

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>軟件工程</u>
學 號	<u>1183710109</u>
班 級	<u>1837101</u>
學 生	<u>郭茁寧</u>
指 導 教 師	<u>史先俊</u>

計算機科學與技術學院

2019 年 12 月

摘 要

本论文将 CSAPP 课程所学内容通过 hello 小程序的一生，对我们所学进行全面的梳理与回顾。我们主要在 Ubuntu 下进行相关操作，合理运用了 Ubuntu 下的操作工具，进行细致的历程分析，目的是加深对计算机系统的了解。

关键词：hello；程序的一生；计算机系统；Ubuntu

（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

摘 要.....	- 1 -
目 录.....	- 2 -
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	- 9 -
3.3 HELLO 的编译结果解析.....	- 9 -
3.3.1 数据和赋值.....	- 9 -
3.3.2 算术操作.....	- 10 -
3.3.3 关系操作和控制转移.....	- 11 -
3.3.4 数组/指针/结构操作.....	- 11 -
3.3.5 函数操作.....	- 12 -
3.4 本章小结.....	- 14 -
第 4 章 汇编.....	- 15 -
4.1 汇编的概念与作用	- 15 -
4.2 在 UBUNTU 下汇编的命令.....	- 15 -
4.3 可重定位目标 ELF 格式.....	- 15 -
4.3.1 命令:	- 15 -
4.3.2 ELF 头:	- 15 -
4.3.3 节头目表:	- 16 -
4.3.4 重定位节:	- 17 -
4.3.5 符号表:	- 18 -
4.4 HELLO.O 的结果解析	- 18 -
4.5 本章小结.....	- 20 -
第 5 章 链接.....	- 21 -

5.1 链接的概念与作用	- 21 -
5.2 在 UBUNTU 下链接的命令	- 21 -
5.3 可执行目标文件 HELLO 的格式	- 21 -
5.4 HELLO 的虚拟地址空间	- 23 -
5.5 链接的重定位过程分析	- 24 -
5.6 HELLO 的执行流程	- 25 -
5.7 HELLO 的动态链接分析	- 26 -
5.8 本章小结	- 27 -
第 6 章 HELLO 进程管理	- 28 -
6.1 进程的概念与作用	- 28 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	- 28 -
6.3 HELLO 的 FORK 进程创建过程	- 29 -
6.4 HELLO 的 EXECVE 过程	- 29 -
6.5 HELLO 的进程执行	- 29 -
6.6 HELLO 的异常与信号处理	- 31 -
6.7 本章小结	- 34 -
第 7 章 HELLO 的存储管理	- 35 -
7.1 HELLO 的存储器地址空间	- 35 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 35 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 36 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	- 36 -
7.5 三级 CACHE 支持下的物理内存访问	- 37 -
7.6 HELLO 进程 FORK 时的内存映射	- 38 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 38 -
7.8 缺页故障与缺页中断处理	- 38 -
7.9 动态存储分配管理	- 39 -
7.10 本章小结	- 39 -
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	- 42 -
8.4 GETCHAR 的实现分析	- 43 -
8.5 本章小结	- 44 -
结 论	- 45 -
附 件	- 46 -
参考文献	- 47 -

第 1 章 概述

1.1 Hello 简介

P2P:

- Program: 在 editor 中键入代码得到 hello.c 程序
- Process: hello.c (在 Linux 中), 经过过 cpp 的预处理、ccl 的编译、as 的汇编、ld 的链接最终成为可执目标程序 hello。在 shell 中键入启动命令后, shell 为其 fork, 产生子进程。

O2O:

- shell 为 hello 进程 execve, 映射虚拟内存, 进入程序入口后程序开始载入物理内存。
- 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: 处理器: Intel® Core™ i7-8550U CPU @ 1.80GHz 1.99GHz

RAM: 8.00GB 系统类型: 64 位操作系统, 基于 x64 的处理器

软件环境: Windows10 64 位; Ubuntu 19.04

开发与调试工具: gcc, as, ld, vim, edb, readelf, VScode

1.3 中间结果

文件的作用	文件名
预处理后的文件	hello.i
编译之后的汇编文件	hello.s
汇编之后的可重定位目标文件	hello.o
链接之后的可执行目标文件	Hello
Hello.o 的 ELF 格式	Elf.txt
Hello.o 的反汇编代码	Disas_hello.s
hello 的 ELF 格式	hello1.elf
hello 的反汇编代码	hello1_objdump.s

1.4 本章小结

本章对 hello 进行了一个总体的概括，首先介绍了 P2P、O2O 的意义和过程，介绍了作业中的硬件环境、软件环境和开发工具，最后简述了从.c 文件到可执行文件中间经历的过程。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

预处理的概念：

预处理中会展开以#起始的行，试图解释为预处理指令(preprocessing directive)，其中 ISO C/C++要求支持的包括#if、#ifdef、#ifndef、#else、#elif、#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的#（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

预处理的作用：

1. 将源文件中用#include 形式声明的文件复制到新的程序中。比如 hello.c 第 6-8 行中的#include<stdio.h> 等命令告诉预处理器读取系统头文件 stdio.h unistd.h stdlib.h 的内容，并把它直接插入到程序文本中。
2. 用实际值替换用#define 定义的字符串
3. 根据#if 后面的条件决定需要编译的代码
4. 特殊符号，预编译程序可以识别一些特殊的符号，预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

2.2 在 Ubuntu 下预处理的命令

命令：cpp hello.c > hello.i



图 1 预处理命令

2.3 Hello 的预处理结果解析

可以发现整个程序已经拓展为 3044 行，原来 hello.c 的程序出现在 3027 行及之后。在这之前出现的是头文件 stdio.h unistd.h stdlib.h 的依次展开。以 stdio.h 的展开为例：stdio.h 是标准库文件，cpp 到 Ubuntu 中默认的环境变量下寻找 stdio.h，打开文件/usr/include/stdio.h，发现其中依然使用了#define

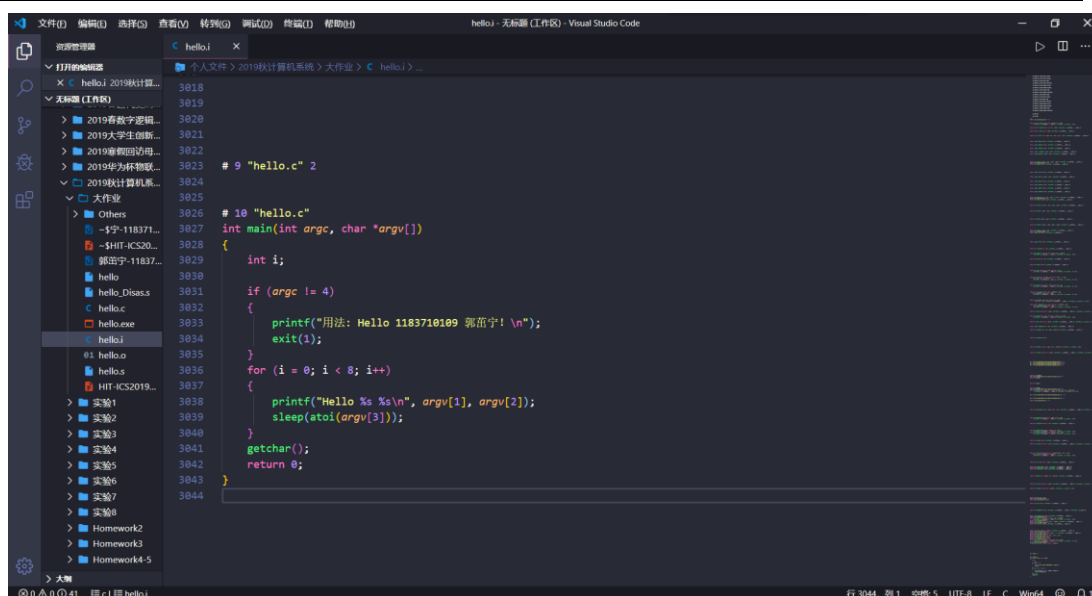


图 4 hello.i源代码部分

2.4 本章小结

本章主要介绍了预处理（包括头文件的展开、宏替换、去掉注释、条件编译）的概念和应用功能，以及 Ubuntu 下预处理的两个指令，同时具体到我们的 hello.c 文件的预处理结果 hello.i 文本文件解析，详细了解了预处理的内涵。

（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：

编译程序所要作的工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。编译器将文本文件 `hello.i` 翻译成文本文件 `hello.s`。

编译的作用：

编译包括以下基本流程：

1. 语法分析：编译程序的语法分析器以单词符号作为输入，分析单词符号串是否形成符合语法规则的语法单位，方法分为两种：自上而下分析法和自下而上分析法。
2. 中间代码：源程序的一种内部表示，或称中间语言。中间代码的作用是可使编译程序的结构在逻辑上更为简单明确，特别是可使目标代码的优化比较容易实现中间代码。
3. 代码优化：指对程序进行多种等价变换，使得从变换后的程序出发，能生成更有效的目标代码。
4. 目标代码：生成是编译的最后一个阶段。目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。此处指汇编语言代码，须经过汇编程序汇编后，成为可执行的机器语言代码。

