

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机类

学　　 号 1170300720

班　　 级 1703007

学 生 陶飞宇

指 导 教 师 郑贵滨

**计算机科学与技术学院**

**2018年12月**

**摘 要**

摘要是论文内容的高度概括，应具有独立性和自含性，即不阅读论文的全文，就能获得必要的信息。摘要应包括本论文的目的、主要内容、方法、成果及其理论与实际意义。摘要中不宜使用公式、结构式、图表和非公知公用的符号与术语，不标注引用文献编号，同时避免将摘要写成目录式的内容介绍。

**关键词：**关键词1；关键词2；……；

**（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 -](#_Toc532238396)

[1.1 Hello简介 - 4 -](#_Toc532238397)

[1.2 环境与工具 - 4 -](#_Toc532238398)

[1.3 中间结果 - 4 -](#_Toc532238399)

[1.4 本章小结 - 4 -](#_Toc532238400)

[第2章 预处理 - 5 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 5 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 5 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 5 -](#_Toc532238404)

[2.4 本章小结 - 5 -](#_Toc532238405)

[第3章 编译 - 6 -](#_Toc532238406)

[3.1 编译的概念与作用 - 6 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 6 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 6 -](#_Toc532238409)

[3.4 本章小结 - 6 -](#_Toc532238410)

[第4章 汇编 - 7 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 7 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 7 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 7 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 7 -](#_Toc532238415)

[4.5 本章小结 - 7 -](#_Toc532238416)

[第5章 链接 - 8 -](#_Toc532238417)

[5.1 链接的概念与作用 - 8 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 8 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 8 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 8 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 8 -](#_Toc532238422)

[5.6 hello的执行流程 - 8 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 8 -](#_Toc532238424)

[5.8 本章小结 - 9 -](#_Toc532238425)

[第6章 hello进程管理 - 10 -](#_Toc532238426)

[6.1 进程的概念与作用 - 10 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 10 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 10 -](#_Toc532238429)

[6.4 Hello的execve过程 - 10 -](#_Toc532238430)

[6.5 Hello的进程执行 - 10 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 10 -](#_Toc532238432)

[6.7本章小结 - 10 -](#_Toc532238433)

[第7章 hello的存储管理 - 11 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 11 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 11 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 11 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 11 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 11 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 11 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 11 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 11 -](#_Toc532238442)

[7.9动态存储分配管理 - 11 -](#_Toc532238443)

[7.10本章小结 - 12 -](#_Toc532238444)

[第8章 hello的IO管理 - 13 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 13 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 13 -](#_Toc532238447)

[8.3 printf的实现分析 - 13 -](#_Toc532238448)

[8.4 getchar的实现分析 - 13 -](#_Toc532238449)

[8.5本章小结 - 13 -](#_Toc532238450)

[结论 - 14 -](#_Toc532238451)

[附件 - 15 -](#_Toc532238452)

[参考文献 - 16 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

**P2P**:hello.c经过预处理，编译，汇编，链接，生成可执行目标程序hello，在shell中启动，fork产生一个子进程，这就是**From Program to Process**的过程。

**O2O**：execve加载hello，映射虚拟内存，载入物理内存，进入main函数执行目标代码，CPU为运行的hello分配时间片，执行逻辑控制流，运行结束后，回收hello进程，删除相关的结构。

以上整个过程为**From Zero-0 to Zero-0。**

## 1.2 环境与工具

## **硬件环境**：Intel Core i7-8750HQ CPU @ 2.20GHz,16G RAM,256G SSD +1T HDD.

## **软件环境**：Ubuntu18.04.1 LTS

## **开发与调试工具**：vim，gcc，as，ld，edb，readelf，HexEdit

## 1.3 中间结果



## 1.4 本章小结

本章介绍了 hello 的 p2p，020 过程，列出了本次实验的环境和中间结果。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

**概念**：预处理器根据以字符#开头的命令，修改原始的C程序

**作用**：

1. **宏定义**；

宏定义又称为宏代换、宏替换，简称“宏”。

格式：

#define标识符文本

其中的标识符就是所谓的符号常量，也称为“宏名”。

预处理（预编译）工作也叫做宏展开：将宏名替换为文本（这个文本可以是字符串、可以是代码等）。

掌握"宏"概念的关键是“换”。一切以换为前提、做任何事情之前先要换，准确理解之前就要“换”。

例：

#define PI 3.1415926

把程序中全部的标识符PI换成3.1415926

1. **文件包含**；

一个文件包含另一个文件的内容

格式：

#include "文件名"

或

#include <文件名>

编译时以包含处理以后的文件为编译单位，被包含的文件是源文件的一部分。

编译以后只得到一个目标文件.obj

被包含的文件又被称为“标题文件”或“头部文件”、“头文件”，并且常用.h作扩展名。

修改头文件后所有包含该文件的文件都要重新编译

头文件的内容除了函数原型和宏定义外，还可以有结构体定义，全局变量定义

例：

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

**3.条件编译**。

有些语句希望在条件满足时才编译。

使用条件编译可以使目标程序变小，运行时间变短。

例：

格式：（1）

#ifdef 标识符

程序段1

#else

程序段2

#endif

或

#ifdef

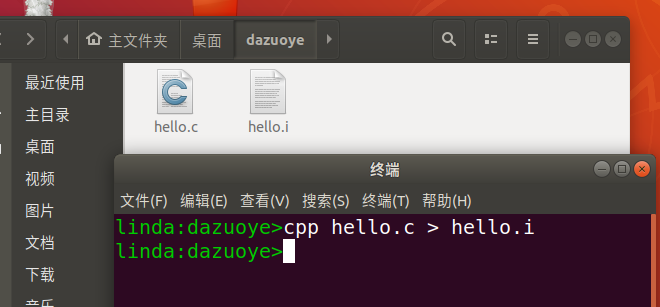
程序段1

#endif

当标识符已经定义时，程序段1才参加编译。

## 2.2在Ubuntu下预处理的命令

命令：cpp hello.c > hello.i

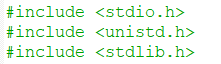


## 2.3 Hello的预处理结果解析



hello.i中变成了几千行的文件，最后的是main函数。

前面的几千行是hello.c中



的体现，三句语句告诉预处理器读取这些系统头文件，直接插入文本中，得到了hello.i的文件。

## 2.4 本章小结

本章介绍了预处理的定义与作用。并且实践了hello.i的生成，亲身体会了#include语句是如何将系统头文件插入程序文本中的。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译器将文本文件 hello.i 翻译成文本文件 hello.s，它包含一个汇编语言程序。

