哈爾濱工業大學

计算机系统

大作业

题　　目　程序人生-Hello’s P2P

专　　业　计算机系

学　　号　120L010911

班　　级　2003002

学　　生　王梓尧

指 导 教 师　史先俊

计算机科学与技术学院

2022 年 5 月

摘　要

本文通过对 hello 在 Linux 从编译到运行的各个环节进行较为深入的分析，结合《深入理解计算机系统》中的内容，运用 gcc edb 等工具进行实践，展示了对各部分的理解。

关键词：CS:APP 计算机系统 汇编 编译 进程 C 语言

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

第 1 章 概述 ..... - 4 -

1.1 HELLO 简介 ..... - 4 -
1.2 环境与工具 ..... - 5 -
1.3 中间结果 ..... - 6 -
1.4 本章小结 ..... - 6 -

第 2 章 预处理 ..... - 7 -

2.1 预处理的概念与作用 ..... - 7 -
2.2 在 UBUNTU 下预处理的命令 ..... - 7 -
2.3 HELLO 的预处理结果解析 ..... - 7 -
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 -
3.4 本章小结 ..... - 12 -

第 4 章 汇编 ..... - 13 -

4.1 汇编的概念与作用 ..... - 13 -
4.2 在 UBUNTU 下汇编的命令 ..... - 13 -
4.3 可重定位目标 ELF 格式 ..... - 13 -
4.4 HELLO.O 的结果解析 ..... - 16 -
4.5 本章小结 ..... - 16 -

第 5 章 链接 ..... - 18 -

5.1 链接的概念与作用 ..... - 18 -
5.2 在 UBUNTU 下链接的命令 ..... - 18 -
5.3 可执行目标文件 HELLO 的格式 ..... - 18 -
5.4 HELLO 的虚拟地址空间 ..... - 22 -
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 ..... - 24 -

第 6 章 HELLO 进程管理 ..... - 25 -

6.1 进程的概念与作用 ..... - 25 -

6.2 简述壳 SHELL-BASH 的作用与处理流程 - 25 -
6.3 HELLO 的 FORK 进程创建过程 - 25 -
6.4 HELLO 的 EXECVE 过程 - 25 -
6.5 HELLO 的进程执行 - 26 -
6.6 HELLO 的异常与信号处理 - 27 -
6.7 本章小结 - 30 -
第 7 章 HELLO 的存储管理 - 31 -
7.1 HELLO 的存储器地址空间 - 31 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理 - 31 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理 - 32 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换 - 32 -
7.5 三级 CACHE 支持下的物理内存访问 - 33 -
7.6 HELLO 进程 FORK 时的内存映射 - 34 -
7.7 HELLO 进程 EXECVE 时的内存映射 - 34 -
7.8 缺页故障与缺页中断处理 - 34 -
7.9 动态存储分配管理 - 34 -
7.10 本章小结 - 35 -
第 8 章 HELLO 的 IO 管理 - 36 -
8.1 LINUX 的 IO 设备管理方法 - 36 -
8.2 简述 UNIX IO 接口及其函数 - 36 -
8.3 PRINTF 的实现分析 - 37 -
8.4 GETCHAR 的实现分析 - 38 -
8.5 本章小结 - 39 -
结论 - 40 -
附件 - 41 -
参考文献 - 42 -

第 1 章 概述

1.1 Hello 简介

1.1.1 P2P -From Program to Process

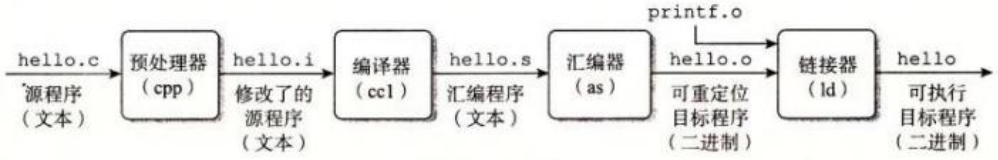
程序：计算机程序（Program）是一组计算机能识别和执行的指令，运行于电子计算机上，满足人们某种需求的信息化工具。

它以某些程序设计语言编写，运行于某种目标结构体系上。一般的，以英语文本为基础的计算机程序要经过编译、链接而成为人难以解读，但可轻易被计算机所解读的数字格式，然后放入运行。

为了使计算机程序得以运行，计算机需要加载代码，同时也要加载数据。从计算机的底层来说，这是由高级语言（例如 Java，C/C++，C#等）代码转译成机器语言而被 CPU 所理解，进行加载。

在一个符合大多数的计算机上，操作系统例如 Windows、Linux 等，加载并执行很多程序的情况下，每一个程序是一个单独的映射，并不是计算机上的所有可执行程序。它是指为了得到某种结果而可以由计算机等具有信息处理能力的装置执行的代码化指令序列，或者可以被自动转换成代码化指令序列的符号化指令序列或者符号化语句序列。同一计算机程序的源程序和目标程序为同一作品。

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



GCC 编译器读取源代码文件 hello.c，并且将其翻译成一个可执行文件 hello，过程如下：

预处理阶段：预处理器（cpp）根据以字符#开头的命令，修改原始的 C 程序，将#include 命令包含的头文件/源代码读取并把它直接插入程序文本中。

结果就得到了另一个 C 程序 hello.i；

编译阶段：编译器（ccl）将文本文件 hello.i 翻译成文本文件 hello.s，它包含一个汇编语言程序；

汇编阶段：接下来，汇编器（as）将 hello.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保存在目标文件 hello.o 中。它是一个二进制文件；

链接阶段：最后，链接器（ld）将程序与函数库中需要使用的二进制文件进行链接，形成可执行目标程序二进制文件。

生成可执行文件 hello 之后，该文件加载在内存中，由系统执行，在 shell 中键入启动命令后，shell 为其 fork 形成一个子进程，分配相应的内存资源，使用 execve 函数加载进程（Process），于是 hello 便从 Program 变为 Process，这便是 P2P 的过程。

1.1.2 020- From Zero to Zero

从“零”到“零”指的是一个程序从开始执行到结束，从“零”始到“零”终。

为执行 hello 程序，系统 Shell 首先 fork 一个子进程，再用 execve 函数加载目标程序，并用目标程序的进程取代当前进程。在这之前，在程序头部表的引导下，加载器将可执行文件的片复制到代码段和数据段，栈空间和堆空间也都被重新初 始化，其余各段也被替代。接着，CPU 为新进程进程分配时间片执行逻辑控制流，系统为进程映射虚拟内存，然后跳转到程序的入口点，也就是_start 函数的地址。这个函数是在系统目标文件 ctrl.o 中定义的，对所有 C 程序来说都是如此。_start 函数调用系统启动函数__libc_start_main，该函数定义在 libc.o 中。它初始化执行环境，调用 main 函数，处理 main 函数的返回值，并在及程序结束后把控制返回给内核。进程结束后，父进程将其回收，避免资源浪费，整个系统恢复到程序执行之前，即为 020。

1.2 环境与工具

1.2.1 硬件环境

Legion R7000P 2020
CPU：AMD Ryzen 7 4800H with Radeon Graphics　2.90 GHz
RAM：16.0GB

1.2.2 软件环境

OS：Windows 10 家庭版 21H2
VirtualBox 6.1.32 r149290 (Qt5.6.2)

OS：Ubuntu 20.04.4 LTS

1.2.3 开发工具

Gcc：gcc version 8.1.0 (x86_64-posix-seh-rev0, Built by MinGW-W64 project)
Vim 8.1.3741
Visual Studio Code 1.67.1

1.3 中间结果

文件名	作用
hello.c	源代码
hello.i	hello.c 预处理生成的文本文件
hello.s	hello.i 编译后得到的汇编语言文本文件
hello.o	hello.s 汇编后得到的可重定位目标文件
hello	hello.o 链接后得到的汇编语言文本文件
hello.objdump	hello.o 的反汇编结果
hello.elf	hello.o 的 ELF 结构
hellor.elf	hello 的 ELF 结构

1.4 本章小结

阐述了程序和进程的概念、hello.c 到可执行文件的编译过程以及可执行文件执行过程，提供了本实验的软硬件环境、开发工具等信息，列出了程序编译的中间过程的文件名称及作用。

（第 1 章 0.5 分）

第 2 章 预处理

2.1 预处理的概念与作用

预处理器 cpp 根据以字符#开头的命令（宏定义、条件编译），修改原始的 C 程序，将引用的所有库展开合并成为一个完整的文本文件。主要功能如下：

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 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

