

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>計算機類</u>
學 號	<u>1190200523</u>
班 級	<u>1903002</u>
學 生	<u>石翔宇</u>
指 導 教 師	<u>鄭貴濱</u>

計算機科學與技術學院

2021 年 6 月

摘 要

本文通过对 `hello` 程序在计算机上产生、运行、结束等过程的深入分析，对本课程所学知识进行系统的回顾与梳理。以更加熟练地掌握计算机系统相关的知识，加深对计算机系统的了解。

关键词：计算机；计算机系统；`hello` 程序；

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 -
3.3.1 数据.....	- 8 -
3.3.2 算术操作.....	- 9 -
3.3.3 关系操作和控制转移.....	- 9 -
3.3.4 数组/指针/结构操作.....	- 9 -
3.3.5 函数操作.....	- 10 -
3.4 本章小结.....	- 11 -
第 4 章 汇编	- 12 -
4.1 汇编的概念与作用.....	- 12 -
4.2 在 UBUNTU 下汇编的命令.....	- 12 -
4.3 可重定位目标 ELF 格式.....	- 12 -
4.3.1 ELF 头.....	- 13 -
4.3.2 节头部表.....	- 13 -
4.3.3 重定位条目.....	- 14 -
4.3.4 符号表.....	- 15 -
4.4 HELLO.O 的结果解析.....	- 15 -
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 -
5.3.1 ELF 头.....	- 19 -

5.3.2 节头部表	- 19 -
5.3.3 符号表	- 20 -
5.4 HELLO 的虚拟地址空间	- 21 -
5.5 链接的重定位过程分析	- 23 -
5.6 HELLO 的执行流程	- 23 -
5.7 HELLO 的动态链接分析	- 24 -
5.8 本章小结	- 25 -
第 6 章 HELLO 进程管理	- 26 -
6.1 进程的概念与作用	- 26 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	- 26 -
6.3 HELLO 的 FORK 进程创建过程	- 27 -
6.4 HELLO 的 EXECVE 过程	- 27 -
6.5 HELLO 的进程执行	- 28 -
6.6 HELLO 的异常与信号处理	- 29 -
6.7 本章小结	- 32 -
第 7 章 HELLO 的存储管理	- 33 -
7.1 HELLO 的存储器地址空间	- 33 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 33 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 33 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	- 34 -
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	- 36 -
7.8 缺页故障与缺页中断处理	- 37 -
7.9 动态存储分配管理	- 37 -
7.10 本章小结	- 39 -
第 8 章 HELLO 的 IO 管理	- 40 -
8.1 LINUX 的 IO 设备管理方法	- 40 -
8.2 简述 UNIX IO 接口及其函数	- 40 -
8.3 PRINTF 的实现分析	- 41 -
8.4 GETCHAR 的实现分析	- 42 -
8.5 本章小结	- 43 -
结论	- 44 -
附件	- 45 -
参考文献	- 46 -

第 1 章 概述

1.1 Hello 简介

P2P:

Program: 编写代码保存成 `hello.c`。

Process: 将 `hello.c` 预处理、编译、汇编、链接成为可执行目标程序 `hello`。

O2O:

shell 执行 `execve` 创建 `hello` 进程，映射虚拟内存。进入程序入口执行目标代码，CPU 为程序分配时间片，高速缓存机制加速程序的运行。当程序运行结束后，shell 回收 `hello` 进程，内核删除相关数据结构。

1.2 环境与工具

硬件环境:

Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz

16GB RAM

1TB HDD + 512G SSD

软件环境:

Windows 10 21H1

Ubuntu 20.04 LTS

开发工具:

VSCode, CodeBlocks, gcc, gdb, readelf

1.3 中间结果

文件名	文件作用
<code>hello.i</code>	<code>hello.c</code> 经过预处理后的文件
<code>hello.s</code>	<code>hello.i</code> 经过编译后的汇编文件
<code>hello.o</code>	<code>hello.s</code> 经过汇编后的可重定位目标文件
<code>hello</code>	<code>hello.o</code> 经过链接之后的可执行目标文件
<code>hello-elf.txt</code>	<code>hello.o</code> 的 ELF 格式
<code>hello-h.txt</code>	<code>hello.o</code> 的 ELF 格式的 ELF 头
<code>hello-r.txt</code>	<code>hello.o</code> 的 ELF 格式的重定位条目
<code>hello-s.txt</code>	<code>hello.o</code> 的 ELF 格式的符号表
<code>hello-sec.txt</code>	<code>hello.o</code> 的 ELF 格式的节头部表
<code>hello-dis.s</code>	<code>hello.o</code> 的反汇编文件

hello-elf2.txt	hello 的 ELF 格式
hello-h2.txt	hello 的 ELF 格式的 ELF 头
hello-r2.txt	hello 的 ELF 格式的重定位条目
hello-s2.txt	hello 的 ELF 格式的符号表
hello-sec2.txt	hello 的 ELF 格式的节头部表
hello-dis2.s	hello 的反汇编文件

1.4 本章小结

本章对 hello 进行了简述，介绍了 hello 的 P2P、O2O 的过程，也介绍了实验所用到的环境及工具。最后介绍了完成实验所需的中间文件的名称及其作用。

第 2 章 预处理

2.1 预处理的概述与作用

概念：

预处理是 C 语言的一个重要功能，它由预处理程序负责完成。当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分作处理，处理完毕自动进入对源程序的编译。C 语言提供多种预处理功能，主要处理#开始的预编译指令，如宏定义(#define)、文件包含(#include)、条件编译(#ifdef)等。

作用：

条件编译：条件编译的功能是根据条件有选择性的保留或者放弃源文件中的内容。常见的条件包含#if、#ifdef、#ifndef 指令开始，以#endif 结束。用#undef 指令可对用#define 定义的标识符取消定义。

源文件包含：源文件包含指令的功能是搜索指定的文件，并将它的内容包含进来，放在当前所在的位置。源文件包含有两种，包含系统文件以及用户自定义文件。

宏替换：宏的作用是把一个标识符指定为其他一些成为替换列表的预处理记号，当这个标识符出现在后面的文本中时，将用对应的预处理记号把它替换掉，宏的本质是替换。

行控制：行控制指令以“#”和“line”引导，后面是行号和可选的字面串。它用于改变预定义宏“__LINE__”的值，如果后面的字面串存在，则改变“__FILE__”的值。

抛错：抛错指令是以“#”和“error”引导，抛错指令用于在预处理期间发出一个诊断信息，在停止转换。抛错是人为的动作。

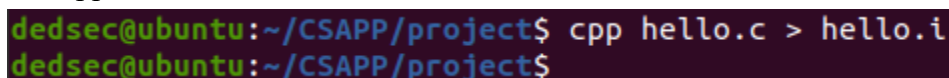
杂注：杂注指令用于向 C 实现传递额外的信息（编译选项），对程序的某些方面进行控制。

空指令：空指令只有一个“#”，自成一，空指令的使用没有效果。

2.2 在 Ubuntu 下预处理的命令

预处理命令：

Linux > cpp hello.c > hello.i



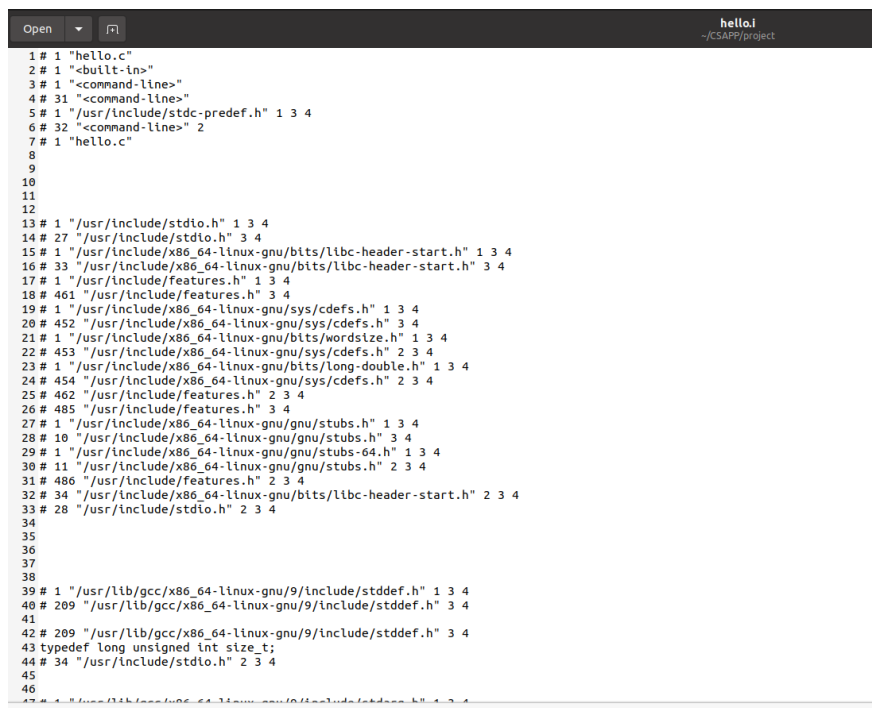
```
dedsec@ubuntu:~/CSAPP/project$ cpp hello.c > hello.i
dedsec@ubuntu:~/CSAPP/project$
```

图 1 Ubuntu 下预处理的命令

2.3 Hello 的预处理结果解析

文件由 28 行的 `hello.c` 变成了 3065 行的 `hello.i`。

在 `hello.i` 中，首先是对文件包含中系统头文件的寻址和解析。接着是 `hello.c` 引用的头文件中内容的递归式的寻址和展开，包括结构体类型、枚举类型、通过 `extern` 关键字调用并声明外部的结构体及函数定义。文件的最后部分则是原来的 `hello.c` 文件中的 `main` 函数内容。



```
1 # 1 "hello.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "hello.c"
8
9
10
11
12
13 # 1 "/usr/include/stdio.h" 1 3 4
14 # 27 "/usr/include/stdio.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
16 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
17 # 1 "/usr/include/features.h" 1 3 4
18 # 461 "/usr/include/features.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
20 # 452 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 453 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
23 # 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
24 # 454 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 462 "/usr/include/features.h" 2 3 4
26 # 485 "/usr/include/features.h" 3 4
27 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
28 # 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
29 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
30 # 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
31 # 486 "/usr/include/features.h" 2 3 4
32 # 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
33 # 28 "/usr/include/stdio.h" 2 3 4
34
35
36
37
38
39 # 1 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 1 3 4
40 # 209 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 3 4
41
42 # 209 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 3 4
43 typedef long unsigned int size_t;
44 # 34 "/usr/include/stdio.h" 2 3 4
45
46
47 # 1 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 1 3 4
```