**主要作用：**

1.扫描（词法分析）

2.语法分析

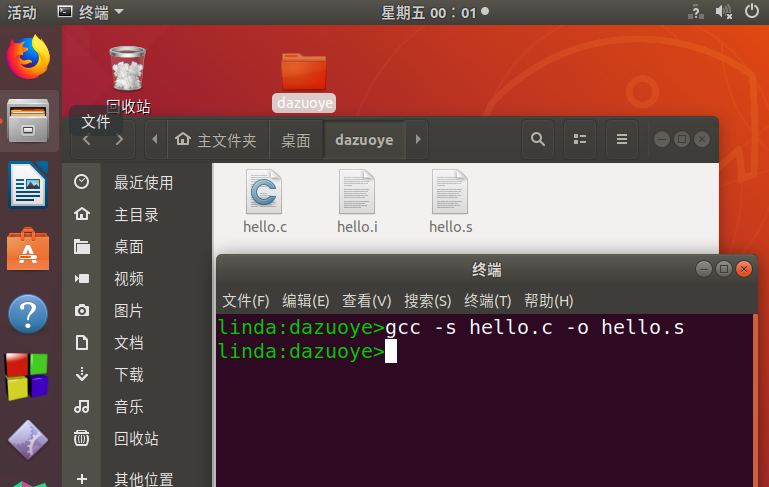
3.语义分析

4.源代码优化（中间语言生成）

5.代码生成，目标代码优化。

## 3.2 在Ubuntu下编译的命令

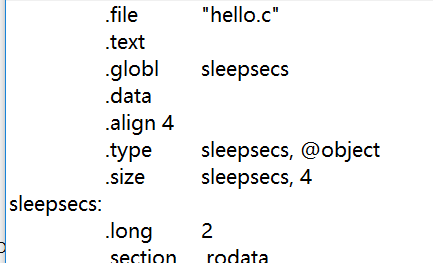
命令：gcc -S hello.i -o hello.s

3.3 Hello的编译结果解析

3.3.1 数据

**整数**

1. int sleepsecs（全局变量）



在最开始声明。

.globl sleepsecs定义其为全局变量

.type sleepsecs, @object将其设置为对象而不是函数

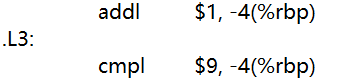
.align 4 设置对齐方式为4字节对齐

.size sleepsecs, 4将其大小设置为4字节

.long 2将其设置为long类型，大小为2

（int变为long可能是编译器的设置）

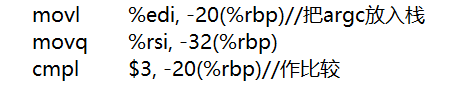
1. int i（for循环中）



i被放在栈中，即-4(%rbp)的位置

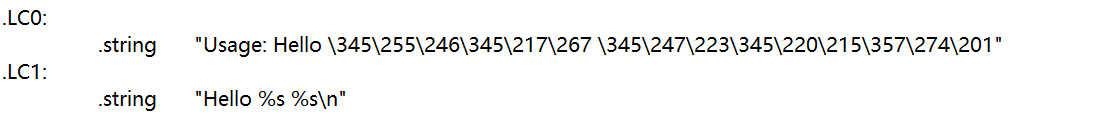
1. int argc

作为main函数的第一个参数放在%rdi寄存器中，后续与3的比较验证了这一点，如图：



1. 其他整数，其他的整数，如3,10这些都作为汇编语言中的立即数出现。

**字符串**

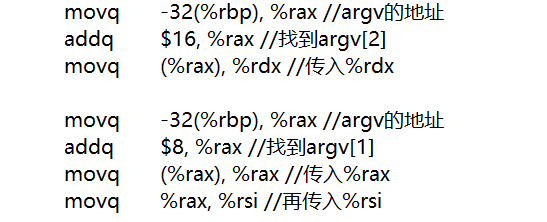


声明了两个字符串，第一个是"Usage: Hello 学号 姓名！\n"，其中学号，姓名两个字符串被声明为UTF-8格式。

第二个是"Hello %s %s\n"，两个%s为格式化参数

3.3.2数组

存储着一个字符串类型的指针数组，一个指针8字节，在.s以此找到argv[1]和argv[2]的位置，如图：



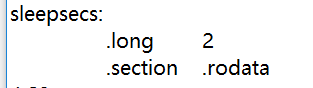
3.3.3 算术操作

程序中出现的算术操作有：

1. addl $1, -4(%rbp)，对应i++操作，后面之所以是4，是因为i为 4字节。
2. leaq .LC1(%rip), %rdi，取地址运算，取出"Hello %s %s\n"作为printf的参数。

3.3.4 隐式类型转换

int sleepsecs=2.5;



2.5的默认类型为double，所以这里把double转换为int（实为long）。

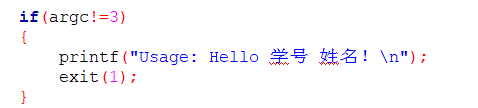
采取的舍入方式是向零舍入，2.5变为2。

3.3.5关系操作

①argc!=3

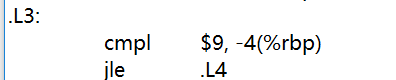


-20(%rbp)中为argc，与3进行比较，相等则跳入L2，不等则继续，决定了是否执行以下语句：



②i<10

这里处理为<=9



3.3.6函数操作

**参数传递**

64位，前6个参数在寄存器中，按顺序为%rdi,%rsi,%rdx,%rcx,%r8,%r9.

