

```

    addl    $1, -4(%rbp)
.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT

```

I 的值为-4(%rbp)通过 `cmpl` 指令，比较 i 的值和 9 的值，即用减法操作来设置 ZF 标志位，然后和下一条指令的 `jle`(判断是否小于等于)发生跳转。

再比如：

```

    cmpl    $3, -20(%rbp)
    je      .L2
    jmp     100(%rip) @medi

```

判断是否等于 3

其对应 C 语言中的：

```

if(argc!=3)
{

```

3.3.4 条件及循环的跳转（控制转移）：

1、Hello.c 出现的 if

在汇编中的控制转移是出现 `jmp` 等类似的跳转而实现控制转移。

```

    cmpl    $3, -20(%rbp)
    je      .L2
    jmp     100(%rip) @medi

```

```

if(argc!=3)
{

```

如上述图，在汇编中，使用 `je` 指令如果满足即跳转到.L2。

2、Hello.c 出现的 for

```

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

```

使用 for 语句，首先判断是否满足条件，还要知道满足循环跳转（即往回跳）。

```

.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)
.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4

```

如上图所示，在程序运行到 `jle .L4` 时判读条件决定是否往回跳转，以实现循环操作。

3.3.5 函数操作：

涉及到的函数有 `printf` `getchar` `main` `sleep`

在汇编中调用函数的时候会先将下一条的指令入栈，然后使用 `call` 指令来进行跳转，在函数结束的时候会使用 `call` 指令来进行跳转。

例如使用 `printf`

```

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

```

主函数也是函数所以也有 `ret`

```

leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc

```

Getchar 函数的调用:

```
jle    .L4
call   getchar@PLT
movl   $0, %eax
```

Sleep 函数调用:

```
movl    %eax, %edi
call    sleep@PLT
        .L4:
```

3.3.6 数组操作:

For 循环中出现了循环, 即 `argv[1]` `argv[2]`

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

在汇编中, 由于数组中的元素是连续存储的, 汇编中通过地址的加减得到数组元素所在的位置, 如下图:

```

.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
```

上图中的 `rdi,rsi` 即函数传参时的寄存器, 即所示的 `argv[1]`, `argv[2]`。

3.4 本章小结

本章主要描述了机器是如何将预处理之后的 `.i` 文件变成汇编文件。即将高级语翻译成汇编语言。在变成汇编之后, 编译器可能会对文件进行优化 (优化代码)。

代码的可读性比高级语言难懂, 但是对于机器而言, 会更加易于机器理解, 因为其实每一条汇编指令都对应着机器指令。

(第3章2分)

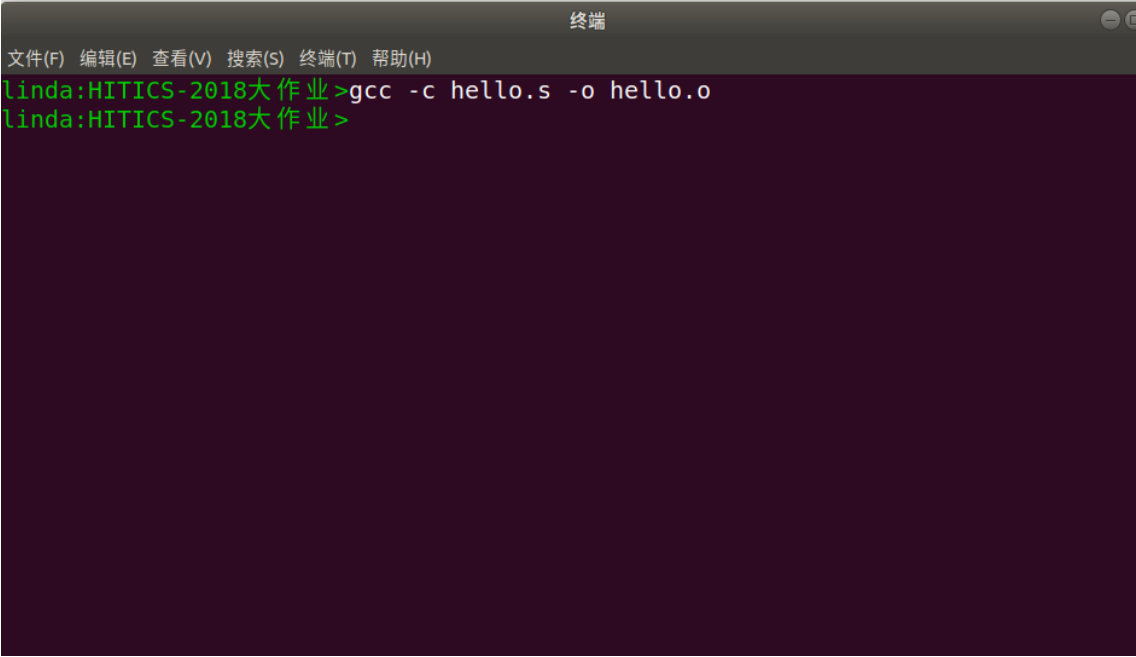
第 4 章 汇编

4.1 汇编的概念与作用

汇编：即 `as`，把汇编语言翻译成机器语言的过程称为汇编。并将其打包为可重定向目标程序（`elf` 格式），产生一个 `.o` 文件（机器指令）。这个过程称为汇编。汇编通过将汇编指令进一步底层化，能够得到机器可以识别的文件即 `.o` 文件。

4.2 在 Ubuntu 下汇编的命令

使用指令 `gcc -c hello.s -o hello.o`



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
linda:HITICS-2018大作业>gcc -c hello.s -o hello.o
linda:HITICS-2018大作业>
```

4.3 可重定位目标 `elf` 格式

可重定位文件使用指令 `gcc -c hello.s -o hello.o`

从而得到文本文档（包含.o 文件信息）截图如下：

1、elf 头（由于是中文的 ubuntu）

elf 头包含了生成该文件的系统字的大小和字节顺序。

```

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:      REL (可重定位文件)
系统架构:   Advanced Micro Devices X86-64
版本:      0x1
入口点地址: 0x0
程序头起点: 0 (bytes into file)
Start of section headers: 1160 (bytes into file)
标志:      0x0
本头的大小: 64 (字节)
程序头大小: 0 (字节)
Number of program headers: 0
节头大小:   64 (字节)
节头数量:   13
字符串表索引节头: 12

```

2、节头部表

节头部表描述的是各个节的信息，包含各个节所在的位置等相关信息