3.2 在 Ubuntu 下编译的命令

命令：`gcc -S hello.i -o hello.s`

A terminal window with a dark background and light text. The title bar shows 'gzn@ubuntu: ~/Desktop/csapp大作业'. The terminal contains three lines of text: 'gzn@ubuntu:~/Desktop/csapp大作业\$ cpp hello.c > hello.i', 'gzn@ubuntu:~/Desktop/csapp大作业\$ gcc -S hello.i -o hello.s', and 'gzn@ubuntu:~/Desktop/csapp大作业\$' followed by a cursor. The text is color-coded: green for the prompt and blue for the command and file names.

```
gzn@ubuntu:~/Desktop/csapp大作业$ cpp hello.c > hello.i
gzn@ubuntu:~/Desktop/csapp大作业$ gcc -S hello.i -o hello.s
gzn@ubuntu:~/Desktop/csapp大作业$
```

图 5 编译命令

3.3 Hello 的编译结果解析

3.3.1 数据和赋值

3.3.1.1 常量

在 if 语句

```
if (argc != 4)
```

中，常量 4 的值保存的位置在 .text 中，作为指令的一部分

```
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $4, -20(%rbp)
je      .L2
```

同理可得

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

中的数字 0、8、1、2、3 也被存储在 .text 节中；

在下述函数中：

```
printf("用法: Hello 1183710109 郭茁宁! \n");
```

printf()、scanf() 中的字符串则被存储在 .rodata 节中

```
.LC0:
.string "\347\224\250\346\263\225: Hello 1183710109 \351
\203\255\350\214\201\345\256\201\357\274\201"
```

3.3.1.2 变量

全局变量：

初始化的全局变量储存在 .data 节，它的初始化不需要汇编语句，而是直接完成的。

局部变量：

局部变量存储在寄存器或栈中。程序中的局部变量 i 定义

```
int i;
```

在汇编代码中

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

此处是循环前 i=0 的操作，i 被保存在栈当中、%rsp-4 的位置上。

3.3.2 算术操作

在循环操作中，使用了自加++操作符：

```
for (i = 0; i < 8; i++)
```

在每次循环执行的内容结束后，对 i 进行一次自加，栈上存储变量 i 的值加 1

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

3.3.3 关系操作和控制转移

程序第 14 行中判断传入参数 `argc` 是否等于 4，源代码为

```
if (argc != 4)
{
    printf("用法: Hello 1183710109 郭苗宁! \n");
    exit(1);
}
```

汇编代码为

```
cmpl    $4, -20(%rbp)
je      .L2
```

`je` 用于判断 `cmpl` 产生的条件码，若两个操作数的值不相等则跳转到指定地址；

for 循环中的循环执行条件

```
for (i = 0; i < 8; i++)
```

汇编代码为

```
.L3:
    cmpl    $7, -4(%rbp)
    jle     .L4
```

`jle` 用于判断 `cmpl` 产生的条件码，若后一个操作数的值小于等于前一个则跳转到指定地址；

3.3.4 数组/指针/结构操作

主函数 `main` 的参数中有指针数组 `char *argv[]`

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

在 `argv` 数组中，`argv[0]` 指向输入程序的路径和名称，`argv[1]` 和 `argv[2]` 分别表示两个字符串。

因为 `char*` 数据类型占 8 个字节，根据

```
.LFB6:
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl    %edi, -20(%rbp) //argc 存储在%edi
    movq    %rsi, -32(%rbp) //argv 存储在%rsi
.L4:
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
.LC1:
    .string "Hello %s %s\n"
```

```
.text
.globl main
.type main, @function
```

对比原函数可知通过%rsi-8 和%rax-16, 分别得到 argv[1]和 argv[2]两个字符串。

3.3.5 函数操作

X86-64 中, 过程调用传递参数规则: 第 1~6 个参数一次储存在%rdi、%rsi、%rdx、%rcx、%r8、%r9 这六个寄存器中, 剩下的参数保存在栈当中。

main 函数:

参数传递: 传入参数 argc 和 argv[], 分别用寄存器%rdi 和%rsi 存储。

函数调用: 被系统启动函数调用。

函数返回: 设置%eax 为 0 并且返回, 对应 return 0 。

源代码:

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

汇编代码:

```
main:
.LFB6:
.cfi_startproc
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
```

可见 argc 存储在%edi 中, argv 存储在%rsi 中;

printf 函数:

参数传递: call puts 时只传入了字符串参数首地址; for 循环中 call printf 时传入了 argv[1]和 argv[2]的地址。

函数调用: if 判断满足条件后调用, 与 for 循环中被调用。

源代码 1:

```
printf("用法: Hello 1183710109 郭苗宁! \n");
```

汇编代码 1:

```
.LC0:
.string "\347\224\250\346\263\225: Hello 1183710109 \351\203\255\350\214\201\345\256\201\357\274\201"
```

.LFB6:

```
    cmpb    $4, -20(%rbp)
    je      .L2
    leaq     .LC0(%rip), %rdi
    call     puts@PLT
```

源代码 2:

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

汇编代码 2:

.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
```

exit 函数:

参数传递: 传入的参数为 1, 再执行退出命令

函数调用: if 判断条件满足后被调用.

源代码:

```
    exit(1);
```

汇编代码:

.LFB6:

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

sleep 函数:

参数传递: 传入参数 atoi(argv[3]),

函数调用: for 循环下被调用, call sleep

源代码:

```
    sleep(atoi(argv[3]));
```

汇编代码:

.L4:

```
    movq     -32(%rbp), %rax
    addq     $24, %rax
    movq     (%rax), %rax
```

```
movq    %rax, %rdi
call    atoi@PLT
movl    %eax, %edi
call    sleep@PLT
```

getchar 函数:

函数调用：在 main 中被调用，call getchar

源代码：

```
getchar();
```

汇编代码：

```
.L3:
    call    getchar@PLT
```

3.4 本章小结

本章主要介绍了编译的概念以及过程。同时通过示例函数表现了 c 语言如何转换为汇编代码。介绍了汇编代码如何实现变量、常量、传递参数以及分支和循环。编译程序所做的工作，就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码表示。包括之前对编译的结果进行解析，都令我更深刻地理解了 C 语言的数据与操作，对 C 语言翻译成汇编语言有了更好的掌握。因为汇编语言的通用性，这也相当于掌握了语言间的一些共性。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

概念

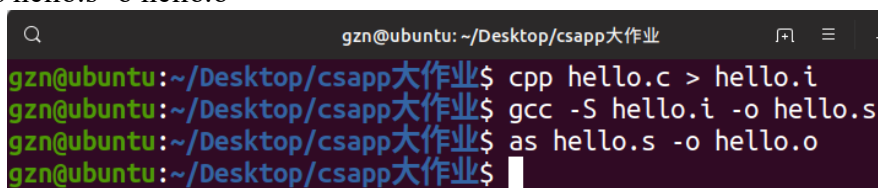
驱动程序运行汇编器 `as`, 将汇编语言(这里是 `hello.s`)翻译成机器语言(`hello.o`)的过程称为汇编, 同时这个机器语言文件也是可重定位目标文件。

作用

汇编就是将高级语言转化为机器可直接识别执行的代码文件的过程, 汇编器将 `.s` 汇编程序翻译成机器语言指令, 把这些指令打包成可重定位 目标程序的格式, 并将结果保存在 `.o` 目标文件中, `.o` 文件是一个二进制文件, 它 包含程序的指令编码。

4.2 在 Ubuntu 下汇编的命令

命令: `as hello.s -o hello.o`



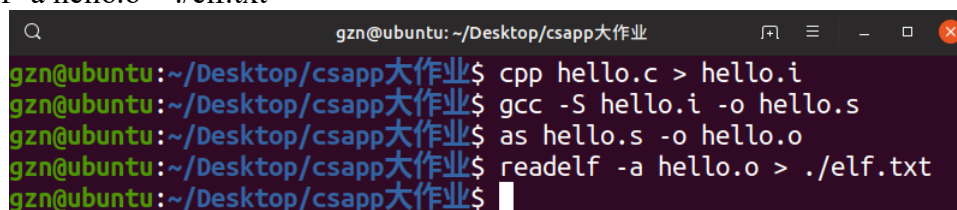
```
gzn@ubuntu: ~/Desktop/csapp大作业
gzn@ubuntu:~/Desktop/csapp大作业$ cpp hello.c > hello.i
gzn@ubuntu:~/Desktop/csapp大作业$ gcc -S hello.i -o hello.s
gzn@ubuntu:~/Desktop/csapp大作业$ as hello.s -o hello.o
gzn@ubuntu:~/Desktop/csapp大作业$
```

图 6 汇编命令

4.3 可重定位目标 elf 格式

4.3.1 命令:

`readelf -a hello.o > ./elf.txt`



```
gzn@ubuntu: ~/Desktop/csapp大作业
gzn@ubuntu:~/Desktop/csapp大作业$ cpp hello.c > hello.i
gzn@ubuntu:~/Desktop/csapp大作业$ gcc -S hello.i -o hello.s
gzn@ubuntu:~/Desktop/csapp大作业$ as hello.s -o hello.o
gzn@ubuntu:~/Desktop/csapp大作业$ readelf -a hello.o > ./elf.txt
gzn@ubuntu:~/Desktop/csapp大作业$
```

图 7 生成并导出elf文件命令

4.3.2 ELF 头:

包含了系统信息, 编码方式, ELF 头大小, 节的大小和数量等一系列信息。Elf 头内容如下:

ELF Header:


```

Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
Class:                                     ELF64
Data:                                       2's complement, little endian
Version:                                  1 (current)
OS/ABI:                                   UNIX - System V
ABI Version:                              0
Type:                                     REL (Relocatable file)
Machine:                                  Advanced Micro Devices X86-64
Version:                                  0x1
Entry point address:                      0x0
Start of program headers:                  0 (bytes into file)
Start of section headers:                  1152 (bytes into file)
Flags:                                     0x0
Size of this header:                       64 (bytes)
Size of program headers:                   0 (bytes)
Number of program headers:                 0
Size of section headers:                   64 (bytes)
Number of section headers:                 13
Section header string table index:         12

```

4.3.3 节头目表:

描述了.o文件中出现的各个节的类型、位置、所占空间大小等信息。

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
	000000000000008e	0000000000000000	AX 0 0	1
[2]	.rela.text	RELA	0000000000000000	00000340
	00000000000000c0	0000000000000018	I 10 1	8
[3]	.data	PROGBITS	0000000000000000	000000ce
	0000000000000000	0000000000000000	WA 0 0	1
[4]	.bss	NOBITS	0000000000000000	000000ce
	0000000000000000	0000000000000000	WA 0 0	1
[5]	.rodata	PROGBITS	0000000000000000	000000d0
	0000000000000033	0000000000000000	A 0 0	8
[6]	.comment	PROGBITS	0000000000000000	00000103

	0000000000000024	0000000000000001	MS	0	0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	00000127		
	0000000000000000	0000000000000000		0	0	1
[8]	.eh_frame	PROGBITS	0000000000000000	00000128		
	0000000000000038	0000000000000000	A	0	0	8
[9]	.rela.eh_frame	RELA	0000000000000000	00000400		
	0000000000000018	0000000000000018	I	10	8	8
[10]	.symtab	SYMTAB	0000000000000000	00000160		
	0000000000000198	0000000000000018		11	9	8
[11]	.strtab	STRTAB	0000000000000000	000002f8		
	0000000000000048	0000000000000000		0	0	1
[12]	.shstrtab	STRTAB	0000000000000000	00000418		
	0000000000000061	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)

4.3.4 重定位节:

表述了各个段引用的外部符号等, 在链接时, 需要通过重定位节对这些位置的地址进行修改。链接器会通过重定位条目的类型判断该使用什么养的方法计算正确的地址值, 通过偏移量等信息计算出正确的地址。

本程序需要重定位的信息有: .rodata 中的模式串, puts, exit, printf, slepsecs, sleep, getchar 这些符号。

Relocation section '.rela.text' at offset 0x340 contains 8 entries:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
00000000001d	000b00000004	R_X86_64_PLT32	0000000000000000	puts - 4
000000000027	000c00000004	R_X86_64_PLT32	0000000000000000	exit - 4
000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 22
00000000005a	000d00000004	R_X86_64_PLT32	0000000000000000	printf - 4
00000000006d	000e00000004	R_X86_64_PLT32	0000000000000000	atoi - 4
000000000074	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4
000000000083	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

Relocation section '.rela.eh_frame' at offset 0x400 contains 1 entry:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
--------	------	------	------------	--------------------

```
000000000020 000200000002 R_X86_64_PC32 0000000000000000 .text + 0
```

4.3.5 符号表:

.symtab 是一个符号表，它存放在程序中定义和引用的函数和全局变量的信息。

Symbol table '.symtab' contains 17 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	hello.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
9:	0000000000000000	142	FUNC	GLOBAL	DEFAULT	1	main
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_GLOBAL_OFFSET_TABLE_
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	puts
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	exit
13:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf
14:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	atoi
15:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sleep
16:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	getchar

4.4 Hello.o 的结果解析

命令: `objdump -d -r hello.o > Disas_hello.s`

hello.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:

```

0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)

```

```

b: 48 89 75 e0      mov    %rsi,-0x20(%rbp)
f: 83 7d ec 04      cmpl   $0x4,-0x14(%rbp)
13: 74 16            je     2b <main+0x2b>
15: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi      # 1c <main+0x1c>
18: R_X86_64_PC32 .rodata-0x4
1c: e8 00 00 00 00      callq 21 <main+0x21>
1d: R_X86_64_PLT32 puts-0x4
21: bf 01 00 00 00      mov    $0x1,%edi
26: e8 00 00 00 00      callq 2b <main+0x2b>
27: R_X86_64_PLT32 exit-0x4
2b: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
32: eb 48            jmp    7c <main+0x7c>
34: 48 8b 45 e0      mov    -0x20(%rbp),%rax
38: 48 83 c0 10      add    $0x10,%rax
3c: 48 8b 10      mov    (%rax),%rdx
3f: 48 8b 45 e0      mov    -0x20(%rbp),%rax
43: 48 83 c0 08      add    $0x8,%rax
47: 48 8b 00      mov    (%rax),%rax
4a: 48 89 c6      mov    %rax,%rsi
4d: 48 8d 3d 00 00 00 00 lea    0x0(%rip),%rdi      # 54 <main+0x54>
50: R_X86_64_PC32 .rodata+0x22
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: 48 8b 45 e0      mov    -0x20(%rbp),%rax
62: 48 83 c0 18      add    $0x18,%rax
66: 48 8b 00      mov    (%rax),%rax
69: 48 89 c7      mov    %rax,%rdi
6c: e8 00 00 00 00      callq 71 <main+0x71>
6d: R_X86_64_PLT32 atoi-0x4
71: 89 c7      mov    %eax,%edi
73: e8 00 00 00 00      callq 78 <main+0x78>
74: R_X86_64_PLT32 sleep-0x4
78: 83 45 fc 01      addl   $0x1,-0x4(%rbp)
7c: 83 7d fc 07      cmpl   $0x7,-0x4(%rbp)
80: 7e b2      jle    34 <main+0x34>
82: e8 00 00 00 00      callq 87 <main+0x87>
83: R_X86_64_PLT32 getchar-0x4
87: b8 00 00 00 00      mov    $0x0,%eax

```

```
8c: c9          leaveq
```

```
8d: c3          retq
```

分析 `hello.o` 的反汇编，并与第 3 章的 `hello.s` 进行对照分析：

1. 数的表示：`hello.s` 中的操作数是十进制，`hello.o` 反汇编代码中的操作数是十六进制。
2. 分支转移：跳转语句之后，`hello.s` 中是 `.L2` 和 `.LC1` 等段名称，而反汇编代码中跳转指令之后是相对偏移的地址，也即间接地址。
3. 函数调用：`hello.s` 中，`call` 指令使用的是函数名称，而反汇编代码中 `call` 指令使用的是 `main` 函数的相对偏移地址。因为函数只有在链接之后才能确定运行执行的地址，因此在 `.rela.text` 节中为其添加了重定位条目。

4.5 本章小结

本章对汇编结果进行了详尽的介绍。经过汇编器的操作，汇编语言转化为机器语言，`hello.o` 可重定位目标文件的生成为后面的链接做了准备。通过对比 `hello.s` 和 `hello.o` 反汇编代码的区别，令人更深刻地理解了汇编语言到机器语言实现地转变，和这过程中为链接做出的准备，对可重定位目标 `elf` 格式进行了详细的分析，侧重点在重定位项目上。同时对 `hello.o` 文件进行反汇编，将 `Disas_hello.s` 与之前生成的 `hello.s` 文件进行了对比。使得我们对该内容有了更加深入地理解。

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

链接器概念：

链接是将各种不同文件的代码和数据部分收集（符号解析和重定位）起来并组合成一个单一文件的过程。

链接器作用：

令源程序节省空间而未编入的常用函数文件（如 `printf.o`）进行合并，生成可以正常工作的可执行文件。这令分离编译成为可能，节省了大量的工作空间。

5.2 在 Ubuntu 下链接的命令

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

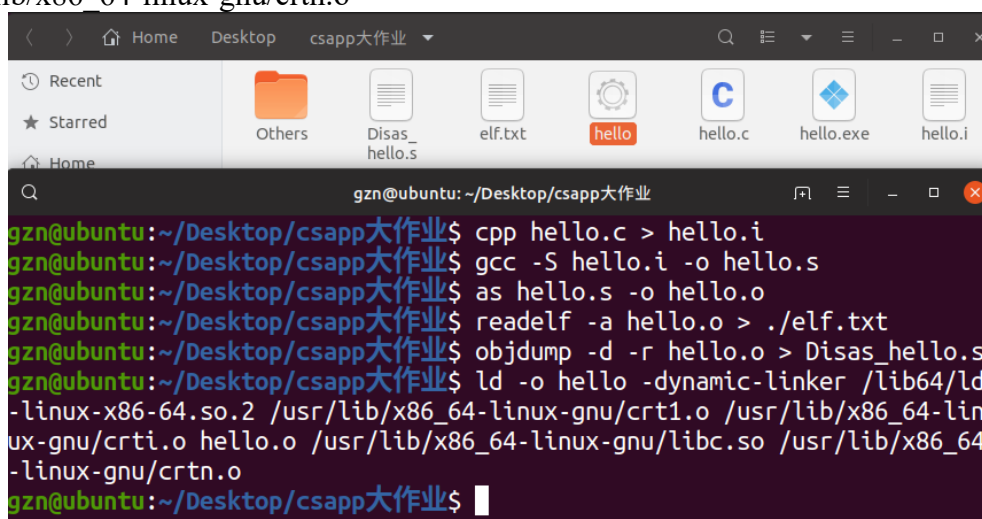


图 8 链接命令

5.3 可执行目标文件 hello 的格式

命令：`readelf -a hello > hello1.elf`

ELF 文件头：

ELF Header:

```

Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
Class:                               ELF64
Data:                                   2's complement, little endian
Version:                             1 (current)

```

```

OS/ABI:                UNIX - System V
ABI Version:            0
Type:                  EXEC (Executable file)
Machine:               Advanced Micro Devices X86-64
Version:               0x1
Entry point address:    0x401090
Start of program headers: 64 (bytes into file)
Start of section headers: 14120 (bytes into file)
Flags:                 0x0
Size of this header:    64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 10
Size of section headers: 64 (bytes)
Number of section headers: 25
Section header string table index: 24

```

节头:

描述了各个节的大小、偏移量和其他属性。链接器链接时，会将各个文件的相同段合并成一个大段，并且根据这个大段的大小以及偏移量重新设置各个符号的地址。

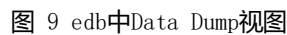
Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0 0	
[1]	.interp	PROGBITS	0000000000400270	00000270
	000000000000001c	0000000000000000	A 0 0 1	
[2]	.note.ABI-tag	NOTE	000000000040028c	0000028c
	0000000000000020	0000000000000000	A 0 0 4	
[3]	.hash	HASH	00000000004002b0	000002b0
	0000000000000038	0000000000000004	A 5 0 8	
[4]	.gnu.hash	GNU_HASH	00000000004002e8	000002e8
	000000000000001c	0000000000000000	A 5 0 8	
[5]	.dynsym	DYNSYM	0000000000400308	00000308
	00000000000000d8	0000000000000018	A 6 1 8	
[6]	.dynstr	STRTAB	00000000004003e0	000003e0
	000000000000005c	0000000000000000	A 0 0 1	
[7]	.gnu.version	VERSYM	000000000040043c	0000043c
	0000000000000012	0000000000000002	A 5 0 2	

[8]	.gnu.version_r	VERNEED	0000000000400450	00000450
	0000000000000020	0000000000000000	A	6 1 8
[9]	.rela.dyn	RELA	0000000000400470	00000470
	0000000000000030	0000000000000018	A	5 0 8
[10]	.rela.plt	RELA	00000000004004a0	000004a0
	0000000000000090	0000000000000018	AI	5 19 8
[11]	.init	PROGBITS	0000000000401000	00001000
	0000000000000017	0000000000000000	AX	0 0 4
[12]	.plt	PROGBITS	0000000000401020	00001020
	0000000000000070	0000000000000010	AX	0 0 16
[13]	.text	PROGBITS	0000000000401090	00001090
	0000000000000121	0000000000000000	AX	0 0 16
[14]	.fini	PROGBITS	00000000004011b4	000011b4
	0000000000000009	0000000000000000	AX	0 0 4
[15]	.rodata	PROGBITS	0000000000402000	00002000
	000000000000003b	0000000000000000	A	0 0 8
[16]	.eh_frame	PROGBITS	0000000000402040	00002040
	00000000000000fc	0000000000000000	A	0 0 8
[17]	.dynamic	DYNAMIC	0000000000403e50	00002e50
	00000000000001a0	0000000000000010	WA	6 0 8
[18]	.got	PROGBITS	0000000000403ff0	00002ff0
	0000000000000010	0000000000000008	WA	0 0 8
[19]	.got.plt	PROGBITS	0000000000404000	00003000
	0000000000000048	0000000000000008	WA	0 0 8
[20]	.data	PROGBITS	0000000000404048	00003048
	0000000000000004	0000000000000000	WA	0 0 1
[21]	.comment	PROGBITS	0000000000000000	0000304c
	0000000000000023	0000000000000001	MS	0 0 1
[22]	.symtab	SYMTAB	0000000000000000	00003070
	0000000000000498	0000000000000018		23 28 8
[23]	.strtab	STRTAB	0000000000000000	00003508
	0000000000000158	0000000000000000		0 0 1
[24]	.shstrtab	STRTAB	0000000000000000	00003660
	00000000000000c5	0000000000000000		0 0 1

5.4 hello 的虚拟地址空间

使用 edb 加载 hello，Data Dump 窗口可以查看加载到虚拟地址中的 hello 程序。查看 ELF 格式文件中的 Program Headers，它告诉链接器运行时加载的内容，并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间



The diagram illustrates the memory layout of a process, divided into two main sections: **内核虚拟内存** (Kernel Virtual Memory) and **进程虚拟内存** (Process Virtual Memory).

内核虚拟内存 (Kernel Virtual Memory):

- 与进程相关的数据结构 (如页表, task和mm结构, 内核栈):** This region is unique to each process (*对每个进程都不相同*).
- 物理内存:** This region is shared by all processes (*对每个进程都一样*).
- 内核代码和数据:** This region is also shared by all processes (*对每个进程都一样*).

进程虚拟内存 (Process Virtual Memory):

- 用户栈 (User Stack):** The top of the process's virtual memory, starting at address **%rsp**. It is shown as a green box with a downward arrow indicating growth.
- 共享库的内存映射区域 (Shared Library Memory Mapping Region):** A gray box representing memory mapped from shared libraries.
- 运行时堆 (malloc) (Runtime Heap):** A green box that grows upwards, indicated by an upward arrow and the **brk** pointer.
- 未初始化的数据 (.bss) (Uninitialized Data):** A light blue box.
- 已初始化的数据 (.data) (Initialized Data):** A dark blue box.
- 代码 (.text) (Code):** A yellow box at the bottom of the process's virtual memory.

The address **0x00400000** is marked at the bottom of the diagram, and the address **0** is at the very bottom.

图 10 Linux 进程的虚拟地址空间

5.5 链接的重定位过程分析

命令: `objdump -d -r hello > hello_objdump.s`

```
hello:      file format elf64-x86-64
```

Disassembly of section .init:

```
000000000401000 <_init>:
  401000: 48 83 ec 08          sub    $0x8,%rsp
  401004: 48 8b 05 ed 2f 00 00 mov    0x2fed(%rip),%rax
# 403ff8 <__gmon_start__>
  40100b: 48 85 c0            test   %rax,%rax
  40100e: 74 02              je     401012 <_init+0x12>
  401010: ff d0             callq  *%rax
  401012: 48 83 c4 08        add    $0x8,%rsp
  401016: c3                retq
```

分析 **hello** 与 **hello.o** 的不同:

1. 链接增加新的函数:

在 **hello** 中链接加入了在 **hello.c** 中用到的库函数, 如 **exit**、**printf**、**sleep**、**getchar** 等函数。

2. 增加的节:

hello 中增加了 **.init** 和 **.plt** 节, 和一些节中定义的函数。

3. 函数调用:

hello 中无 **hello.o** 中的重定位条目, 并且跳转和函数调用的地址在 **hello** 中都变成了虚拟内存地址。对于 **hello.o** 的反汇编代码, 函数只有在链接之后才能确定运行执行的地址, 因此在 **.rela.text** 节中为其添加了重定位条目。

4. 地址访问:

hello.o 中的相对偏移地址变成了 **hello** 中的虚拟内存地址。而 **hello.o** 文件中对于某些地址的定位是不明确的, 其地址也是在运行时确定的, 因此访问也需要重定位, 在汇编成机器语言时, 将操作数全部置为 0, 并且添加重定位条目。

链接的过程:

根据 **hello** 和 **hello.o** 的不同, 分析出链接的过程为:

链接就是链接器 (**ld**) 将各个目标文件 (各种 **.o** 文件) 组装在一起, 文件中的各个函数段按照一定规则累积在一起。

5.6 **hello** 的执行流程

子函数名和地址 (后 6 位)

```
401000 <_init>
401020 <_plt>
401030 <puts@plt>
401040 <printf@plt>
401050 <getchar@plt>
401060 <atoi@plt>
```

```

401070 <exit@plt>
401080 <sleep@plt>
401090 <_start>
4010c0 <_dl_relocate_static_pie>
4010c1 <main>
401150 <__libc_csu_init>
4011b0 <__libc_csu_fini>
4011b4 <_fini>

```

通过 edb 的调试，一步一步地记录下 call 命令进入的函数。

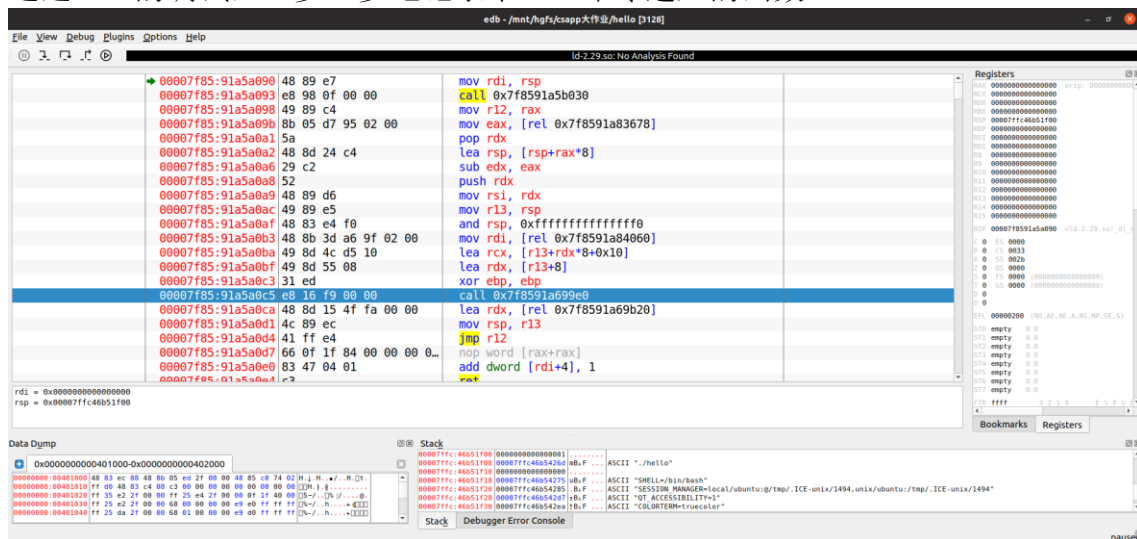


图 11 edb运行程序

5.7 Hello 的动态链接分析

在 elf 文件中可以找到：