如main函数，%rdi中为argc的值，%rsi中为argv数组的头地址

**函数调用**

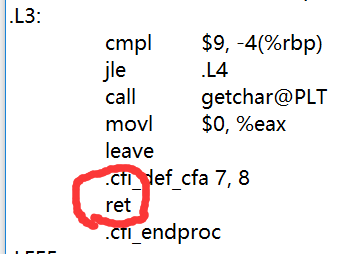
利用call指令，如：



把调用指令的下一条指令的地址压入栈中

**函数返回**

利用ret指令，如：

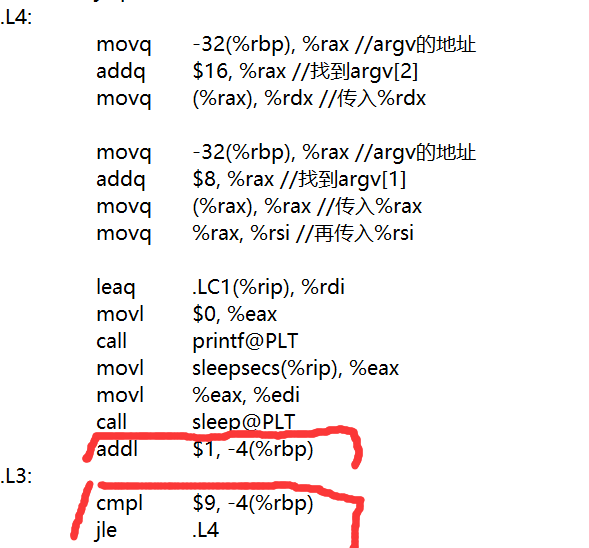


把call压入栈中的指令弹出并跳到这条指令的位置。

3.3.7控制转移

1. if(argc!=3)同上
2. for (i=0;i<10;i++)

L4为循环体，L3为循环控制结构



## 3.4 本章小结

本章主要介绍了编译器是如何处理 C 语言的各个数据类型以及各类操作。

编译器将.i编译为.s 的汇编代码。经过编译之后，我们的 hello 自C 语言变为汇编语言。

**（第3章2分）**

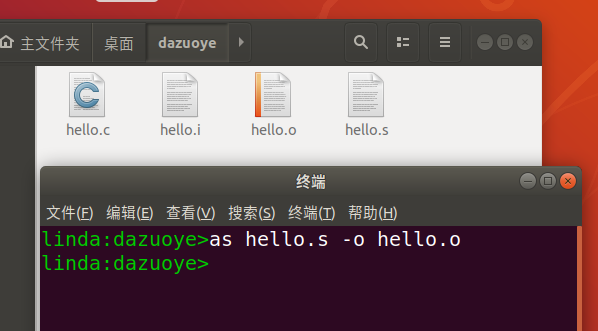
# 第4章 汇编

## 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。

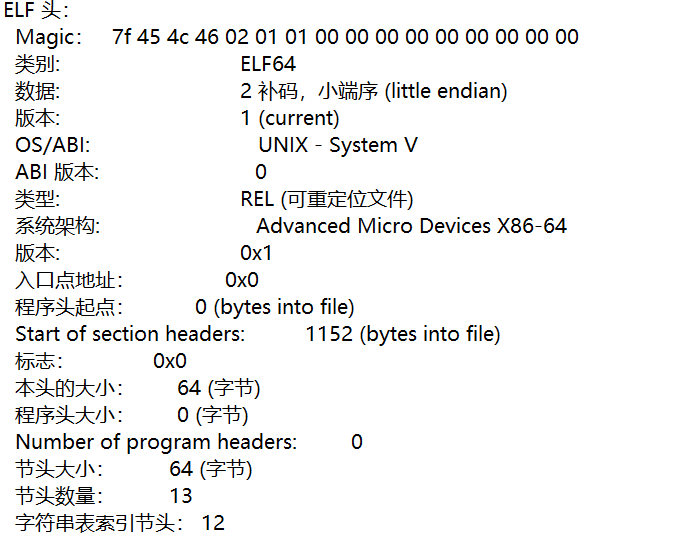
## 4.2 在Ubuntu下汇编的命令

指令：as hello.s -o hello.o

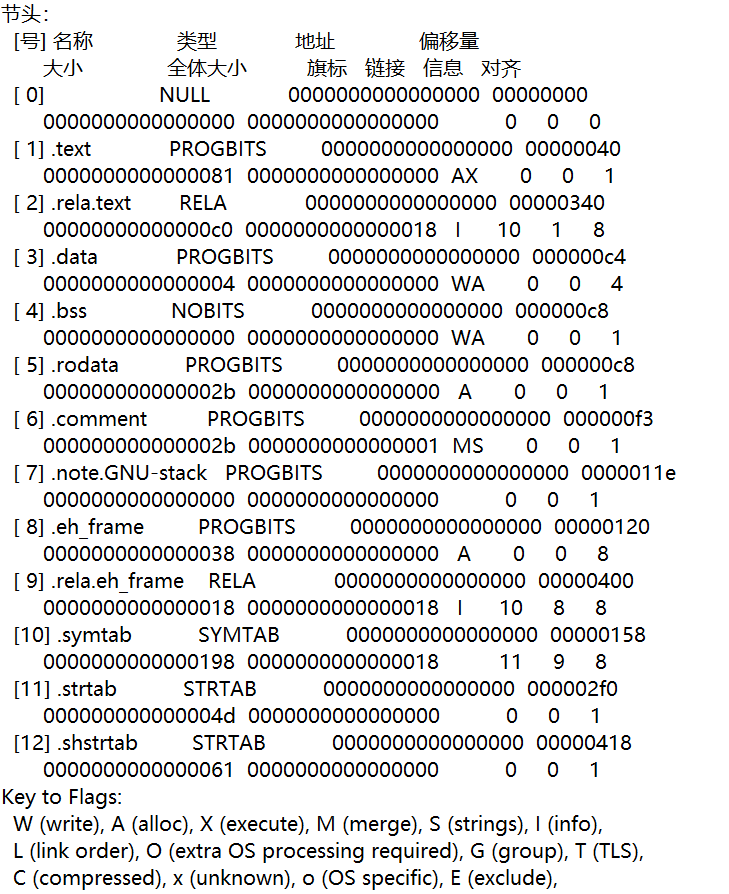


## 4.3 可重定位目标elf格式

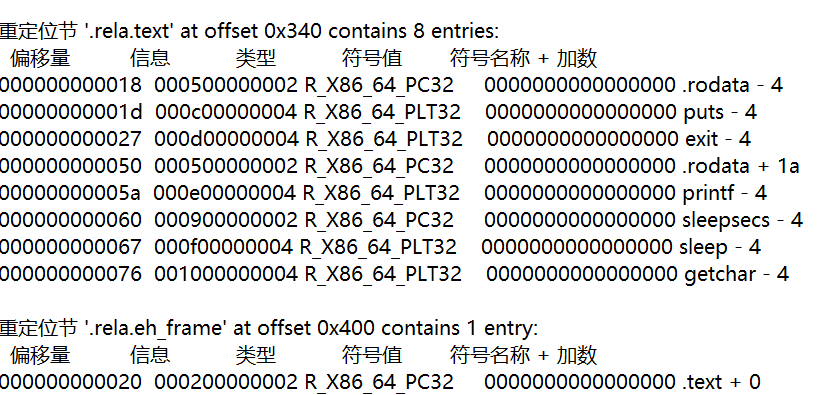
**ELF头：**以一个16字节的序列开始，这个序列描述了生成该文件的系统的字的大小和字节顺序