```

节头:
[号] 名称      类型      地址      偏移量
      大小      全体大小  旗标  链接  信息  对齐
[ 0] 0000000000000000 NULL 0000000000000000 0 0 0
[ 1] .text 0000000000000081 PROGBITS 0000000000000000 AX 0 0 1
[ 2] .rela.text 00000000000000c0 RELA 0000000000000000 I 10 1 8
[ 3] .data 0000000000000004 PROGBITS 0000000000000000 WA 0 0 4
[ 4] .bss 0000000000000000 NOBITS 0000000000000000 WA 0 0 1
[ 5] .rodata 0000000000000034 PROGBITS 0000000000000000 A 0 0 8
[ 6] .comment 000000000000002b PROGBITS 0000000000000000 MS 0 0 1
[ 7] .note.GNU-stack 0000000000000000 PROGBITS 0000000000000000 0 0 1
[ 8] .eh_frame 0000000000000038 PROGBITS 0000000000000000 A 0 0 8
[ 9] .rela.eh_frame 0000000000000018 RELA 0000000000000000 I 10 8 8
[10] .symtab 0000000000000198 SYMTAB 0000000000000000 11 9 8
[11] .strtab 000000000000004d STRTAB 0000000000000000 0 0 1
[12] .shstrtab 0000000000000061 STRTAB 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)

```

There are no section groups in this file.

2、重定位

重定位节 '.rela.text' at offset 0x348 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
0000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
000000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4
0000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4
0000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 23
000000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4
0000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
0000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4
0000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

重定位节 '.rela.eh_frame' at offset 0x408 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
0000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

如上图所示，重定位节包含了偏移量（offset），信息（info），类型（type），符号值（sym.value）

Pc 的重定位的方式有两种，一种是 pc 相对寻址，一种是 pc 直接寻址

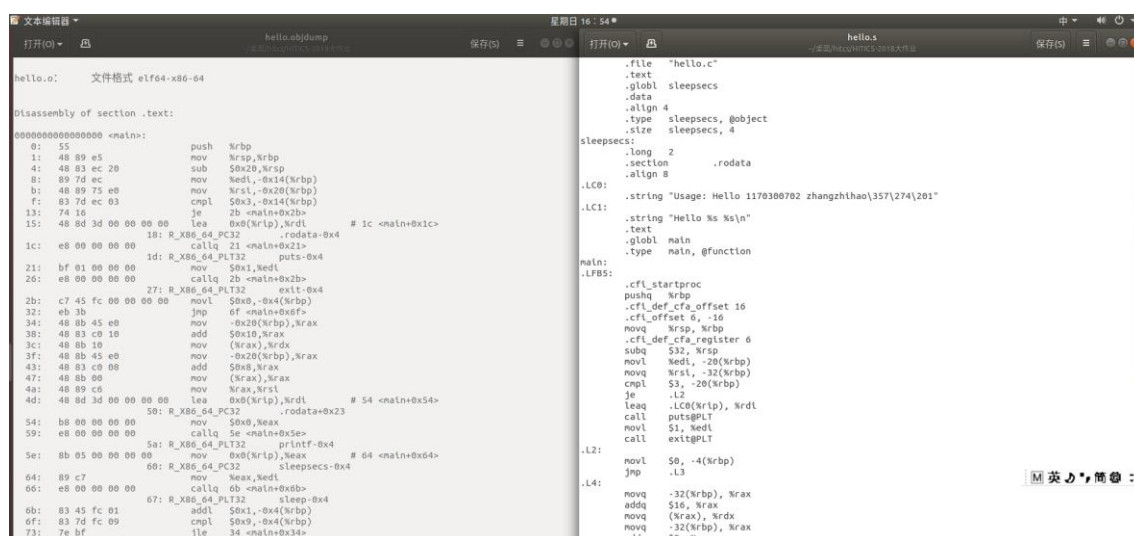
相对寻址的公式为：

$$*refptr = (\text{unsigned}) (\text{ADDR}(\text{r.symbol}) + \text{r.addend} - \text{refaddr})$$

链接器通过信息找到某一个重定位的信息，然后在符号节中找到那个符号，然后根据重定位节中的偏移量，然后根据相对寻址的公式找到目标。

而直接寻址的方式相对而言比较简单，他直接就告诉链接器其符号的位置在哪个地方。

4.4 Hello.o 的结果解析



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

可以看出一下不同：

1、函数调用和跳转的助记符消失。

通过汇编之后的文件比原来的文件短了。很多标记消失了。比如在.s 文件中含有很多.L2 .L3，但是在.o 文件都没有了。而相比原来的.s 文件这些助记符，现在的.o 文件已经改为确定的地址了。

2、操作数的改变，在.s 文件中操作数是十进制的，但是在.o 文件中操作数都是 16 进制的

3、在汇编成为机器语言时，将操作数设置为全 0 并添加重定位条目

在.s 文件中，访问只读数据段比如说 `printf` 所输出的字符串，使用段名称+`%rip`，在.o 文件中 `0+%rip`，因为 `rodata` 中数据地址也是在运行时确定，故访问也需要重定位。

4.5 本章小结

本章主要介绍了.o 文件的格式以及 elf 格式中的具体结构，以及用 `objdump` 命令下的 elf 格式的查看。分析了.s 和.o 文件之间的不同。了解了从汇编语言到机器语言之间的过程和.o 文件中的重定位和与.s 文件之间的映射。

(第 4 章 1 分)

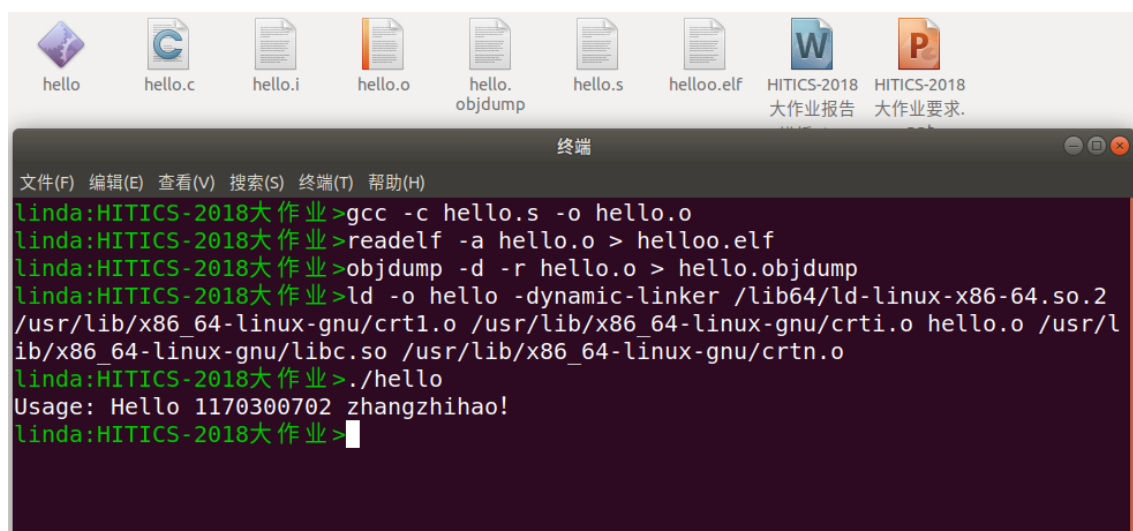
第 5 章 链接

5.1 链接的概念与作用

链接是通过链接器将多个文件整合到一个文件的过程。

在链接的过程会合并多个.o 文件，将多个.o 文件整合成一个可执行文件，在整合的过程中，会将相同的段进行合并。

5.2 在 Ubuntu 下链接的命令



在链接时通过与多个文件整合链接，命令为：`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`

5.3 可执行目标文件 hello 的格式

使用 `readelf -a hello > hello1.elf` 得到 hello 的 elf 格式。

而 hello 的 elf 的格式结构以及各个段的信息（比如说每一段的地址等）可以通过节头表得知：

如下是节头表的截图：

节头:

[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标 链接 信息	对齐
[0]	0000000000000000	NULL	0000000000000000 0 0	00000000 0
[1]	.interp 000000000000001c	PROGBITS	000000000400200 A 0 0	00000200 1
[2]	.note.ABI-tag 0000000000000020	NOTE	00000000040021c A 0 0	0000021c 4
[3]	.hash 0000000000000034	HASH	000000000400240 A 5 0	00000240 8
[4]	.gnu.hash 000000000000001c	GNU_HASH	000000000400278 A 5 0	00000278 8
[5]	.dynsym 00000000000000c0	DYNSYM	000000000400298 A 6 1	00000298 8
[6]	.dynstr 0000000000000057	STRTAB	000000000400358 A 0 0	00000358 1
[7]	.gnu.version 0000000000000010	VERSYM	0000000004003b0 A 5 0	000003b0 2
[8]	.gnu.version_r 0000000000000020	VERNEED	0000000004003c0 A 6 1	000003c0 8
[9]	.rela.dyn 0000000000000030	RELA	0000000004003e0 A 5 0	000003e0 8
[10]	.rela.plt 0000000000000078	RELA	000000000400410 AI 5 19	00000410 8
[11]	.init 0000000000000017	PROGBITS	000000000400488 AX 0 0	00000488 4
[12]	.plt 0000000000000060	PROGBITS	0000000004004a0 AX 0 0	000004a0 16
[13]	.text 0000000000000132	PROGBITS	000000000400500 AX 0 0	00000500 16
[14]	.fini 0000000000000009	PROGBITS	000000000400634 AX 0 0	00000634 4
[15]	.rodata 000000000000003c	PROGBITS	000000000400640 A 0 0	00000640 8
[16]	.eh_frame 00000000000000fc	PROGBITS	000000000400680 A 0 0	00000680 8
[17]	.dynamic 00000000000001a0	DYNAMIC	000000000600e50 WA 6 0	00000e50 8
[18]	.got 0000000000000010	PROGBITS	000000000600ff0 WA 0 0	00000ff0 8
[19]	.got.plt 0000000000000040	PROGBITS	000000000601000 WA 0 0	00001000 8
[20]	.data 0000000000000008	PROGBITS	000000000601040 WA 0 0	00001040 4
[21]	.comment 000000000000002a	PROGBITS	0000000000000000 MS 0 0	00001048 1
[22]	.symtab 0000000000000498	SYMTAB	0000000000000000 23 28	00001078 8
[23]	.strtab 0000000000000150	STRTAB	0000000000000000 0 0	00001510 1
[24]	.shstrtab 00000000000000c5	STRTAB	0000000000000000 0 0	00001660 1

如图是节头表，他描述了段的名称和地址和偏移量和对齐信息等。

5.4 hello 的虚拟地址空间