图 2 `hello.i` 文件部分内容

2.4 本章小结

本章介绍了 C 语言由源代码生成可执行文件过程中的预处理过程，包括预处理的`概念`、`作用`等内容。预处理过程为后面生成汇编语言程序操作做了准备工作。

第 3 章 编译

3.1 编译的概念与作用

概念：

利用编译程序从源语言编写的源程序，通过对代码的语法和语义的分析，生成汇编代码产生目标程序的过程。

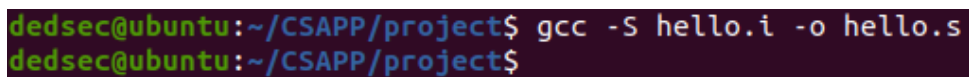
作用：

将便于人编写、阅读、维护的高级计算机语言所写作的源代码程序，翻译为计算机能解读、运行的低阶机器语言的程序，也就是可执行文件。编译器将原始程序作为输入，翻译产生使用目标语言的等价程序。

3.2 在 Ubuntu 下编译的命令

编译命令：

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



```
dedsec@ubuntu:~/CSAPP/project$ gcc -S hello.i -o hello.s
dedsec@ubuntu:~/CSAPP/project$
```

图 3 Ubuntu下编译的命令

3.3 Hello 的编译结果解析

3.3.1 数据

3.3.1.1 常量

源代码中的 `if(argc!=3)` 中的常量 3，对应于汇编代码中第 30 行 .text 段中 `cmpl $3, -20(%rbp)` 的 \$3。

源代码中的 `exit(1)` 中的常量 1，对应于汇编代码中第 34 行 .text 段中 `movl $1, %edi` 中的 \$1。

源代码中的 `for(i=0;i<10;i++)` 中的常量 0 和 10，分别对应于汇编代码中第 37 行 .text 段中 `movl $0, -4(%rbp)` 中的 \$0；第 55 行 .text 段中 `cmpl $9, -4(%rbp)` 中的 \$9（汇编代码中为 10-1）。

源代码中的 `printf("Hello %s %s\n", argv[1], argv[2])` 中的常量 "Hello %s %s\n"，对应于汇编代码中第 14 行 .rodata 段中的 .string "Hello %s %s\n"。

源代码中的 `printf("Usage: Hello 学号 姓名! \n");` 中的常量 "Usage: Hello 学号 姓名! \n", 对应于汇编代码中第 12 行 `.rodata` 段中的 `.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"`。

3.3.1.2 变量

全局变量：源代码中的 `int sleepsecs=2.5` 定义的是一个全局变量，初始化的全局变量 `sleepsecs` 储存在 `.data` 节中 `sleepsecs: .long 2`。

局部变量：源代码中的 `int i` 定义的是一个局部变量，存储在栈中，位于 `-4(%rbp)` 处。

3.3.1.3 类型转换

源代码中 `int sleepsecs=2.5` 进行了隐式的类型转换，将 `float` 类型转换为 `(long) int` 类型，在汇编代码中的值为 2。

3.3.2 算术操作

`hello.c` 文件中涉及的算术操作只有“++”：`for(i=0;i<10;i++)`。在汇编代码中表示为 `addl $1, -4(%rbp)`，其中 `-4(%rbp)` 中存储的是 `i`，`i` 存储在栈上。

3.3.3 关系操作和控制转移

源代码中第 16 行的表达式 `if(argc!=3)` 表示一个不等于的关系操作，在汇编代码中表示为 `cmpl $3, -20(%rbp) je .L2`，其中用 `je` 代表表达式中的“!=”，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的地址。

源代码中第 21 行的表达式 `for(i=0;i<10;i++)` 中包含一个小于的关系操作，在汇编代码中表示为 `cmpl $9, -4(%rbp) jle .L4`，其中用 `jle` 代表表达式中的“<”，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的地址。

3.3.4 数组/指针/结构操作

源代码中第 12 行 `int main(int argc, char *argv[])` 定义了 `char *argv[]`，类型为 `char *` 的数组，在第 23 行 `printf("Hello %s %s\n", argv[1], argv[2])` 引用了该数组的第 1 和第 2 个元素（下标从 0 开始）。

由汇编代码：

```

movq %rsi, -32(%rbp)
和
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

```

可知 `char *argv[0]` 存储于 `-32(%rbp)` 处，对其分别进行 +8 和 +16 操作，再到相应的地址中分别取到 `argv[1]` 和 `argv[2]`。

3.3.5 函数操作

3.3.5.1 main 函数

参数传递：源代码中第 12 行 `int main(int argc, char *argv[])` 表示了 main 函数的参数，对应汇编代码第 28、29 行，`movl %edi, -20(%rbp)` `movq %rsi, -32(%rbp)`，其中 `%edi` 存储的是 `argc`，`%rsi` 存储的是 `argv[]`。

函数调用：由系统启动函数调用。

函数返回：源代码中第 27 行 `return 0` 是 main 函数的返回语句，对应汇编代码第 58、61 行 `movl $0, %eax` `ret`。汇编代码中将 `%eax` 设置为 0，即 main 函数的返回值，之后返回。

3.3.5.2 printf 函数

参数传递：在源代码第 18 行 `printf("Usage: Hello 学号 姓名! \n")`，传入一个字符串（地址），对应汇编代码第 32、33 行 `leaq .LC0(%rip), %rdi` `call puts@PLT`；在源代码第 23 行 `printf("Hello %s %s\n", argv[1], argv[2])`，传入一个字符串（地址）和两个参数 `argv[1]` 和 `argv[2]`，对应汇编代码第 42、46、47、49 行 `movq (%rax), %rdx` `movq %rax, %rsi` `leaq .LC1(%rip), %rdi` `call printf@PLT`。

函数调用：在源代码第 18 行，满足第 16 行 if 判断条件则调用；在源代码第 23 行，进入第 21 行的 for 语句则调用。

函数返回：源代码及汇编代码中不涉及 printf 函数的返回值。

3.3.5.3 exit 函数

参数传递：在源代码第 19 行 exit(1)，传入一个数字 1，对应汇编代码第 34、35 行 movl \$1,%edi call exit@PLT。

函数调用：在源代码第 19 行，满足第 16 行 if 判断条件则调用。

函数返回：源代码及汇编代码中不涉及 exit 函数的返回值。

3.3.5.4 sleep 函数

参数传递：在源代码第 24 行 sleep(sleepsecs)，传入数字 sleepsecs，对应汇编代码第 50、51、52 行 movl sleepsecs(%rip),%eax movl %eax,%edi call sleep@PLT。

函数调用：在源代码第 24 行，进入第 21 行的 for 语句则调用。

函数返回：源代码及汇编代码中不涉及 sleep 函数的返回值。

3.3.5.5 getchar 函数

参数传递：在源代码第 26 行 getchar()，无参数，对应汇编代码第 57 行 call getchar@PLT。

函数调用：程序运行到源代码第 26 行则调用。

函数返回：源代码及汇编代码中不涉及 getchar 函数的返回值。

3.4 本章小结

本章介绍了 C 语言编译的概念和作用。本章以 hello.c 的编译结果为例，将源代码与汇编代码对应起来，介绍了编译器对常量、变量、表达式等的处理，以及对于类型转换、算术操作、关系操作、数组操作、控制转移、函数操作等内容的具体实现方法。我们可以发现，编译器并不是逐条语句翻译成汇编文件的。编译器不仅会对我们的代码做一些隐式的优化，而且会将原来代码中用到的跳转，循环等操作作用控制转移等方法进行解析。

第 4 章 汇编

4.1 汇编的概念与作用

概念：

使用汇编语言编写的源代码，然后通过相应的汇编程序将它们转换成可执行的机器代码。

作用：

典型的现代汇编器建造目标代码，由解译组语指令集的助记符到操作码，并解析符号名称成为存储器地址以及其它的实体。汇编器将编译过的.s 文件中的汇编语言指令转化成机器语言指令形成.o 可重定向文件。

4.2 在 Ubuntu 下汇编的命令

汇编命令：

Linux > as hello.s -o hello.o

```
dedsec@ubuntu:~/CSAPP/project$ as hello.s -o hello.o
dedsec@ubuntu:~/CSAPP/project$
```

图 4 Ubuntu下汇编的命令

4.3 可重定位目标 ELF 格式

我们可以使用命令 Linux > readelf -a hello.o > hello-elf.txt 获得 hello.o 的完整结构，并将其保存在 hello-elf.txt 中。

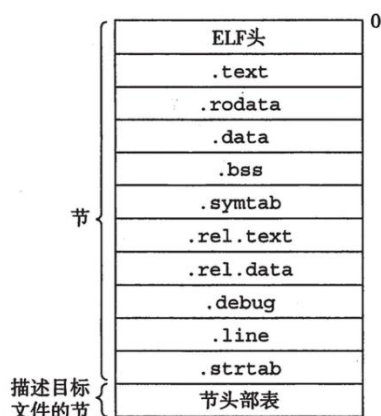


图 5 典型的ELF可重定位目标文件

4.3.1 ELF 头

我们可以使用命令 `Linux > readelf -h hello.o > hello-h.txt` 获得 ELF 头，并将其保存在 `hello-h.txt` 中。ELF 头描述了生成该文件的系统的字的大小，字节顺序，ELF 头的大小，目标文件的类型，机器类型，节头部表的文件偏移，以及节头部表中条目的大小和数量等。

```
1 ELF Header:
2 Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 Class:                               ELF64
4 Data:                                2's complement, little endian
5 Version:                             1 (current)
6 OS/ABI:                               UNIX - System V
7 ABI Version:                         0
8 Type:                                REL (Relocatable file)
9 Machine:                             Advanced Micro Devices X86-64
10 Version:                             0x1
11 Entry point address:                 0x0
12 Start of program headers:            0 (bytes into file)
13 Start of section headers:           1232 (bytes into file)
14 Flags:                               0x0
15 Size of this header:                 64 (bytes)
16 Size of program headers:             0 (bytes)
17 Number of program headers:           0
18 Size of section headers:            64 (bytes)
19 Number of section headers:           14
20 Section header string table index: 13
```