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



重定位节.rela.text代码的重定位条目，告诉链接器在将目标文件合并成可执行文件时如何修改这个应用。



**偏移量（offset）**：需要被修改的引用的节偏移。

**信息（info）**：提供了符号表中的一个位置，同时还包括重定位类型的有个信息。这是通过将值划分为两部分来达到的。该值的低8位表示重定位入口的类型， 高24位表示重定位入口的符号在符号表重的下标。

**类型（type）**：告知连接器如何修改新的引用

**符号名称（name）**：重定位目标的名称

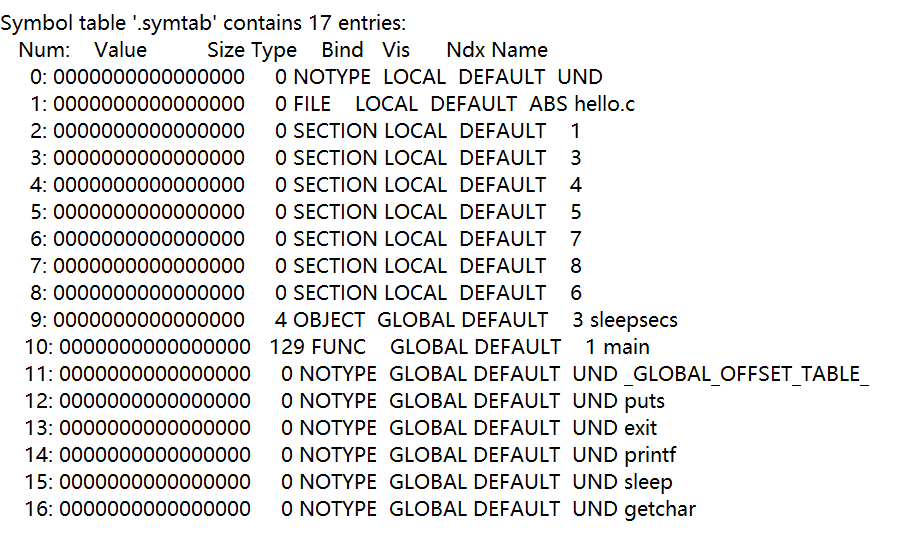
**加数（addend）**：一个有符号常数，一些类型的重定位要使用它对被修改引用的值做便宜调整

**重定位节.rela.eh\_frame**

eh\_frame的重定位节，同上

符号表：

每个重定位目标模块都有一个符号表，它包含其定义和引用的符号的信息



## 4.4 Hello.o的结果解析

反汇编和.s文件在内容上差别不大，有一些细节差别：

分支跳转：（左.s，右反汇编）



反汇编不使用段名称这样的助记符，直接使用地址。

函数调用：（左.s，右反汇编）



在.s 文件中，函数调用之后直接跟着函数名称，在反汇编程中，call 的目标地址是当前下一条指令。

因为 hello.c 中调用的都是共享库中的函数，需要通过动态链接器确定地址，在汇编成为机器语言的时候，将其call指令后的相对地址设置为0

## 4.5 本章小结

本章介绍了 hello 从 hello.s 到 hello.o 的过程，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。

**（第4章1分）**

# 第5章 链接

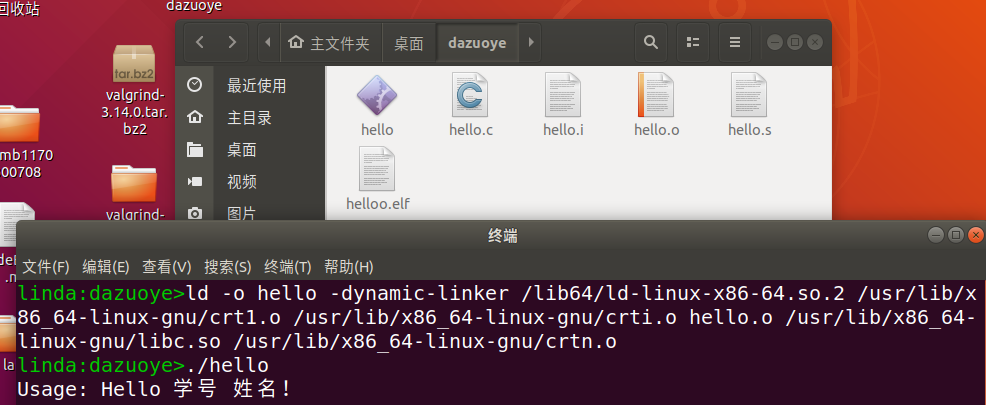
## 5.1 链接的概念与作用

## 链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可 被加载到内存并执行。链接可以执行于编译时，也就是在源代码被编译成机器代 码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至 于运行时，也就是由应用程序来执行。链接是由叫做链接器的程序执行的。链接 器使得分离编译成为可能。

## 5.2 在Ubuntu下链接的命令

命令：

ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o hello.o /usr/lib/x86\_64-linux-gnu/libc.so /usr/lib/x86\_64-linux-gnu/crtn.o



## 5.3 可执行目标文件hello的格式



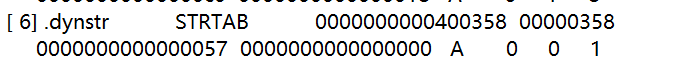
节头部表中包含了各节的基本信息：

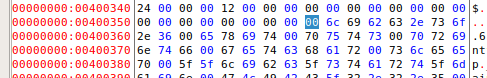
大小（size），类型（type），地址（address），offset（偏移量）