```

Disassembly of section .text:

0000000000000000 <main>:
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 20       sub     $0x20,%rsp
 8: 89 7d ec          mov     %edi,-0x14(%rbp)
 b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
 f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16             je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq   21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq   2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b             jmp     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 lea     0x0(%rip),%rdi    # 54 <main+0x54>
50: R_X86_64_PC32    .rodata+0x23
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

```

区别如下：

hello.o 的反汇编只出现了 main，hello 的反汇编出现了很多其他的函数。

如 init，还有基本的 puts@plt 等

对偏移地址变成了虚拟内存地址

Hello 中是从 hello.elf 开始，而 hello.o 是从.text 节开始

过程如下：

编译器先任意随便看一个地址，而其他的函数则会使用偏移量的方法来定位。而链接器在完成地址和空间的分配之后可以根据符号表中的内容从而可以确定所有符号的虚拟地址。然后就可以根据符号的地址对那些需要重定位的符号等信息进行修正。在 elf 文件中有一个重定位的表，他指示了哪些地方需要重定位，他的作用是描述如何修改相应的段中的内容。例如代码段.text 有需要被重定位的地方，

那么会有一个相应的叫`.rel.text` 的段保存代码段的重定位表。重定位表的结构是一个结构体类型的数组，每个数组元素对应一个重定位入口。

5.6 hello 的执行流程

(以下格式自行编排, 编辑时删除)

```

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

```

5.7 Hello 的动态链接分析

(以下格式自行编排, 编辑时删除)

动态链接库中的函数在编译的运行地址是未知的。需要添加到`.plt` 与`.got` 中等待动态链接器处理。`.got` 中存放函数目标地址, `.plt` 使用`.got` 中地址跳转到目标函数, 由于 Linux 采用了延迟绑定技术, 可执行文件中`got.plt` 中的地址并不是目标地址, 而是动态链接器中的地址

dl_init 之前的截图：

00000000:00600fc0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600fd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600ff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601000	50 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00	P.....
00000000:00601010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00@.....
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00@.....
00000000:00601040	00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62GCC: (Ub
00000000:00601050	75 6e 74 75 20 37 2e 33 2e 30 2d 32 37 75 62 75	untu 7.3.0-27ubu
00000000:00601060	6e 74 75 31 7e 31 38 2e 30 34 29 20 37 2e 33 2e	ntu1~18.04) 7.3.
00000000:00601070	30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0.....

之后：

00000000:00600fc0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600fd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00600ff0	b0 da a4 a7 56 7f 00 00 00 00 00 00 00 00 00 00V.....
00000000:00601000	50 0e 60 00 00 00 00 00 70 61 04 a8 56 7f 00 00	P.....pa.uV...
00000000:00601010	80 46 e3 a7 56 7f 00 00 b6 04 40 00 00 00 00 00	.F.V.....@.....
00000000:00601020	c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00@.....
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00@.....
00000000:00601040	00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62GCC: (Ub
00000000:00601050	75 6e 74 75 20 37 2e 33 2e 30 2d 32 37 75 62 75	untu 7.3.0-27ubu
00000000:00601060	6e 74 75 31 7e 31 38 2e 30 34 29 20 37 2e 33 2e	ntu1~18.04) 7.3.
00000000:00601070	30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0.....

不同之处已经标注：0x601008 和 0x601010 处的两个 8B 数据分别发生改变为 0x7fda 9fd3f170 和 0x7fda 9fb2d680。

5.8 本章小结

本章描述了链接的过程，讲述了一个文件与其他的文件一起链接，从而形成了一个独立的可执行文件的过程。在链接时讲述了重定位的信息其的空间地址分布问题。

(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：进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

6.2 简述壳 Shell-bash 的作用与处理流程

（以下格式自行编排，编辑时删除）

Shell 是 linux 下用 C 语言写出来的一个应用程序。他提供用户一种可以直接和操作系统对接的一个程序。

处理流程：

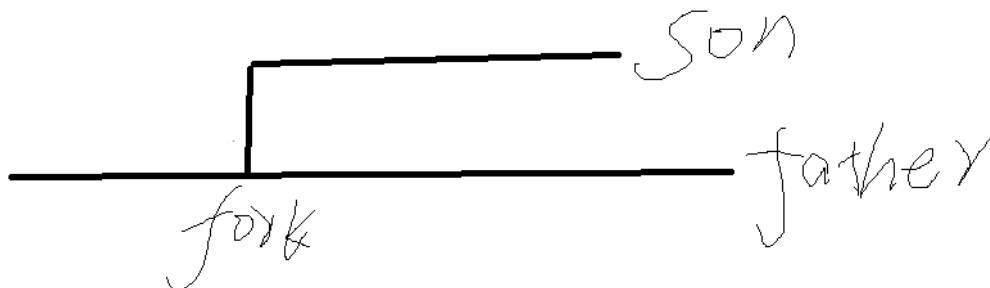
- 1、用输入一串命令
- 2、shell 首先切分这一串命令以获得用户输入的相关参数。
- 3、判断是否是内置命令
- 4、如果是内置命令，则直接执行这条命令。
- 5、若不是则使用 fork 和 execve 还创建一个进程并替换
- 6、在执行过程中仍然接受信号，并判断是否对这些信号做出反应以及做出什么样的反应。

6.3 Hello 的 fork 进程创建过程

用户要求在 shell 中输入./hello 此时 shell 程序会判断是否是内置命令。显然不是，是一个外部命令（为本目录下的一个程序）此时 shell 会对用户输入的进行分析。

此时 shell 会使用 fork 函数来创建一个子进程。子进程会得到与父进程相同的虚拟的地址空间，而且还会拥有与父进程相同的参数等信息，和父进程相同的副本。但是子进程和父进程的 PID 是不一样。但是系统是并发运行的。内

核能交替执行它们的逻辑控制流。



6.4 Hello 的 execve 过程

(以下格式自行编排, 编辑时删除)

子进程使用 `execve` 函数使得子进程被替换成另一个进程。将子进程的内容替换成 `Hello` 的信息。此时子进程就完全变成了 `hello`。

至于是如何完全替换成 `hello` 的过程如下:

加载器会删除现有的虚拟内存段, 新建一组并初始化新的堆栈。

此时在 `rip` 指向 `_start` 时, 起始函数会调用 `hello` 的 `main` 函数

通过将虚拟地址空间中的页映射到可执行文件的页大小的片, 新的代码和数据段被初始化为可执行文件中的内容。

与 `fork` 一次调用返回两次不同, `execve` 调用一次并从不返回。

6.5 Hello 的进程执行

(以下格式自行编排, 编辑时删除)

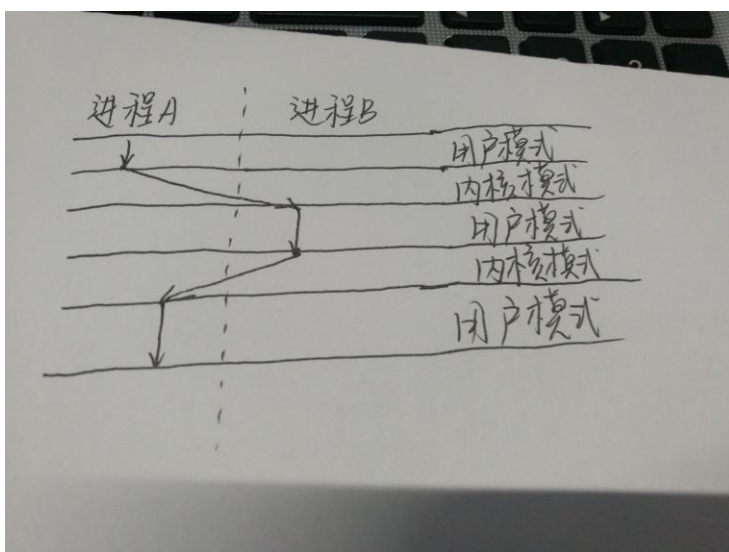
上下文: 上下文是由程序正确运行所需要的状态组成的, 这个状态包括存放在内存中的程序的代码和数据, 它的栈、通用寄存器的内容、程序、环境变量以及打开文件描述符的集合。

一个独立的逻辑流: 提供一种假象, 好像我们的程序独占使用处理器。

一个私有的地址空间: 好像我们的程序独占内存系统。

上下文切换, 进程调度:

在进程执行的某些时刻, 内核可以决定抢占当前进程, 并重新开始先前被抢占了的进程。这种决策交调度。



如图所示，操作系统在执行进程 A 到进程 B 之间的切换，当执行进程 A 时，是处于用户模式，而后进程 A 中断，操作系统会保存进程 A 的信息，而后进入内核模式从进程 A 转换到进程 B，在进程 B 执行一段时间之后又引发中断。转而又进行进程 A。

结合进程上下文信息、进程时间片，阐述进程调度的过程，用户态与核心态转换等等。

比如在 hello 函数中有 getchar 函数，此时可能会发生调度，上下文切换。

执行输入流 stdin 系统调用 read，陷入内核，内核中的陷阱处理程序请求来自缓冲区的处理，并且安排当缓存区的字符到达内存中去之后，中断处理器。此时才又开始重新在次运行 hello 程序。

6.6 hello 的异常与信号处理

由于 hello 中最后有一个 getchar 所以该进程需要输入一个字符才能正常结束。

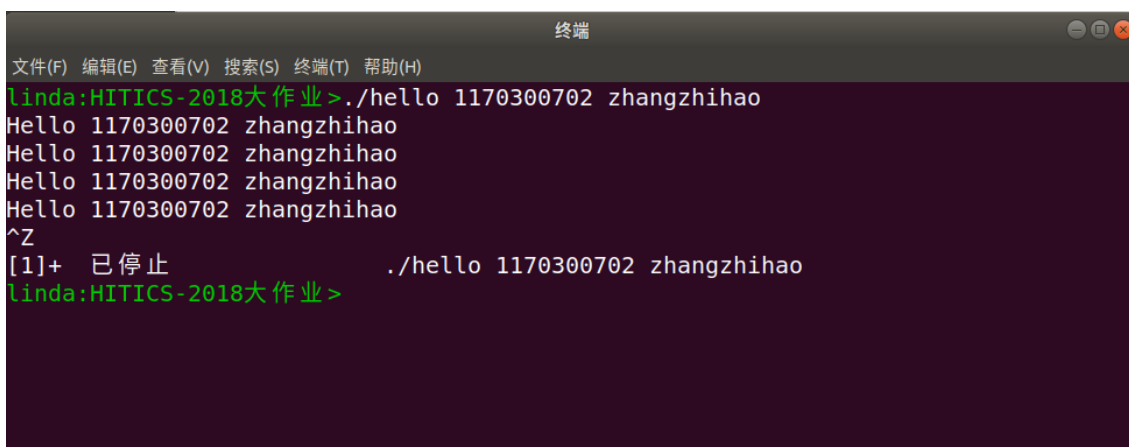
1、在运行的过程中输入 ctrl + c,z 含义是终止运行。如图：