图 6 hello.o 的 ELF 头

4.3.2 节头部表

我们可以使用命令 `Linux > readelf -S hello.o > hello-sec.txt` 获得节头部表，并将其保存在 `hello-sec.txt` 中。节头部表描述了不同节的位置和大小。

```

1  There are 14 section headers, starting at offset 0x4d0:
2
3  Section Headers:
4  [Nr] Name                Type                Address              Offset
5      Size                EntSize            Flags  Link  Info  Align
6  [ 0]                      NULL               0000000000000000    00000000
7      0000000000000000    0000000000000000    0  0  0
8  [ 1] .text                 PROGBITS           0000000000000000    00000040
9      0000000000000085    0000000000000000    AX  0  0  1
10 [ 2] .rela.text            RELA               0000000000000000    00000380
11      00000000000000c0    0000000000000018    I  11  1  8
12 [ 3] .data                 PROGBITS           0000000000000000    000000c8
13      0000000000000004    0000000000000000    WA  0  0  4
14 [ 4] .bss                  NOBITS             0000000000000000    000000cc
15      0000000000000000    0000000000000000    WA  0  0  1
16 [ 5] .rodata               PROGBITS           0000000000000000    000000cc
17      000000000000002b    0000000000000000    A  0  0  1
18 [ 6] .comment              PROGBITS           0000000000000000    000000f7
19      000000000000002b    0000000000000001    MS  0  0  1
20 [ 7] .note.GNU-stack        PROGBITS           0000000000000000    00000122
21      0000000000000000    0000000000000000    0  0  1
22 [ 8] .note.gnu.propert      NOTE               0000000000000000    00000128
23      0000000000000020    0000000000000000    A  0  0  8
24 [ 9] .eh_frame             PROGBITS           0000000000000000    00000148
25      0000000000000038    0000000000000000    A  0  0  8
26 [10] .rela.eh_frame         RELA               0000000000000000    00000440
27      0000000000000018    0000000000000018    I  11  9  8
28 [11] .symtab               SYMTAB             0000000000000000    00000180
29      000000000000001b0    0000000000000018    12 10  8
30 [12] .strtab               STRTAB             0000000000000000    00000330
31      000000000000004d    0000000000000000    0  0  1
32 [13] .shstrtab              STRTAB             0000000000000000    00000458
33      0000000000000074    0000000000000000    0  0  1
34 Key to Flags:
35 W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
36 L (link order), O (extra OS processing required), G (group), T (TLS),
37 C (compressed), x (unknown), o (OS specific), E (exclude),
38 l (large), p (processor specific)

```

图 7 hello.o 的节头部表

4.3.3 重定位条目

我们可以使用命令 `Linux > readelf -r hello.o > hello-r.txt` 获得重定位条目，并将其保存在 `hello-r.txt` 中。重定位条目包括了 `.rela.text` 和 `.rela.eh_frame`。`.rela.text` 包含了代码段中引用的外部函数和全局变量的重定位条目。一般而言，任何调用外部函数或者引用全局变量的指令都需要修改。`hello.o` 中需要重定位的信息有 `puts`、`exit`、`printf`、`sleepsecs` 和 `getchar`。`.rela.eh_frame` 包含了 `eh_frame` 的重定位信息。

```

2  Relocation section '.rela.text' at offset 0x380 contains 8 entries:
3  | Offset          Info          Type                Sym. Value    Sym. Name + Addend
4  | 000000000000001c 000500000002 R_X86_64_PC32      0000000000000000 .rodata - 4
5  | 0000000000000021 000d00000004 R_X86_64_PLT32      0000000000000000 puts - 4
6  | 000000000000002b 000e00000004 R_X86_64_PLT32      0000000000000000 exit - 4
7  | 0000000000000054 000500000002 R_X86_64_PC32      0000000000000000 .rodata + 1a
8  | 000000000000005e 000f00000004 R_X86_64_PLT32      0000000000000000 printf - 4
9  | 0000000000000064 000a00000002 R_X86_64_PC32      0000000000000000 sleepsecs - 4
10 | 000000000000006b 001000000004 R_X86_64_PLT32      0000000000000000 sleep - 4
11 | 000000000000007a 001100000004 R_X86_64_PLT32      0000000000000000 getchar - 4
12
13 Relocation section '.rela.eh_frame' at offset 0x440 contains 1 entry:
14 | Offset          Info          Type                Sym. Value    Sym. Name + Addend
15 | 0000000000000020 000200000002 R_X86_64_PC32      0000000000000000 .text + 0

```

图 8 hello.o 的重定位条目

4.3.4 符号表

我们可以使用命令 `Linux > readelf -s hello.o > hello-s.txt` 获得符号表，并将其保存在 `hello-s.txt` 中。符号表存放在程序中定义和应用的函数和全局变量的信息。

```

2 Symbol table '.symtab' contains 18 entries:
3
4   Num:      Value              Size Type Bind Vis      Ndx Name
5   0: 0000000000000000          0 NOTYPE LOCAL DEFAULT UND
6   1: 0000000000000000          0 FILE   LOCAL DEFAULT ABS hello.c
7   2: 0000000000000000          0 SECTION LOCAL DEFAULT 1
8   3: 0000000000000000          0 SECTION LOCAL DEFAULT 3
9   4: 0000000000000000          0 SECTION LOCAL DEFAULT 4
10  5: 0000000000000000          0 SECTION LOCAL DEFAULT 5
11  6: 0000000000000000          0 SECTION LOCAL DEFAULT 7
12  7: 0000000000000000          0 SECTION LOCAL DEFAULT 8
13  8: 0000000000000000          0 SECTION LOCAL DEFAULT 9
14  9: 0000000000000000          0 SECTION LOCAL DEFAULT 6
15 10: 0000000000000000          4 OBJECT GLOBAL DEFAULT 3 sleepsecs
16 11: 0000000000000000        133 FUNC   GLOBAL DEFAULT 1 main
17 12: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
18 13: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND puts
19 14: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND exit
20 15: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND printf
21 16: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND sleep
   17: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UND getchar

```

图 9 hello.o 的符号表

4.4 Hello.o 的结果解析

使用命令 `Linux > objdump -d -r hello.o > hello-dis.s` 获取 `hello.o` 的反汇编文件。


```

2  hello.o:      file format elf64-x86-64
3
4
5  Disassembly of section .text:
6
7  0000000000000000 <main>:
8      0: f3 0f 1e fa      endbr64
9      4: 55              push    %rbp
10     5: 48 89 e5        mov     %rsp,%rbp
11     8: 48 83 ec 20     sub     $0x20,%rsp
12     c: 89 7d ec        mov     %edi,-0x14(%rbp)
13     f: 48 89 75 e0     mov     %rsi,-0x20(%rbp)
14    13: 83 7d ec 03     cmpl    $0x3,-0x14(%rbp)
15    17: 74 16          je      2f <main+0x2f>
16    19: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # 20 <main+0x20>
17    | 1c: R_X86_64_PC32 .rodata-0x4
18    20: e8 00 00 00 00     callq   25 <main+0x25>
19    | 21: R_X86_64_PLT32 puts-0x4
20    25: bf 01 00 00 00     mov     $0x1,%edi
21    2a: e8 00 00 00 00     callq   2f <main+0x2f>
22    | 2b: R_X86_64_PLT32 exit-0x4
23    2f: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
24    36: eb 3b          jmp     73 <main+0x73>
25    38: 48 8b 45 e0     mov     -0x20(%rbp),%rax
26    3c: 48 83 c0 10     add     $0x10,%rax
27    40: 48 8b 10       mov     (%rax),%rdx
28    43: 48 8b 45 e0     mov     -0x20(%rbp),%rax
29    47: 48 83 c0 08     add     $0x8,%rax
30    4b: 48 8b 00       mov     (%rax),%rax
31    4e: 48 89 c6       mov     %rax,%rsi
32    51: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # 58 <main+0x58>
33    | 54: R_X86_64_PC32 .rodata+0x1a
34    58: b8 00 00 00 00     mov     $0x0,%eax
35    5d: e8 00 00 00 00     callq   62 <main+0x62>
36    | 5e: R_X86_64_PLT32 printf-0x4
37    62: 8b 05 00 00 00 00     mov     0x0(%rip),%eax      # 68 <main+0x68>
38    | 64: R_X86_64_PC32 sleepsecs-0x4
39    68: 89 c7          mov     %eax,%edi
40    6a: e8 00 00 00 00     callq   6f <main+0x6f>
41    | 6b: R_X86_64_PLT32 sleep-0x4
42    6f: 83 45 fc 01     addl    $0x1,-0x4(%rbp)
43    73: 83 7d fc 09     cmpl    $0x9,-0x4(%rbp)
44    77: 7e bf          jle     38 <main+0x38>
45    79: e8 00 00 00 00     callq   7e <main+0x7e>
46    | 7a: R_X86_64_PLT32 getchar-0x4
47    7e: b8 00 00 00 00     mov     $0x0,%eax
48    83: c9          leaveq
49    84: c3          retq
50

```

图 10 hello.o的反汇编文件hello-dis.s

hello.s 和 hello-dis.s 的不同:

1. 分支转移与函数调用: 跳转(jmp)或者调用(call)时 hello.s 使用的是 L2、L3、L4 等标号或者函数名称, 而 hello-dis.s 使用的是地址偏移量或者为其在.rela.text 节中添加重定位条目。
2. 数据表示: hello.s 中使用的操作数等数字是十进制的形式, 而 hello-dis.s 使用的是十六进制的形式。
3. 数据访问: hello.s 中对 sleepsecs 等数据的访问使用的是其名称, 而 hello-dis.s 为其在.rela.text 节中添加重定位条目。

4.5 本章小结

本章介绍了 C 语言汇编的概念和作用。本章以可重定义文件 `hello.o` 为例，利用 `readelf` 工具获得 `hello.o` 的完整结构，分别详细介绍了 ELF 头，节头部表，重定位条目，符号表等具体部分的内容。也利用 `objdump` 工具获取到 `hello.o` 的反汇编文件，通过与 `hello.s` 的对比，介绍了反汇编文件与 `hello.s` 的不同。

第 5 章 链接

5.1 链接的概念与作用

概念：

链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。

作用：

链接使分离编译成为可能。当我们改变大型软件模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其他文件。

5.2 在 Ubuntu 下链接的命令

链接命令：

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

```
dedsec@ubuntu:~/CSAPP/project$ ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
dedsec@ubuntu:~/CSAPP/project$
```

图 11 Ubuntu下链接的命令

5.3 可执行目标文件 hello 的格式

我们可以使用命令 `Linux > readelf -a hello > hello-elf2.txt` 获得 hello 的完整结构，并将其保存在 hello-elf2.txt 中。