2.2 在 Ubuntu 下预处理的命令

预处理过程我们有两种选择：

A、gcc -E hello.c -o hello.i

B、cpp hello.c > hello.i

```
partychicken@partychicken-VirtualBox:~$ gcc -E hello.c -o hello.i
partychicken@partychicken-VirtualBox:~$ cpp hello.c > hello1.i
partychicken@partychicken-VirtualBox:~$ diff -sb hello.i hello1.i
Files hello.i and hello1.i are identical
```

2.3 Hello 的预处理结果解析

```
1 // 大作业的 hello.c 程序
2 // gcc -m64 -Og -no-pie -fno-PIC hello.c -o hello
3 // 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。
4 // 可以 运行 ps  jobs  pstree fg 等命令
5
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9
10 int main(int argc,char *argv[]){
11     int i;
12
13     if(argc!=4){
14         printf("用法: Hello 学号 姓名 秒数！\n");
15         exit(1);
16     }
17     for(i=0;i<8;i++){
18         printf("Hello %s %s\n",argv[1],argv[2]);
19         sleep(atoi(argv[3]));
20     }
21     getchar();
22     return 0;
23 }
```

Hello.c 的代码如上所示

```
3042
3043 # 9 "hello.c" 2
3044
3045
3046 # 10 "hello.c"
3047 int main(int argc,char *argv[]){
3048  int i;
3049
3050  if(argc!=4){
3051   printf("用法: Hello 学号 姓名 秒数！\n");
3052   exit(1);
3053  }
3054  for(i=0;i<8;i++){
3055   printf("Hello %s %s\n",argv[1],argv[2]);
3056   sleep(atoi(argv[3]));
3057  }
3058  getchar();
3059  return 0;
3060 }
```

```
10 int main(int argc,char *argv[]){
11     int i;
12
13     if(argc!=4){
14         printf("用法: Hello 学号 姓名 秒数！\n");
15         exit(1);
16     }
17     for(i=0;i<8;i++){
18         printf("Hello %s %s\n",argv[1],argv[2]);
19         sleep(atoi(argv[3]));
20     }
21     getchar();
22     return 0;
23 }
```

经过预处理后的 hello.i 文件为 3060 行，远大于原先 hello.c 的 23 行。 通过观察我们发现之前的#include 部分内容全部消失，取而代之的是各个头文 件的具体内容。以 stdio.h 的展开为例，cpp 到默认的环境变量下寻找 stdio.h，打开 /usr/include/stdio.h 发现其中依然使用了#define 语句，cpp 对此递归展开，所以最终.i 程序中是没有#define 的。而且发现其中使用了大量的#ifdef #ifndef 的语句，cpp 会对条件值进行判断来决定是否执行包含其中的逻辑。同时 hello.c 中对于代码 翻译无意义的注释也消失。代码的主体内容得以保留。

上图展示了 hello.i 与 hello.c 代码内容相同的部分。

2.4 本章小结

我们介绍了对于.c 文件的预处理的指令以及预处理的效果，解析了预处理之后的结果，为后续步骤的进行做好基础

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译器会将某一种编程语言携程的源代码转换为另外一种编程语言，该过程我们称之为编译。

在 gcc 编译 C 语言源程序的过程中，ccl 将预处理后的.i 文件翻译为.s 文本文件，该文件包含一个汇编语言程序。

高级计算机语言便于人编写，阅读，维护。低阶机器语言是计算机能直接解读、运行的。编译器主要的目的是将便于人编写，阅读，维护的高级计算机语言所写作的源代码，翻译为计算机能解读、运行的低阶机器语言的程序。编译器将原始程序（Source program）作为输入，翻译产生使用目标语言（Target language）的等价程序。源代码一般为高阶语言 (High-level language), 如 Pascal、C、C++、C#、Java 等，而目标语言则是汇编语言或目标机器的目标代码（Object code），有时也称作机器代码（Machine code）。

编译程序把一个源程序翻译成目标程序的工作过程分为六个阶段：词法分析；语法分析；语义分析；中间代码生成；代码优化；目标代码生成。

3.2 在 Ubuntu 下编译的命令

gcc -S hello.i -o hello.s

```
partychicken@partychicken-VirtualBox:~$ gcc -S hello.i -o hello.s
```

3.3 Hello 的编译结果解析

3.3.1 数据

hello.s 中使用到的数据类型有：整数、字符串、数组。

3.3.1.1 整数

hello.c 中的整数数据有 argc 和 i。

i：在 main 的最开始由 `pushq %rbp` 命令申请出内存空间作为 int 类型的循环变量，在循环体中以%ebp 寄存器调用

```
45      addl    $1, %ebp
46  .L2:
47      cmpl    $7, %ebp
48      jle .L3
```

argc: 作为第一个参数，argc 最初在寄存器%edi 中，比较后被直接丢弃，未

保存。

3.3.1.2 字符串

程序中的字符串分别是：

1) “用法: Hello 学号 姓名 秒数！\n”，第一个 printf 传入的输出格式化参数，在 hello.s 中声明如图，可以发现字符串被编码成 UTF-8 格式，一个汉字在 utf-8 编码中占三个字节，一个\代表一个字节。

2) “Hello %s %s\n”，第二个 printf 传入的输出格式化参数，在 hello.s 中声明

```
    .section    .rodata.str1.8,"aMS",@progbits,1
    .align 8
.LC0:
    .string "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \347\247
\222\346\225\260\357\274\201"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC1:
    .string "Hello %s %s\n"
```

字符串都声明在.section 与.rodata 中。

3.3.1.3 数组

程序中的数组是 char *argv[]，其地址作为 main 的第二个参数被存在%rsi 中，而后转存到%rbx `movq %rsi, %rbx` 。

3.3.2 赋值

程序中的赋值操作有：

i=0：整型数据的赋值通过 mov 指令来完成，由于 i 是 int 类型,因此使用 movl 指令。

```
movl    $0, %ebp
```

3.3.3 算术操作

代码中的算术操作为 i 的自增，通过 addl 指令来实现。

```
addl    $1, %ebp
```

3.3.4 关系操作

hello 中的关系比较操作有：“!=”和“<”。

1) argc!=4：判断 argc 是否不等于 4，通过 cmpl 函数来设置条件码 ZF，若 ZF 等于 1 则两个数字相等，反之不等，为其他指令判断该条件是否成立做准备。

```
cmpl    $4, %edi
```

2)) i<8：判断 i 是否小于 8，同样通过 cmpl 函数，将 i 与 7 进行比较来设置条件码，若后者小于等于前者则表示循环未结束，接着进行循环，否则跳出。

```
cmpl    $7, %ebp
jle .L3
```

（以下格式自行编排，编辑时删除）

3.3.5 数组操作

数组只有 argv[]，调用时作为 printf 的参数，分别调用了 argv[1],argv[2]，以及作为 atoi 的参数调用了 argv[3]。由于是 64 位系统，每个指针占 8 位，所以分别以 `movq 8(%rbx), %rsi` `movq 16(%rbx), %rdx` `movq 24(%rbx), %rdi` 的形式调用

3.3.6 控制转移

程序中涉及转移的有：

1) if 语句中如果 argc!=4 将执行花括号中的代码片段，这里通过 if 实现了控制的跳转。

```
cmpl    $4, %edi
jne .L6
```

2) for 语句中在最后判断循环终止条件时，如果不满足则跳转到循环开始的地方重新执行，否则跳出循环。

```
    cmpl    $7, %ebp
    jle .L3
```

3.3.8 函数操作函数是一种过程，过程提供了一种封装代码的方式，用一组指定的参数和可选的返回值实现某种功能。P 中调用函数 Q 包含以下动作：

1) 传递控制：进行过程 Q 的时候，程序计数器必须设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。

2) 传递数据：P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 中返回一个值。

3) 分配和释放内存：在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些空间。

程序中涉及函数操作的有：

1) main 函数：

a) 传递控制，main 函数因为被调用 call 才能执行（被系统启动函数__libc_start_main 调用），call 指令将下一条指令的地址 dest 压栈，然后跳转到 main 函数。

b) 传递数据，外部调用过程向 main 函数传递参数 argc 和 argv，分别使用%rdi 和%rsi 存储，函数正常出口为 return 0，将%eax 设置 0 返回。

c) 分配和释放内存，使用%rsp 记录栈顶，函数分配栈帧空间令%rsp 减去对应字节即可，程序结束时， 先把 rsp 加回来，然后 pop 所有 push 过的寄存器，恢复栈

