

哈爾濱工業大學

計算機系統

大作業

題 目 程序人生-Hello's P2P

專 業 計算機科學與技術

學 號 1170300603

班 級 1703006

學 生 聞永言

指 導 教 師 吳銳

計算機科學與技術學院

2018 年 12 月

摘 要

本文根据 `hello` 的独白，主要根据 CSAPP 的介绍顺序，系统的阐述了 `hello` 从键盘输入到屏幕显示结果的完整过程。分析了 `hello` 的预处理、编译原理、汇编过程、链接过程、Linux 操作系统的进程管理、虚拟内存和物理内存控制以及 I/O 管理等关键内容，系统地阐明了 `hello` 的完整的运行机制，并探讨了部分技术细节和原理。

关键词：Linux；预处理；编译；汇编；链接；重定位；进程；虚拟内存；物理内存；`shell`；缺页；信号；I/O；页表；

目 录

第 1 章 概述	- 4 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 4 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 6 -
2.3 HELLO 的预处理结果解析	- 6 -
2.4 本章小结	- 6 -
第 3 章 编译	- 7 -
3.1 编译的概念与作用	- 7 -
3.2 在 UBUNTU 下编译的命令	- 7 -
3.3 HELLO 的编译结果解析	- 7 -
3.4 本章小结	- 9 -
第 4 章 汇编	- 10 -
4.1 汇编的概念与作用	- 10 -
4.2 在 UBUNTU 下汇编的命令	- 10 -
4.3 可重定位目标 ELF 格式	- 10 -
4.4 HELLO.o 的结果解析	- 13 -
4.5 本章小结	- 16 -
第 5 章 链接	- 17 -
5.1 链接的概念与作用	- 17 -
5.2 在 UBUNTU 下链接的命令	- 17 -
5.3 可执行目标文件 HELLO 的格式	- 17 -
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	- 20 -
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	27 -
第 7 章 HELLO 的存储管理	30 -
7.1 HELLO 的存储器地址空间	30 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	30 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	30 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	32 -
7.5 三级 CACHE 支持下的物理内存访问	32 -
7.6 HELLO 进程 FORK 时的内存映射	33 -
7.7 HELLO 进程 EXECVE 时的内存映射	33 -
7.8 缺页故障与缺页中断处理	34 -
7.9 动态存储分配管理	35 -
7.10 本章小结	35 -
第 8 章 HELLO 的 IO 管理	37 -
8.1 LINUX 的 IO 设备管理方法	37 -
8.2 简述 UNIX IO 接口及其函数	37 -
8.3 PRINTF 的实现分析	38 -
8.4 GETCHAR 的实现分析	38 -
8.5 本章小结	38 -
结论	39 -
附件	41 -
参考文献	42 -

第 1 章 概述

1.1 Hello 简介

输入 `hello.c` 的源代码，保存到硬盘中的 `hello.c`，并编译运行。开始进行预处理、编译、汇编、链接，最终生成可执行文件，并开始运行。在壳（`Bash`）里，OS（进程管理）进行 `fork`（`Process`）和 `execve`、`mmap` 等操作控制进程，分配时间片，使得该程序在硬件（`CPU/RAM/IO`）上执行（取指译码执行/流水线等）。通过 OS（存储管理）与 MMU 翻译 VA 到 PA 并通过 TLB、4 级页表、3 级 Cache，Pagefile 等软硬件结合加速程序的执行；IO 管理与信号处理协同管理进程，最终产生输出，翻译为 RGB 信号显示到屏幕上。最终执行完毕后，操作系统回收该进程，完成整个操作。

1.2 环境与工具

硬件：CPU：Core i7-7920HQ

RAM：16GB

操作系统：macOS Mojave 10.14.2

Ubuntu 18.04 LTS

开发环境：Vmware Fushion Pro 11.0.1，CLion 2018.3.2

调试工具：GDB，EDB debugger，Hopper Disassembler v4

1.3 中间结果

中间结果文件名	文件作用
hello.i	预处理得到的中间结果
hello.s	hello.i 编译后得到的汇编语言文本文件
Hello_o.o	hello.s 汇编后得到的可重定位目标文件
a.out	链接后得到的可执行目标文件
hello.o	hello_o.o 链接后得到的可重定位目标文件
hello.objdump	hello 的反汇编代码
hello.elf	hello 的 elf 格式文件

1.4 本章小结

介绍了 hello 的运行过程以及 hello 运行的硬件环境以及软件环境，生成的中间结果文件。

第 2 章 预处理

2.1 预处理的观念与作用

预处理的观念：在编译之前进行的处理。

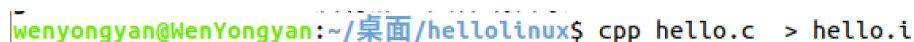
预处理器（cpp）根据以字符#开头的命令，修改原始的 C 程序。结果得到另一个 C 程序，通常以 .i 作为文件拓展名。

预处理主要有三个方面的内容： 1.宏定义；2.文件包含；3.条件编译。

预处理命令以符号“#”开头。

使用条件编译可以使目标程序变小，运行时间变短。同时有利于代码的模块化。

2.2 在 Ubuntu 下预处理的命令



```
wenyongyan@WenYongyan:~/桌面/hellolinux$ cpp hello.c > hello.i
```

图 2.1 预处理命令

2.3 Hello 的预处理结果解析

将头文件 stdio.h，unistd.h，stdlib.h 导入，并进行宏替换。

2.4 本章小结

本阶段完成了对 hello.c 的预处理工作。使用 Ubuntu 下的预处理指令可以将其转换为.i 文件。完成该阶段转换后，可以进行下一阶段的汇编处理。

第 3 章 编译

3.1 编译的概念与作用

编译是利用编译程序从源语言编写的源程序产生目标程序的过程。编译程序把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误，给出提示信息。

3.2 在 Ubuntu 下编译的命令

```
wenyongyan@WenYongyan:~/桌面/hellolinux$ gcc -v hello.i
```

图 3.1 生成 hello.i

```
wenyongyan@WenYongyan:~/桌面/hellolinux$ gcc -S hello.i -o hello.s
```

图 3.2 生成 hello.s

3.3 Hello 的编译结果解析

3.3.1 全局变量 int sleepsecs

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

图 3.3 全局变量

储存在数据段中

3.3.2 数据段

```
.LC0:
.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
.LC1:
.string "Hello %s %s\n"
```

图 3.4 字符串

保存待打印的字符串，如姓名，学号等信息。

3.3.3 main 函数


```

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)

```

图 3.5 主函数

首先将%rbp 压入栈中，并将%rsp 栈指针的值传给%rbp，接下来%rsp 分配栈帧，并将参数%edi, %rsi 压入栈中。

3.3.4 判断与跳转语句

```

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

```

图 3.6 跳转语句

对应源代码中的 if(argc!=3)。

当 argc 参数（栈中保存）等于 3 时跳转到.L2，开始执行源代码中的 for 循环。

```

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

```

图 3.7 跳转语句

将 0 传给栈中的参数，作为循环的初始值（i = 0）。

3.3.5 for 循环

```

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

```

图 3.8 for 循环的判断条件

很显然，判断循环的退出条件是-4(%rbp)>9，即 i<10。

不满足时进入循环，即跳转到.L4。

```

.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)