```

终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
linda:HITICS-2018大作业>./hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
^C
linda:HITICS-2018大作业>

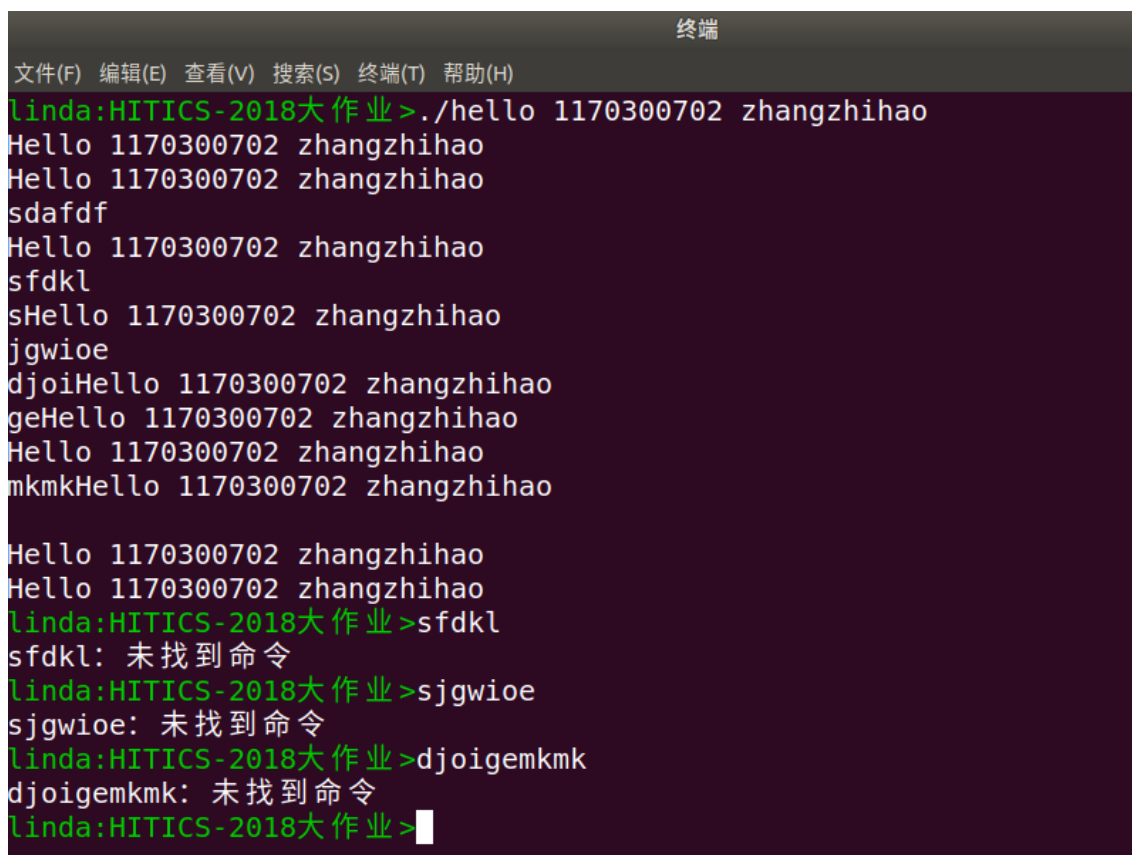
```

2、在运行中键盘输入 `ctrl + z` 发出将程序挂起的命令，截图如下：



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
linda:HITICS-2018大作业>./hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
^Z
[1]+  已停止                  ./hello 1170300702 zhangzhihao
linda:HITICS-2018大作业>
```

3、不停的乱按，包括回车：



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
linda:HITICS-2018大作业>./hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
sda fdf
Hello 1170300702 zhangzhihao
sfdkl
sHello 1170300702 zhangzhihao
jgwioe
djo iHello 1170300702 zhangzhihao
geHello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
mkmkHello 1170300702 zhangzhihao

Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
linda:HITICS-2018大作业>sfdkl
sfdkl: 未找到命令
linda:HITICS-2018大作业>sjgwioe
sjgwioe: 未找到命令
linda:HITICS-2018大作业>djoigemkmk
djoigemkmk: 未找到命令
linda:HITICS-2018大作业>
```

在乱按中，由于从键盘读入的字符都会到缓存区中，所以没有影响。但是当我们按下回车键时，在回车键之后的命令都会当作命令。这是由于回车键对 `getchar` 有影响所致。

4、 按下回车键之后按 ps, jobs , pstree ,fg ,kill

Ps 是显示现在 shell 中所包含的进程。

Jobs 与 ps 类似。

Pstree 是显示进程树。

Fg 是将后台的程序移到前台执行。

Kill 是杀死某个进程。

以上的截图如下：