其中address为被载入到的虚拟地址的起始地址，size为大小

## 5.4 hello的虚拟地址空间

在data dumo块中可以看到，以.dynstr和.init为例，起始地址的确对应着被载入后的虚拟地址



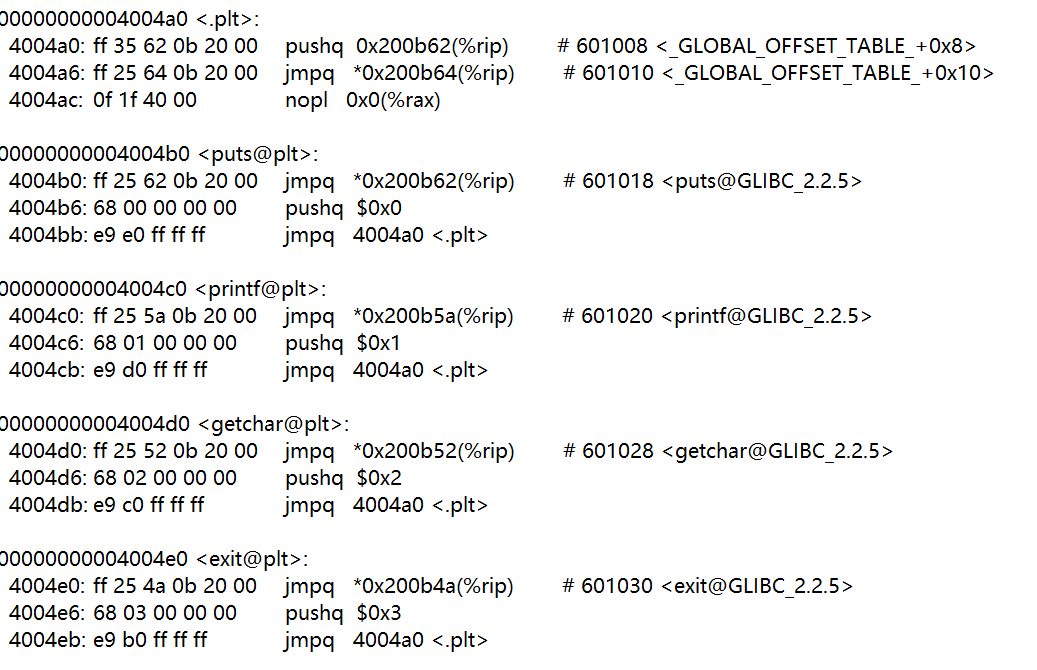






使用edb加载hello，查看本进程的虚拟地址空间各段信息，并与5.3对照分析说明。

## 5.5 链接的重定位过程分析



可执行文件中，与.o文件的差别在于：

1. 多出了很多函数，这些函数是链接器从共享库中提取出来的，main函数所调用的函数，把它加入可执行目标文件使其完整
2. 调用函数的方式：call后面跟上的是所调用的函数的实际地址，而不是下一条指令的地址，因为链接器已经帮我们计算出了位置。

## 5.6 hello的执行流程

程序名：

ld-2.27.so!\_dl\_start

ld-2.27.so!\_dl\_init

hello!\_start

libc-2.27.so!\_\_libc\_start\_main

-libc-2.27.so!\_\_cxa\_atexit

-libc-2.27.so!\_\_libc\_csu\_init

hello!\_init

libc-2.27.so!\_setjmp

-libc-2.27.so!\_sigsetjmp

--libc-2.27.so!\_\_sigjmp\_save

hello!main

hello!puts@plt

hello!exit@plt

\*hello!printf@plt

\*hello!sleep@plt

\*hello!getchar@plt

ld-2.27.so!\_dl\_runtime\_resolve\_xsave

-ld-2.27.so!\_dl\_fixup

--ld-2.27.so!\_dl\_lookup\_symbol\_x

libc-2.27.so!exit

## 5.7 Hello的动态链接分析

## 对于动态共享链接库中 PIC 函数，编译器没有办法预测函数的运行时地址，所以需要添加重定位记录，等待动态链接器处理，为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表 PLT+全局偏移量表GOT实现函数的动态链接，GOT中存放函数目标地址，PLT使用GOT中地址跳转到目标函数。

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

## 5.8 本章小结

在本章中主要介绍了链接的概念与作用、hello 的 ELF 格式，hello 的 虚拟地址空间、重定位过程、执行流程、动态链接过程。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

## 进程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。

## 进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需要的状态组成的。这个状态包括存放在内存中的程序的代码和数据，他的栈，通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

## 6.2 简述壳Shell-bash的作用与处理流程

## 作用：Shell 是一个用 C 语言编写的程序，他是用户使用 Linux 的桥梁。 Shell 是指一种应用程序，Shell 应用程序提供了一个界面，用户通过这个界面访问 操作系统内核的服务。

## **处理流程：**

## 1）从终端读入输入的命令。

## 2）将输入字符串切分获得所有的参数

## 3）如果是内置命令则立即执行

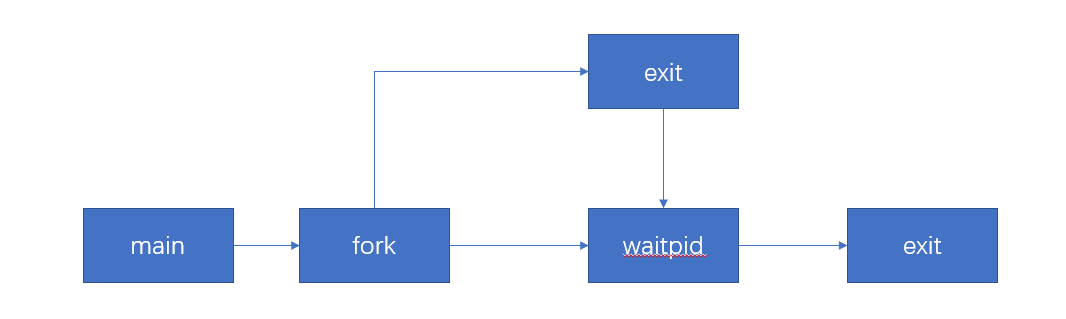
## 4）否则调用相应的程序为其分配子进程并运行

## 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

## 输入./hello之后，调用 fork 函数创建一个新的运行的子进程，新创建的子进程几乎但不完全 与父进程相同，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的） 一份副本，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。父进程与子进程之间最大的区别在于它们拥有不同的 PID。