```

[18] .got                PROGBITS                0000000000403ff0  00002ff0
      0000000000000010  0000000000000008  WA          0      0      8
[19] .got.plt             PROGBITS                0000000000404000  00003000
      0000000000000048  0000000000000008  WA          0      0      8
0x0000000000000003 (PLTGOT)                0x404000

```

进入 edb 查看：

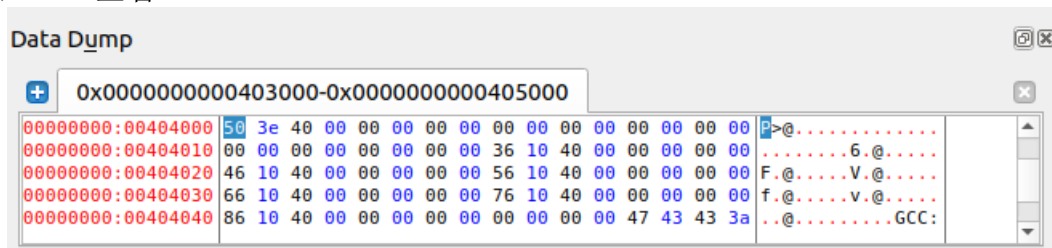


图 12 edb执行init之前的地址

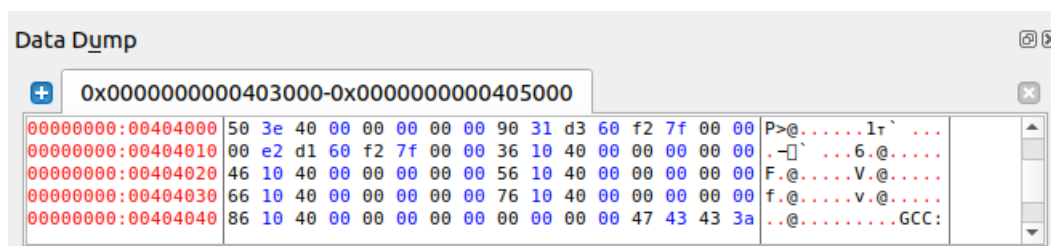


图 13 edb在执行init之后的地址

对于变量而言，我们利用代码段和数据段的相对位置不变的原则计算正确地址。对于库函数而言，需要 `plt`、`got` 合作，`plt` 初始存的是一批代码，它们跳转到 `got` 所指示的位置，然后调用链接器。初始时 `got` 里面存的都是 `plt` 的第二条指令，随后链接器修改 `got`，下一次再调用 `plt` 时，指向的就是正确的内存地址。`plt` 就能跳转到正确的区域。

5.8 本章小结

本章主要了解温习了在 `linux` 中链接的过程。通过查看 `hello` 的虚拟地址空间，并且对比 `hello` 与 `hello.o` 的反汇编代码，更好地掌握了链接与之中重定位的过程。不过，链接远不止本章所涉及的这么简单，就像是 `hello` 会在它运行时要求动态链接器加载和链接某个共享库，而无需在编译时将那些库链接到应用中。

(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：

进程是执行中程序的抽象。

进程的作用：

- 每次运行程序时，shell 创建一新进程，在这个进程的上下文切换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的关键抽象：一个独立的逻辑控制流，如同程序独占处理器；一个私有的地址空间，如同程序独占内存系统。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：解释命令，连接用户和操作系统以及内核

流程：

shell 先分词，判断命令是否为内部命令，如果不是，则寻找可执行文件进行执行，重复这个流程：

1. Shell 首先从命令行中找出特殊字符（元字符），在将元字符翻译成间隔符号。元字符将命令行划分成小块 tokens。Shell 中的元字符如下所示：SPACE, TAB, NEWLINE, &, ;, (,), <, >, |
2. 程序块 tokens 被处理，检查他们是否是 shell 中所引用到的关键字。
3. 当程序块 tokens 被确定以后，shell 根据 aliases 文件中的列表来检查命令的第一个单词。如果这个单词出现在 aliases 表中，执行替换操作并且处理过程回到第一步重新分割程序块 tokens。
4. Shell 对~符号进行替换。
5. Shell 对所有前面带有\$符号的变量进行替换。
6. Shell 将命令行中的内嵌命令表达式替换成命令；他们一般都采用\$(command)标记法。
7. Shell 计算采用\$(expression)标记的算术表达式。
8. Shell 将命令字符串重新划分为新的块 tokens。这次划分的依据是栏位分割符号，称为 IFS。缺省的 IFS 变量包含有：SPACE, TAB 和换行符号。
9. Shell 执行通配符*?[]的替换。
10. shell 把所有從處理的結果中用到的注释删除，並且按照下面的顺序实行命令的检查：

- I. 内建的命令
 - II. shell 函数（由用户自己定义的）
 - III. 可执行的脚本文件（需要寻找文件和 PATH 路径）
11. 在执行前的最后一步是初始化所有的输入输出重定向。
 12. 最后，执行命令。

6.3 Hello 的 fork 进程创建过程

根据 shell 的处理流程，可以推断，输入命令执行 hello 后，父进程如果判断不是内部指令，即会通过 fork 函数创建子进程。子进程与父进程近似，并得到一份与父进程用户级虚拟空间相同且独立的副本——包括数据段、代码、共享库、堆和用户栈。父进程打开的文件，子进程也可读写。二者之间最大的不同或许在于 PID 的不同。Fork 函数只会被调用一次，但会返回两次，在父进程中，fork 返回子进程的 PID，在子进程中，fork 返回 0。

6.4 Hello 的 execve 过程

execve 函数在加载并运行可执行目标文件 Hello，且带列表 argv 和环境变量列表 envp。该函数的作用就是在当前进程的上下文中加载并运行一个新的程序。只有当出现错误时，例如找不到 Hello 时，execve 才会返回到调用程序，这里与一次调用两次返回的 fork 不同。

在 execve 加载了 Hello 之后，它调用启动代码。启动代码设置栈，并将控制传递给新程序的主函数，该主函数有如下的原型：