空间为调用之前的状态，然后 ret 返回，ret 相当 pop IP，将下一条要执行指令的地址设置为 dest。

2) printf 函数：

a) 传递数据：第一次 printf 将%rdi 设置为“用法: Hello 学号 姓名 秒数！\n”字符串的首地址。第二次 printf 设置%rdi 为“Hello %s %s\n”的首地址，设置%rsi 为 argv[1]，%rdx 为 argv[2]。

b) 控制传递：第一次 printf 因为只有一个字符串参数，所以 call puts@PLT；第二次 printf 使用 call printf@PLT。

3) exit 函数：

a) 传递数据：将%edi 设置为 1。

b) 控制传递：call exit@PLT。

4) sleep 函数：

a) 传递数据：将%edi 设置为 atoi()的返回值（存在%eax 中）。

b) 控制传递：call sleep@PLT。

5) getchar 函数：

a) 控制传递：call gethcar@PLT

6）atoi 函数

a) 传递数据：将%rdi 设置为 argv[3] `movq 24(%rbx), %rdi`。

b) 控制传递：call atoi@PLT。

3.4 本章小结

本章了解了代码有高级源代码转换为低级汇编代码的编译过程，ccl 将 hello.c 转换为 hello.s 的过程即为编译，此外解析了 hello 的编译结果，涵盖了许多常见的指令。由于编译平台的不同，导致最终的运行效率产生差异。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

汇编器（as）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。这个过程称为汇编，亦即汇编的作用。通过这个过程，我们可以将汇编代码转化为机器可以理解的机器码。

4.2 在 Ubuntu 下汇编的命令

预处理过程我们有两种指令：

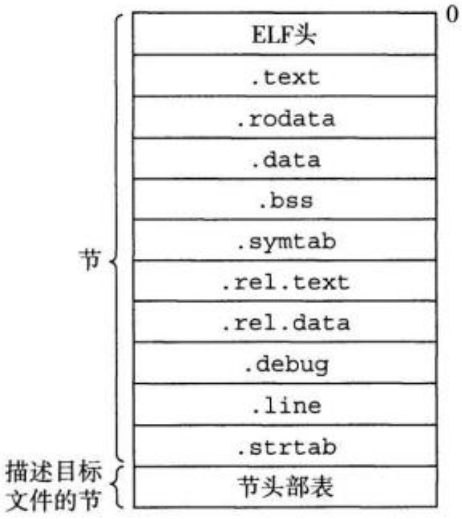
A、as hello.s -o hello.o

B、gcc hello.s -c -o hello.o

```
partychicken@partychicken-VirtualBox:~$ as hello.s -o hello.o
```

4.3 可重定位目标 elf 格式

4.3.0 elf 的结构与指令



ELF 是一种 Unix 二进制文件，它可能是可链接文件，也可能是可执行文件。用如下指令可以获得可阅读的文件

```
partychicken@partychicken-VirtualBox:~$ readelf -a hello.o > hello.elf
```

4.3.1 ELF Header

ELF 头以一个 16B 的序列 Magic 开始，Magic 描述了生成该文件的系统的字的大小和字节顺序，ELF 头剩下的部分包含帮助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。

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          1328 (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:         15
  Section header string table index: 14
```

4.3.2 Section Headers

节头部表，用于描述目标文件的节，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。

```
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
       0000000000000071  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  000003c8
       00000000000000c0  0000000000000018   I      12     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000b1
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  000000b1
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata.str1.8    PROGBITS         0000000000000000  000000b8
       0000000000000026  0000000000000001 AMS       0     0     8
  [ 6] .rodata.str1.1    PROGBITS         0000000000000000  000000de
       000000000000000d  0000000000000001 AMS       0     0     1
  [ 7] .comment          PROGBITS         0000000000000000  000000eb
       000000000000002c  0000000000000001  MS       0     0     1
  [ 8] .note.GNU-stack   PROGBITS         0000000000000000  00000117
       0000000000000000  0000000000000000           0     0     1
  [ 9] .note.gnu.propert NOTE             0000000000000000  00000118
       0000000000000020  0000000000000000   A       0     0     8
  [10] .eh_frame         PROGBITS         0000000000000000  00000138
       0000000000000040  0000000000000000   A       0     0     8
  [11] .rela.eh_frame    RELA             0000000000000000  00000488
       0000000000000018  0000000000000018   I      12    10     8
  [12] .symtab           SYMTAB           0000000000000000  00000178
       00000000000001f8  0000000000000018          13    13     8
  [13] .strtab           STRTAB           0000000000000000  00000370
       0000000000000052  0000000000000000           0     0     1
  [14] .shstrtab         STRTAB           0000000000000000  000004a0
       000000000000008a  0000000000000000           0     0     1
```

4.3.3 重定位节

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。如图中 8 条重定位信息分别是对.LC0（第一个 printf 中的字符串）、puts 函数、exit 函数、.LC1（第二个 printf 中的字符串）、printf 函数、atoi 函数、sleep 函数、getchar 函数进行重定位声明。

```
Relocation section '.rela.text' at offset 0x3c8 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000001c  000a00000002 R_X86_64_PC32     0000000000000000 .LC0 - 4
000000000021  000f00000004 R_X86_64_PLT32    0000000000000000 puts - 4
00000000002b  001000000004 R_X86_64_PLT32    0000000000000000 exit - 4
00000000003a  000b00000002 R_X86_64_PC32     0000000000000000 .LC1 - 4
000000000044  001100000004 R_X86_64_PLT32    0000000000000000 printf - 4
00000000004d  001200000004 R_X86_64_PLT32    0000000000000000 atoi - 4
000000000054  001300000004 R_X86_64_PLT32    0000000000000000 sleep - 4
000000000061  001400000004 R_X86_64_PLT32    0000000000000000 getchar - 4
```

1) 偏移量：指所引用的符号的相对偏移，或者说符号应该填在程序的哪个位置。例如，第二行中，puts 的偏移量为 0x000000000021。这就相当于告诉链接器，需要修改开始于偏移量 0x21 处的 32 位 PC 相对引用，使它在运行时指向 puts 函数。
2) 信息，包括符号和类型两部分，共占 8 个字节。其中，前 4 个字节表示符号，后 4 个字节表示类型。符号代表重定位到的目标在.symtab 节中的偏移量，类型则包括相对地址引用和绝对地址应用。
3) 类型，就是对第二列中类型信息的翻译。
4) 符号值，就是符号代表的值。
5) 符号名称是重定位目标的名字，可能是节名、变量名、函数名等；加数则是用于对被修改的引用值做偏移调整。

4.3.4 符号表

符号表保存了程序中所用的各种符号的信息，包括文件名、函数名、全局变量名、静态（私有）变量名等。

```
Symbol table '.symtab' contains 21 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    6
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    9
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT   10
    10: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    5 .LC0
    11: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    6 .LC1
    12: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
    13: 0000000000000000   113 FUNC    GLOBAL DEFAULT    1 main
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND atoi
    19: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sleep
    20: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND getchar
```

4.4 Hello.o 的结果解析

我们使用命令 objdump -d -r hello.o > hello.objdump 获得反汇编代码。

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


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   53                      push   %rbx
   6:   48 83 ec 08             sub    $0x8,%rsp
   a:   83 ff 04                cmp    $0x4,%edi
   d:   75 0a                   jne    19 <main+0x19>
   f:   48 89 f3                mov    %rsi,%rbx
  12:   bd 00 00 00 00          mov    $0x0,%ebp
  17:   eb 42                   jmp    5b <main+0x5b>
  19:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 20 <main+0x20>
            1c: R_X86_64_PC32   .LC0-0x4
  20:   e8 00 00 00 00          callq  25 <main+0x25>
            21: R_X86_64_PLT32  puts-0x4
  25:   bf 01 00 00 00          mov    $0x1,%edi
  2a:   e8 00 00 00 00          callq  2f <main+0x2f>
            2b: R_X86_64_PLT32  exit-0x4
  2f:   48 8b 53 10             mov    0x10(%rbx),%rdx
  33:   48 8b 73 08             mov    0x8(%rbx),%rsi
  37:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 3e <main+0x3e>
            3a: R_X86_64_PC32   .LC1-0x4
  3e:   b8 00 00 00 00          mov    $0x0,%eax
  43:   e8 00 00 00 00          callq  48 <main+0x48>
            44: R_X86_64_PLT32  printf-0x4
  48:   48 8b 7b 18             mov    0x18(%rbx),%rdi
  4c:   e8 00 00 00 00          callq  51 <main+0x51>
            4d: R_X86_64_PLT32  atoi-0x4
  51:   89 c7                   mov    %eax,%edi
  53:   e8 00 00 00 00          callq  58 <main+0x58>
            54: R_X86_64_PLT32  sleep-0x4
  58:   83 c5 01                add    $0x1,%ebp
  5b:   83 fd 07                cmp    $0x7,%ebp
  5e:   7e cf                   jle    2f <main+0x2f>
  60:   e8 00 00 00 00          callq  65 <main+0x65>
            61: R_X86_64_PLT32  getchar-0x4
  65:   b8 00 00 00 00          mov    $0x0,%eax
  6a:   48 83 c4 08             add    $0x8,%rsp
  6e:   5b                      pop    %rbx
  6f:   5d                      pop    %rbp
  70:   c3                      retq
```