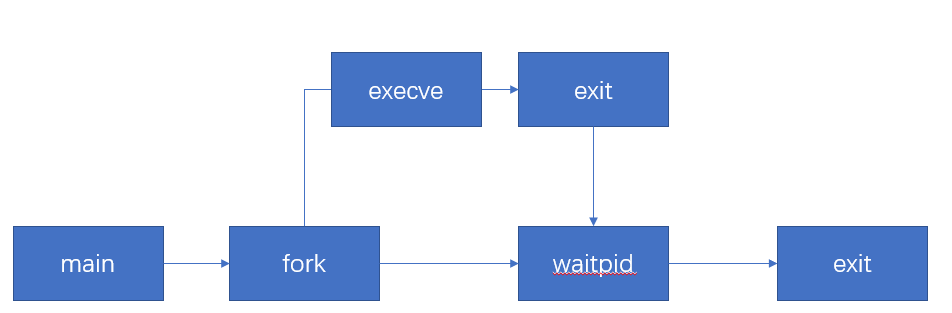
如图：



## 6.4 Hello的execve过程

调用fork函数之后，在子进程中加载execve函数，载入并运行hello函数。

覆盖当前进程的代码，数据，栈。保留相同的PID，继承已打开的文件描述符和信号上下文。



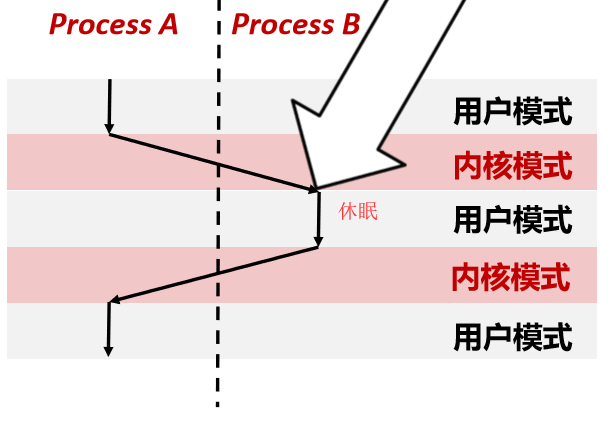
## 6.5 Hello的进程执行

## 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。

## 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

以调用sleep函数为例，如图：



调用sleep之后，

1. 内核处理sleep发出的休眠请求，将hello挂起。（内核）
2. hello被挂起，其他进程获得当前进程控制权（用户）
3. 休眠结束，收到中断信号（内核）
4. hello不再挂起重新恢复，进行自己的逻辑控制流。

## 6.6 hello的异常与信号处理

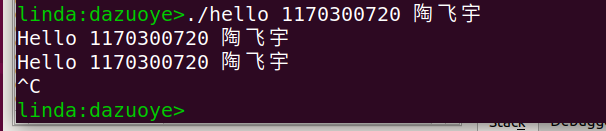
**乱按结果：**



乱按其中有一个回车，在第四行的最后。

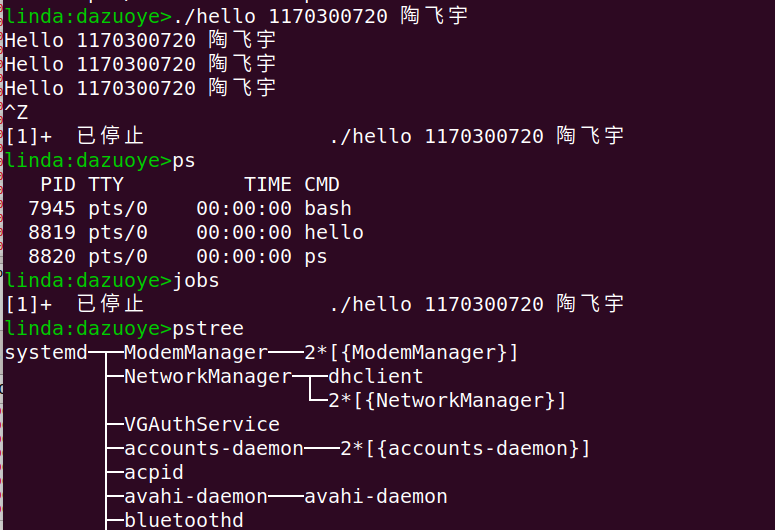
可以看到，程序只是把输入缓存在stdin中，由于程序最后有一个getchar，读取这之前的一个字符串作为输入，剩下的则在执行完毕之后全部打印在屏幕上。

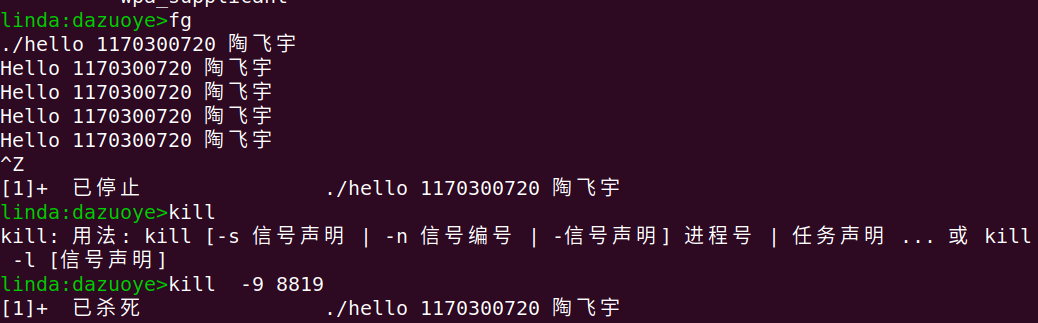
**输入ctrl+C：**



如图，打印两条信息后按ctrl+C，进程被中断收回。

**输入ctrl+Z及命令：**





键入Ctrl+z进程被挂起，有ps可以看出这一点，调用jobs也能说明其当前的状态。再调用pstree查看当前的进程树。

调用fg命令把刚才挂起的hello进程转到前台，可以看到其在继续运行。

最后调用一个kill指令发送终止信号杀死了hello进程。

## 6.7本章小结

本章中，介绍了进程的定义与作用， Shell 的一般处理流程，调用 fork 创建新进程，调用 execve 执行 hello，hello 的进程执行，hello 的异常与信号处理等内容

**（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

**物理地址(physical address)**  
用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。

**虚拟内存(virtual memory)**  
这是对整个内存的抽象描述。它是相对于物理内存来讲的，可以直接理解成“不直实的”，“假的”内存

**逻辑地址(logical address)**  
Intel为了兼容，将远古时代的段式内存管理方式保留了下来。逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。

**线性地址(linear address)或也叫虚拟地址(virtual address)**  
跟逻辑地址类似，它也是一个不真实的地址，如果逻辑地址是对应的硬件平台段式管理转换前地址的话，那么线性地址则对应了硬件页式内存的转换前地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

**基本原理**

        在段式存储管理中，将程序的地址空间划分为若干个段(segment)，这样每个进程有一个二维的地址空间。在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在段式存储管理系统中，则为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。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。