```

图 3.9 循环体

循环体如图所示。

将栈中的数值%rsi 传给寄存器%rax，并+16，将一个参数取地址传给%rdx。对于另外一个参数做同样处理，这样便将 print 需要的两个参数从栈中取出保存到了寄存器%rdi 和%rsi 当中。接下来再将 0 写入%eax，开始调用 printf。

3.3.6 函数调用

call printf 后完成字符串的输出并返回到调用前的下一条指令，继续执行循环，对于下面的 call sleep 同理，只需先将参数先传给%eax 再进行调用。

最后将计数器%rbx+1。

3.3.7 函数的返回

完成循环或判断语句后，调用了 getchar，并将 0 写入%eax 作为主函数 main 的返回值，最后 ret 返回，结束。

3.4 本章小结

在本章中，了解了汇编语言这一更接近底层的形式，而产生汇编代码则需要进行编译过程。

编译是利用编译程序从源语言编写的源程序产生目标程序的过程或用编译程序产生目标程序的动作。编译可以把高级语言变成计算机可以识别的二进制语言。编译程序把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误，给出提示信息。

第 4 章 汇编

4.1 汇编的概念与作用

概念：

将汇编代码（hello.s）翻译为机器语言的二进制目标代码。目标代码是机器代码的一种形式，它包含所有指令的二进制表示，但是还没有填入全局值的地址。

作用：

将汇编代码转化成二进制目标代码文件。

4.2 在 Ubuntu 下汇编的命令

```
wenyongyan@WenYongyan:/mnt/hgfs/大作业$ as hello.s -o hello.o
```

图 4.1 汇编命令

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。

```
wenyongyan@WenYongyan:/mnt/hgfs/大作业$ readelf -a hello.o
ELF 头:
  Magic:      7f 45 4c 46 02 01 01 00 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:               1152 (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
```

图 4.2 文件头

节头:

[号]	名称 大小	类型 全体大小	地址 旗标	链接	偏移量 信息	对齐
[0]	0000000000000000	NULL	0000000000000000		00000000	
	0000000000000000			0	0	0
[1]	.text	PROGBITS	0000000000000000		00000040	
	0000000000000081	0000000000000000	AX	0	0	1
[2]	.rela.text	RELA	0000000000000000		00000340	
	00000000000000c0	0000000000000018	I	10	1	8
[3]	.data	PROGBITS	0000000000000000		000000c4	
	0000000000000004	0000000000000000	WA	0	0	4
[4]	.bss	NOBITS	0000000000000000		000000c8	
	0000000000000000	0000000000000000	WA	0	0	1
[5]	.rodata	PROGBITS	0000000000000000		000000c8	
	0000000000000032	0000000000000000	A	0	0	8
[6]	.comment	PROGBITS	0000000000000000		000000fa	
	0000000000000024	0000000000000001	MS	0	0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000		0000011e	
	0000000000000000	0000000000000000		0	0	1
[8]	.eh_frame	PROGBITS	0000000000000000		00000120	
	0000000000000038	0000000000000000	A	0	0	8
[9]	.rela.eh_frame	RELA	0000000000000000		00000400	
	0000000000000018	0000000000000018	I	10	8	8
[10]	.symtab	SYMTAB	0000000000000000		00000158	
	0000000000000198	0000000000000018		11	9	8
[11]	.strtab	STRTAB	0000000000000000		000002f0	
	000000000000004d	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)

There are no section groups in this file.

本文件中没有程序头。

There is no dynamic section in this file.

图 4.3 节头部分

重定位节 '.rela.text' at offset 0x340 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
00000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4
000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4
000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 21
00000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4
000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4
000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

重定位节 '.rela.eh_frame' at offset 0x400 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not currently supported.

Symbol table '.symtab' contains 17 entries:

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

No version information found in this file.

图 4.4 重定位节部分

其组成如下:

ELF Header: 以 16B 的序列 Magic 开始, Magic 描述了生成该文件的系统的大小和字节顺序, ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息,其中包括 ELF 头的大小、目标文件的类型、机器类型、字节头部表(section header table)的文件偏移,以及节头部表中条目的大小和数量等信息。

Section Headers: 节头部表,包含了文件中出现的各个节的语义,包括节的类型、位置和大小等信息。

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 4.4,图中 8 条重定位信息分别是对.L0(第一个 printf 中的字符串)、puts 函数、exit 函数、.L1(第二个 printf 中的字符串)、printf 函数、sleepsecs、sleep 函数、getchar 函数进行重定位声明。

offset	需要进行重定向的代码在.text 或.data 节中的偏移位置,8 个字节。
--------	--

Info	包括 symbol 和 type 两部分,其中 symbol 占前 4 个字节, type 占后 4 个字节, symbol 代表重定位到的目标在 .symtab 中的偏移量, type 代表重定位的类型
Addend	计算重定位位置的辅助信息, 共占 8 个字节
Type	重定位到的目标的类型
Name	重定向到的目标的名称

.rela.eh_frame : eh_frame 节的重定位信息。

.symtab: 符号表, 用来存放程序中定义和引用的函数和全局变量的信息。重定位需要引用的符号都在其中声明。

可以观察到, 在文件头中得到节头表的信息, 然后再使用节头表中的字节偏移信息得到各节在文件中的起始位置, 以及各节所占空间的大小; 同时代码段是可执行的, 但是不能写; 数据段和只读数据段都不可执行, 而且只读数据段也不可写。

4.4 Hello.o 的结果解析

hello.o: 文件格式 elf64-x86-64

Disassembly of section .text:

```
0000000000000000 <main>:
 0: 55                push    %rbp
 1: 48 89 e5          mov     %rsp,%rbp
 4: 48 83 ec 20       sub     $0x20,%rsp
 8: 89 7d ec          mov     %edi,-0x14(%rbp)
 b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
 f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16            je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # 1c <main+0x1c>
1c: e8 00 00 00 00    callq   21 <main+0x21>
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq   2b <main+0x2b>
2b: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
43: 48 83 c0 08       add     $0x8,%rax
47: 48 8b 00          mov     (%rax),%rax
4a: 48 89 c6          mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # 54 <main+0x54>
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq   5e <main+0x5e>
5e: 8b 05 00 00 00 00    mov     0x0(%rip),%eax      # 64 <main+0x64>
64: 89 c7            mov     %eax,%edi
66: e8 00 00 00 00    callq   6b <main+0x6b>
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>
7a: b8 00 00 00 00    mov     $0x0,%eax
7f: c9              leaveq  %rax,%rsi
80: c3              retq    ..
```