```

linda:HITICS-2018大作业>ps
  PID TTY          TIME CMD
  2481 pts/0        00:00:00 bash
  2489 pts/0        00:00:00 ps
linda:HITICS-2018大作业>clear
linda:HITICS-2018大作业>./hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
^Z
[1]+  已停止                  ./hello 1170300702 zhangzhihao
linda:HITICS-2018大作业>ps
  PID TTY          TIME CMD
  2481 pts/0        00:00:00 bash
  2491 pts/0        00:00:00 hello
  2492 pts/0        00:00:00 ps
linda:HITICS-2018大作业>jobs
[1]+  已停止                  ./hello 1170300702 zhangzhihao
linda:HITICS-2018大作业>fg 1
./hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
Hello 1170300702 zhangzhihao
i
linda:HITICS-2018大作业>pstree
systemd--ModemManager--2*[{ModemManager}]
        --NetworkManager--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
        --dbus-daemon
        --fwupd--4*[{fwupd}]
        --gdm3--gdm-session-wor--gdm-wayland-ses--gnome-session-b--gnome-sh+
                                                --gsd-ally+
                                                --gsd-clip+
                                                --gsd-colo+
                                                --gsd-date+
                                                --gsd-hous+
                                                --gsd-keyb+
                                                --gsd-medi+
                                                --gsd-mous+
                                                --gsd-powe+
                                                --gsd-print+
                                                --gsd-rfki+
                                                --gsd-scre+
                                                --gsd-shar+
                                                --gsd-smar+
                                                --gsd-soun+
                                                --gsd-waco+

```



6.7 本章小结

本章是异常处理程序。讲的是在程序运行过程中如果出现了异常操作系统怎么处理（包括 `ctrl + c` 等）。也叙述了 `fork` 函数和 `execve` 函数以及 `shell` 的功能以及实现。还有当需要操作系统需要处理上下文切换时操作系统的行为。

（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：出现在汇编中地址。如在.s 文件中逻辑地址由选择符 和偏移量组成。

物理地址：和寻址总线有关，电路根据输入的地址到相应地址的物理内存中的数据放到数据总线中传输。如果是写，电路根据这个地址每位的值就在相应地址的物理内存中放入数据总线上的内容。

线性地址：如果是写，电路根据这个地址每位的值就在相应地址的物理内存中放入数据总线上的内容。分页机制中线性地址作为输入

虚拟地址：CPU 启动保护模式后，程序运行在虚拟地址空间中。CPU 通过一个虚拟地址来访问主存，这个虚拟地址会先被操作系统处理成物理地址然后再根据物理地址来寻址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