将 hello.s 与 hello.objdump 对比之后我们发现：

1) 伪指令消失：hello.s 中许多“.”开头的伪指令在 hello.objdump 消失。

2) 条件分支变化：在 hello.s 中的段名称（如.L1、.L2）全部消失，取而代之的则是确定的相对偏移地址。如 hello.s 中的 jle .L3 在 hello.objdump 中变为了 jle 2f <main+0x2f>。

3) 函数调用变化：在 hello.s 我们调用 puts 等来自静态库或者其他的文件，需要进行链接才能调用的函数时，直接 call+函数名，但是在反汇编的代码中我们发现对应的机器码是 e8 00 00 00 00（即 call 0），反汇编的汇编指令是 call+相对地址，后面添加注释便于重定位。

4) 数据访问变化：在 hello.s 中，我们访问字符串常量是通过一些助记符访问的。而在反汇编的代码中是通过 0x0(%rip)访问，同样有注释，便于重定位。如 hello.s 中的 leaq .LC0(%rip), %rdi 在 hello.objdump 中变为了 lea 0x0(%rip),%rdi。

4.5 本章小结

本章介绍了从 hello.s 到 hello.o 的汇编过程，汇编器（as）将汇编代码 hello.s 转化为可重定位目标文件 hello.o，得到一个可以用于链接的二进制文件，通过查看 hello.o 的 elf 格式和使用 objdump 得到反汇编代码与 hello.s 进行比较的方式，间接了解到从汇编语言映射到机器语言汇编器需要实现的转换。

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

链接是 C 源代码编译的最后一步，它是指为了生成可执行文件，将有关的目标文件连接起来，使得这些目标文件成为操作系统可以装载执行的统一整体的过程。例如，在 A 文件中引用的符号将同该符号在 B 文件中的定义连接起来，从而形成一个可执行的整体。

链接分为两种模式：

(1) 静态链接。静态链接时，外部函数的代码将从其所在的静态链接库中拷贝到最终的可执行程序中。这样，程序执行时，这些代码就会被装入到对应进程的虚拟内存空间里。这里的静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。

(2) 动态链接。动态链接中，外部函数的代码被放到动态链接库或共享对象的某个目标文件中（通常以.so 为后缀名）。链接器在链接时所做的只是在生成的可执行文件中记下共享对象的名字等少量信息。在可执行文件运行行时，动态链接库的全部内容将被映射到相应进程的虚拟内存空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

5.2 在 Ubuntu 下链接的命令

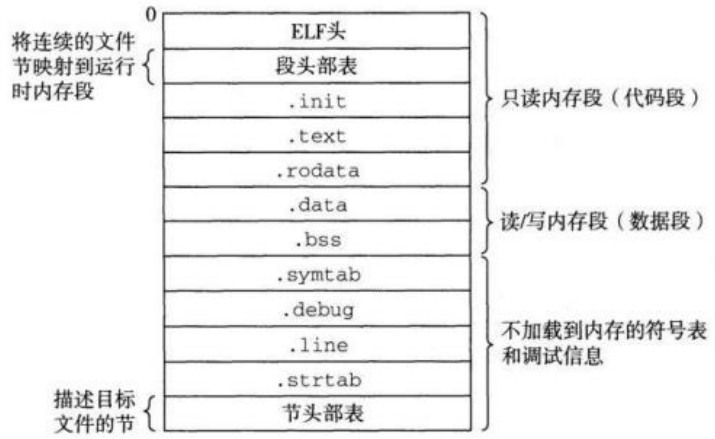
执行的命令为：ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o

```
partychicken@partychicken-VirtualBox:~$ ld -o hello -dynamic-linker /lib64/ld-l
inux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crt
i.o hello.o /usr/lib/x86_64-linux-gnu/libc.so  /usr/lib/x86_64-linux-gnu/crtn.o
```

5.3 可执行目标文件 hello 的格式

5.3.0 ELF 结构与指令

使用 readelf -a hello > hello.elf 命令生成 hello 程序的 ELF 格式文件。



5.3.1 ELF Header

也可以直接输入命令 readelf -a hello 查看 ELF 头信息。
ELF 头以一个 16 字节的序列开始，这个序列描述了生成该文件的系统的节的大小和字节顺序。ELF 头剩下的部分还包括 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、以及节头部表的大小和数目。

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4010f0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14208 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         12
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

5.3.2 Section Headers

我们会发现相比与 hello.o，hello 的节头部表有 26 项，比 hello.o 多了 12 项。节头部表对 hello 中所有的节信息进行了声明，包括大小 size 以及在程序中的偏移量 offset，大小、全体大小、旗标、链接、信息、对齐等信息，并可以根据此定位各个节所占的区间。由于链接时使用的是动态链接，所以 hello 可执行文件仍然具有.rela.*的重定位节，便于使用动态链接共享库。.init 节定义了一个小函数_init，程序的初始化代码会调用它。

```
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         00000000004002e0  000002e0
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.propert NOTE             0000000000400300  00000300
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.ABI-tag     NOTE             0000000000400320  00000320
       0000000000000020  0000000000000000   A       0     0     4
  [ 4] .hash             HASH             0000000000400340  00000340
       0000000000000038  0000000000000004   A       6     0     8
  [ 5] .gnu.hash         GNU_HASH         0000000000400378  00000378
       000000000000001c  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           0000000000400398  00000398
       00000000000000d8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000400470  00000470
       000000000000005c  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000004004cc  000004cc
       0000000000000012  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          00000000004004e0  000004e0
       0000000000000020  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000400500  00000500
       0000000000000030  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000400530  00000530
       0000000000000090  0000000000000018  AI       6    21     8
  [12] .init             PROGBITS         0000000000401000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000401020  00001020
       0000000000000070  0000000000000010  AX       0     0     16
  [14] .plt.sec          PROGBITS         0000000000401090  00001090
       0000000000000060  0000000000000010  AX       0     0     16
  [15] .text             PROGBITS         00000000004010f0  000010f0
       0000000000000125  0000000000000000  AX       0     0     16
  [16] .fini             PROGBITS         0000000000401218  00001218
       000000000000000d  0000000000000000  AX       0     0     4
  [17] .rodata           PROGBITS         0000000000402000  00002000
       000000000000003b  0000000000000000   A       0     0     8
  [18] .eh_frame         PROGBITS         0000000000402040  00002040
       0000000000000104  0000000000000000   A       0     0     8
  [19] .dynamic          DYNAMIC          0000000000403e50  00002e50
       00000000000001a0  0000000000000010  WA       7     0     8
  [20] .got              PROGBITS         0000000000403ff0  00002ff0
       0000000000000010  0000000000000008  WA       0     0     8
```

5.3.3 程序头表

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000002a0 0x00000000000002a0  R      0x8
  INTERP         0x00000000000002e0 0x00000000004002e0 0x00000000004002e0
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000005c0 0x00000000000005c0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x0000000000000225 0x0000000000000225  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
                 0x0000000000000144 0x0000000000000144  R      0x1000
  LOAD           0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
                 0x00000000000001fc 0x00000000000001fc  RW     0x1000
  DYNAMIC        0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
                 0x00000000000001a0 0x00000000000001a0  RW     0x8
  NOTE           0x0000000000000300 0x0000000000400300 0x0000000000400300
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000320 0x0000000000400320 0x0000000000400320
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_PROPERTY   0x0000000000000300 0x0000000000400300 0x0000000000400300
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002e50 0x0000000000403e50 0x0000000000403e50
                 0x00000000000001b0 0x00000000000001b0  R      0x1
```

程序头部表描述了可执行文件的连续的片和连续的内存段的映射关系。

5.3.4 其余信息

段映射：

```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .note.gnu.property .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .
gnu.version_r .rela.dyn .rela.plt
  03     .init .plt .plt.sec .text .fini
  04     .rodata .eh_frame
  05     .dynamic .got .got.plt .data
  06     .dynamic
  07     .note.gnu.property
  08     .note.ABI-tag
  09     .note.gnu.property
  10
  11     .dynamic .got
```

动态节项目：