图 4.5 hello.o 的反汇编代码

以上为 hello.o 的反汇编代码，可以发现已经进行了重定位，跳转的位置已经有了偏移量。与下面的 hello.s 进行对比，还可以发现一些差别：跳转指令跳转位置已不再使用标记，而是变成了确定的地址；在开始位置，汇编代码分配了 0x32 字节栈帧，反汇编代码分配了 0x20 字节，造成其储存的数值位置也发生了变化；此外，数字格式发生了变化，如 4 改为 0x4 的十六进制表示；callq 每次返回下一条指令的相对地址。

```

19  main: --
20  .LFB6:
21      .cfi_startproc
22      pushq   %rbp
23      .cfi_def_cfa_offset 16
24      .cfi_offset 6, -16
25      movq    %rsp, %rbp
26      .cfi_def_cfa_register 6
27      subq    $32, %rsp
28      movl    %edi, -20(%rbp)
29      movq    %rsi, -32(%rbp)
30      cmpl    $3, -20(%rbp)
31      je      .L2
32      leaq    .LC0(%rip), %rdi
33      call    puts@PLT
34      movl    $1, %edi
35      call    exit@PLT
36  .L2:
37      movl    $0, -4(%rbp)
38      jmp     .L3
39  .L4:
40      movq    -32(%rbp), %rax
41      addq    $16, %rax
42      movq    (%rax), %rdx
43      movq    -32(%rbp), %rax
44      addq    $8, %rax
45      movq    (%rax), %rax
46      movq    %rax, %rsi
47      leaq    .LC1(%rip), %rdi
48      movl    $0, %eax
49      call    printf@PLT
50      movl    sleepsecs(%rip), %eax
51      movl    %eax, %edi
52      call    sleep@PLT
53      addl    $1, -4(%rbp)
54  .L3:
55      cmpl    $9, -4(%rbp)
56      jle     .L4
57      call    getchar@PLT
58      movl    $0, %eax
59      leave
60      .cfi_def_cfa 7, 8
61      ret
62      .cfi_endproc

```

图 4.6 汇编代码

机器语言只由 0 和 1 构成，计算机能直接识别和执行。一条指令就是机器语言的一个语句，它是一组有意义的二进制代码，指令的基本格式如，操作码字段和地址码字段，其中操作码指明了指令的操作性质及功能，地址码则给出了操作数或操作数的地址。

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。

4.5 本章小结

在本章中，窥见到了 C 语言提供的抽象层面以下的内容，以进一步了解机器级编程。通过让编译器产生机器级程序的汇编代码，我们可以更好地理解编译器的工作内容与细节并且能够更好地了解程序如何将数据存储在不同的内存区域中。

机器级程序和他们的汇编代码表示，与 C 程序的差别很大。各种数据类型之间的差别很小。程序是以指令序列来表示的，每一条指令都表示一条单独的操作。部分程序状态，如寄存器和运行时栈，对程序员来说是直接可见的。编译器为了完成复杂的控制流程，必须使用多条指令来产生和操作各种数据结构，以及像条件、循环和过程这样的控制结构。通过了解汇编语言，我们可以更好地分析程序的行为，对各种异常进行更加深入的、底层的分析。

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）到内存中并执行。

作用：将分别在不同的目标文件中编译或汇编的代码收集到一个可直接执行的文件中。它还连接目标程序和用于标准库函数的代码，以及连接目标程序和由计算机的操作系统提供的资源（例如，存储分配程序及输入与输出设备）。