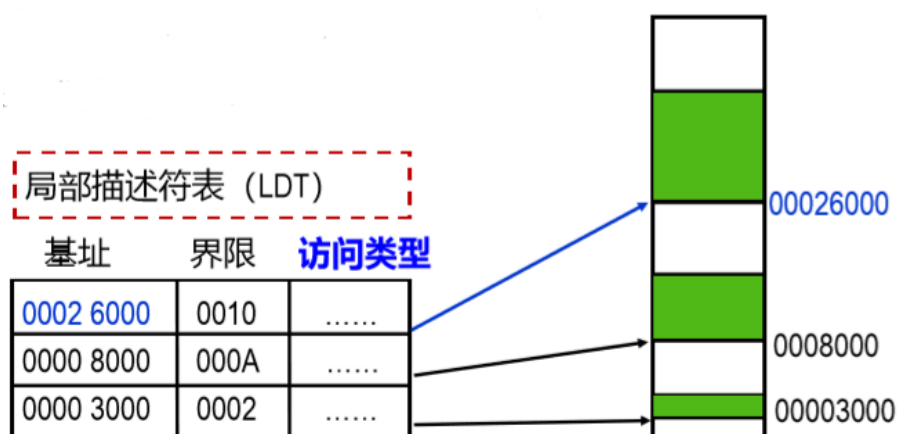
(以下格式自行编排，编辑时删除)

1、实地址：

处理器需要多位的地址，而系统不能提供操作系统所需的，所以采用内存分段的解决办法。段寄存器存放真实段基址，同时给出地址偏移量，则可以访问真实物理内存。

2、保护地址：

段寄存器无法放下 32 位段基址，所以它们被称作选择符，用于引用段描述符表中的表项来获得描述符。分段机制就可以描述为：通过解析段寄存器中的段选择符在段描述符表中根据 Index 选择目标描述符条目 Segment Descriptor，从目标描述符中提取出目标段的基地址 Base address，最后加上偏移量 offset 共同构成线性地址



给定一个完整的逻辑地址[段选择符：段内偏移地址]。

首先找到段描述符得到基址。取出段选择符中前 13 位，在数组中查找到对应的段描述符，得到基地址，再加上偏移量得到线性地址。

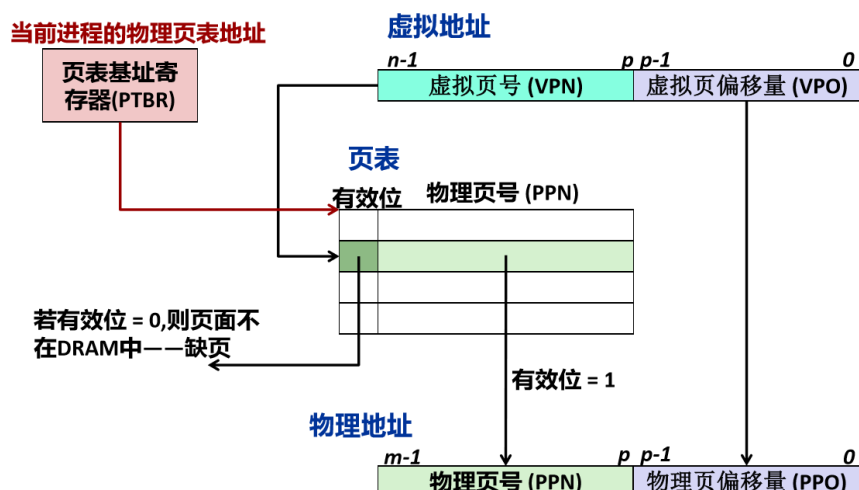
公式为：线性地址 = 基地址 + 偏移量

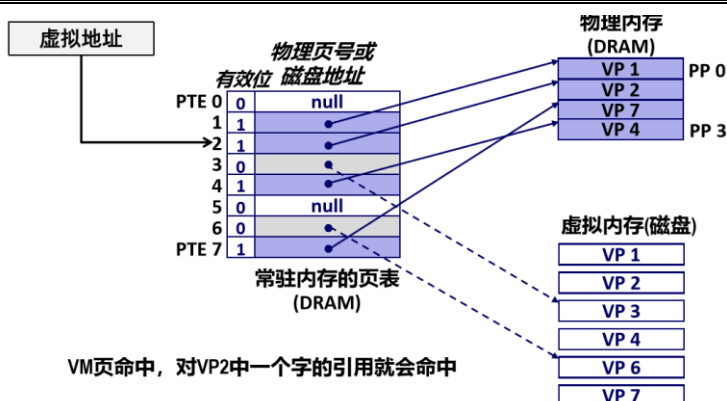
7.3 Hello 的线性地址到物理地址的变换-页式管理

(以下格式自行编排, 编辑时删除)

线性地址到物理地址的变换(页式管理)是由操作系统、MMU 和存放在物理内存中的页表共同实现的。他实现了实现虚拟内存到物理内存的映射。

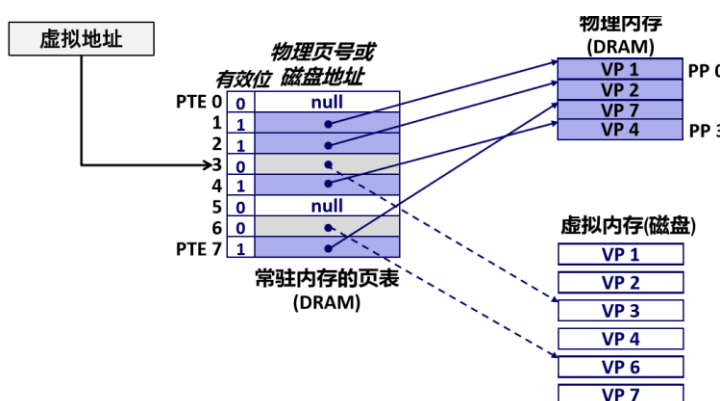
在访问是首先访问页表基地址，他存在在一个叫做页表基地址的一个寄存器当中。当知道了页表基地址之后，虚拟地址通过在页表中匹配页表条目来获取 PPN，然后再加上物理页的偏移地址就可以得到物理地址。



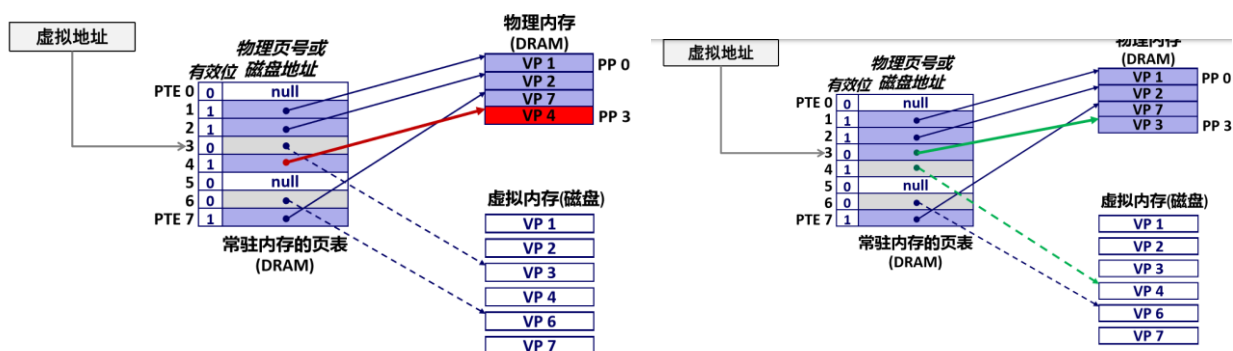


如果在匹配页表条目是发生不匹配现象则称为缺页，会引发缺页中断，缺页处理程序将需要的页表条目放入页面中，更新 PTE，再次进行访问。

如下图所示：



缺页处理过程如下：



7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

(以下格式自行编排，编辑时删除)

36 位 VPN 被划分成四个 9 位的部分。

此时虚拟地址被分成 4 个 VPN 和一个 VPO，第 i 个 VPN 都是一个到第 i 级页表的索引，然后再由他们里面的值再指向下一个页面。在翻译虚拟地址时通过四级页表查询到 PPN，与 VPO 结合成 PA。

The diagram illustrates the multi-stage address translation process in x86-64 architecture:

- Virtual Address (VA) Split:** A 32-bit virtual address is split into a 36-bit Virtual Page Number (VPN) and a 12-bit Virtual Page Offset (VPO).
- TLB Check:** The 36-bit VPN is used to check the TLB (Translation Lookaside Buffer). The TLB is divided into a 32-bit TLBT (TLB Tag) and a 4-bit TLBI (TLB Index).
 - TLB Hit:** If the TLB contains the VPN, the 40-bit Physical Page Number (PPN) is retrieved directly from the TLB.
 - TLB Miss:** If not found, the process proceeds to the next stage.
- Page Table Walk:** The 36-bit VPN is split into four 9-bit segments (VPN1, VPN2, VPN3, VPN4). These are used to traverse the CR3 register and a series of Page Tables (PTEs) to find the base address of the final page table.
- Final Page Table Access:** The 12-bit VPO is used to access the specific byte in the final page table, yielding the 12-bit Physical Page Offset (PPO).
- Physical Address (PA) Construction:** The 40-bit PPN and 12-bit PPO are combined to form the 52-bit Physical Address (PA).
- Caching and Final Output:**
 - The 52-bit PA is used to access the L1 d-cache (64 sets, 8 rows/set).