```
Dynamic section at offset 0x2e50 contains 21 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x401000
 0x000000000000000d (FINI)               0x401218
 0x0000000000000004 (HASH)               0x400340
 0x000000006ffffef5 (GNU_HASH)           0x400378
 0x0000000000000005 (STRTAB)             0x400470
 0x0000000000000006 (SYMTAB)             0x400398
 0x000000000000000a (STRSZ)              92 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x404000
 0x0000000000000002 (PLTRELSZ)           144 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x400530
 0x0000000000000007 (RELA)               0x400500
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x4004e0
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x4004cc
 0x0000000000000000 (NULL)               0x0
```

重定位节：

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

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

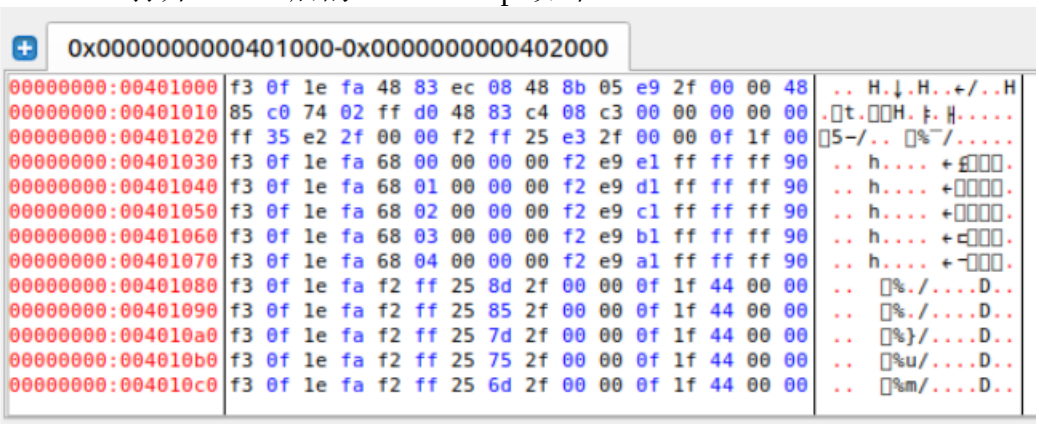
符号表：（节选）

```
Symbol table '.dynsym' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getchar@GLIBC_2.2.5 (2)
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND atoi@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND exit@GLIBC_2.2.5 (2)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 51 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000004002e0     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400300     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400320     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400340     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000400378     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400398     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400470     0 SECTION LOCAL  DEFAULT    7
     8: 00000000004004cc     0 SECTION LOCAL  DEFAULT    8
     9: 00000000004004e0     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000400500     0 SECTION LOCAL  DEFAULT   10
    11: 0000000000400530     0 SECTION LOCAL  DEFAULT   11
    12: 0000000000401000     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000401020     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000401090     0 SECTION LOCAL  DEFAULT   14
    15: 00000000004010f0     0 SECTION LOCAL  DEFAULT   15
```

5.4 hello 的虚拟地址空间

Edb 打开 hello 后的 Data Dump 如下：



由上图我们可以看到 hello 隔断的虚拟地址空间被限制在 0x401000 到 0x402000 之间。

查看图 5-5 我么可以看到 ELF 格式文件中的 Program Headers，程序头表在执行的时候被使用，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，程序包含 7 个段：

1) PHDR 保存程序头表。

2) 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）。

3) 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。
4) DYNAMIC 保存了由动态链接器使用的信息。
5) NOTE 保存辅助信息。
6) GNU_STACK：权限标志，标志栈是否是可执行的。
7) 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

使用 objdump -d -r hello > hello.d 得到 hello 的反汇编的代码。

将 hello.d 与 hello.objdump 进行对比我们可以发现：

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、crtn.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。

。

5.6 hello 的执行流程

程序名	程序地址
加载 hello	
ld-linux-x86-64.so!_dl_start	0x00007ffff7fd4d30
ld-linux-x86-64.so!_dl_init	0x00007ffff7fe27b0
hello!_start	0x0000000000401090
hello!__libc_csu_init	0x00000000004010d0
hello!_init	0x0000000000401000

libc.so!_setjmp	0x00007ffff7e08b10
程序运行	
hello!main	0x0000000000401149
hello!puts@plt	0x0000000000401030
退出程序	
hello!exit@plt	0x0000000000401070
libc.so!exit	0x00007ffff7e0b840
hello!_fini	0x00000000004011d4

5.7 Hello 的动态链接分析

函数调用一个由共享库定义的函数时，编译器无法预先判断出函数的地址，因为定义它的共享模块在运行时可以加载到任意位置。GNU 编译系统使用延迟绑定的方式解决该问题，在运行时动态载入。

延迟绑定通过两个数据结构之间简洁但又有些复杂的交互来实现，即过程链接表（PLT）和全局偏移量表（GOT）。

过程链接表（PLT）：PLT 是一个数组，其中每个条目是 16 字节代码。PLT [0]是一个特殊条目，它跳转到动态链接器中。每个被可执行程序调用的库函数都有它自己的 PLT 条目。每个条目都负责调用一个具体的函数。每个条目都负责调用一个具体的函数。

全局偏移量表（GOT）：GOT 是一个数组，其中每个条目是 8 字节地址。和 PLT 联合使用时，GOT [0]和 GOT [1]包含动态链接器在解析函数地址时会使用的信息。GOT [2]是动态链接器在 1d-linux.so 模块中的入口点。其余的每个条目对应于一个被调用的函数，其地址需要在运行时被解析。每个条目都有一个相匹配的 PLT 条目。

5.8 本章小结

本章介绍了链接的概念及作用，分析了 hello 的 ELF 格式，深入学习了 hello.o 可重定位文件到 hello 可执行文件的流程，和链接的各个过程介绍了链接器如何将 hello.o 可重定向文件与动态库函数链接起来，链接技术可以允许我们把大项目分解成小模块、小功能。当我们改变某个小模块时，只需简单地重新编译它并重新链接应用，而无需编译其他文件，正是因为有了技术的存在，代码的编译才会变得更加高效。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：进程是一个执行中的程序的实例，每一个进程都有它自己的地址空间，一般情况下，包括文本区域、数据区域、和堆栈。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储区着活动过程调用的指令和本地变量。

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一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。每次用户通过 shell 输入一个可执行目标文件的名字，运行程序时，shell 就会创建一个新的进程，应用程序也可以创建新进程，并且在这个新进程的上下文中运行它们自己的代码或者其他应用程序。这使得我们可以同时运行多个程序。

6.2 简述壳 Shell-bash 的作用与处理流程

作用：Shell 是一种壳层与命令行界面，是操作系统下传统的用户和计算机的交互界面，使可以通过这个界面访问内核提供的服务。

处理流程：

1) 从终端读入输入的命令。
2) 将输入字符串切分并解析获得所有的参数。
3) 如果是内置命令则立即调用相应的函数执行。
4) 否则创立一个子进程并在子进程中使用 execve 运行该程序。
5) shell 应该接受键盘输入信号，并对这些信号进行相应处理。

6.3 Hello 的 fork 进程创建过程

父进程可以通过 fork 函数创建一个新子进程。函数原型为 pid_t fork(void);
函数返回值分两种情形，父进程内返回子进程的 PID，子进程内返回 0。新创建的子进程与父进程几乎完全相同。子进程得到与父进程用户级虚拟地址空间相同（但是独立的）一份副本，包括代码段和数据段、堆、共享库以及用户栈。子进程还会获得父进程所打开的文件描述符的副本，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

差别在于它们有不同的 PID。

fork 函数调用一次，返回两次。父进程调用一次 fork，一次是返回到父进程，而另一次是返回到子进程的。父进程和子进程是并发运行的独立进程，内核可以以任意方式交替执行它们的逻辑控制流中的指令。我们不能对不同进程中指令的交替执行做任何假设。两个进程有相同的用户栈、运行时堆和本地变量值等，但它们对各自内存空间的修改是相互独立的。事实上，在物理内存中，一开始，两个进程指向的地址确实是相同的；但是，一旦一方对部分共享空间做了修改，这部分空间就会被拷贝出去，不再共享。共享文件。子进程会继承父进程打开的所有文件。

6.4 Hello 的 execve 过程

shell 会通过 execve 调用 hello。execve 函数加载并运行可执行目标文件 hello，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 hello，execve 才会返回到调用函数。所以与 fork 调用一次返回两次不一样，execve 调用一次并且从不返回。

execve 函数在当前进程的上下文中加载并运行一个新的函数。它会覆盖当前进程的地址空间，但并没有创建一个新进程。新的函数仍然有相同的 PID，并且继承了调用 execve 函数时已打开的所有文件描述符。

载入并执行 hello 需要以下几个步骤：

1) 删除已存在的使用者区域：删除当前程序虚拟地址的使用者部分中已存在的区域结构。

2) 对映私有区域：为新程序的代码、数据、.bss 和栈区域建立新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据被对映为 hello 档案中的.text 和.data 区。.bss 区域是请求二进位制零的，对映到初值为 0 的匿名文件，其大小包含在 hello 中。栈和堆区域也是请求二进制零的，初始长度为零。

3) 对映共享区域：如果 hello 程式与共享物件连结，那么这些物件都是动态连结到这个程式的，然后再对映到使用者虚拟地址空间中的共享区域内。

4) 设定程式计数器：设定当前程序上下文中的程序计数器，使之指向代码区域的入口点。下一次运行这个程序时，它将从这个入口点开始执行

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，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

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：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区

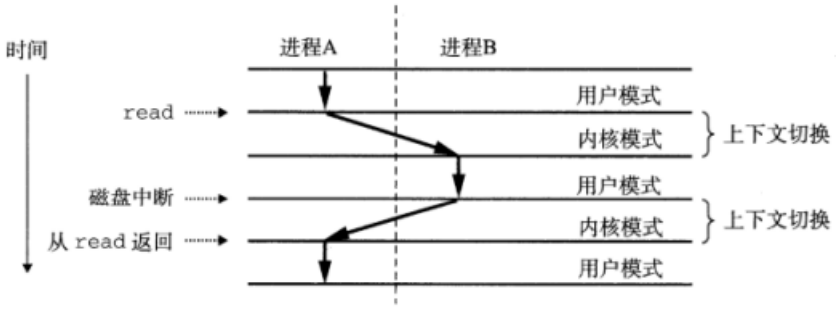
内的代码和数据；设置模式位时，进程处于内核模式，该进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

上下文信息：上下文就是内核重新启动一个被抢占的进程所需要的状态，它由通用寄存器、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等对象的值构成。

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

内核为每个进程维护了一个上下文。当内核选择的一个新的进程运行时，我们说内核调度了这个进程。所以当内核调度了 hello 这个新的进程运行后，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，上下文切换会首先保存当前进程的上下文，然后恢复新恢复进程被保存的上下文，最后控制传递给这个新恢复的进程 ，来完成上下文切换。

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。



6.6 hello 的异常与信号处理

hello 执行过程中出现的可能的异常种类会有四种：

(1) **中断** 中断是异步发生的，是来自处理器外部的 I/O 设备的信号的结果。硬件中断的异常处理程序被称为中断处理程序。

(2) **陷阱** 陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令。陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。

(3) **故障** 故障由错误情况引起，它可能能够被故障处理程序修正。当故障发生时，处理器将控制转移给故障处理程序。如果处理程序能够修正这个错误情况，它就将控制返回到引起故障的指令，从而重新执行它。否则处理程序返回到内核中的 abort 例程，abort 例程会终止引起故障的应用程序。

(4) **终止** 终止是不可恢复的致命错误造成的结果，通常是一些硬件错误，比如 DRAM 或者 SRAM 位被损坏时发生的奇偶错误。终止处理程序从不将控制返

回给应用程序。处理程序将控制返回给一个 abort 例程，该例程会终止这个应用程序。

hello 执行时，还可以发送或接收信号。信号是一种系统消息，它用于通知进程系统中发生了某种类型的事件，是一种更高层的软件形式的异常。不同的事件对应不同的信号类型。信号传送到目的进程由发送和接收两个步骤组成。信号的发送者一般是内核，接收者是进程。

发送信号可以有如下两种原因：

(1) 内核检测到一个系统事件（如除零错误或者子进程终止）；

(2) 一个进程调用了 kill 函数，显式地要求内核发送一个信号给目的进程。

接收信号是内核强迫目的进程做出的反应。进程可以以默认方式做出反应，也可以通过信号处理程序捕获这个信号。每个信号只会被处理一次。

待处理信号指的是已经发送而没有接收的信号。任何时候，一种信号类型至多有一个待处理信号，即信号不会排队。

进程可以有选择性地阻塞接收某种信号。被阻塞的信号仍可以发出，但不会被目标进程接收。

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截屏，说明异常与信号的处理。

正常运行：由于最后有 getchar 需要输入字符才能结束