5.2 在 Ubuntu 下链接的命令

```
wenyongyan@WenYongyan:/mnt/hgfs/大作业$ 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 5.1 链接命令

5.3 可执行目标文件 hello 的格式

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	00000000
[1]	.interp	PROGBITS	0000000000400270	00000270
[2]	.note.ABI-tag	NOTE	000000000040028c	0000028c
[3]	.hash	HASH	00000000004002b0	000002b0
[4]	.gnu.hash	GNU_HASH	00000000004002e8	000002e8
[5]	.dynsym	DYNSYM	0000000000400308	00000308
[6]	.dynstr	STRTAB	00000000004003c8	000003c8
[7]	.gnu.version	VERSYM	0000000000400420	00000420
[8]	.gnu.version_r	VERNEED	0000000000400430	00000430
[9]	.rela.dyn	RELA	0000000000400450	00000450
[10]	.rela.plt	RELA	0000000000400480	00000480
[11]	.init	PROGBITS	0000000000401000	00001000
[12]	.plt	PROGBITS	0000000000401020	00001020
[13]	.text	PROGBITS	0000000000401080	00001080
[14]	.fini	PROGBITS	00000000004011a4	000011a4
[15]	.rodata	PROGBITS	0000000000402000	00002000
[16]	.eh_frame	PROGBITS	0000000000402040	00002040
[17]	.dynamic	DYNAMIC	0000000000403e50	00002e50
[18]	.got	PROGBITS	0000000000403ff0	00002ff0
[19]	.got.plt	PROGBITS	0000000000404000	00003000
[20]	.data	PROGBITS	0000000000404040	00003040
[21]	.comment	PROGBITS	0000000000000000	00003048
[22]	.symtab	SYMTAB	0000000000000000	00003070
[23]	.strtab	STRTAB	0000000000000000	00003508
[24]	.shstrtab	STRTAB	0000000000000000	00003658

图 5.2 链接后的节头

在 ELF 格式文件中，Section Headers 对 hello 中所有的节信息进行了声明，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

信息我们就可以用 HexEdit 定位各个节所占的区间(起始位置,大小)。其中 Address 是程序被载入到虚拟地址的起始地址。

5.4 hello 的虚拟地址空间

程序头:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040	0x000000000000400040	0x0000000000400040
	0x0000000000000230	0x0000000000000230	R 0x8
INTERP	0x0000000000000270	0x000000000000400270	0x0000000000400270
	0x000000000000001c	0x000000000000001c	R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x000000000000400000	0x0000000000400000
	0x00000000000004f8	0x00000000000004f8	R 0x1000
LOAD	0x0000000000000100	0x000000000000401000	0x0000000000401000
	0x00000000000001ad	0x00000000000001ad	R E 0x1000
LOAD	0x0000000000000200	0x000000000000402000	0x0000000000402000
	0x000000000000013c	0x000000000000013c	R 0x1000
LOAD	0x00000000000002e50	0x000000000000403e50	0x0000000000403e50
	0x00000000000001f8	0x00000000000001f8	RW 0x1000
DYNAMIC	0x00000000000002e50	0x000000000000403e50	0x0000000000403e50
	0x00000000000001a0	0x00000000000001a0	RW 0x8
NOTE	0x000000000000028c	0x00000000000040028c	0x000000000040028c
	0x0000000000000020	0x0000000000000020	R 0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x00000000000002e50	0x000000000000403e50	0x0000000000403e50
	0x00000000000001b0	0x00000000000001b0	R 0x1

图 5.3 程序头部分

在 0x400000~0x401000 段中, 程序被载入, 自虚拟地址 0x400000 开始, 自 0x400fff 结束, 这之间每个节(开始 ~ .eh_frame 节)的排列即开始结束同图 5.2 中 Address 中声明。

如图, 查看 ELF 格式文件中的 Program Headers 信息, 程序头表在执行时被使用, 它指出链接器运行时加载的内容并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。在下面可以看出, 程序共包含 8 个段:

PHDR: 保存程序头表。

INTERP: 指定在程序已经从可执行文件映射到内存之后, 必须调用的解释器(如动态链接器)。

LOAD 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量数据(如字符串)、程序的目标代码等。

DYNAMIC 保存了由动态链接器使用的信息。

NOTE 保存辅助信息。

GNU_STACK: 权限标志, 标志栈是否是可执行的。

GNU_RELRO: 指定在重定位结束之后那些内存区域是需要设置只读。

5.5 链接的重定位过程分析

链接后, `hello.o` 的反汇编文件中包含了如下节。

节名称	描述
<code>.interp</code>	保存 <code>ld.so</code> 的路径
<code>.note.ABI-tag</code>	Linux 下特有的 section
<code>.hash</code>	符号的哈希表
<code>.gnu.hash</code>	GNU 拓展的符号的 hash 表
<code>.dynsym</code>	运行时/动态符号表
<code>.dynstr</code>	存放 <code>.dynsym</code> 节中的符号名称
<code>.gnu.version</code>	符号版本
<code>.gnu.version_r</code>	符号引用版本
<code>.rela.dyn</code>	运行时/动态重定位表
<code>.rela.plt</code>	<code>.plt</code> 节的重定位条目
<code>.init</code>	程序初始化需要执行的代码
<code>.plt</code>	动态链接-过程链接表
<code>.fini</code>	当程序正常终止时需要执行的代码
<code>.eh_frame</code>	包含异常展开和源语言信息
<code>.dynamic</code>	存放被 <code>ld.so</code> 使用的动态链接信息
<code>.got</code>	动态链接-全局偏移量表-存放变量
<code>.got.plt</code>	动态链接-全局偏移量表-存放函数
<code>.data</code>	初始化了的数据
<code>.comment</code>	一串包含编译器的 NULL-terminated 字符串

与原始的 `hello.o.o` 反汇编文件信息进行对比, 有如下不同:

- [1] 函数个数: 在使用 `ld` 命令链接的时候, 指定了动态链接器为 64 的 `/lib64/ld-linux-x86-64.so.2`, `crt1.o`、`crti.o`、`crtm.o` 中主要定义了程序入口 `_start`、初始化函数 `_init`, `_start` 程序调用 `hello.c` 中的 `main` 函数, `libc.so` 是动态链接共享库, 其中定义了 `hello.c` 中用到的 `printf`、`sleep`、`getchar`、

exit 函数和 `_start` 中调用的 `__libc_csu_init` , `__libc_csu_fini` , `__libc_start_main`。链接器将上述函数加入。