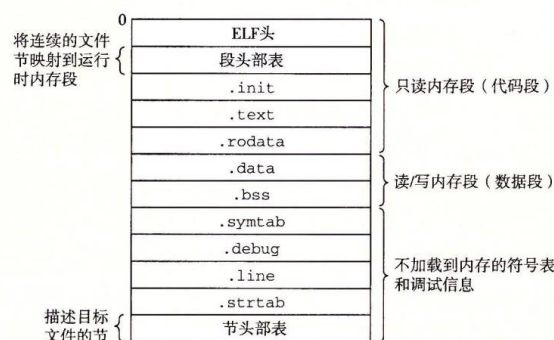


图 12 典型的ELF可执行目标文件

5.3.1 ELF 头

我们可以使用命令 `Linux > readelf -h hello > hello-h2.txt` 获得 ELF 头，并将其保存在 `hello-h2.txt` 中。ELF 头描述了生成该文件的系统的字的大小，字节顺序，ELF 头的大小，目标文件的类型，机器类型，节头部表的文件偏移，以及节头部表中条目的大小和数量等。

```
1 ELF Header:
2 Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
3 Class:                                ELF64
4 Data:                                  2's complement, little endian
5 Version:                              1 (current)
6 OS/ABI:                                UNIX - System V
7 ABI Version:                           0
8 Type:                                  EXEC (Executable file)
9 Machine:                               Advanced Micro Devices X86-64
10 Version:                              0x1
11 Entry point address:                   0x4010d0
12 Start of program headers:              64 (bytes into file)
13 Start of section headers:              14200 (bytes into file)
14 Flags:                                 0x0
15 Size of this header:                    64 (bytes)
16 Size of program headers:                56 (bytes)
17 Number of program headers:              12
18 Size of section headers:                64 (bytes)
19 Number of section headers:              27
20 Section header string table index: 26
```

图 13 hello的ELF头

5.3.2 节头部表

我们使用命令 `Linux > readelf -S hello > hello-sec2.txt` 获得节头部表，并将其保存在 `hello-sec2.txt` 中。节头部表描述了不同节的位置和大小。可以看到 `hello` 中节的数量比 `hello.o` 的多了很多，说明了在链接后有很多文件添加进来了。

```

1   There are 27 section headers, starting at offset 0x3778:
2
3   Section Headers:
4   [Nr] Name                Type              Address            Offset
5       Size                EntSize          Flags Link Info Align
6   [ 0]                      NULL             0000000000000000 00000000
7       0000000000000000 0000000000000000 0 0 0
8   [ 1] .interp                PROGBITS         00000000004002e0 000002e0
9       000000000000001c 0000000000000000 A 0 0 1
10  [ 2] .note.gnu.proptet      NOTE             0000000000400300 00000300
11       0000000000000020 0000000000000000 A 0 0 8
12  [ 3] .note.ABI-tag          NOTE             0000000000400320 00000320
13       0000000000000020 0000000000000000 A 0 0 4
14  [ 4] .hash                 HASH             0000000000400340 00000340
15       0000000000000034 0000000000000004 A 6 0 8
16  [ 5] .gnu.hash              GNU_HASH         0000000000400378 00000378
17       000000000000001c 0000000000000000 A 6 0 8
18  [ 6] .dynsym                DYNSYM           0000000000400398 00000398
19       00000000000000c0 0000000000000018 A 7 1 8
20  [ 7] .dynstr                STRTAB           0000000000400458 00000458
21       0000000000000057 0000000000000000 A 0 0 1
22  [ 8] .gnu.version           VERSYM           00000000004004b0 000004b0
23       0000000000000010 0000000000000002 A 6 0 2
24  [ 9] .gnu.version_r          VERNEED          00000000004004c0 000004c0
25       0000000000000020 0000000000000000 A 7 1 8
26  [10] .rela.dyn               RELA             00000000004004e0 000004e0
27       0000000000000030 0000000000000018 A 6 0 8
28  [11] .rela.plt              RELA             0000000000400510 00000510
29       0000000000000078 0000000000000018 AI 6 21 8
30  [12] .init                 PROGBITS         0000000000401000 00001000
31       000000000000001b 0000000000000000 AX 0 0 4
32  [13] .plt                    PROGBITS         0000000000401020 00001020
33       0000000000000060 0000000000000010 AX 0 0 16
34  [14] .plt.sec                PROGBITS         0000000000401080 00001080
35       0000000000000050 0000000000000010 AX 0 0 16
36  [15] .text                  PROGBITS         00000000004010d0 000010d0
37       0000000000000135 0000000000000000 AX 0 0 16
38  [16] .fini                   PROGBITS         0000000000401208 00001208
39       000000000000000d 0000000000000000 AX 0 0 4
40  [17] .rodata                 PROGBITS         0000000000402000 00002000
41       000000000000002f 0000000000000000 A 0 0 4
42  [18] .eh_frame               PROGBITS         0000000000402030 00002030
43       00000000000000fc 0000000000000000 A 0 0 8
44  [19] .dynamic                 DYNAMIC          0000000000403e50 00002e50
45       00000000000001a0 0000000000000010 WA 7 0 8
46  [20] .got                     PROGBITS         0000000000403ff0 00002ff0
47       0000000000000010 0000000000000008 WA 0 0 8
48  [21] .got.plt                 PROGBITS         0000000000404000 00003000
49       0000000000000040 0000000000000008 WA 0 0 8
50  [22] .data                    PROGBITS         0000000000404040 00003040
51       0000000000000008 0000000000000000 WA 0 0 4
52  [23] .comment                 PROGBITS         0000000000000000 00003048
53       000000000000002a 0000000000000001 MS 0 0 1
54  [24] .symtab                  SYMTAB           0000000000000000 00003078
55       000000000000004c8 0000000000000018 25 30 8
56  [25] .strtab                  STRTAB           0000000000000000 00003540
57       0000000000000150 0000000000000000 0 0 1
58  [26] .shstrtab                STRTAB           0000000000000000 00003690
59       00000000000000e1 0000000000000000 0 0 1
60
61   Key to Flags:
62   W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
63   L (link order), O (extra OS processing required), G (group), T (TLS),
64   C (compressed), x (unknown), o (OS specific), E (exclude),
65   l (large), p (processor specific)

```

图 14 hello的节头部表

5.3.3 符号表

我们可以使用命令 `Linux > readelf -s hello > hello-s2.txt` 获得符号表，并将其保存在 `hello-s2.txt` 中。符号表存放在程序中定义和应用的函数和全局变量的信息。可以看到 `hello` 中符号的数量比 `hello.o` 的多了很多，也说明了在链接后有很多文件

添加了进来。

```

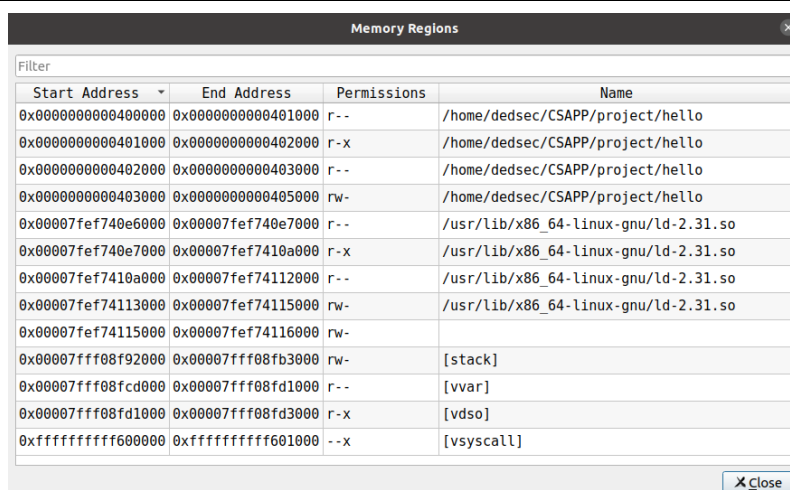
1
2 Symbol table '.dynsym' contains 8 entries:
3   Num:   Value                               Size Type Bind Vis Ndx Name
4   0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
5   1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.2.5 (2)
6   2: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.2.5 (2)
7   3: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
8   4: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getchar@GLIBC_2.2.5 (2)
9   5: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
10  6: 0000000000000000 0 FUNC GLOBAL DEFAULT UND exit@GLIBC_2.2.5 (2)
11  7: 0000000000000000 0 FUNC GLOBAL DEFAULT UND sleep@GLIBC_2.2.5 (2)
12
13 Symbol table '.symtab' contains 51 entries:
14   Num:   Value                               Size Type Bind Vis Ndx Name
15   0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
16   1: 00000000004002e0 0 SECTION LOCAL DEFAULT 1
17   2: 0000000000400300 0 SECTION LOCAL DEFAULT 2
18   3: 0000000000400320 0 SECTION LOCAL DEFAULT 3
19   4: 0000000000400340 0 SECTION LOCAL DEFAULT 4
20   5: 0000000000400378 0 SECTION LOCAL DEFAULT 5
21   6: 0000000000400398 0 SECTION LOCAL DEFAULT 6
22   7: 0000000000400458 0 SECTION LOCAL DEFAULT 7
23   8: 00000000004004b0 0 SECTION LOCAL DEFAULT 8
24   9: 00000000004004c0 0 SECTION LOCAL DEFAULT 9
25  10: 00000000004004e0 0 SECTION LOCAL DEFAULT 10
26  11: 0000000000400510 0 SECTION LOCAL DEFAULT 11
27  12: 0000000000401000 0 SECTION LOCAL DEFAULT 12
28  13: 0000000000401020 0 SECTION LOCAL DEFAULT 13
29  14: 0000000000401080 0 SECTION LOCAL DEFAULT 14
30  15: 00000000004010d0 0 SECTION LOCAL DEFAULT 15
31  16: 0000000000401208 0 SECTION LOCAL DEFAULT 16
32  17: 0000000000402000 0 SECTION LOCAL DEFAULT 17
33  18: 0000000000402030 0 SECTION LOCAL DEFAULT 18
34  19: 0000000000403e50 0 SECTION LOCAL DEFAULT 19
35  20: 0000000000403ff0 0 SECTION LOCAL DEFAULT 20
36  21: 0000000000404000 0 SECTION LOCAL DEFAULT 21
37  22: 0000000000404040 0 SECTION LOCAL DEFAULT 22
38  23: 0000000000000000 0 SECTION LOCAL DEFAULT 23
39  24: 0000000000000000 0 FILE LOCAL DEFAULT ABS hello.c
40  25: 0000000000000000 0 FILE LOCAL DEFAULT ABS
41  26: 0000000000403e50 0 NOTYPE LOCAL DEFAULT 19 __init_array_end
42  27: 0000000000403e50 0 OBJECT LOCAL DEFAULT 19 __DYNAMIC
43  28: 0000000000403e50 0 NOTYPE LOCAL DEFAULT 19 __init_array_start
44  29: 0000000000404000 0 OBJECT LOCAL DEFAULT 21 _GLOBAL_OFFSET_TABLE_
45  30: 0000000000401200 5 FUNC GLOBAL DEFAULT 15 __libc_csu_fini
46  31: 0000000000404040 0 NOTYPE WEAK DEFAULT 22 data_start
47  32: 0000000000000000 0 FUNC GLOBAL DEFAULT UND puts@@GLIBC_2.2.5
48  33: 0000000000404044 4 OBJECT GLOBAL DEFAULT 22 sleepsecs
49  34: 0000000000404048 0 NOTYPE GLOBAL DEFAULT 22 _edata
50  35: 0000000000401208 0 FUNC GLOBAL HIDDEN 16 _fini
51  36: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@@GLIBC_2.2.5
52  37: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@@GLIBC_
53  38: 0000000000404040 0 NOTYPE GLOBAL DEFAULT 22 __data_start
54  39: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getchar@@GLIBC_2.2.5
55  40: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
56  41: 0000000000402000 4 OBJECT GLOBAL DEFAULT 17 _IO_stdin_used
57  42: 0000000000401190 101 FUNC GLOBAL DEFAULT 15 __libc_csu_init
58  43: 0000000000404048 0 NOTYPE GLOBAL DEFAULT 22 _end
59  44: 0000000000401100 5 FUNC GLOBAL HIDDEN 15 _dl_relocate_static_pie
60  45: 00000000004010d0 47 FUNC GLOBAL DEFAULT 15 _start
61  46: 0000000000404048 0 NOTYPE GLOBAL DEFAULT 22 __bss_start
62  47: 0000000000401105 133 FUNC GLOBAL DEFAULT 15 main
63  48: 0000000000000000 0 FUNC GLOBAL DEFAULT UND exit@@GLIBC_2.2.5
64  49: 0000000000000000 0 FUNC GLOBAL DEFAULT UND sleep@@GLIBC_2.2.5
65  50: 0000000000401000 0 FUNC GLOBAL HIDDEN 12 _init