```
partychicken@partychicken-VirtualBox:~$ ./hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
a
```

乱按（无回车）：最后需要回车结束

```
partychicken@partychicken-VirtualBox:~$ ./hello 120L010911 wzy 3
Hello 120L010911 wzy
akdfjakjfjea Hello 120L010911 wzy
Hello 120L010911 wzy
jajfekajjjHello 120L010911 wzy
kafjfejjHello 120L010911 wzy
,zjfHello 120L010911 wzy
jejlHello 120L010911 wzy
ajfiahfehthHello 120L010911 wzy
aiejeji
```

乱按（有回车）：

```
partychicken@partychicken-VirtualBox:~$ ./hello 120L010911 wzy 3
Hello 120L010911 wzy
kajefi ajefj
aeifjijefa;lijei
jaeifjaejfaljleawjiHello 120L010911 wzy
f
jgaiejgiawjgija
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
sjfjiHello 120L010911 wzy

jkdjfajf
Hello 120L010911 wzy
Hello 120L010911 wzy
partychicken@partychicken-VirtualBox:~$ aeifjijefa;lijei
aeifjijefa: command not found
lijei: command not found
partychicken@partychicken-VirtualBox:~$ jaeifjaejfaljleawjif
jaeifjaejfaljleawjif: command not found
partychicken@partychicken-VirtualBox:~$ jgaiejgiawjgija
jgaiejgiawjgija: command not found
partychicken@partychicken-VirtualBox:~$ sjfji
sjfji: command not found
partychicken@partychicken-VirtualBox:~$ jkdjfajf
jkdjfajf: command not found
partychicken@partychicken-VirtualBox:~$
```

可以发现除了第一次回车前的内容被 getchar 吸收掉以外，剩余键盘输入均被保存起来，当作 shell 输入的命令，然后在程序结束后试图运行这些命令。

Ctrl-C ：

```
partychicken@partychicken-VirtualBox:~$ ./hello 120L010911
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
^C
```

Ctrl-C 操作向进程发送了一个 SIGINT 信号，让进程终止，输入 ps 指令可以发现 hello 进程已经被回收。

Ctrl-Z：