[2] 函数调用：链接器解析重定条目时发现对外部函数调用的类型为 `R_X86_64_PLT32` 的重定位，此时动态链接库中的函数已经加入到了 PLT 中，`.text` 与 `.plt` 节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为 PLT 中相应函数与下条指令的相对地址，指向对应函数。对于此类重定位链接器为其构造 `.plt` 与 `.got.plt`。

[3] `.rodata` 引用：链接器解析重定条目时发现两个类型为 `R_X86_64_PC32` 的对 `.rodata` 的重定位（`printf` 中的两个字符串），`.rodata` 与 `.text` 节之间的相对距离确定，因此链接器直接修改 `call` 之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。对于其他的 `.rodata` 引用，函数调用原理类似。

5.6 hello 的执行流程

函数名称	地址
ld-2.27.so!_dl_start	0x7fce 8cc38ea0
ld-2.27.so!_dl_init	0x7fce 8cc47630
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

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。虽然动态链接把链接过程推迟到了程序运行时，但是在形成可执行文件时（注意形成可执行文件和执行程序是两个概念），还是需要用到动态链接库。比如我们在形成可执行程序时，发现引用了一个外部的函数，此时会检查动态链接库，发现这个函数名是一个动态链接符号，此时可执行程序就不对这个符号进行重定位，而把这个过程留到装载时再进行。

在 `dl_init` 调用之前，对于每一条 PIC 函数调用，调用的目标地址都实际指向 PLT 中的代码逻辑，GOT 存放的是 PLT 中函数调用指令的下一条指令地址。如图所示。

[illegible]

0x601008 和 0x601010 处的两个 8B 数据分别发生改变为 0x7fd9 d3925170 和 0x7fd9 d3713680。

– 22 –

5.8 本章小结

链接可以在编译时由静态编译器来完成，也可以在加载时和运行时由动态链接器来完成。链接器处理称为目标文件的二进制文件，它有 3 种不同的形式：可重定位的、可执行的和共享的。可重定位的目标文件由静态链接器合并成一个可执行的目标文件，它可以加载到内存中并执行。共享目标文件（共享库）是在运行时由动态链接器链接和加载的，或者隐含地在调用程序被加载和开始执行时，或者根据需要在程序调用 `dlopen` 库的函数时。

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

静态链接器是由像 GCC 这样的编译驱动程序调用的。它们将多个可重定位目标文件合并成一个单独的可执行目标文件。多个目标文件可以定义相同的符号，而链接器用来悄悄地解析这些多重定义的规则可能在用户程序中引入微妙的错误。

多个目标文件可以被连接到一个单独的静态库中。链接器用库来解析其他目标模块中的符号引用。许多链接器通过从左到右的顺序扫描来解析符号引用，这是另一个引起令人迷惑的链接时错误的来源。

加载器将可执行文件的内容映射到内存，并运行这个程序。链接器还可能生成部分链接的可执行目标文件，这样的文件中有对定义在共享库中的例程和数据的未解析的引用。在加载时，加载器将部分链接的可执行文件映射到内存，然后调用动态链接器，它通过加载共享库和重定位程序中的引用来完成链接任务。

被编译为位置无关代码的共享库可以加载到任何地方，也可以在运行时被多个进程共享。为了加载、链接和访问共享库的函数和数据，应用程序也可以在运行时使用动态链接器。

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，上下文是由正确运行的所需的状态组成的。

进程的作用：

提供一个独立的逻辑控制流：它提供一个假象，好像我们的程序独占地使用处理器；

提供一个私有的地址空间：它提供一个假象，好像我们的程序独占地使用内存。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：

shell 是用户和 Linux 内核之间的接口程序。在提示符下输入的每个命令都由 shell 先解释然后传给 Linux 内核。shell 是一个命令语言解释器，拥有自己内建的 shell 命令集。此外，shell 也能被系统中其他有效的 Linux 实用程序和应用程序所调用。

处理流程：

首先 shell 检查命令是否是内部命令，若不是再检查是否是一个应用程序（这里的应用程序可以是 Linux 本身的实用程序，如 ls 和 rm；也可以是购买的商业程序，如 xv；或者是自由软件，如 emacs）。然后 shell 在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果输入的命令不是一个内部命令且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能找到命令，该内部命令或应用程序分解后将被系统调用并传给 Linux 内核。

6.3 Hello 的 fork 进程创建过程

父进程通过 fork() 创建一个新的运行的子进程。新创建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程

任何打开文件描述符相同的副本，这就意味着当父进程调用 `fork()` 时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大的区别在于它们有不同的 PID。

`fork` 函数只被调用一次，却会返回两次，一次是在调用进程（父进程）中，一次是在新创建的子进程中。在父进程中，`fork` 返回子进程的 PID，在子进程中，`fork` 返回 0。因为子进程的 PID 总是为非 0，返回值就提供一个明确的方法来分辨程序是在父进程中还是在子进程中运行。

6.4 Hello 的 `execve` 过程

`execve` 函数在当前进程的上下文中加载并运行一个新程序。首先函数加载并运行可执行目标文件 `filename`，且带参数列表 `argv` 和环境变量列表 `envp`。只有当出现错误时，例如找不到 `filename`，`execve` 才会返回到调用程序。所以 `execve` 调用一次并且从不返回。

在 `execve` 加载了 `filename` 之后，他调用启动代码，启动代码设置栈，并将控制传递给新程序的主函数，该主函数有如下形式：

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

当 `main` 开始执行时用户栈的组织结构

6.5 Hello 的进程执行

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。

新进程的创建，首先在内存中为新进程创建一个 `task_struct` 结构，然后将父进程的 `task_struct` 内容复制其中，再修改部分数据。分配新的内核堆栈、新的 PID、再将 `task_struct` 这个 `node` 添加到链表中。然后将可执行文件装入内核的 `linux_binprm` 结构体。进程调用 `execve` 时，该进程执行的程序完全被替换，新的程序从 `main` 函数开始执行。调用 `execve` 并不创建新进程，只是替换了当前进程的代码区、数据区、堆和栈。在进程调用了 `exit` 之后，该进程并非马上就消失掉，而是留下了一个成为僵尸进程的数据结构，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。

为了控制进程的执行，内核必须有能力挂起正在 CPU 上执行的进程，并恢复以前挂起的某个进程的执行，这叫做进程切换。进程上下文切换由以下 4 个步骤组成：

(1) 决定是否作上下文切换以及是否允许作上下文切换。包括对进程调度原因的检查分析, 以及当前执行进程的资格和 CPU 执行方式的检查等。在操作系统中, 上下文切换程序并不是每时每刻都在检查和分析是否可作上下文切换, 它们设置有适当的时机。

(2) 保存当前执行进程的上下文。这里所说的当前执行进程, 实际上是指调用上下文切换程序之前的执行进程。如果上下文切换不是被那个当前执行进程所调用, 且不属于该进程, 则所保存的上下文应是先前执行进程的上下文, 或称为“老”进程上下文。显然, 上下文切换程序不能破坏“老”进程的上下文结构。

(3) 使用进程调度算法, 选择一处于就绪状态的进程。

(4) 恢复或装配所选进程的上下文, 将 CPU 控制权交到所选进程手中。

6.6 hello 的异常与信号处理

hello 执行过程中会出现哪几类异常, 会产生哪些信号, 又怎么处理的。

程序运行过程中可以按键盘, 如不停乱按, 包括回车, Ctrl-Z, Ctrl-C 等, Ctrl-z 后可以运行 `ps jobs pstree fg kill` 等命令, 请分别给出各命令及运行结果截图, 说明异常与信号的处理。

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

在发生异常时会产生信号。例如缺页故障会导致 OS 发生 SIGSEGV 信号给用户进程, 而用户进程以段错误退出。常见信号种类如下表所示。

ID	名称	默认行为	相应事件
2	SIGINT	终止	来自键盘的中断
9	SIGKILL	终止	杀死程序 (该信号不能被捕获不能被忽略)
11	SIGSEGV	终止	无效的内存引用 (段故障)
14	SIGALRM	终止	来自 alarm 函数的定时器信号
17	SIGCHLD	忽略	一个子进程停止或者终止