```

图 15 hello的符号表

5.4 hello 的虚拟地址空间

使用 edb 加载 hello，通过 Memory Regions 查看 hello 的虚拟地址空间各段信息。



Start Address	End Address	Permissions	Name
0x0000000000400000	0x0000000000401000	r--	/home/dedsec/CSAPP/project/hello
0x0000000000401000	0x0000000000402000	r-x	/home/dedsec/CSAPP/project/hello
0x0000000000402000	0x0000000000403000	r--	/home/dedsec/CSAPP/project/hello
0x0000000000403000	0x0000000000405000	rw-	/home/dedsec/CSAPP/project/hello
0x00007fef740e6000	0x00007fef740e7000	r--	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007fef740e7000	0x00007fef7410a000	r-x	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007fef7410a000	0x00007fef74112000	r--	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007fef74113000	0x00007fef74115000	rw-	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007fef74115000	0x00007fef74116000	rw-	
0x00007fff08f92000	0x00007fff08fb3000	rw-	[stack]
0x00007fff08fcd000	0x00007fff08fd1000	r--	[vvar]
0x00007fff08fd1000	0x00007fff08fd3000	r-x	[vdso]
0xfffffffff6000000	0xfffffffff6010000	--x	[vsyscall]

图 16 虚拟地址空间各段信息

由 5.3.2 节的节头部表可得到各个节的地址，与上图对照可得到各段中包含的节：

1. 0x400000-0x401000 : .interp ; .note.gnu.proptert ; .note.ABI-tag; .hash; .gnu.hash; .dynsym; .dynstr; .gnu.version; .gnu.version_r; .rela.dyn; .rela.plt。上述节在 0x400000-0x401000 中，且只能读不能写或执行。
2. 0x401000-0x402000: .init; .plt; .plt.sec; .text; .fini。上述节都在 0x401000-0x402000 中，能读和执行，不能写。
3. 0x402000-0x403000: .rodata; .eh_frame。上述节在 0x402000-0x403000 中，只能读不能写或执行。0x400000-0x403000 对应下图中的只读代码段。
4. 0x403000-0x405000: .dynamic; .got; .got.plt; .data。上述节在 0x403000-0x405000 中，可以读写，不能执行。0x403000-0x405000 对应下图中的读/写段。

0x400000-0x405000 也就是由 hello 加载映射到的内存区域。剩下的段要么是贡献库的内存映射区域，要么是运行时的栈或者堆等内容。

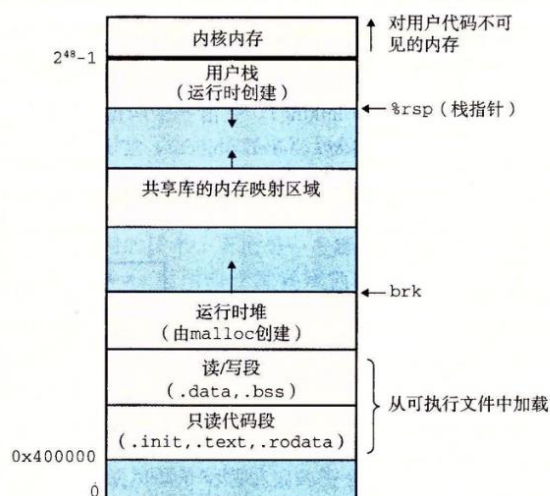


图 17 Linux x86-64运行时内存映像

5.5 链接的重定位过程分析

使用命令 `Linux > objdump -d -r hello > hello-dis2.s` 获取 `hello` 的反汇编文件。

通过 `hello-dis.s` 和 `hello-dis2.s` 我们可以分析 `hello.o` 和 `hello` 的不同：

1. 增加了新的节：在 `hello` 中增加了 `.init` 节、`.plt` 节、`.plt.sec` 节和 `.fini` 节。
2. 加入了新的函数：在 `hello` 中多了引用的库函数，`exit`、`printf`、`sleep`、`getchar` 等。
3. 填充了重定位条目：在 `hello` 中的跳转（`jmp`）和调用（`call`）命令的操作数地址都被填充为虚拟内存地址；在 `hello` 中对 `sleepsecs` 等全局变量的访问变成了虚拟内存地址。

链接的过程：

1. 符号解析：我们在代码中会声明变量及函数，之后会调用变量及函数，所有的符号声明都会被保存在符号表中，而符号表会保存在由汇编器生成的 `object` 文件中。符号表实际上是一个结构体数组，每一个元素包含名称、大小和符号的位置。
2. 重定位：这一步所做的工作是把原先分开的代码和数据片段汇总成一个文件，会把原先在 `.o` 文件中的相对位置转换成在可执行程序中的绝对位置，并且据此更新对应的引用符号（才能找到新的位置）。

5.6 `hello` 的执行流程

地址	函数名
----	-----

0x00000000004010d0	__start
0x00007f43d72d7fc0	__libc_start_main
0x00007f43d72faf60	__cxa_atexit
0x0000000000401190	__libc_csu_init
0x0000000000401000	_init
0x00007f43d72f6e00	_setjmp
0x00007f43d72f6d30	__sigsetjmp
0x0000000000401105	main
0x0000000000401030	puts
0x0000000000401060	exit
0x00007f43d72fabe0	on_exit
0x0000000000401200	__libc_csu_fini
0x0000000000401208	_fini

5.7 Hello 的动态链接分析

在 hello-sec2.txt 中我们可以找到.got 节和.got.plt 节的位置：

```
[20] .got          PROGBITS          0000000000403ff0  00002ff0
      0000000000000010 0000000000000008  WA      0      0      8
[21] .got.plt      PROGBITS          0000000000404000  00003000
      0000000000000040 0000000000000008  WA      0      0      8
```

初始时每个 GOT 条目都指向对应 PLT 条目的第二条指令。

00000000:00403fc0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00403fd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00403fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00403ff0	c0 3f 17 92 62 7f 00 00 00 00 00 00 00 00 00 00	[]?..b.....
00000000:00404000	50 3e 40 00 00 00 00 00 00 00 90 41 38 92 62 7f 00 00	P>@.....A8.b...
00000000:00404010	b0 db 36 92 62 7f 00 00 30 10 40 00 00 00 00 00	z_6.b...0.@....
00000000:00404020	40 10 40 00 00 00 00 00 50 10 40 00 00 00 00 00	@.@.....P.@....
00000000:00404030	60 10 40 00 00 00 00 00 70 10 40 00 00 00 00 00	'.@.....p.@....
00000000:00404040	00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62GCC: (Ub
00000000:00404050	75 6e 74 75 20 39 2e 33 2e 30 2d 31 37 75 62 75	untu 9.3.0-17ubu

图 18 库函数被调用前

当库函数被调用后，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。

00000000:00403fc0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00403fd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00403fe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00403ff0	c0 9f e6 77 3b 7f 00 00 00 00 00 00 00 00 00 00	[]?w;.....
00000000:00404000	50 3e 40 00 00 00 00 00 90 a1 07 78 3b 7f 00 00	P>@.....X;...
00000000:00404010	b0 3b 06 78 3b 7f 00 00 a0 a5 ec 77 3b 7f 00 00	z;.x;...u w;...
00000000:00404020	40 10 40 00 00 00 00 00 50 10 40 00 00 00 00 00	@.@.....P.@....
00000000:00404030	60 10 40 00 00 00 00 00 70 10 40 00 00 00 00 00	'.@.....p.@....
00000000:00404040	00 00 00 00 02 00 00 00 47 43 43 3a 20 28 55 62GCC: (Ub
00000000:00404050	75 6e 74 75 20 39 2e 33 2e 30 2d 31 37 75 62 75	untu 9.3.0-17ubu

图 19 库函数被调用后

5.8 本章小结

本章介绍了链接的概念及作用。本章以 `hello.o` 程序为例，将可重定位目标文件 `hello.o` 与动态库链接生成可执行目标文件 `hello`，并利用 `readelf` 来获取 `hello` 的 ELF 头、节头部表等部分的内容，深入分析了各部分的组成。还利用 `edb` 查看了 `hello` 的虚拟地址空间，并对 `hello` 的执行过程和动态连接过程进行了分析。

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：

进程就是执行中的程序的实例。

作用：

进程为用户提供了以下假象，我们的程序好像是系统中当前运行的唯一程序，我们的程序好像是独占地使用处理器和内存，处理器就好像是无间断地执行指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

进程提供给程序抽象：独立的逻辑控制流，每个程序似乎独占 CPU；私有的空间地址，每个程序似乎独占内存。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：

Shell 是一个命令处理器，通常运行于文本窗口中，并能执行用户直接输入的命令。Shell 还能从文件中读取命令，这样的文件称为脚本。它支持文件名替换（通配符匹配）、管道、here 文档、命令替换、变量，以及条件判断和循环遍历的结构控制语句。

处理流程：

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。
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。
9. Shell 执行通配符*?[]的替换。
10. Shell 把所有处理的结果中用到的注释删除，并且按照下面的顺序实行命令的