```
partychicken@partychicken-VirtualBox:~$ ./hello 120L010911 wzy 3
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
^Z
[1]+  Stopped                 ./hello 120L010911 wzy 3
partychicken@partychicken-VirtualBox:~$ ps
    PID TTY          TIME CMD
   2039 pts/0    00:00:00 bash
   8187 pts/0    00:00:00 hello
   8188 pts/0    00:00:00 ps
partychicken@partychicken-VirtualBox:~$ jobs
[1]+  Stopped                 ./hello 120L010911 wzy 3
partychicken@partychicken-VirtualBox:~$ fg 1
./hello 120L010911 wzy 3
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
Hello 120L010911 wzy
a
partychicken@partychicken-VirtualBox:~$
```

Ctrl-Z 操作向进程发送了一个 SIGTSTP 信号，让进程暂时挂起，输入 jobs、ps 指令可以发现 hello 进程在后台挂起，且进程号为 1，通过 fg 指令可以恢复运行。

6.7 本章小结

本章描述了 hello 进程管理，介绍了进程的概念和作用，以及 shell 如何运行 hello 程序。为了高效地描述系统中发生的各类事件，则需要用到信号，这是一种更高层级的软件形式的异常。利用信号，内核和进程之间得以高效地传递信息并对各类事件做出相应的反应。

（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

计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有唯一的物理地址，它是指在地址总线上、以电子形式存在的、使得数据总线可以访问主存的某个特定存储单元的内存地址。利用物理地址寻址，是 CPU 最自然的访问内存的方式。接下来是几个具体概念：

1) 逻辑地址：在计算机体系结构中逻辑地址是指应用程序角度看到的内存单元、存储单元、网络主机的地址，即 hello.o 里面的相对偏移地址。逻辑地址往往不同于物理地址，通过地址翻译器或映射函数可以把逻辑地址转化为物理地址。

2) 线性地址：线性地址是逻辑地址到物理地址变换之间的中间层，即 hello 中的虚拟地址，等于逻辑地址加上基地址。逻辑地址可转化为线性地址，其地址空间是一个非负整数地址的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间。

3) 虚拟地址：虚拟地址是程序用于访问物理内存的逻辑地址，即线性地址，在 hello 中为虚拟地址。

4) 物理地址：计算机的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每个字节都有一个唯一的物理地址。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

内存分段是为了支持多任务并发执行，每一个任务对应各自的段空间，段之间支持保护访问限制，实现了程序和数据从物理地址空间到虚拟地址空间的重映射，从而达到隔离的效果。

如上所述，在段式内存管理中，程序的地址空间被划分为若干段，每个进程都有一个“二维”的地址空间。系统为每个段分配一个连续分区，而进程中的各个段可以不连续地存放在内存的各个分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。

为了实现段式管理，系统需要进程段表、系统段表和空闲段表等数据结构，来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

1) 程序段表：描述组成程序地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址，即段内地址。

2) 系统段表：系统所有占用段（已经分配的段）。

3) 空闲段表：记忆体中所有空闲段，可以结合到系统段表中。

在段式管理系统中，整个进程的地址空间是“二维”的，逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址。这个

过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。

7.3 Hello 的线性地址到物理地址的变换-页式管理

虚拟内存系统将程序的虚拟地址空间划分为固定大小的虚拟页，物理内存被划分为同样大小的物理页（也被称作页帧）。在页式存储管理中，虚拟地址由两部构成，高位部分是页号，低位部分是页内地址（偏移量）。

在任意时刻，虚拟页面的集合都分为三个不相交的子集：

1) 未分配页。虚拟内存系统还未分配（或者创建）的页。未分配的块没有任何数据和它们相关联，因此也就不占用任何磁盘空间。

2) 缓存页。当前已缓存在物理内存中的已分配页

3) 未缓存页。未缓存在物理内存中的已分配页

物理内存对应存储器金字塔的 DRAM（主存）一级，而虚拟页则是存储在磁盘上。与读写高速缓存一样，从虚拟内存和主存之间也存在着缓存关系，因而也拥有类似的命中、不命中的概念。

页式管理方式的优点是：没有外部碎片；一个程序不必连续存放；便于改变程序占用空间的大小（主要指随着程序执行，动态生成的数据增多，所要求的地址空间相应增长）。其缺点是：要求程序全部装入内存，没有足够的内存，程序就不能执行。

在页式系统中程序建立时，系统为程序中所有的页分配页帧。当程序撤销时收回所有分配给它的页帧。在程序的执行期间，如果允许程序动态地申请空间，系统还要为程序申请的空间分配物理页帧。系统为了完成这些功能，必须记录系统内存中实际的页帧使用情况。系统还要在程序切换时，需要正确地切换两个不同的程序地址空间到物理内存空间的对映。这就要求系统要记录每个程序页表的相关信息。

为了完成上述的功能，一个页式系统中，一般要采用如下的数据结构：

页表：页表将虚拟内存对映到物理页。每次内存管理单元将一个虚拟地址转换为物理地址时，都会读取页表。页表是一个页表条目（PTE）的数组。虚拟地址空间的每个页在页表中一个固定偏移量处都有一个 PTE。假设每个 PTE 是由一个有效位和一个 n 位地址栏位组成的。有效位表明了该虚拟页当前是否被缓存在 DRAM 中。如果设定了有效位，那么地址字段就表示 DRAM 中相应的物理页的起始位置，这个物理页中缓存了该虚拟页。如果没有设定有效位，那么一个空地址表示这个虚拟页还未被分配。否则，这个地址就指向该虚拟页在磁盘上的起始位置。

MMU 利用虚拟页号（VPN）来选择适当的 PTE，将列表条目中物理页号（PPN）和虚拟地址中的虚拟页偏移量（VPO）串联起来，就得到相应的物理地址。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

在 Intel Core i7 环境下研究 VA 到 PA 的地址翻译问题。前提如下：

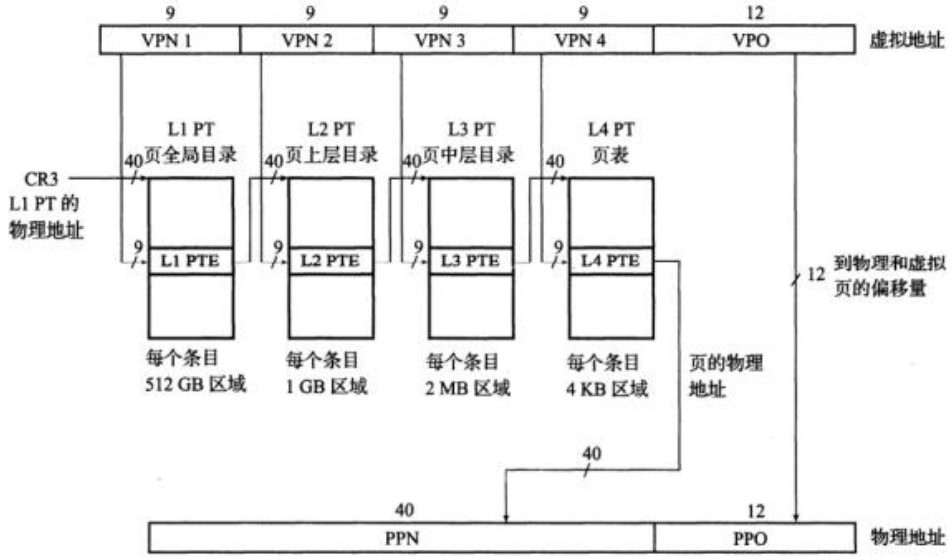
虚拟地址空间 48 位，物理地址空间 52 位，页表大小 4KB，4 级页表。TLB 4 路 16 组相联。CR3 指向第一级页表的起始位置（上下文一部分）。

解析前提条件：由一个页表大小 4KB，一个 PTE 条目 8B，共 512 个条目，使用 9 位二进制索引，一共 4 个页表共使用 36 位二进制索引，所以 VPN 共 36 位，因为 VA 48 位，所以 VPO 12 位；因为 TLB 共 16 组，所以 TLBI 需 4 位，因为 VPN 36 位，所以 TLBT 32 位。

CPU 产生虚拟地址 VA，VA 传送给 MMU，MMU 使用前 36 位 VPN 作为 TLBT（前 32 位）+TLBI（后 4 位）向 TLB 中匹配，如果命中，则得到 PPN（40bit）与 VPO（12bit）组合成 PA（52bit）。

如果 TLB 中没有命中，MMU 向页表中查询，CR3 确定第一级页表的起始地址，VPN1（9bit）确定在第一级页表中的偏移量，查询出 PTE，如果在物理内存中且权限符合，确定第二级页表的起始地址，以此类推，最终在第四级页表中查询到 PPN，与 VPO 组合成 PA，并且向 TLB 中添加条目。

如果查询 PTE 的时候发现不在物理内存中，则引发缺页故障。如果发现权限不够，则引发段错误。



7.5 三级 Cache 支持下的物理内存访问

前提：只讨论 L1 Cache 的寻址细节，L2 与 L3Cache 原理相同。L1 Cache 是 8 路 64 组相联。块大小为 64B。

解析前提条件：因为共 64 组，所以需要 6bit CI 进行组寻址，因为共有 8 路，因为块大小为 64B 所以需要 6bit CO 表示数据偏移位置，因为 VA 共 52bit，所以 CT 共 40bit。

在上一步中我们已经获得了物理地址 VA，使用 CI（后六位再后六位）进行组索引，每组 8 路，对 8 路的块分别匹配 CT（前 40 位）如果匹配成功且块的 valid 标志位为 1，则命中（hit），根据数据偏移量 CO（后六位）取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是 1，则不命中（miss），向下一级缓存中查询数据（L2 Cache->L3 Cache->主存）。查询到数据之后，一种简单的放置策略如下：如果映射到的组内有空闲块，则直接放置，否则组内都是有效块，产生冲突（evict），则采用最近最少使用策略 LFU 进行替换。

7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 调用时，内核为 hello 创建各种数据结构，并分配一个唯一的 PID。为了给 hello 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 hello 进程中返回时，hello 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的页面，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

execve 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序。加载并运行 hello 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。

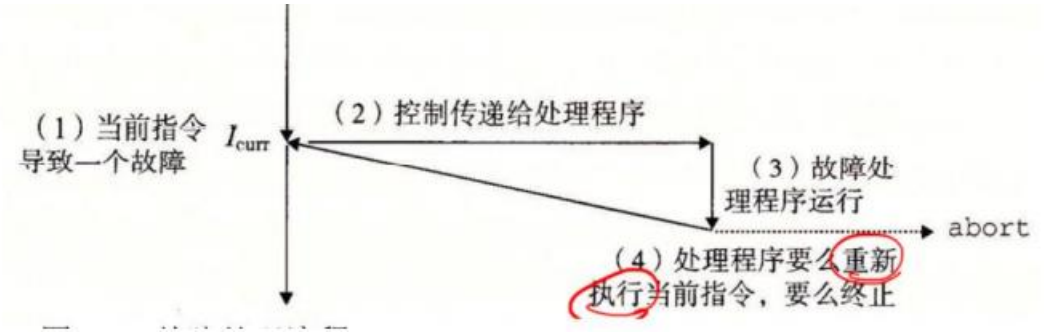
2) 映射私有区域，为新程序的代码、数据、bss 和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 hello 文件中的.text 和.data 区，bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中，栈和堆地址也是请求二进制零的，初始长度为零。

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

4) 设置程序计数器（PC），execve 做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就会发生故障。其处理流程遵循下图所示的故障处理流程。



7.9 动态存储分配管理

动态内存分配器用于分配和维护一个进程的虚拟内存区域，称为堆。堆在系统内存中向上生长。对于每个进程，系统维护着一个堆顶指针 brk。
分配器将堆视为不同大小的块组成的集合。每个块就是一段连续的虚拟内存片（chunk），要么是已分配的（allocated），要么是空闲的（free）。已分配的块显式地保留给应用程序使用，空闲块留待分配。
显式分配器：要求应用显式地释放任何已分配的块。
隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。

7.9.1 隐式空闲链表

我们可以将堆组织为一个连续的已分配块和空闲块的序列，空闲块是通过头部中的大小字段隐含地连接着的，这种结构为隐式空闲表。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地集合。一个块是由一个字的头部、有效载荷、可能的填充和一个字的脚部，其中脚部就是头部的一个副本。头部编码了这个块的大小以及这个块是已分配还是空闲的。分配器就可以通过检查它的头部和脚部，判断前后块的起始位置和状态。

7.9.2 显式空闲链表

将堆组成一个双向空闲链表，在每个空闲块中，都包含一个 pred 和 succ 指针。一种方法是用后进先出（LIFO）的顺序来维护链表，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。
一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小快大小，也潜在的提高了内部碎片的程度。

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，以 intel Core7 在指定环境下介绍了 VA 到 PA 的变换、物理内存访问，还介绍了 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 I0 管理

8.1 Linux 的 I0 设备管理方法

所有的 I/O 装置（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做对相应档案的读和写来执行。这种将装置优雅地映射为文件的方式，允许 Linux 核心引出一个简单、低阶的应用接口，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

设备的模型化：文件

设备管理：unix io 接口

8.2 简述 Unix I0 接口及其函数

Unix I/O 接口统一操作：

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。

2) Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。

3) 改变当前的文件位置：对于每个打开的文件，内核保持着一个文件位置 k，初始为 0，这个文件位置是从文件开头起始的字节偏移量，应用程序能够通过执行 seek，显式地将改变当前文件位置 k。

4) 读写文件：一个读操作就是从文件复制 n>0 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 k+n，给定一个大小为 m 字节的而文件，当 k>=m 时，触发 EOF。类似一个写操作就是从内存中复制 n>0 个字节到一个文件，从当前文件位置 k 开始，然后更新 k。

5) 关闭文件，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去。

Unix I/O 函数：

1) int open(char* filename,int flags,mode_t mode)，进程通过调用 open 函数来打开一个存在的文件或是创建一个新文件的。open 函数将 filename 转换为一个文件描述符，并且返回描述符数字，返回的描述符总是在进程中当前没有打开的最小描述符，flags 参数指明了进程打算如何访问这个文件，mode 参数指定了新文件的访问权限位。

2) int close(fd)，fd 是需要关闭的文件的描述符，close 返回操作结果。

3) ssize_t read(int fd,void *buf,size_t n)，read 函数从描述符为 fd 的当前文件位置赋

值最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，0 表示 EOF，否则返回值表示的是实际传送的字节数量。

4) ssize_t wirte(int fd,const void *buf,size_t n)，write 函数从内存位置 buf 复制至多 n 个字节到描述符为 fd 的当前文件位置。

8.3 printf 的实现分析

https://www.cnblogs.com/pianist/p/3315801.html

printf 函数实现如下：