```
int main(int argc, char **argv, char *envp);
```

结合虚拟内存和内存映射过程，可以更详细地说明 execve 函数实际上是如何加载和执行程序 Hello：

1. 删除已存在的用户区域（自父进程独立）。
2. 映射私有区：为 Hello 的代码、数据、.bss 和栈区域创建新的区域结构，所有这些区域都是私有的、写时才复制的。
3. 映射共享区：比如 Hello 程序与标准 C 库 libc.so 链接，这些对象都是动态链接到 Hello 的，然后再用户虚拟地址空间中的共享区域内。
4. 设置 PC：execve 做的最后一件事就是设置当前进程的上下文中的程序计数器，使之指向代码区域的入口点。

6.5 Hello 的进程执行

逻辑控制流：

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

用户模式和内核模式：

处理器使用一个寄存器提供两种模式的区分。用户模式的进程不允许执行特殊指令，不允许直接引用地址空间中内核区的代码和数据；内核模式进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

上下文：

上下文就是内核重新启动一个被抢占的进程所需要恢复的原来的状态，由寄存器、程序计数器、用户栈、内核栈和内核数据结构等对象的值构成。

示例：sleep 进程的调度过程

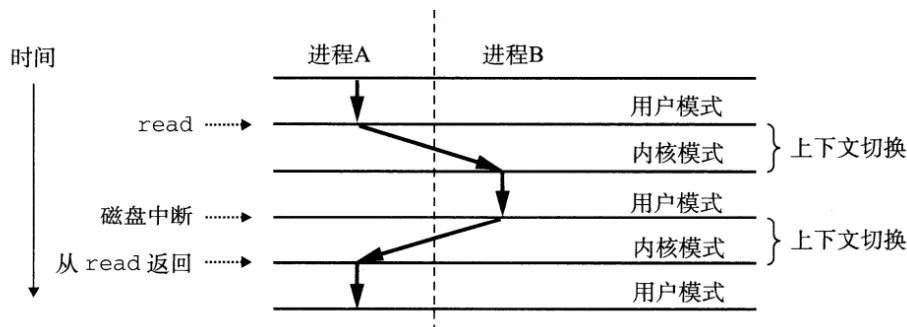


图 14 进程上下文切换

初始时，控制流再 `hello` 内，处于用户模式

调用系统函数 `sleep` 后，进入内核态，此时间片停止。

2s 后，发送中断信号，转回用户模式，继续执行指令。

调度的过程：

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了了的进程，这种决策就叫做调度，是由内核中称为调度器的代码处理的。当内核选择一个新的进程运行，我们说内核调度了这个进程。在内核调度了一个新的进程运行了之后，它就抢占了当前进程，并使用上下文切换机制来将控制转移到新的进程。

以执行 `sleep` 函数为例，`sleep` 函数请求调用休眠进程，`sleep` 将内核抢占，进入倒计时，当倒计时结束后，`hello` 程序重新抢占内核，继续执行。

用户态与核心态转换：

为了能让处理器安全运行，不至于损坏操作系统，必然需要先知应用程序可执行指令所能访问的地址空间范围。因此，就存在了用户态与核心态的划分，核心态可以说是“创世模式”，拥有最高的访问权限，处理器以一个寄存器当做模式位来描述当前进程的特权。进程只有故障、中断或陷入系统调用时才会得到内核访问权限，其他情况下始终处于用户权限之中，保证了系统的安全性。

6.6 hello 的异常与信号处理

正常运行状态:

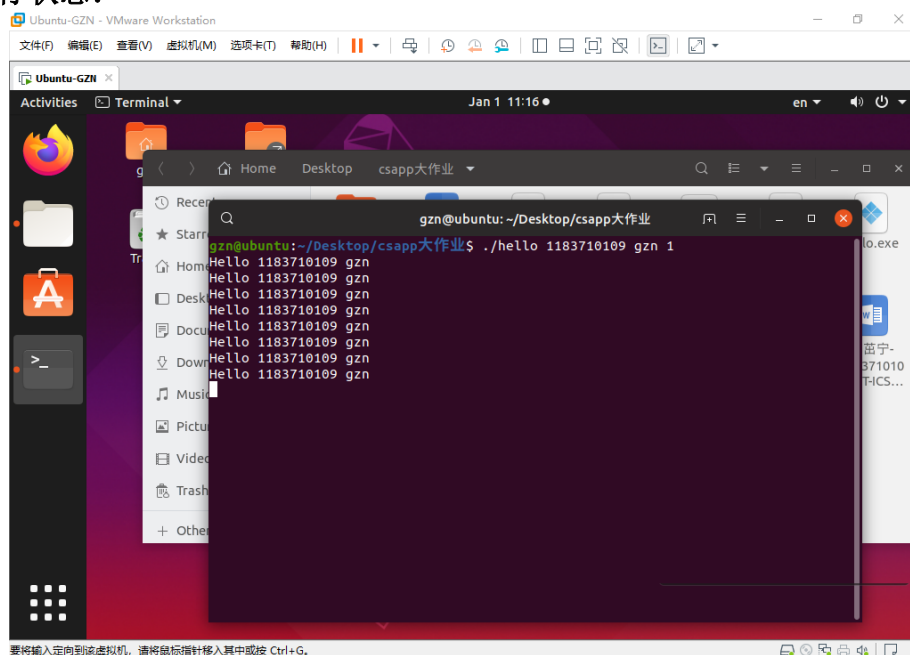


图 15 正常运行状态

异常类型:

类别	原因	异步/同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

处理方式:

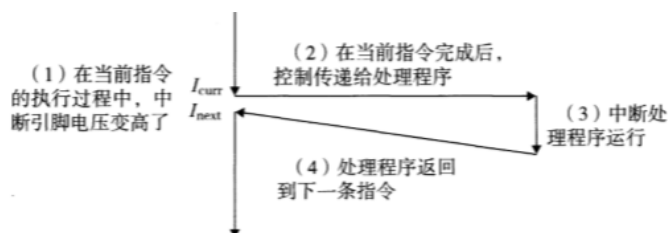


图 16 中断处理方式

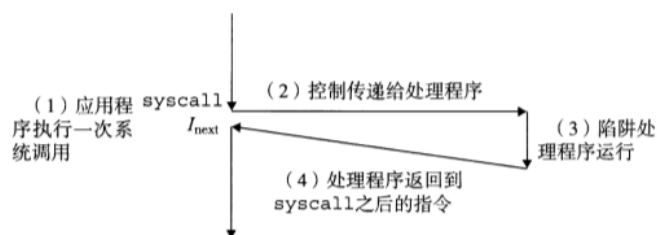


图 17 陷阱处理方式

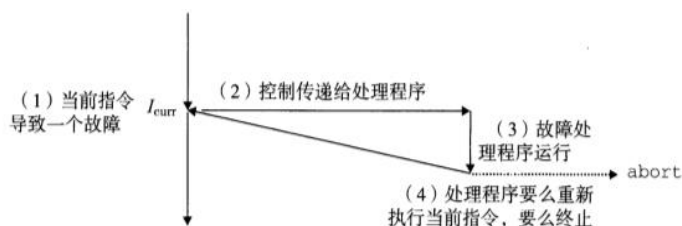


图 18 故障处理方式

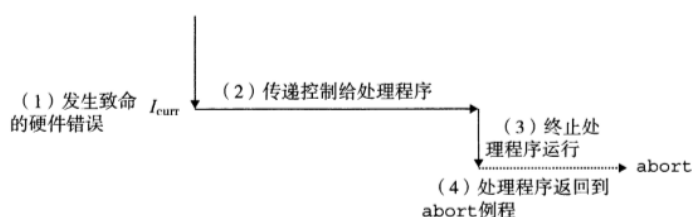


图 19 终止处理方式

按下 **Ctrl+Z**：进程收到 SIGSTP 信号，hello 进程挂起。用 ps 查看其进程 PID，可以发现 hello 的 PID 是 2681；再用 jobs 查看此时 hello 的后台 job 号是 1，调用 fg 1 将其调回前台。

```

gzn@ubuntu: ~/Desktop/csapp大作业
gzn@ubuntu:~/Desktop/csapp大作业$ ./hello 1183710109 gzn 1
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
^Z
[1]+  Stopped                  ./hello 1183710109 gzn 1
gzn@ubuntu:~/Desktop/csapp大作业$ ps
  PID TTY          TIME CMD
 2674 pts/0    00:00:00 bash
 2681 pts/0    00:00:00 hello
 2682 pts/0    00:00:00 ps
gzn@ubuntu:~/Desktop/csapp大作业$ jobs
[1]+  Stopped                  ./hello 1183710109 gzn 1
gzn@ubuntu:~/Desktop/csapp大作业$ fg 1
./hello 1183710109 gzn 1
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
  
```

图 20 按下Ctrl+Z运行状态

Ctrl+C：进程收到 SIGINT 信号，结束 hello。在 ps 中查询不到其 PID，在 jobs 中也没有显示，可以看出 hello 已经被彻底结束。

```

gzn@ubuntu: ~/Desktop/csapp大作业
gzn@ubuntu:~/Desktop/csapp大作业$ ./hello 1180300821 gzn 2
Hello 1180300821 gzn
Hello 1180300821 gzn
Hello 1180300821 gzn
Hello 1180300821 gzn
Hello 1180300821 gzn
Hello 1180300821 gzn
Hello 1180300821 gzn
Hello 1180300821 gzn
^C
gzn@ubuntu:~/Desktop/csapp大作业$ ps
  PID TTY          TIME CMD
 2674 pts/0    00:00:00 bash
 2708 pts/0    00:00:00 ps
gzn@ubuntu:~/Desktop/csapp大作业$ jobs
gzn@ubuntu:~/Desktop/csapp大作业$ fg 1
bash: fg: 1: no such job
gzn@ubuntu:~/Desktop/csapp大作业$

```

图 21 按下Ctrl+C运行状态

中途乱按：只是将屏幕的输入缓存到缓冲区。乱码被认为是命令。

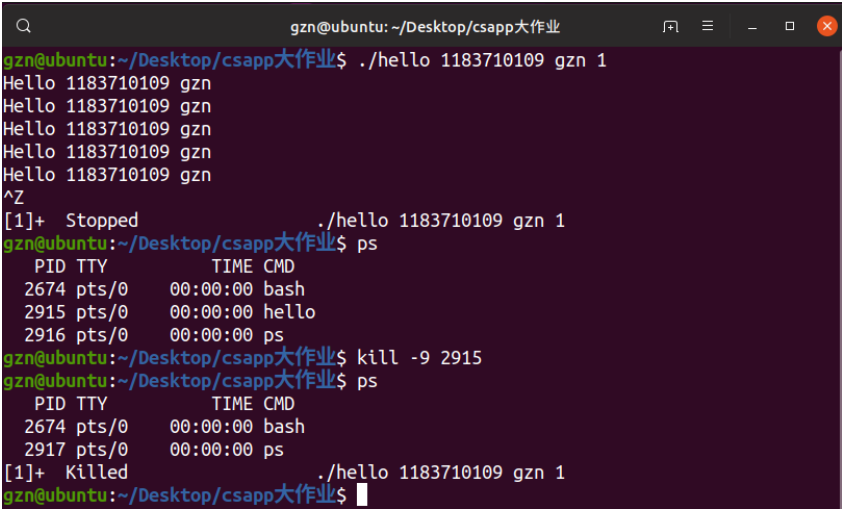
```

gzn@ubuntu: ~/Desktop/csapp大作业
gzn@ubuntu:~/Desktop/csapp大作业$ ./hello 1183710109 gzn 1
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
qmnHello 1183710109 gzn
xceHello 1183710109 gzn
rw
rwerewHello 1183710109 gzn
nwenrew
ewvHello 1183710109 gzn
c
x
vw
e
Hello 1183710109 gzn
g
g
sgzn@ubuntu:~/Desktop/csapp大作业$ rwerewnwenrew
d
rwerewnwenrew: command not found
gzn@ubuntu:~/Desktop/csapp大作业$ ewvc
fewvc: command not found
gzn@ubuntu:~/Desktop/csapp大作业$ x
x: command not found
gzn@ubuntu:~/Desktop/csapp大作业$ vw
Command 'vw' not found, but can be installed with:
sudo apt install vovpal-wabbit
gzn@ubuntu:~/Desktop/csapp大作业$ e
Command 'e' not found, but can be installed with:
sudo apt install e-wrapper
gzn@ubuntu:~/Desktop/csapp大作业$ g
g: command not found
gzn@ubuntu:~/Desktop/csapp大作业$ g
g: command not found
gzn@ubuntu:~/Desktop/csapp大作业$ sd
sd: command not found
gzn@ubuntu:~/Desktop/csapp大作业$ F^C
gzn@ubuntu:~/Desktop/csapp大作业$

```

图 22 中途乱按运行状态

Kill 命令：挂起的进程被终止，在 ps 中无法查到其 PID。