检查:

- a) 内建的命令
- b) shell 函数 (由用户自己定义的)
- c) 可执行的脚本文件 (需要寻找文件和 PATH 路径)

11. 初始化所有的输入输出重定向。

12. 执行命令。

6.3 Hello 的 fork 进程创建过程

创建过程:

1. 给新进程分配一个标识符
2. 在内核中分配一个 PCB, 将其挂在 PCB 表上
3. 复制它的父进程的环境 (PCB 中大部分的内容)
4. 为其分配资源 (程序、数据、栈等)
5. 复制父进程地址空间里的内容 (代码共享, 数据写时拷贝)
6. 将进程置成就绪状态, 并将其放入就绪队列, 等待 CPU 调度。

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。父进程和子进程之间最大的区别在于它们有不同的 PID。

6.4 Hello 的 execve 过程

execve 函数加载并运行 hello, 且带参数列表 argv 和环境变量列表 envp。在 execve 加载了 hello 之后, 它调用启动代码。启动代码设置栈, 并将控制传递给程序的主函数, 该主函数有如下形式的原型 `int main(int argc, char **argv, char **envp);`

当 main 开始执行时, 用户栈的组织结构如下图所示。

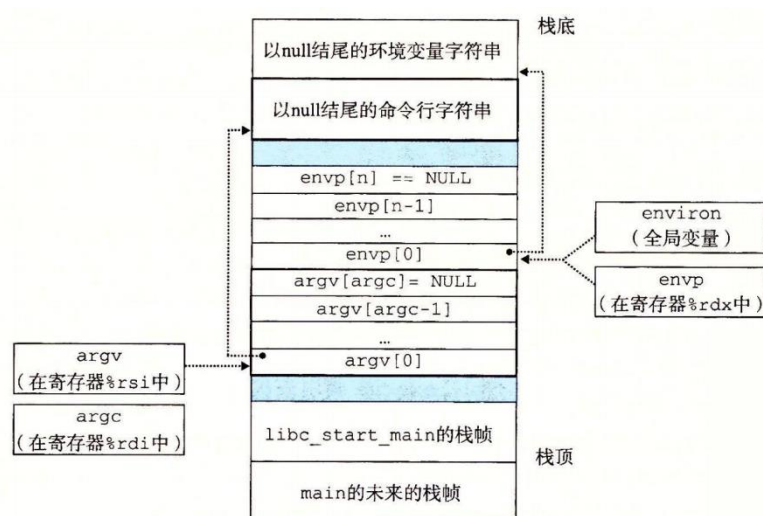


图 20 一个新程序开始时，用户栈的典型组织结构

6.5 Hello 的进程执行

进程的上下文信息：上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

进程时间片：一个进程执行它的控制流的一部分的每一时间段叫做时间片。

用户模式和内核模式：为了使操作系统内核提供一个无懈可击的进程抽象，处理器必须提供一种机制，限制一个应用可以执行的指令以及它可以访问的地址空间范围。处理器通常是用某个控制寄存器中的一个模式位来提供这种功能的，该寄存器描述了进程当前享有的特权。当设置了模式位时，进程就运行在内核模式中。一个运行在内核模式的进程可以执行指令集中的任何指令，并且可以访问系统中的任何内存位置。没有设置模式位时，进程就运行在用户模式中。用户模式中的进程不允许执行特权指令，比如停止处理器、改变模式位，或者发起一个 I/O 操作。也不允许用户模式中的进程直接引用地址空间中内核区内的代码和数据。任何这样的尝试都会导致致命的保护故障。

上下文切换与进程的调度：在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程。这种决策就叫做调度。在内核调度了一个新的进程运行后，它就抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程，上下文切换 1) 保存当前进程的上下文，2) 恢复某个先前被抢占的进程被保存的上下文，3) 将控制传递给这个新恢复的进程。

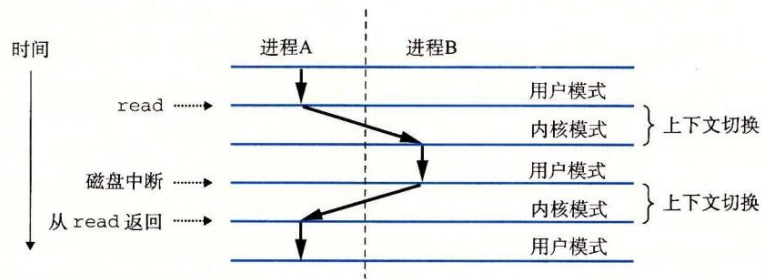


图 21 进程的上下文切换

6.6 hello 的异常与信号处理

hello 执行过程中会出现异常的类型如下图所示。

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

图 22 异常的分类

异常的处理方式如下图所示。

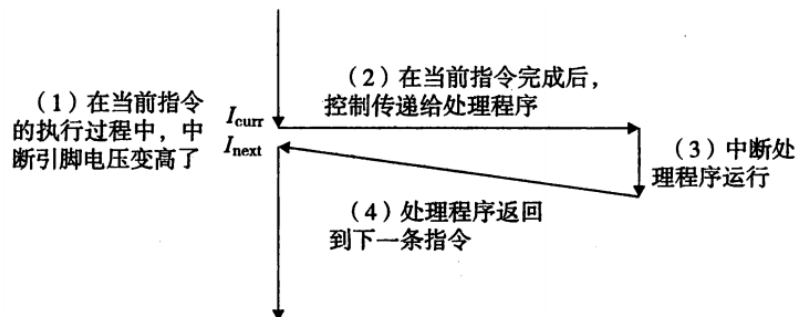


图 23 中断处理

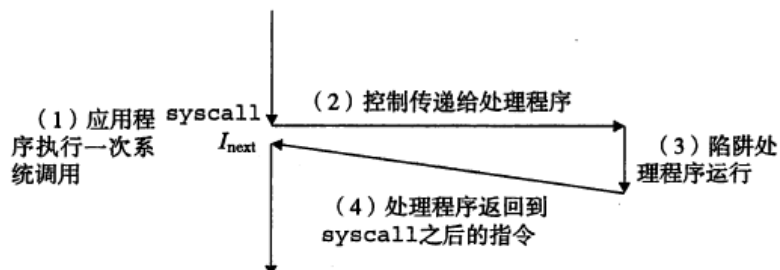


图 24 陷阱处理

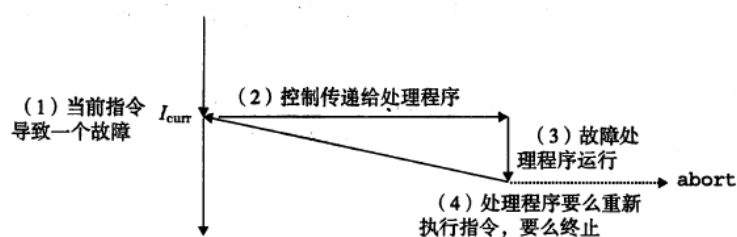


图 25 故障处理

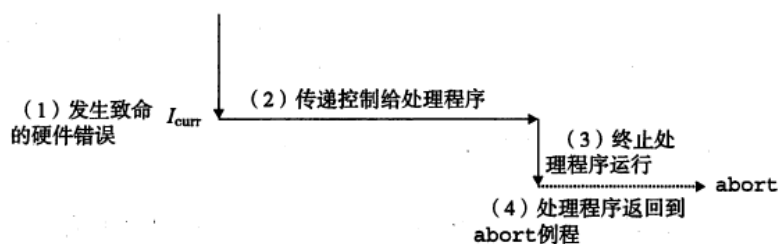


图 26 终止处理

在 hello 运行过程中不停乱按，输入只是被写入缓存区，待 hello 结束后被解析为命令，如下图所示。

```

dedsec@ubuntu:~/CSAPP/project$ ./hello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy
erydfg
dydgfHello 1190200523 sxy
kyu
hdrhjguk
Hello 1190200523 sxy
sfhftjk
dgrrdHello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy

Hello 1190200523 sxy
Hello 1190200523 sxy
dedsec@ubuntu:~/CSAPP/project$ dydgfkyu
dydgfkyu: command not found
dedsec@ubuntu:~/CSAPP/project$ hdrhjguk
hdrhjguk: command not found
dedsec@ubuntu:~/CSAPP/project$ sfhftjk
sfhftjk: command not found
dedsec@ubuntu:~/CSAPP/project$ dgrrd

Command 'dgrrd' not found, did you mean:

  command 'dgord' from deb ptscotch (6.0.9-1)

Try: sudo apt install <deb name>
dedsec@ubuntu:~/CSAPP/project$
  
```

图 27 hello运行过程中不停乱按

按下 Ctrl+Z 后，进程会收到 SIGSTP 信号，进程会被挂起。可以通过 ps、jobs 命令查看其状态，使用 fg 命令将其调回前台；通过 pstree 命令以树状图显示进程间的关系


```
dedsec@ubuntu:~/CSAPP/project$ ./hello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy
^C
dedsec@ubuntu:~/CSAPP/project$ ps
  PID TTY          TIME CMD
  91754 pts/0    00:00:00 bash
 100009 pts/0    00:00:00 ps
dedsec@ubuntu:~/CSAPP/project$ jobs
dedsec@ubuntu:~/CSAPP/project$
```

图 30 hello收到SIGINT信号

按下 Ctrl+Z 后运行 kill 命令，进程被终止，如下图所示。

```
dedsec@ubuntu:~/CSAPP/project$ ./hello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy
Hello 1190200523 sxy
^Z
[1]+  Stopped                  ./hello 1190200523 sxy
dedsec@ubuntu:~/CSAPP/project$ ps
  PID TTY          TIME CMD
  91754 pts/0    00:00:00 bash
 100302 pts/0    00:00:00 hello
 100303 pts/0    00:00:00 ps
dedsec@ubuntu:~/CSAPP/project$ kill -9 100302
dedsec@ubuntu:~/CSAPP/project$ ps
  PID TTY          TIME CMD
  91754 pts/0    00:00:00 bash
 100304 pts/0    00:00:00 ps
[1]+  Killed                   ./hello 1190200523 sxy
dedsec@ubuntu:~/CSAPP/project$
```

图 31 按下Ctrl+Z后运行kill命令

6.7 本章小结

本章介绍了进程的概念与作用和 Shell 的作用与处理流程。本章以 hello 为例，介绍了 fork 和 execve 的执行具体过程，也从进程时间片、上下文切换用户模式和内核模式多角度全方面介绍了 hello 程序执行时的调度问题。本章还涉及了异常与信号处理，介绍了四种异常及其相应的处理方式；也展示了向进程发送不同的信号后进程的行为。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：指由 hello 程序产生的段内偏移。