 - L1 Hit:** The 32-bit physical address is returned to the CPU.
 - L1 Miss:** The address is passed to L2, L3 caches, and finally to main memory.
 - The final 32-bit result is sent back to the CPU.

Core i7 页表翻译



7.5 三级 Cache 支持下的物理内存访问

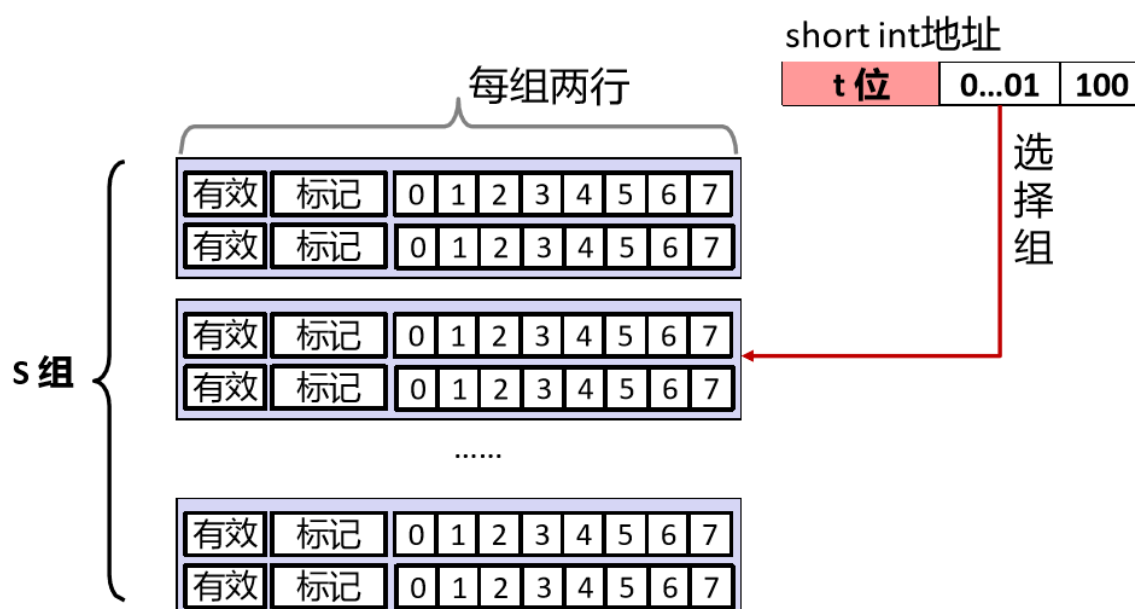
在英特尔构架下有三级缓存，这些缓存存在与 cpu 中，分别为 L1，L2，L3。第 k 级的缓存包含第 $k+1$ 层的一个子集的副本。Cpu 给一个地址，会先到第 K 中去寻找，如果在第 k 层中没有找到，此时将会发生替换，将第 $k+1$ 层的数据替换到第 k 层中去。然后再把数据送到 cpu。

每个缓存将被分成 S 组，每组有 E 行。每一行的结构如下：

有效位	标记位	数据块
-----	-----	-----

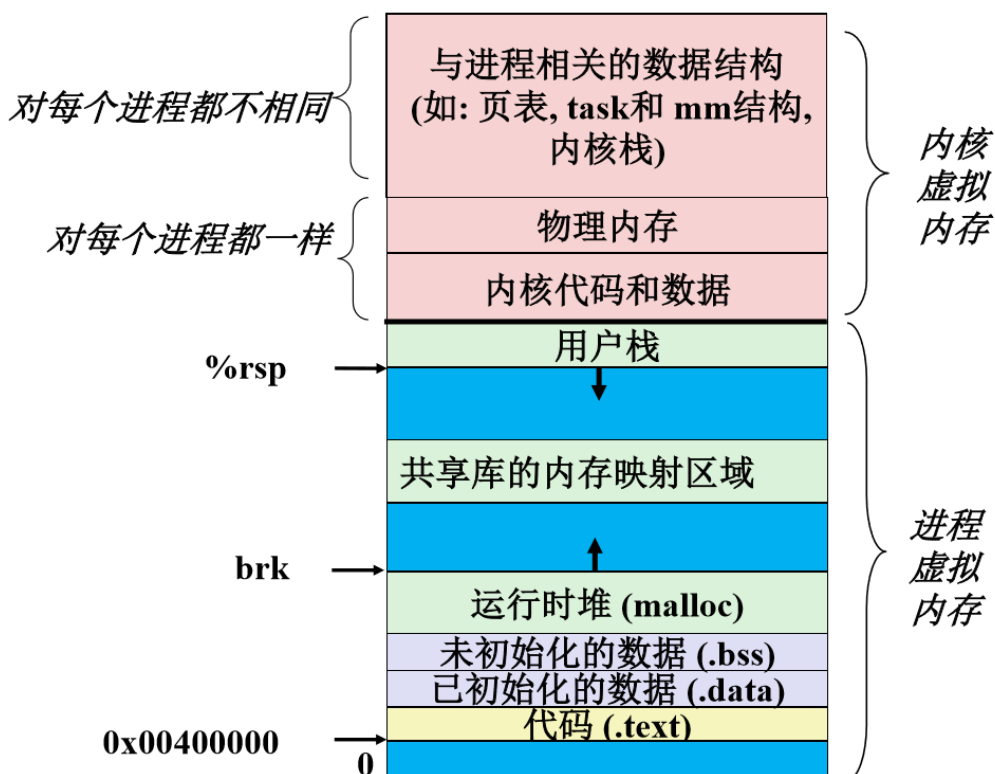
根据 CPU 给出的地址格式，先找到缓存中的特定的组，然后在这些组中根据 CPU 给出地址中的标记位寻找到这个组中的匹配行。即 CPU 给出的地址中的标记与这个组中的某一行的标记相等。此时再根据偏移量得到数据块。

如下图所示：



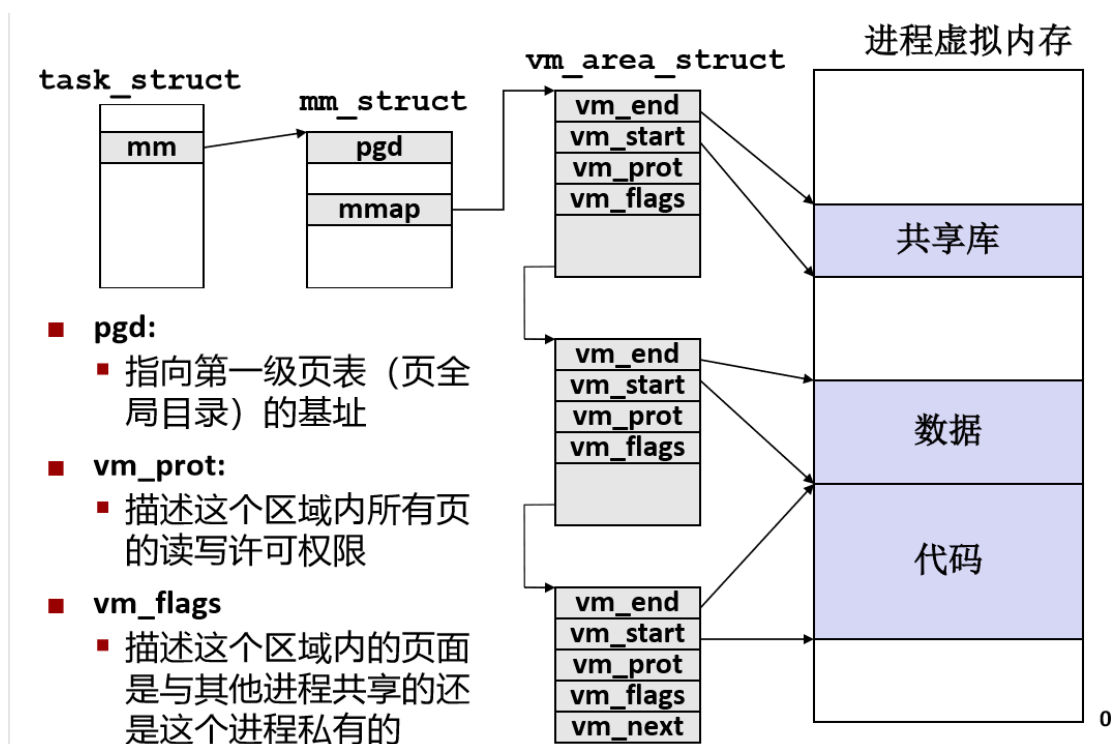
7.6 hello 进程 fork 时的内存映射

一个 Linux 进程的虚拟地址空间如下所示：



- 1、**task_struct:** 构元素包含或指向内核运行该进程所需要的所有信息，例如 PID、指向用户栈的指针、可执行目标文件的名称、程序计数器等，其中一个条目指向 `mm_struct`，
- 2、**mm_struct:** 它描述了虚拟内存的当前状态。其中两个元素 `pgd` 指向第一级页表的基址
- 3、**mmap:** 它指向一个区域结构的链表，其中每个 `vm_area_struct` 都描述了当前虚拟地址空间的一个区域。当内核运行这个进程时，就将 `pgd` 存放在 CR3 控制寄存器中

这些区域集合以及映射到内存中的关系如下图所示：

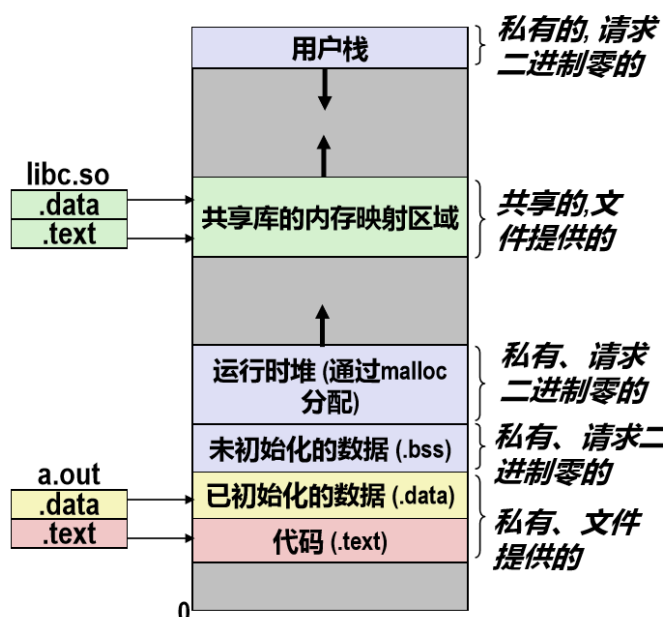


下面解释当 `fork` 函数进行是创建的子进程：

当 `fork` 函数被 `shell` 调用时，内核为 `hello` 进程创建各种数据结构，并分配给它一个唯一的 `PID`。为了给 `hello` 进程创建虚拟内存，它创建了 `hello` 进程的 `mm_struct`、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

`Fork` 函数在创建子进程的时候，子进程的地址空间的映射和 `Linux` 下的进程地址空间一样，如上述所言。

7.7 `hello` 进程 `execve` 时的内存映射



`execve` 函数在当前进程中 加载并运行新程序 `hello` 的步骤如下:

- 1、删除已存在的用户区域
- 2、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

- 3、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

7.8 缺页故障与缺页中断处理

当 CPU 给出一个虚拟地址, 再根据寄存器寻找到相应的物理地址数据, 在寻找时所要找的数据在页表条目中没有。此时就是缺页故障。

缺页故障的处理步骤如下:

系统将控制权交给缺页处理程序, 他会选择牺牲一个页表条目, 把需要的条目加入进来并且替换掉牺牲的那个条目, 并更新页面。当缺页中断程序返回时, CPU 将重新执行取值令。

上述所言的内容截图请参见 7.3

7.9 动态存储分配管理

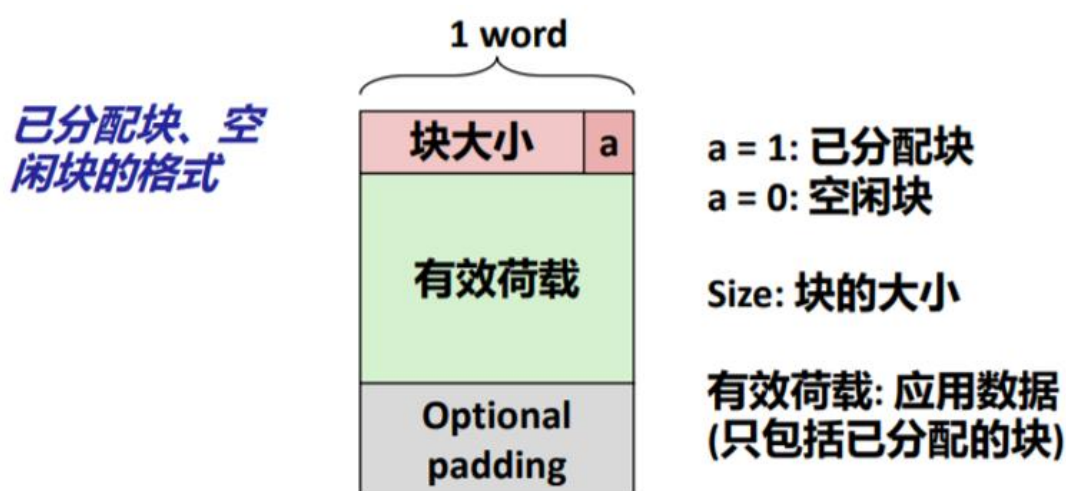
动态内存维护堆，堆由低地址向高地址生长。分配器将堆看作一组由不同大小块的集合。要维护使得操作系统知道哪个内存是空闲的，哪一块内存是被占用的。

动态分配内存分为显式分配器和隐式分配器

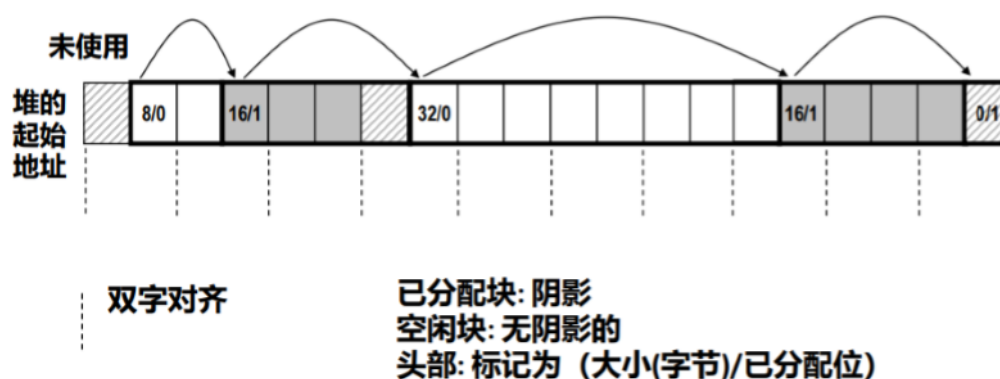
显示分配器：要求显示的释放和分配块

隐式分配器：应用检测到已分配块不再被程序所使用，就释放这个块

1、隐式分配器下，我们需要知道块的大小和分配状态，从而对其进行管理。而且还需要考虑对齐。如下图所示：



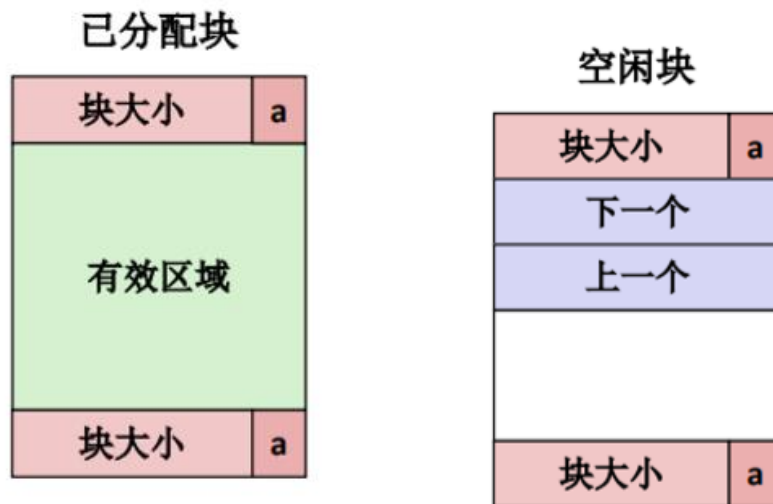
由图所示，存块都是 8 字节对齐的，那么块大小的低三位总是 0，不存储这些 0 位，将它视为已分配/未分配的标志，读大小字段时，将其屏蔽。这样，一个块就分为了两部分，一部分用于存储大小和状态，一部分称为有效荷载，



2、显式分配器下

显示空间链表是类似与双向链表，他只维护空闲的空间块，不能通过块

的大小来进行索引



其索引的方式是通过看空闲块的上一个和下一个来寻找到下一个空闲块

7.10 本章小结

本章描述了存储器的管理模式以及虚拟内存的内容和动态分配内存。讲述了几种地址的概念以及地址的翻译过程和虚拟地址对应于物理地址的对应关系，以及进程与内存之间的映射关系和动态分配内存的内容。