```
gzn@ubuntu: ~/Desktop/csapp大作业
gzn@ubuntu:~/Desktop/csapp大作业$ ./hello 1183710109 gzn 1
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
Hello 1183710109 gzn
^Z
[1]+  Stopped                  ./hello 1183710109 gzn 1
gzn@ubuntu:~/Desktop/csapp大作业$ ps
  PID TTY          TIME CMD
 2674 pts/0        00:00:00 bash
 2915 pts/0        00:00:00 hello
 2916 pts/0        00:00:00 ps
gzn@ubuntu:~/Desktop/csapp大作业$ kill -9 2915
gzn@ubuntu:~/Desktop/csapp大作业$ ps
  PID TTY          TIME CMD
 2674 pts/0        00:00:00 bash
 2917 pts/0        00:00:00 ps
[1]+  Killed                  ./hello 1183710109 gzn 1
gzn@ubuntu:~/Desktop/csapp大作业$
```

图 23 输入kill命令运行状态

6.7 本章小结

本章了解了 hello 进程的执行过程。主要讲 hello 的创建、加载和终止，通过键盘输入。程序是指令、数据及其组织形式的描述，进程是程序的实体。可以说，进程是运行的程序。在 hello 运行过程中，内核有选择对其进行管理，决定何时进行上下文切换。也同样是在 hello 的运行过程中，当接受到不同的异常信号时，异常处理程序将对异常信号做出相应，执行相应的代码，每种信号都有不同的处理机制，对不同的异常信号，hello 也有不同的处理结果。我们对 hello 执行过程中产生信号和信号的处理过程有了更多的认识，对使用 linux 调试运行程序也有了更多的新得。

(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

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)。

- 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。程序 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。

- 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。

- 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 的线性地址就直接成为物理地址了。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部分组成，段标识符，段内偏移量。段标识符是一个 16 位长的字段组成，称为段选择符，其中前 13 位是一个索引号。后面三位包含一些硬件细节。

索引号，可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段。

这里面，我们只关心 Base 字段，它描述了一个段的开始位置的线性地址。

全局的段描述符，放在“全局段描述符表(GDT)”中，一些局部的段描述符，放在“局部段描述符表(LDT)”中。

GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。

给定一个完整的逻辑地址段选择符+段内偏移地址，

看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，它了 Base，即基地址就知道了。

把 Base + offset，就是要转换的线性地址了

7.3 Hello 的线性地址到物理地址的变换-页式管理

页式管理是一种内存空间存储管理的技术，页式管理分为静态页式管理和动态页式管理。将各进程的虚拟空间划分成若干个长度相等的页(page)，页式管理把内存空间按页的大小划分成片或者页面（page frame），然后把页式虚拟地址与内存地址建立一一对应页表，并用相应的硬件地址变换机构，来解决离散地址变换问题。页式管理采用请求调页或预调页技术实现了内外存存储器的统一管理。

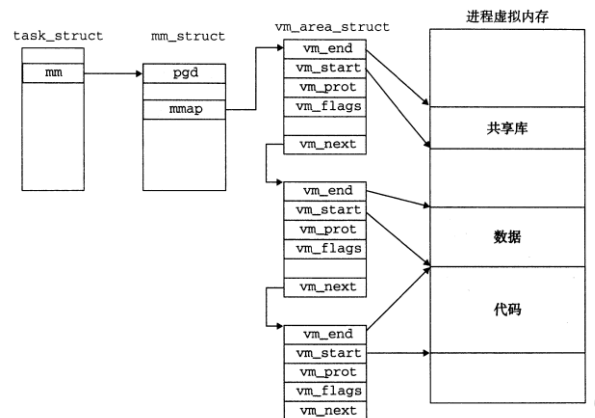


图 24 页式管理流程图

优点：

- 1、由于它不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了碎片问题。
- 2、动态页式管理提供了内存和外存统一管理的虚存实现方式，使用户可以利用的存储空间大大增加。这既提高了主存的利用率，又有利于组织多道程序执行。

缺点：

- 1、要求有相应的硬件支持。例如地址变换机构，缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。这增加了机器成本。
- 2、增加了系统开销，例如缺页中断处理机，
- 3、请求调页的算法如选择不当，有可能产生抖动现象。
- 4、虽然消除了碎片，但每个作业或进程的最后一页内总有一部分空间得不到利用。如果页面较大，则这一部分的损失仍然较大。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

每次 CPU 产生一个虚拟地址，MMU（内存管理单元）就必须查阅一个 PTE（页表条目），以便将虚拟地址翻译为物理地址。在最糟糕的情况下，这会从内存多取

一次数据，代价是几十到几百个周期。如果 PTE 碰巧缓存在 L1 中，那么开销就会下降 1 或 2 个周期。然而，许多系统都试图消除即使是这样的开销，它们在 MMU 中包括了一个关于 PTE 的小的缓存，称为翻译后备缓存器（TLB）。

多级页表：

将虚拟地址的 VPN 划分为相等大小的不同的部分，每个部分用于寻找由上一级确定的页表基址对应的页表条目。

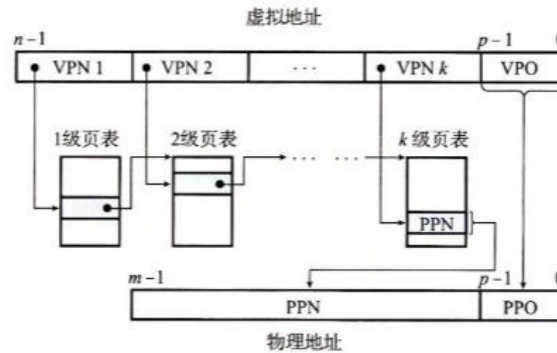


图 25 使用k级页表进行翻译

解析 VA，利用前 m 位 vpn1 寻找一级页表位置，接着一次重复 k 次，在第 k 级页表获得了页表条目，将 PPN 与 VPO 组合获得 PA

7.5 三级 Cache 支持下的物理内存访问

CPU 发送一条虚拟地址，随后 MMU 按照上述操作获得了物理地址 PA。根据 cache 大小组数的要求，将 PA 分为 CT（标记位）CS(组号)，CO（偏移量）。根据 CS 寻找到正确的组，比较每一个 cacheline 是否标记位有效以及 CT 是否相等。如果命中就直接返回想要的的数据，如果不命中，就依次去 L2,L3,主存判断是否命中，当命中时，将数据传给 CPU 同时更新各级 cache 的 cacheline（如果 cache 已满则要采用换入换出策略）。

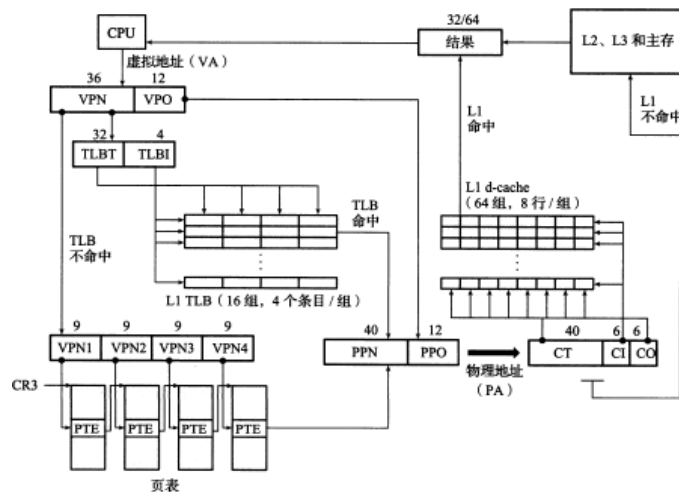


图 26 3级Cache

7.6 hello 进程 fork 时的内存映射

当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的 PID，同时为这个新进程创建虚拟内存。

它创建了当前进程的 mm_struct、区域结构和页表的原样副本。它将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。

当 fork 在新进程中返回时，新进程现在的虚拟内存刚好和调用 fork 时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面。因此，也就为每个进程保持了私有空间地址的抽象概念。

7.7 hello 进程 execve 时的内存映射

- 1) 在 bash 中的进程中执行了如下的 execve 调用：`execve("hello", NULL, NULL);`
- 2) 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：

- 3) 删除已存在的用户区域。
- 4) 映射私有区域
- 5) 映射共享区域
- 6) 设置程序计数器（PC）

execve 做的最后一件事是设置当前进程的上下文中的程序计数器，是指指向代码区域的入口点。而下一次调度这个进程时，他将从这个入口点开始执行。Linux 将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

页面命中完全是由硬件完成的，而处理缺页是由硬件和操作系统内核协作完成的：

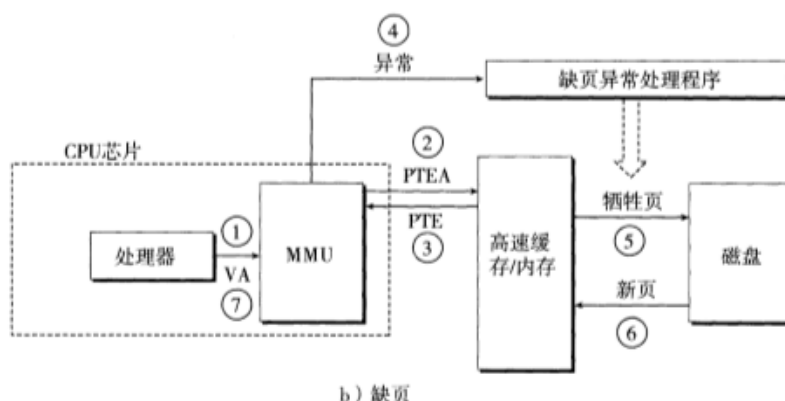


图 27 缺页中断处理

整体的处理流程：

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

动态储存分配管理使用动态内存分配器来进行。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可以用来分配。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的状态，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。动态内存分配主要有两种基本方法与策略：

带边界标签的隐式空闲链表分配器管理

带边界标记的隐式空闲链表的每个块是由一个字的头部、有效载荷、可能的额外填充以及一个字的尾部组成的。

隐式空闲链表：在隐式空闲链表中，因为空闲块是通过头部中的大小字段隐接地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。其中，一个设置了已分配的位而大小为零的终止头部将作为特殊标记的结束块。

当一个应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。分配器有三种放置策略：首次适配、下一次适配合最佳适配。分配完后可以分割空闲块减少内部碎片。同时分配器在面对释放一个已分配块时，可以合并空闲块，其中便利用隐式空闲链表的边界标记来进行合并。

显式空闲链表管理

显式空闲链表是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。如，堆可以组织成一个双向链表，在每个空闲块中，都包含一个前驱与一个后继指针。

显式空闲链表：在显式空闲链表中。可以采用后进先出的顺序维护链表，将最新释放的块放置在链表的开始处，也可以采用按照地址顺序来维护链表，其中链表中每个块的地址都小于它的后继地址，在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。

7.10 本章小结

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

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

设备的模型化

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）

设备管理

unix io 接口

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。

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 IO 接口：

打开文件：内核返回一个非负整数的文件描述符，通过对此文件描述符对文件进行所有操作。

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（文件描述符 0）、标准输出（描述符为 1），标准出错（描述符为 2）。头文件 `<unistd.h>` 定义了常量 `STDIN_FILENO`、`STDOUT_FILENO`、`STDERR_FILENO`，他们可用来代替显式的描述符值。

改变当前的文件位置，文件开始位置为文件偏移量，应用程序通过 `seek` 操作，可设置文件的当前位置为 `k`。

读写文件，读操作：从文件复制 `n` 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`；写操作：从内存复制 `n` 个字节到文件，当前文件位置为 `k`，然后更新 `k`

关闭文件：当应用完成对文件的访问后，通知内核关闭这个文件。内核会释放文件打开时创建的数据结构，将描述符恢复到描述符池中

Unix IO 函数：

1. `open()` 函数

功能描述：用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

函数原型：`int open(const char *pathname, int flags, int perms)`

参数：`pathname`: 被打开的文件名（可包括路径名如 `"dev/ttyS0"`）`flags`: 文件打开方式，

返回值：成功：返回文件描述符；失败：返回 -1

2. `close()` 函数

功能描述：用于关闭一个被打开的文件

所需头文件：`#include <unistd.h>`

函数原型: `int close(int fd)`

参数: `fd` 文件描述符

函数返回值: 0 成功, -1 出错

3. `read()` 函数

功能描述: 从文件读取数据。

所需头文件: `#include <unistd.h>`

函数原型: `ssize_t read(int fd, void *buf, size_t count);`

参数: `fd`: 将要读取数据的文件描述词。`buf`: 指缓冲区, 即读取的数据会被放到这个缓冲区中去。`count`: 表示调用一次 `read` 操作, 应该读多少数量的字符。

返回值: 返回所读取的字节数; 0 (读到 EOF); -1 (出错)。

4. `write()` 函数

功能描述: 向文件写入数据。

所需头文件: `#include <unistd.h>`

函数原型: `ssize_t write(int fd, void *buf, size_t count);`

返回值: 写入文件的字节数 (成功); -1 (出错)

5. `lseek()` 函数

功能描述: 用于在指定的文件描述符中将文件指针定位到相应位置。

所需头文件: `#include <unistd.h>`, `#include <sys/types.h>`

函数原型: `off_t lseek(int fd, off_t offset, int whence);`

参数: `fd`: 文件描述符。`offset`: 偏移量, 每一个读写操作所需要移动的距离, 单位是字节, 可正可负 (向前移, 向后移)

返回值: 成功: 返回当前位移; 失败: 返回 -1

8.3 `printf` 的实现分析

`printf` 函数:

```
int printf(const char *fmt, ...)
{
    int i;
    va_list arg = (va_list)((char *)&fmt + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
    return i;
}
```

所引用的 `vsprintf` 函数

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char *p;
    char tmp[256];
    va_listp_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 函数将所有的参数内容格式化之后存入 buf，然后返回格式化数组的长度。write 函数将 buf 中的 i 个元素写到终端。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。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

getchar 有一个 int 型的返回值。当程序调用 getchar 时，程序就等着用户按键，用户输入的字符被存放在键盘缓冲区中直到用户按回车为止(回车字符也放在缓冲区中)。

当用户键入回车之后，getchar 才开始从 stdio 流中每次读入一个字符。getchar 函数的返回值是用户输入的第一个字符的 ascii 码,如出错返回-1,且将用户输入的字符回显到屏幕。如用户在按回车之前输入了不止一个字符,其他字符会保留在键盘缓存区中,等待后续 getchar 调用读取。也就是说,后续的 getchar 调用不会等待用

户按键,而直接读取缓冲区中的字符,直到缓冲区中的字符读完为后,才等待用户按键。

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` 等调用 `read` 系统函数，通过系统调用读取按键 `ascii` 码，直到接受到回车键才返回。

8.5 本章小结

本章介绍了 Linux 的 I/O 设备的基本概念和管理方法，以及 Unix I/O 接口及其函数。最后分析了 `printf` 函数和 `getchar` 函数的工作过程。

(第 8 章 1 分)

结 论

Hello 的一生可谓变化多端：

1. `hello.c` 经过预编译，拓展得到 `hello.i` 文本文件
2. `hello.i` 经过编译，得到汇编代码 `hello.s` 汇编文件
3. `hello.s` 经过汇编，得到二进制可重定位目标文件 `hello.o`
4. `hello.o` 经过链接，生成了可执行文件 `hello`
5. `bash` 进程调用 `fork` 函数，生成子进程；并由 `execve` 函数加载运行当前进程的上下文中加载并运行新程序 `hello`
6. `hello` 的变化过程中，会有各种地址，但最终我们真正期待的是 PA 物理地址。
7. `hello` 再运行时调用一些函数，比如 `printf` 函数，这些函数与 `linux` I/O 的设备模拟化密切相关
8. `hello` 最终被 `shell` 父进程回收，内核会收回为其创建的所有信息

CSAPP 贵为计算机基础书籍顶级之作，介绍了计算机系统的基本概念，包括最底层的内存中的数据表示、流水线指令的构成、虚拟存储器、编译系统、动态加载库，以及用户应用等。书中提供了大量实际操作，可以帮助读者更好地理解程序执行的方式，改进程序的执行效率。此书以程序员的视角全面讲解了计算机系统，深入浅出地介绍了处理器、编译器、操作系统和网络环境。不愧是这一领域的权威之作！

我们 HITers 要成为优秀的程序员、工程师，而不是码农。我们基于实践在学习计算机，然而却也要基于理论；我们不应该只盯着顶层的实现，而忽视底层的构造，A programmer's perspective 正是对我们本科的最好概况——远见、修养要更重要！

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附 件

文件的作用	文件名
预处理后的文件	hello.i
编译之后的汇编文件	hello.s
汇编之后的可重定位目标文件	hello.o
链接之后的可执行目标文件	Hello
Hello.o 的 ELF 格式	elf.txt
Hello.o 的反汇编代码	Disas_hello.s
hello 的 ELF 格式	hello1.elf
hello 的反汇编代码	hello1_objdump.s

列出所有的中间产物的文件名，并予以说明起作用。

(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>.
- [7] 《深入理解计算机系统》 Randal E.Bryant David R.O' Hallaron 机械工业出版社
- [8] 博客园 printf 函数实现的深入剖析
- [9] CSDN 博客 Ubuntu 系统预处理、编译、汇编、链接指令
- [10] 博客园 从汇编层面看函数调用的实现原理
- [11] CSDN 博客 ELF 可重定位目标文件格式
- [12] 博客园 shell 命令执行过程
- [13] 《步步惊芯——软核处理器内部设计分析》 TLB 的作用及工作过程
- [14] 博客园 [转]printf 函数实现的深入剖析

(参考文献 0 分, 缺失 -1 分)