线性地址：指虚拟地址到物理地址变换的中间层，是处理器可寻址的内存空间中的地址。hello 程序代码会产生逻辑地址，也就是段中的偏移地址，加上相应的段基址就成了线性地址。

虚拟地址：指由 hello 程序产生的由段选择符和段内偏移地址组成的地址。

物理地址：指 CPU 外部地址总线上寻址物理内存的地址信号，是地址变换的最终结果。

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：段标识符和段内偏移量。段标识符是一个 16 位的字段组成，称为段选择符，其中前 13 位是一个索引号，后面 3 位包含一些硬件细节，它用来从段描述符表中选择一个具体的段，某个段描述符表项的 base 字段描述了一个段的开始位置的线性地址。程序过来一个逻辑地址，使用其段选择符的 Index 字段去索引段描述符表，若 TI=0，索引全局段描述符表，TI=1，索引局部段描述符表，表的地址在相应的寄存器中。通过 Index 字段和段描述符表的位置能找到某项具体的段描述符。将段描述符中的 base 字段和逻辑地址中的 offset 字段合并即得到了线性地址。

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。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。整个线性地址就被划分为一个大数组，共有 2^{20} 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。

这里注意到，这个数组有 2^{20} 个成员，每个成员是一个地址，那么要单单要表示这么一个数组，就要占去 4MB 的内存空间。为了节省空间，引入了多级管理模式的机器来组织分页单元。

变换：

1. 分页单元中页目录是唯一的，它的地址放在 CPU 的 CR3 寄存器中，是进行地

址转换的开始点。

2. 每一个活动的进程因为都有其独立的对应的虚拟内存，那么它也对应了一个独立的页目录地址。
3. 每一个 32 位的线性地址被划分为三部分，页目录索引（10 位）、页表索引（10 位）、偏移（12 位）。
4. 依据以下步骤进行转换：
 - a) 从 CR3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；
 - b) 根据线性地址前十位在数组中找到对应的索引项，因为引入了二级管理模式，页目录中的项不再是页的地址而是一个页表的地址。
 - c) 根据线性地址的中间十位，在页表中找到页的起始地址；
 - d) 将页的起始地址与线性地址中最后 12 位拼接，得到最终的物理地址。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

Intel Core i7 的虚拟地址（VA）有 48 位，物理地址（PA）有 52 位，虚拟地址的 0~11 位是 VPO，12~47 位是 VPN。

CPU 产生一个虚拟地址。开始时，MMU 从虚拟地址中抽出 VPN，并检查 TLB，看它是否因为前面的某个内存引用缓存了 VPN 对应的 PTE 的一个副本。若命中，则 TLB 从 VPN 抽取出 TLB 索引和 TLB 标记，然后将缓存的 PPN 返回给 MMU。

若不命中，则 36 位的 VPN 被划分为四个 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。VPN1 提供一个 L1 PTE 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN2 提供一个 L2 PTE 的偏移量，以此类推。最后在四级页表中找到目标页表条目，这个页表条目中存储的就是 PPN。将 PPN 和 PPO 拼接起来就得到了物理地址。

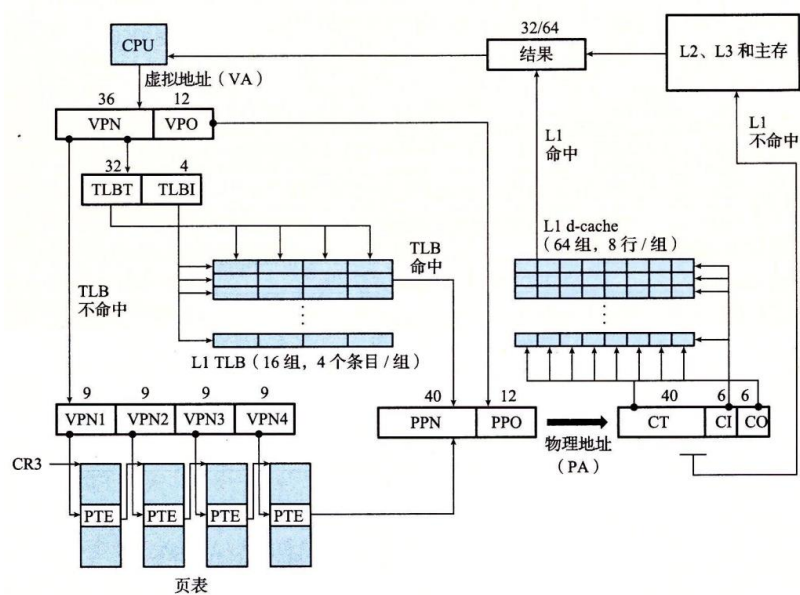


图 32 Core i7地址翻译的概况

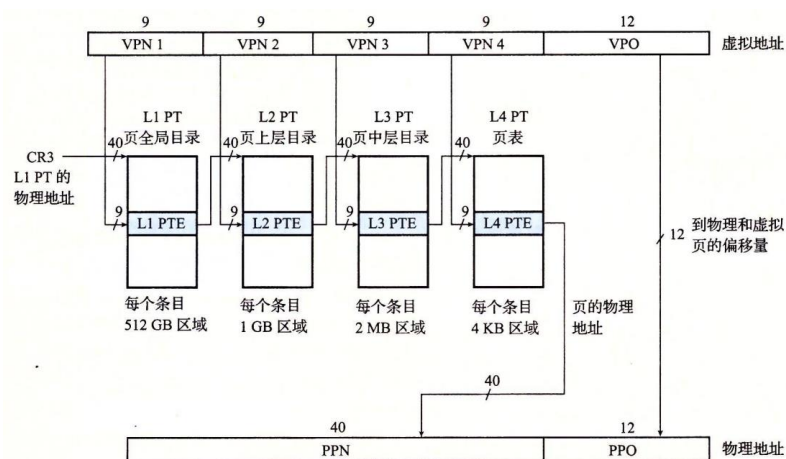


图 33 Core i7页表翻译

7.5 三级 Cache 支持下的物理内存访问

经过上述过程我们得到了物理地址 PA。访问步骤如下：

1. 我们将地址 PA 由高至低划分为四个部分：标签、索引、块内偏移、字节偏移。
2. 用索引定位到相应的缓存块。
3. 用标签尝试匹配该缓存块的对应标签值。如果存在这样的匹配，称为命中；否则称为未命中。
4. 如命中，用块内偏移将已定位缓存块内的特定数据段取出，送回处理器。

5. 如未命中，先用此块地址（标签+索引）从下一级缓存或内存中读取数据并载入到当前缓存块，再用块内偏移将位于此块内的特定数据单元取出，送回处理器。

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。为了给这个新进程创建虚拟内存，它创建了当前进程的 mm_struct、区域结构和页表的原样副本。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。当这两个进程中的人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地址空间的抽象概念。

7.7 hello 进程 execve 时的内存映射

执行 execve 调用加载并运行新程序需要以下几个步骤：

1. 删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。
2. 映射私有区域。为新程序的代码、数据、bss 和栈区域创建新的区域结构。
3. 映射共享区域。
4. 设置程序计数器（PC）。使之指向代码区域的入口点。

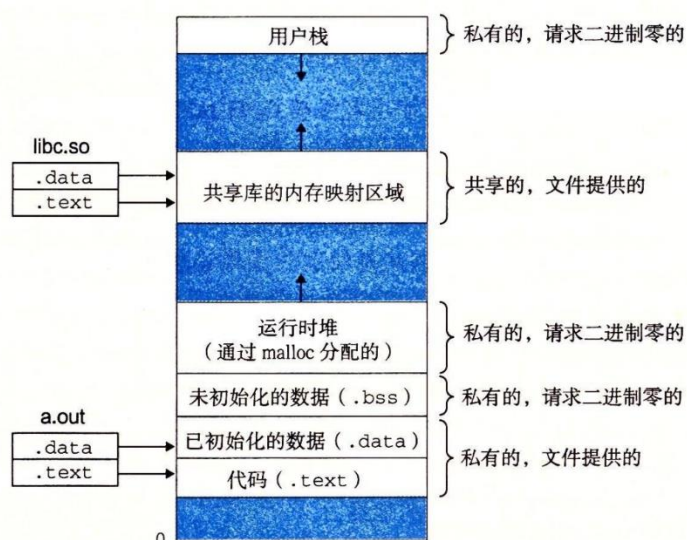


图 34 加载器是如何映射用户地址空间的区域的

7.8 缺页故障与缺页中断处理

处理缺页要求硬件和操作系统内核协作完成，步骤如下：

1. 处理器生成一个虚拟地址，并把它传送给 MMU。
2. MMU 生成 PTE 地址，并从高速缓存/主存请求得到它。
3. PTE 中的有效位是零，所以 MMU 触发了一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。
4. 缺页处理程序确定物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。
5. 缺页处理程序页面调入新的页面，并更新内存中的 PTE。
6. 缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU 将引起缺页的虚拟地址重新发给 MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中，在 MMU 执行了页面命中的操作步骤之后，主存就会将所请求字返回给处理器。

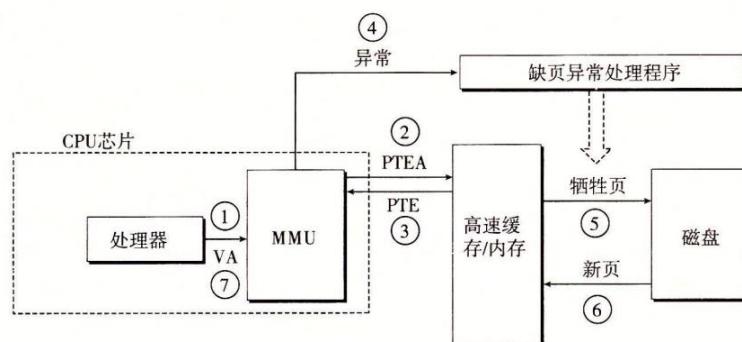


图 35 缺页的处理

7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。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位供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格：显式分配器和隐式分配器。两种分配器都要求应用程序显式地分配块。不同之处在于显式分配器要求应用显式地释放任何已分配的块。我们将主要介绍显式分配器。