      在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法等方法。

      在回收某个段所占用的空间时，要注意将收回的空间与其相邻的空间合并。

      段式存储管理也需要硬件支持，实现逻辑地址到物理地址的映射。

      程序通过分段划分为多个模块，如代码段、数据段、共享段：

      –可以分别编写和编译

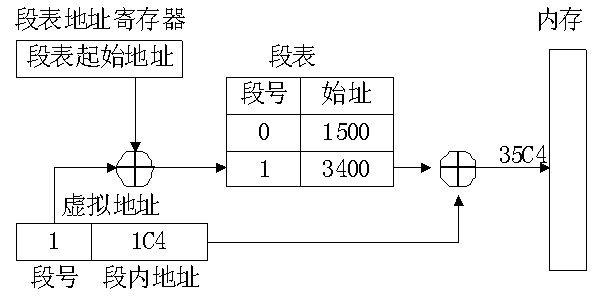
      –可以针对不同类型的段采取不同的保护

      –可以按段为单位来进行共享，包括通过动态链接进行代码共享

      这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。

       总的来说，段式存储管理的优点是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。缺点与页式存储管理的缺点相同，进程必须全部装入内存。

**地址变换**

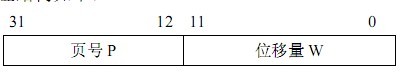


        在段式管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址(见图4—5)。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

**基本原理**

        将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载时，可将任意一页放人内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部构成，前一部分是页号，后一部分为页内地址w（位移量），如图所示：



      页式管理方式的优点是：

       1）没有外碎片，每个内碎片不超过页大比前面所讨论的几种管理方式的最大进步是，

       2）一个程序不必连续存放。

       3）便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。

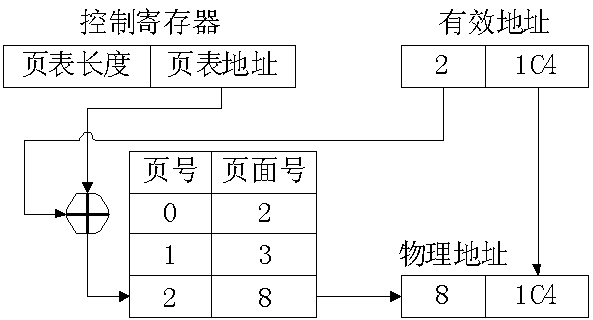
      缺点是：要求程序全部装入内存，没有足够的内存，程序就不能执行。

**地址变换**

 在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。

       原理：CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址(见图4-4)。

        逻辑页号，页内偏移地址－>查进程页表，得物理页号－>物理地址：



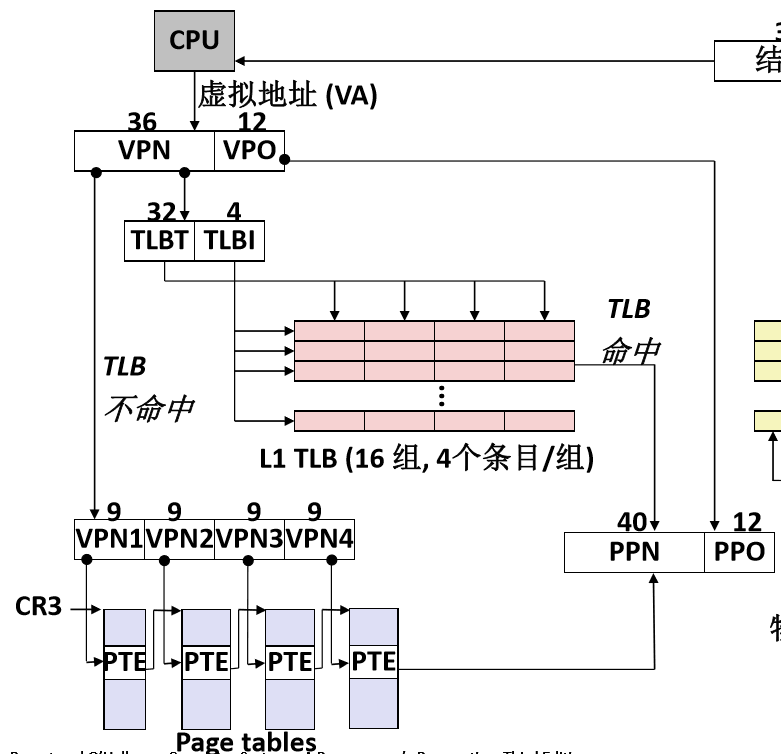
       上述过程通常由处理器的硬件直接完成，不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存：

       第一次用来查找页表将操作数的 逻辑地址变换为物理地址；

       第二次完成真正的读写操作。

## 

## 7.4 TLB与四级页表支持下的VA到PA的变换

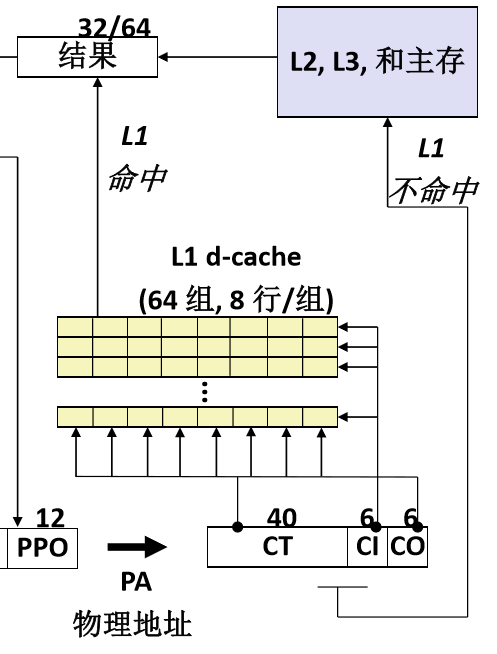


流程如图所示：

①CPU产生虚拟地址，前36位VPN与TLB匹配，如果匹配到，得到PPN与VPO（即PPO）组成物理地址。

②若TLB不命中，进入页表。VPN在四级链表中，每四分之一，即9位是一级链表的偏移量，而每一级链表中（除第四级）存储着的是下一级链表的起始地址，由此一层层进入下一层链表，直到在第四级链表中取出PPN构成物理地址。