```
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;
}
```

用 vsprintf 生成显示信息：

```
char *p;
char tmp[256];
va_list p_next_arg = args;
for (p = buf; *fmt; fmt++) {
    if (*fmt != '%') {
        *p++ = *fmt;
        continue;
    }
    fmt++;
    switch (*fmt) {
        case 'x':
            itoa(tmp, *((int *)p_next_arg));
            strcpy(p, tmp);
            p_next_arg += 4;
            p += strlen(tmp);
            break;
        case 's':
            break;
        default:
            break;
    }
}
return (p - buf);
```

printf 调用的 write 函数汇编如下：

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

write 函数中，%ecx 中存储字符个数，%ebx 中存储字符串首地址，int INT_VECTOR_SYS_CALL 的意思是通过系统调用 sys_call。这个函数的功能就是不断地打印出字符，直到遇到\0。

追踪 sys_call，得到其汇编实现如下：

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

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 的一种实现如下：

```c
#include "libioP.h"
#include "stdio.h"
#undef getchar
int getchar(void) {
    int result;
    if (!_IO_need_lock(stdin))
        return _IO_getc_unlocked(stdin);
    _IO_acquire_lock(stdin);
    result = _IO_getc_unlocked(stdin);
    _IO_release_lock(stdin);
    return result;
}
#ifndef _IO_MTSAFE_IO
#undef getchar_unlocked
weak_alias(getchar, getchar_unlocked)
#endif
```

这个 getchar 每次从标准输入中读取一个字符。具体来说，若当前 I/O 未被锁定，它就调用系统_IO_getc_unlocked 内置宏，读取一个字符。

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

本章主要介绍了 Linux 的 IO 设备管理方法、Unix IO 接口及其函数，分析了 printf 函数和 getchar 函数。

（第 8 章 1 分）

结论

hello 的一生包含如下阶段：

1) 预处理：将 hello.c 根据以字符#开头命令，修改原始 c 程序，得到 hello.i。

2) 编译：将 hello.i 翻译为 hello.s 的汇编程序，中间对代码进行语法检查和优化。

3) 汇编：将 hello.s 翻译为二进制机器码，得到可重定位目标文件 hello.o。

4) 链接：将 hello.o 同等动态库等连接，生成可执行目标文件 hello。

5) 创建进程：通过 shell 运行 hello 程序。shell 通过 fork 创建子进程，通过 execve 运行 hello。

6) 访问内存：通过 MMU 将 hello 中的虚拟地址转换为实际的物理地址，再通过多级缓存读取数据。

7) 异常：程序执行过程中，如果从键盘输入 Ctrl-C 等命令，会给进程发送一个异常信号，然后通过信号处理函数对信号进行处理。

8) 结束：hello 运行完后会由父进程（shell 进程）回收，内核会删除对应的数据结构。

通过一个学期的 CSAPP 学习，我了解了一个代码从产生到实现的每一个过程以及整个过程中经历的各种细节。曾经的我只知道数组开太大会编译错误，int 值太大会变成负数，但从来没由了解过为什么以及怎么样，如今了解了一台电脑底层的活动之后，尤其是储存器结构，虚拟内存，机器级表示的相关知识之后对程序编写有了更加深入的认识。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

文件名	作用
hello.c	源代码
hello.i	hello.c 预处理生成的文本文件
hello.s	hello.i 编译后得到的汇编语言文本文件
hello.o	hello.s 汇编后得到的可重定位目标文件
hello	hello.o 链接后得到的汇编语言文本文件
hello.objdump	hello.o 的反汇编结果
hello.elf	hello.o 的 ELF 结构
hellor.elf	hello 的 ELF 结构

（附件 0 分，缺失 -1 分）

参考文献

[1]. 兰德尔 •E •布莱恩特等著；深入理解计算机系统[M]. 北京：机械工业出版社，2016.7.
[2]. C library - C++ Reference http://www.cplusplus.com/reference/clibrary/.[3]. ArchWiki https://wiki.archlinux.org/.
[4]. Linux man pages https://linux.die.net/man/.
[5]. printf 函数实现的深入剖析 https://www.cnblogs.com/pianist/p/3315801.html
[6]. ELF 构造：https://www.cs.stevens.edu/~jschauma/631/elf.html
[7]. 进程的睡眠、挂起和阻塞：https://www.zhihu.com/question/42962803
[8]. Wikipedia https://www.wikipedia.org/.
[9]. jaywcjlove/linux-command: Linux 命令大全搜索工具 ... - GitHubhttps://github.com/jaywcjlove/linux-command

（参考文献 0 分，缺失 -1 分）