```
wenyongyan@WenYongyan: /mnt/hgfs/大作业$ ./a.out 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
gon
```

图 6.1 正常运行状态

正常运行时

```
wenyongyan@WenYongyan: /mnt/hgfs/大作业$ ps
  PID TTY          TIME CMD
 33511 pts/0    00:00:00 bash
 35537 pts/0    00:00:00 ps
```

图 6.2 ps 命令

输入 ps

```
wenyongyan@WenYongyan: /mnt/hgfs/大作业$ ./a.out 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
^Z
[1]+  已停止                  ./a.out 1170300603 wenyongyan
```

图 6.3 ctrl+z 命令

输入 ctrl+z

```
wenyongyan@WenYongyan: /mnt/hgfs/大作业$ fg
./a.out 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
c
```

图 6.4 fg 命令

输入 fg

```

wenyongyan@WenYongyan:/mnt/hgfs/大作业$ ./a.out 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
Hello 1170300603 wenyongyan
rwwwghkuhes;nslbis;hbirhblsHello 1170300603 wenyongyan
nblebdfkbnlkrnsbnlkbjfnrsljnbsdjHello 1170300603 wenyongyan
nblsknbdskbjnjsnrsjnslnlkrstbnjrstbnHello 1170300603 wenyongyan
snblksrnbksnljkrnsbljknrljnsbkjbnHello 1170300603 wenyongyan
bnlaelbnelbneewrnlkbnlkeHello 1170300603 wenyongyan
lkndslkfn34tg3nlkng3o83fehfwefh

```

图 6.5 运行过程中乱按

运行过程中乱按

6.7 本章小结

异常控制流（ECF）发生在计算机系统的各个层次，是计算机系统中提供并发的基本机制。

在硬件层，异常是由处理器中的事件触发的控制流中的突变。控制流传递给一个软件处理程序，该处理程序进行一些处理，然后返回控制给被中断的控制流。

有四种不同类型的异常：中断、故障、终止和陷阱。当一个外部 I/O 设备（例如定时踩芯片或者磁盘控制器）设置了处理器芯片上的中断管脚时，（对于任意指令）中断会异步地发生。控制返回到故障指令后面的那条指令。一条指令的执行可能导致故障和终止同步发生。故障处理程序会重新启动故障指令，而终止处理程序从不将控制返回给被中断的流。最后，陷阱就像是用来实现向应用提供到操作系统代码的受控的入口点的系统调用的函数调用。

在操作系统层，内核用 ECF 提供进程的基本概念。进程提供给应用两个重要的抽象：

- 1) 逻辑控制流，它提供给每个程序一个假象，好像它是在独占地使用处理器；
- 2) 私有地址空间，它提供给每个程序一个假象，好像它是在独占地使用主存。

在操作系统和应用程序之间的接口处，应用程序可以创建子进程，等待它们的子进程停止或者终止，运行新的程序，以及捕获来自其他进程的信号。信号处理的语义是微妙的，并且随系统不同而不同。然而，在与 Posix 兼容的系统上存在着一些机制，允许程序清楚地指定期望的信号处理语义。

最后, 在应用层, C 程序可以使用非本地跳转来规避正常的调用 / 返回栈规则, 并且直接从一个函数分支到另一个函数。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

物理地址：加载到内存地址寄存器中的地址，内存单元的真正地址。在前端总线上传输的内存地址都是物理内存地址，编号从 0 开始一直到可用物理内存的最高端。这些数字被北桥(Northbridge chip)映射到实际的内存条上。物理地址是明确的、最终用在总线上的编号，不必转换，不必分页，也没有特权级检查(no translation, no paging, no privilege checks)。

逻辑地址：CPU 所生成的地址。逻辑地址是内部和编程使用的，并不唯一。例如，在进行 C 语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于当前进程数据段的地址（偏移地址），不和绝对物理地址相关。

虚拟地址：程序访问存储器所使用的逻辑地址称为虚拟地址，有时我们也把逻辑地址称为虚拟地址。因为和虚拟内存空间的概念类似，逻辑地址也是和实际物理内存容量无关的。逻辑地址和物理地址的“差距”是 0xC0000000，是由于虚拟地址->线性地址->物理地址映射正好差这个值。这个值是由操作系统指定的。逻辑地址（或称为虚拟地址）到线性地址是由 CPU 的段机制自动转换的。如果没有开启分页管理，则线性地址就是物理地址。如果开启了分页管理，那么系统程式需要参和线性地址到物理地址的转换过程。具体是通过设置页目录表和页表项进行的。

线性地址：逻辑地址到物理地址变换之间的中间层。程式代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

- [1] 逻辑地址 = 段选择符 + 偏移量
- [2] 每个段选择符大小为 16 位，段描述符为 8 字节（注意单位）。
- [3] GDT 为全局描述符表，LDT 为局部描述符表。
- [4] 段描述符存放在描述符表中，也就是 GDT 或 LDT 中。
- [5] 段首地址存放在段描述符中。

每个段的首地址都存放在自己的段描述符中，而所有的段描述符都存放在一个描述符表中（描述符表分为全局描述符表 GDT 和局部描述符表 LDT）。而要想找到某个段的描述符必须通过段选择符才能找到。下图是一个段选择符的格式：



图 7.1 段选择符的格式

从上图可以看出段选择符由三个部分组成，从右向左依次是 RPL、TI、index（索引）。RPL 在此不做介绍。先来看 TI，当 TI=0 时，表示段描述符在 GDT 中，当 TI=1 时表示段描述符在 LDT 中。

再来看一下 index 部分。我们可以将描述符表看成是一个数组，每个元素都存放一个段描述符，那 index 就表示某个段描述符在数组中的索引。

现在我们假设有一个段的段选择符，它的 TI=0，index=8。我们可以知道这个段的描述符是在 GDT 数组中，并且他的在数组中的索引是 8。

假设 GDT 的起始位置是 0x00020000，而一个段描述符的大小是 8 个字节，由此我们可以计算出段描述符所在的地址： $0x00020000 + 8 * \text{index}$ ，从而我们就可以找到我们想要的段描述符，从而获取某个段的首地址，然后再将从段描述符中获取到的首地址与逻辑地址的偏移量相加就得到了线性地址。

7.3 Hello 的线性地址到物理地址的变换-页式管理

CPU 的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。从管理和效率的角度出发，线性地址被分为以固定长度为单位的组，称为页(page)，例如一个 32 位的机器，线性地址最大可为 4G，可以用 4KB 为一个页来划分，这页，整个线性地址就被划分为一个 $\text{total_page}[2^{20}]$ 的大数组，共有 2 的 20 个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。另一类“页”，我们称之为物理页，或者是页框、页帧的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