(第7章 2分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

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

8.2 简述 Unix IO 接口及其函数

(以下格式自行编排，编辑时删除)

打开和关闭文件：int open(char *filename,int flags,mode_t mode);

int close(int fd);

读写文件：ssize_t read(int fd,void *buf,size_t n)

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

指示文件要读写位置的偏移量：off_t lseek(int handle, off_t offset, int fromwhere);

8.3 printf 的实现分析

(以下格式自行编排，编辑时删除)

Printf 函数如下所述：

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;

}

从 vsprintf 生成显示信息，到 write 系统函数，到陷阱-系统调用 int 0x80 或 syscall。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。字符显示驱动子程序：从 ASCII 到字模库到显示 vram（存储每一个点的 RGB 颜色信息）。显示芯片按照刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

8.4 getchar 的实现分析

（以下格式自行编排，编辑时删除）

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

异步异常-键盘中断的处理：键盘中断处理子程序。接受按键扫描码转成 ascii 码，保存到系统的键盘缓冲区。getchar 函数中调用了 read 系统函数，通过系统调用读取按键 ascii 码，保存到系统的键盘缓冲区，直到接受到回车键才开始从 stdio 流中每次读入一个字符。若文件结尾则返回-1(EOF)，且将用户输入的字符回显到屏幕。

8.5 本章小结

本章主要讲述了 I/O 接口的问题，分析了 IO 接口的几个函数如 printf 和 getchar 等。了解了在应用程序需要使用文件或者标准化输入输出时的相关知识。

（第 8 章 1 分）

结论

Hello 的一生：

首先利用 IO 设备将文件载入在内存中去

然后 `hello.c` 文件被预处理（`c p p`）成 `.i` 文件成为 `hello.i`

然后 `.i` 文件经过编译形成由汇编指令形成的文件，成为 `.s` 文件即 `hello.s`

然后经过汇编器，将 `.s` 文件变成 `.o` 文件（由机器指令构成的文件）

然后使用链接器，将多个 `.o` 文件链接到一起，生成一个可执行的二进制文件

然后 `shell` 使用 `fork` 函数和 `execve` 函数，`hello` 此时就变成了一个进程了

`Hello` 被映射到内存中去，一条一条指令的执行。

`CPU` 给 `hello` 分配时间片，使其能独立有自己的逻辑流。

如果在执行的过程中出现异常，调用异常处理程序。

在 `hello` 结束时，`shell` 会回收 `hello`，删除由 `hello` 制造的数据

`Hello` 的执行是许许多多的部分构成，从硬件到软件无不为 `hello` 担心。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

Hello	hello.o 链接之后生成的可执行文件
Hello.c	hello 程序的 C 代码
Helloo.elf	临时文件，用来观察 hello 的 readelf 和 hello.o 的 readelf 形式
Hello.i	预处理之后的文件
Hello.o	汇编之后的文件
Hello.s	编译之后的文件
Hello.elf	临时文件 hello 的反汇编代码
Hello.objdump	临时文件 hello.o 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] 林来兴. 空间控制技术[M]. 北京: 中国宇航出版社, 1992: 25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集: A 集[C]. 北京: 中国科学出版社, 1999.
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北: 天下文化出版社, 1998 [1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨: 哈尔滨工业大学, 1992: 8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359) : 2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science , 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.

(参考文献 0 分, 缺失 -1 分)