隐式空闲链表：

任何实际的分配器都需要一些数据结构，允许它来区别块边界，以及区别已分配块和空闲块。大多数分配器将这些信息嵌入块本身。一个简单的方法如下图所示。

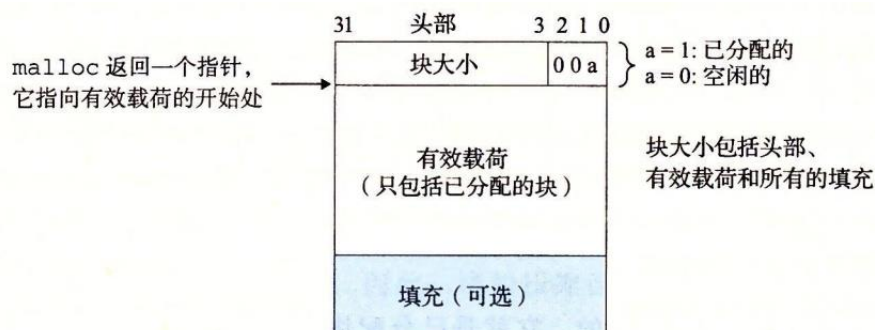


图 36 一个简单的堆块的格式

一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是零。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况下，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。

我们称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。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。

分配器利用首次适配、下一次适配或最佳适配等适配策略来搜索空闲链表并放置已分配的块。分配器使用立即合并或者推迟合并的合并策略来合并相邻的空闲块，以解决假碎片的问题。

显式空闲链表：

另一种方法将空闲块组织成显式空闲链表。我们把堆组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针，如下图所示。

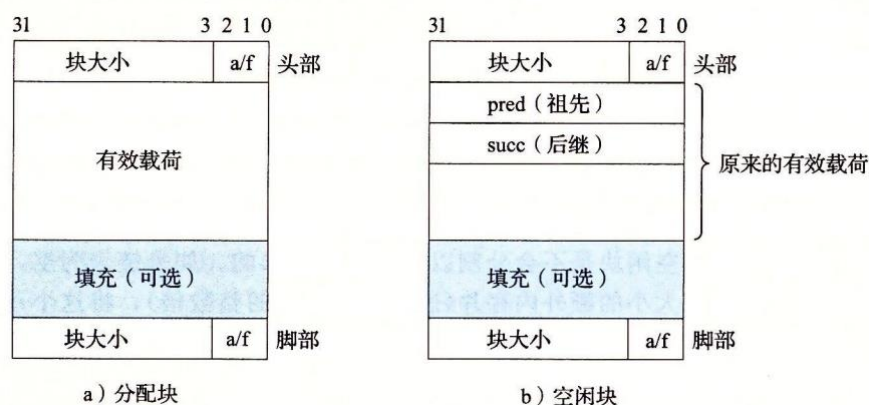


图 37 使用双向空闲链表的堆块的格式

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于我们所选择的空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是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后继的地址。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。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

分离的空闲链表：

一种流行的减少分配时间的方法，通常称为分离存储，就是维护多个空闲链表，其中每个链表中的块有大致相等的大小。一般的思路是将所有可能的块大小分成一些等价类，也叫做大小类。分配器维护着一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列。当分配器需要一个大小为 n 的块时，它就搜索相应的空闲链表。如果不能找到合适的块与之匹配，它就搜索下一个链表，以此类推。两种基本的分离存储方法：简单分离存储和分离适配。

7.10 本章小结

本章主要介绍了虚拟内存、存储器的一些知识。本章以 hello 程序为例，介绍了程序中逻辑地址到线性地址，进而到物理地址的转换，以及在 TLB 和四级页表支持下的虚拟地址到物理地址的转换，进而在三级 cache 支持下的物理内存的访问。还介绍了 fork 和 execve 执行时内存是如何被映射的。最后从隐式空闲链表和显式空闲链表两个角度介绍了动态存储分配的管理机制。

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（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。

设备管理：

Unix I/O 接口。将设备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。

8.2 简述 Unix IO 接口及其函数

Unix IO 接口

1. 打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 I/O 设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录有关这个打开文件的所有信息。应用程序只需记住这个描述符。
2. Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）。头文件 `<unistd.h>` 定义了常量 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，它们可用来代替显式的描述符值。
3. 改变当前的文件位置。对于每个打开的文件，内核保持着一个文件位置 `k`，初始为 0。这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行 `seek` 操作，显式地设置文件的当前位置为 `k`。
4. 读写文件。一个读操作就是从文件复制 `n > 0` 个字节到内存，从当前文件位置 `k` 开始，然后将 `k` 增加到 `k+n`。给定一个大小为 `m` 字节的文件，当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”。
5. 关闭文件。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

Unix IO 接口函数

1. **open 函数**: `int open(char *filename, int flags, mode_t mode);`
 open 函数将 filename 转换为一个文件描述符，并返回描述符数字，若成功则为新文件描述符，若出错为-1。返回的描述符总是在进程中当前没有打开的最小描述符。
2. **close 函数**: `int close(int fd);`
 通过调用 close 函数关闭一个打开的文件，若成功返回 0，否则为-1。注意关闭一个已关闭的描述符会出错。
3. **read 函数**: `ssize_t read(int fd, void *buf, size_t n);`
 read 函数从描述符 fd 的当前文件位置复制最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，返回值 0 表示 EOF。否则返回值表示的是实际传送的字节数量。
4. **write 函数**: `ssize_t write(int fd, const void *buf, size_t n);`
 write 函数从内存位置 buf 复制至多 n 个字节到描述符 fd 的当前文件位置。若成功则返回写的字节数，若出错则返回-1。

8.3 printf 的实现分析

printf 函数如下所示：

```
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;
}
```

printf 函数中主要引用了 vsprintf 函数，该函数如下所示：

```
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 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。

`printf` 函数中还调用了系统函数 `write`。表示写操作，把 `buf` 中的 `i` 个元素的值写到终端。

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()` 函数的作用是从标准的输入 `stdin` 中读取字符。也就是说，`getchar()` 函数以字符为单位对输入的数据进行读取。在控制台中通过键盘输入数据时，以回车键作为结束标志。当输入结束后，键盘输入的数据连同回车键一起被输入到输入缓冲区中。在程序中第一次调用 `getchar()` 函数从输入缓冲区中读取一个字节的的数据。需要注意的是，如果此时在程序中第二次调用 `getchar()` 函数，因为此时输入缓冲区中还有回车键的数据没有被读出，第二个 `getchar()` 函数读出的是回车符。

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 Unix I/O 的概念及其实现方式，还有 I/O 相关的一些函数。本章还借助代码深入分析了 `printf` 函数和 `getchar` 函数的实现方法。

结论

hello 程序的生命周期从源文件 `hello.c` 开始：

1. `hello.c` 通过预处理变成预处理后的 `hello.i`;
2. `hello.i` 利用编译程序,通过对代码的语法和语义的分析,生成汇编文件 `hello.s`;
3. `hello.s` 通过相应的汇编程序被转换成可执行的机器代码形成可重定位目标文件 `hello.o`;
4. `hello.o` 通过与系统库进行链接,生成了可执行文件 `hello`;
5. `shell` 通过 `fork` 函数,加载 `hello` 程序,创建各种数据结构,并分配给它一个唯一的 PID,给新进程 `hello` 创建虚拟内存;
6. `hello` 通过多级页表、TLB 等机制实现从虚拟内存到物理内存的转变,与计算机存储器进行交互;
7. `hello` 通过 Unix I/O 提供的函数实现与 I/O 设备的交互;
8. `hello` 运行结束,被 `shell` 回收,与其相关的资源被回收。

《深入理解计算机系统》这本书先从数据的表示开始,自然而然引出用汇编描述的计算机基本运行的方式,接着介绍控制流、过程调用到跳转等高级话题,以了解了计算机运行的模式。这之后开始涉及诸如内存、编译器、进程、信号、I/O、虚拟内存、动态内存分配等知识。这些东西看起来都非常吓人,但是书中很巧妙地掌握好了度,让我们理解基本原理的同时却不会过早陷入无谓的复杂度,这样以后想要深入学习可以自己探索,为后面的学习打下了坚实的基础。这本书真可谓“比同等质量的黄金更加珍贵”!

附件

文件名	文件作用
hello.i	hello.c 经过预处理后的文件
hello.s	hello.i 经过编译后的汇编文件
hello.o	hello.s 经过汇编后的可重定位目标文件
hello	hello.o 经过链接之后的可执行目标文件
hello-elf.txt	hello.o 的 ELF 格式
hello-h.txt	hello.o 的 ELF 格式的 ELF 头
hello-r.txt	hello.o 的 ELF 格式的重定位条目
hello-s.txt	hello.o 的 ELF 格式的符号表
hello-sec.txt	hello.o 的 ELF 格式的节头部表
hello-dis.s	hello.o 的反汇编文件
hello-elf2.txt	hello 的 ELF 格式
hello-h2.txt	hello 的 ELF 格式的 ELF 头
hello-r2.txt	hello 的 ELF 格式的重定位条目
hello-s2.txt	hello 的 ELF 格式的符号表
hello-sec2.txt	hello 的 ELF 格式的节头部表
hello-dis2.s	hello 的反汇编文件

参考文献

- [1] Randal E. Bryant, David R. O'Hallaron. 深入理解计算机系统. 第三版. 北京: 机械工业出版社[M]. 2018: 1-737
- [2] Dedsecr's Blog. <https://dedsecr.top>
- [3] C++预处理详解. <https://www.cnblogs.com/liangliangh/p/3585326.html>
- [4] 编译原理学习基本概念汇总 . <https://blog.csdn.net/CHENYUFENG1991/article/details/47002525>
- [5] ELF 格式详解 (一) . <https://zhuanlan.zhihu.com/p/73114831>
- [6] readelf elf 文件格式分析 . https://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/readelf.html
- [7] 计算机原理系列之七 —— 链接过程分析 . <https://zhuanlan.zhihu.com/p/52765090>
- [8] Introduction to CSAPP (十五): 过程调用指令与 C 语言函数调用 . <https://zhuanlan.zhihu.com/p/102050997>
- [9] Linux 内核分析 (七) 系统调用 `execve` 处理过程 . <https://blog.csdn.net/yubo112002/article/details/82527157>
- [10] SIGTERM 、 SIGKILL 、 SIGINT 和 SIGQUIT 的区别 . https://blog.csdn.net/dai_xiangjun/article/details/41871647
- [11] linux 下的 context. <https://www.jianshu.com/p/dae774893a30>
- [12] The Memory Layout of a 64-bit Linux Process. <https://simonis.github.io/Memory/>
- [13] Understanding the Memory Layout of Linux Executables. <https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4>
- [14] 虚拟地址、逻辑地址、线性地址、物理地址 . <https://www.jianshu.com/p/8b37d10bc504>
- [15] LINUX 逻辑地址、线性地址、虚拟地址和物理地址 . https://blog.csdn.net/baidu_35679960/article/details/80463445
- [16] CPU 缓存维基百科, 自由的百科全书 . <https://zh.wikipedia.org/wiki/CPU%E7%BC%93%E5%AD%98>