36 位 VPN 被划分成四个 9 位的片，每个片被用作到一个页表的偏移量。CR3 寄存器包含 L1 页表的物理地址。VPN 1 提供到一个 L1 PTE 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN 2 提供到一个 L2 PTE 的偏移量，以此类推。

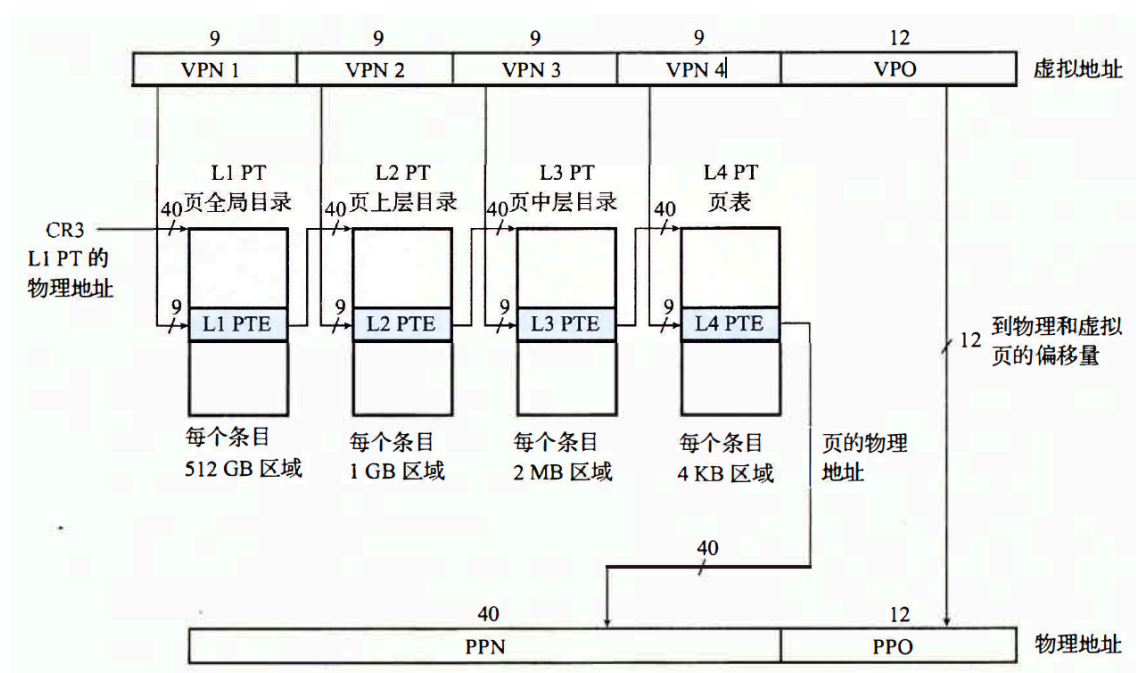


图 7.2 TLB 与四级页表支持下的 VA 到 PA 的变换

7.5 三级 Cache 支持下的物理内存访问

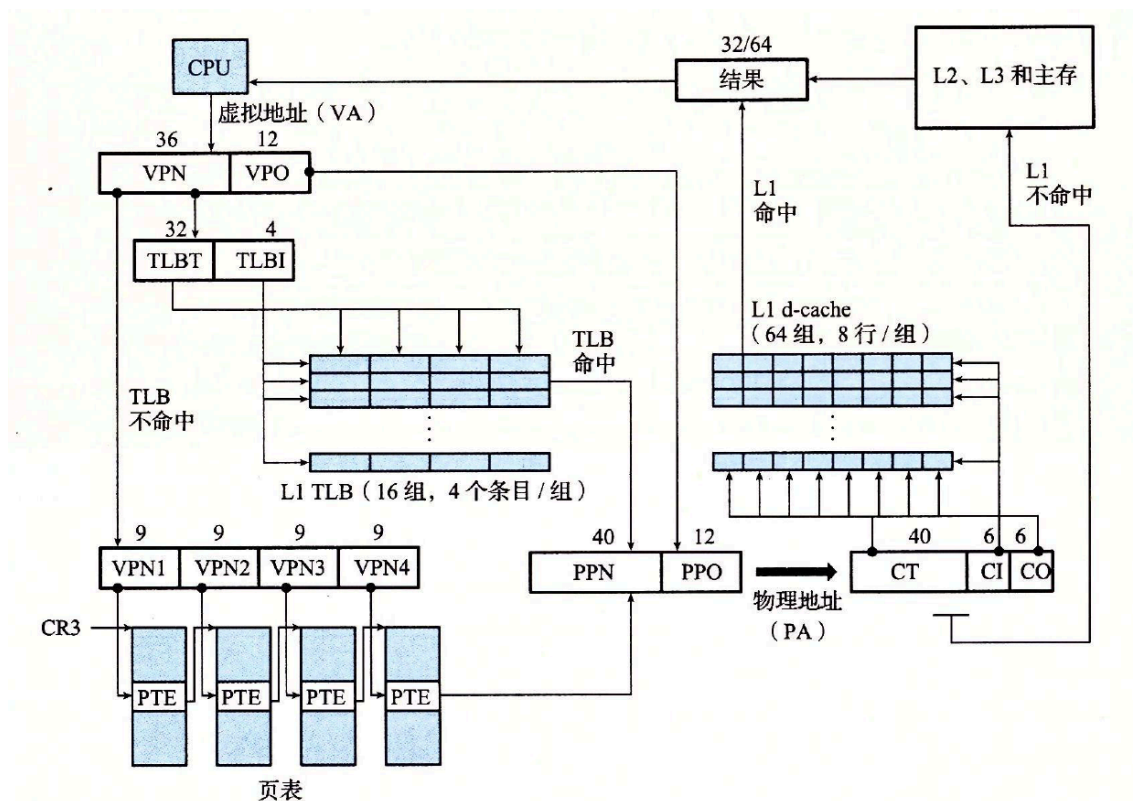


图 7.3 三级 Cache 支持下的物理内存访问

从 CPU 产生虚拟地址的时刻一直到来自内存的数据字到达 CPU。Core i7 采用四级页表层次结构。每个进程有它自己私有的页表层次结构。当一个 Linux 进程在运行时，虽然 Core i7 体系结构允许页表换进换出，但是与已分配了的页相关联的页表都是驻留在内存中的。CR3 控制寄存器指向第一级页表 L1 的起始位置。CR3 的值是每个进程上下文的一部分，每次上下文切换时，CR3 的值都会被恢复。

7.6 hello 进程 fork 时的内存映射

虚拟内存和内存映射解释了 fork 函数如何为每个新进程提供私有的虚拟地址空间。fork 函数为新进程创建虚拟内存。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 (vm_area_struct) 都标记为私有的写时复制 (COW)。在新进程中返回时，新进程拥有与调用 fork 进程相同的虚拟内存，随后的写操作通过写时复制机制创建新页面。

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.out` 的步骤：删除已存在的用户区域，创建新的区域结构，代码和初始化数据映射到 `.text` 和 `.data` 区（目标文件提供），`.bss` 和栈映射到匿名文件，设置 PC，指向代码区域的入口点。Linux 根据需要换入代码和数据页面。

7.8 缺页故障与缺页中断处理

在虚拟内存的习惯说法中，DRAM 缓存不命中称为缺页（page fault）。下图展示了在缺页之前我们的示例页表的状态。CPU 引用了 VP 3 中的一个字，VP 3 并未缓存在 DRAM 中。地址翻译硬件从内存中读取 PTE 3，从有效位推断出 VP 3 未被缓存，并且触发一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，在此例中就是存放在 pp 3 中的 VP4。如果 VP4 已经被修改了，那么内核就会将它复制回磁盘。无论哪种情况，内核都会修改 VP4 的页表条目，反映出 VP4 不再缓存在主存中这一事实。

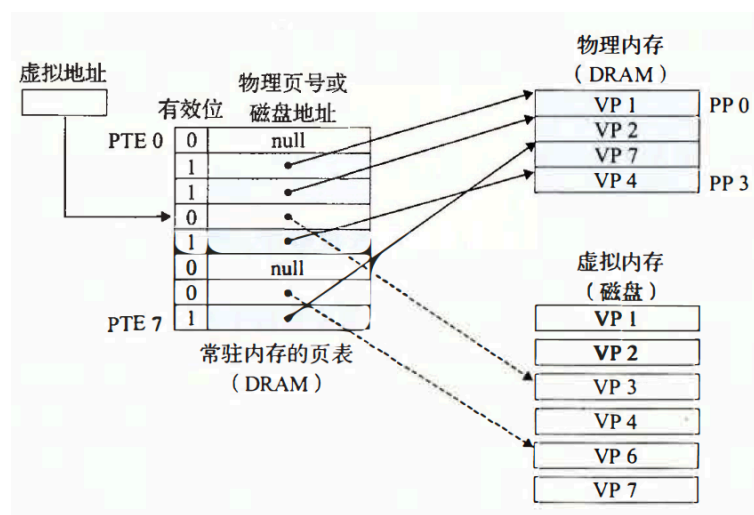


图 7.4 缺页故障与缺页中断处理

接下来，内核从磁盘复制 VP 3 到内存中的 pp 3,更新 PTE 3,随后返回。当异常处理程序返回时，它会重新启动导致缺页的指令，该指令会把导致缺页的虚拟地址重发送到地址翻译硬件。但是现在，VP 3 已经缓存在主存中了，那么页命中也能由地址翻译硬件正常处理了。下图展示了在缺页之后我们的示例页表的状态。

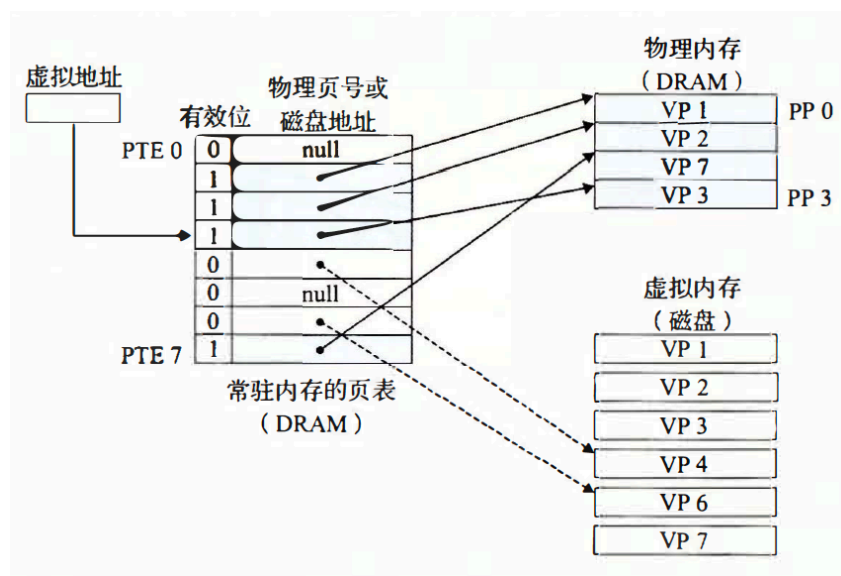


图 7.5 处理后的虚拟内存和物理内存状态

7.9 动态存储分配管理

在程序运行时程序员使用动态内存分配器（如 `malloc`）获得虚拟内存。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，每个块要么是已分配的，要么是空闲的。分配器的类型包括显式分配器和隐式分配器。前者要求应用显式地释放任何已分配的块，后者在检测到已分配块不再被程序所使用时，就释放这个块。

动态内存管理的策略包括首次适配、下一次适配和最佳适配。首次适配会从头开始搜索空闲链表，选择第一个合适的空闲块。搜索时间与总块数（包括已分配和空闲块）成线性关系。会在靠近链表起始处留下小空闲块的“碎片”。下一次适配和首次适配相似，只是从链表中上一次查询结束的地方开始。比首次适应更快，避免重复扫描那些无用块。最佳适配会查询链表，选择一个最好的空闲块，满足适配，且剩余最少空闲空间。它可以保证碎片最小，提高内存利用率。

7.10 本章小结

虚拟内存是对主存的一个抽象。支持虚拟内存的处理器通过使用一种叫做虚拟寻址的间接形式来引用主存。处理器产生一个虚拟地址，在被发送到主存之前，这个地址被翻译成一个物理地址。从虚拟地址空间到物理地址空间的地址翻译要求硬件和软件紧密合作。专门的硬件通过使用页表来翻译虚拟地址，而页表的内容是由操作系统提供的。

虚拟内存提供三个重要的功能。第一，它在主存中自动缓存最近使用的存放磁盘上的虚拟地址空间的内容。虚拟内存缓存中的块叫做页。对磁盘上页的引用会触发缺页，缺页将控制转移到操作系统中的一个。缺页处理程序。缺页处理程序将页面从磁盘复制到主存缓存，如果必要，将写回被驱逐的页。第二，虚拟内存简化了内存管理，进而又简化了链接、在进程间共享数据、进程的内存分配以及程序加载。最后，虚拟内存通过在每条页表条目中加入保护位，从而简化了内存保护。

地址翻译的过程必须和系统中所有的硬件缓存的操作集成在一起。大多数页表条目位于 L1 高速缓存中，但是一个称为 TLB 的页表条目的片上高速缓存，通常会消除访问在 L1 上的页表条目的开销。

现代系统通过将虚拟内存片和磁盘上的文件片关联起来，来初始化虚拟内存片，这个过程称为内存映射。内存映射为共享数据、创建新的进程以及加载程序提供了一种高效的机制。应用可以使用 `mmap` 函数来手工地创建和删除虚拟地址空间的区域。然而，大多数程序依赖于动态内存分配器，例如 `malloc`，它管理虚拟地址空间区域内一个称为堆的区域。动态内存分配器是一个感觉像系统级程序的应用级程序，它直接操作内存，而无需类型系统的很多帮助。分配器有两种类型。显式分配器要求应用显式地释放它们的内存块。隐式分配器（垃圾收集器）自动释放任何未使用的和不可达的块。

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

一个 Linux 文件就是一个 m 个字节的序列：

$$B_0, B_1, \dots, B_k, \dots, B_{m-1}$$

所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。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，这使得所有的输入和输出都能以一种统一且一致的方式来执行：

- 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。应用程序只需记住这个描述符。
- 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`，它们可用来代替显式的描述符值。
- 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` 操作，显式地设置文件的当前位置为 K 。
- 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 $k \geq m$ 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。
- 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。

8.2 简述 Unix IO 接口及其函数

linux 提供如下 IO 接口函数：

read 和 write – 最简单的读写函数；

readn 和 writen – 原子性读写操作；

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

recv 和 send – 允许从进程到内核传递标志；

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

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

8.3 printf 的实现分析

printf 函数代码如下所示：

```
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;
}
```

(char*)&fmt + 4) 表示的是...可变参数中的第一个参数的地址。而 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。接着从 vsprintf 生成显示信息，到 write 系统函数，直到陷阱系统调用 `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

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

Linux 提供了少星的基于 Unix I/O 模型的系统级函数，它们允许应用程序打开、关闭、读和写文件，提取文件的元数据，以及执行 I/O 重定向。Linux 的读和写操

作会出现不足值，应用程序必须能正确地预计和处理这种情况。应用程序不应直接调用 Unix I/O 函数，而应该使用 RIO 包，RIO 包通过反复执行读写操作，直到传送完所有的请求数据，自动处理不足值。

Linux 内核使用三个相关的数据结构来表示打开的文件。描述符表中的表项指向打开文件表中的表项，而打开文件表中的表项又指向 v-node 表中的表项。每个进程都有它自己单独的描述符表，而所有的进程共享同一个打开文件表和 v-node 表。理解这些结构的一般组成就能使我们清楚地理解文件共享和 I/O 重定向。

标准 I/O 库是基于 Unix I/O 实现的，并提供了一组强大的高级 I/O 例程。对于大多数应用程序而言，标准 I/O 更简单，是优于 Unix I/O 的选择。然而，因为对标准 I/O 和网络文件的一些相互不兼容的限制，Unix I/O 比之标准 I/O 更该适用于网络应用程序。

结论

hello.c 通过键盘鼠标等 I/O 设备输入计算机，并存储在内存中。然后预处理器将 hello.c 预处理成为文本文件 hello.i。接着编译器将 hello.i 翻译成汇编语言文件 hello.s。汇编器将 hello.s 汇编成可重定位二进制代码 hello.o。链接器将外部文件和 hello.o 连接起来形成可执行二进制文件 hello.out。shell 通过 fork 和 execve 创建进程，然后把 hello 加载到其中。shell 创建新的内存区域，并加载代码、数据和堆栈。hello 在运行的过程中遇到异常，会接受 shell 的信号完成处理。hello 在运行的过程中需要使用内存，那么就通过 CPU 和虚拟空间进行地址访问。Hello 运行结束后，shell 回收其僵尸进程，从系统中消失。

hello 的完整运行过程如下：

编写：通过键盘将代码输入到 hello.c

预处理：将 hello.c 调用的所有外部库合并到 hello.i 文件中

编译：将 hello.i 编译为汇编文件 hello.s

汇编：将 hello.s 会变成成为可重定位目标文件 hello.o

链接：将 hello.o 与可重定位目标文件与动态链接库链接为可执行目标程序

a.out

运行：在 shell 中输入 ./a.out 1170300603 wenyongyan

创建子进程：shell 进程调用 fork 为其创建子进程

运行：shell 调用 `execve`，`execve` 调用启动加载器，映射虚拟内存，进入程序入口后程序开始载入物理内存，随后进入 `main` 函数。

执行指令：CPU 为其分配时间片，在一个时间片中，hello 享有 CPU 资源，顺序执行自己的控制逻辑流

访问内存：MMU 将程序中使用的虚拟内存地址通过页表映射为物理地址

动态申请内存：`printf` 会调用 `malloc` 向动态内存分配器申请堆中的内存

信号：如果运行途中键入 `ctr-c` `ctr-z` 则调用 shell 的信号处理函数分别停止、挂起。

结束：shell 父进程回收子进程，内核删除为这个进程创建的所有数据结构

附件

中间结果文件名	文件作用
hello.i	预处理得到的中间结果
hello.s	hello.i 编译后得到的汇编语言文本文件
Hello_o.o	hello.s 汇编后得到的可重定位目标文件
a.out	链接后得到的可执行目标文件
hello.o	hello_o.o 链接后得到的可重定位目标文件
hello.objdump	hello 的反汇编代码
hello.elf	hello 的 elf 格式文件

参考文献

- [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] 湛颖. 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(原书第3版)/(美)兰德尔·E.布莱恩特(Randal E. Byrant)
等; 龚奕利, 贺莲译. -北京: 机械工业出版社, 2016.7
- [8] <https://blog.csdn.net/tuxedolinux/article/details/80317419>
- [9] <https://blog.csdn.net/xuwq2015/article/details/48572421>
- [10] <https://www.cnblogs.com/pianist/p/3315801.html>
- [11] <https://blog.csdn.net/tuxedolinux/article/details/80317419>