## 7.5 三级Cache支持下的物理内存访问



如图所示，物理地址被分为CT（标志位）CI（索引位）CO（偏移量）

根据CI在cache中找到对应组，根据CT找到匹配的标志位，若改行的有效位为1，则命中，根据CO找到数据块传输，流程完毕。

若找不到对应tag或者标志位0，则发生了不命中，则向L2,L3,和主存中依次查询数据，将其放到L1当中。放置时，如果有空闲块直接放在空闲块里，如果没有，根据LRU等算法寻找牺牲块替换。

## 7.6 hello进程fork时的内存映射

## 当 fork 函数被 shell 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。将这两个进程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

## 7.7 hello进程execve时的内存映射

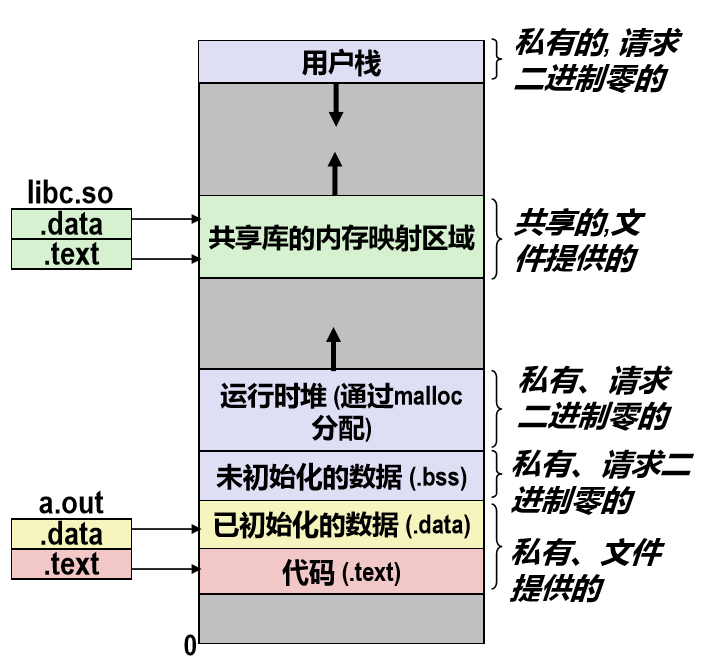
execve函数在当前进程中 加载并运行新程序hello的 步骤

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。

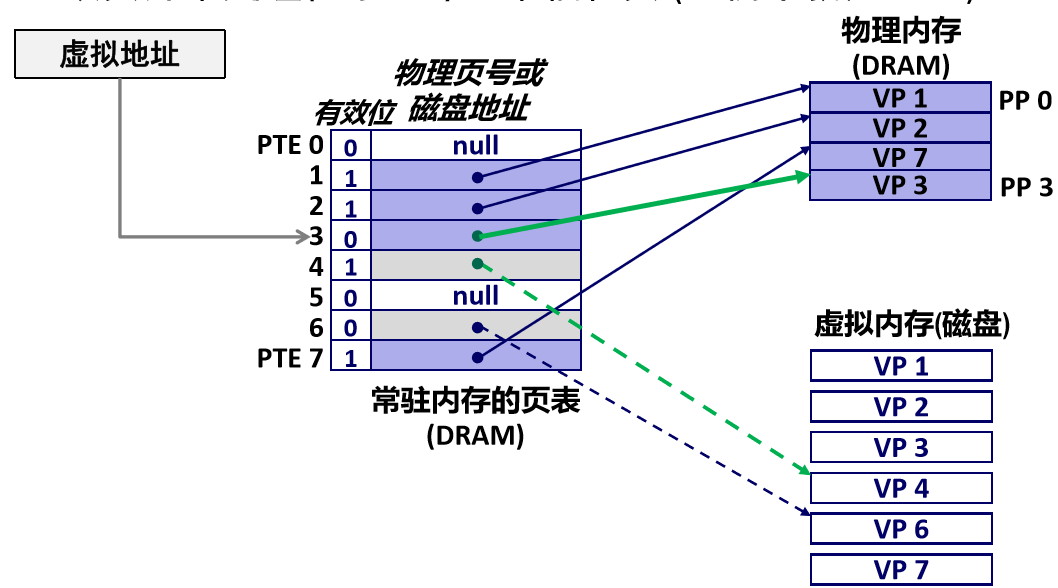


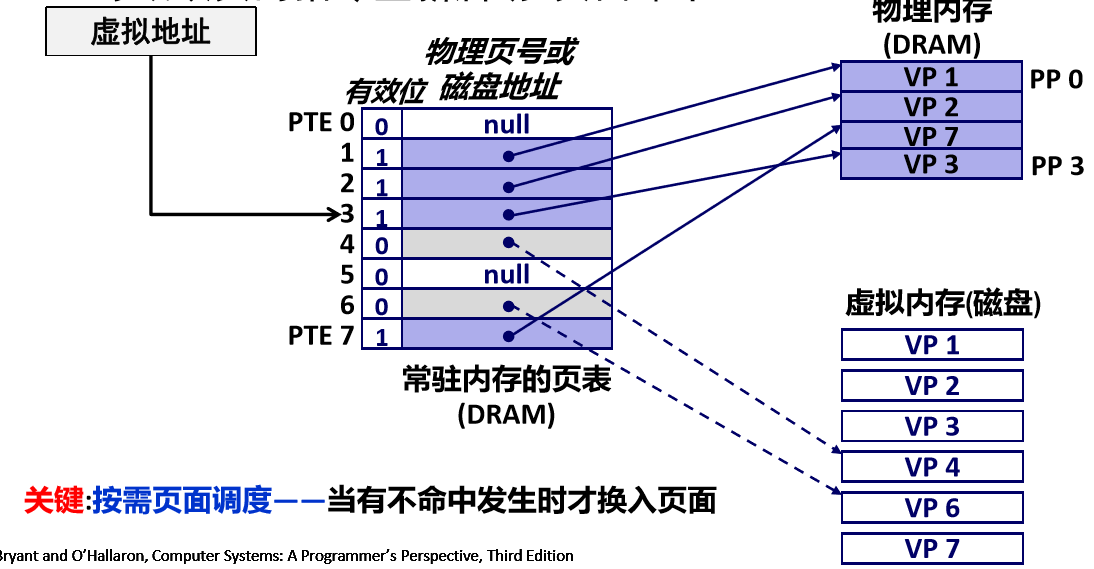
## 7.8 缺页故障与缺页中断处理

页面不命中导致缺页(缺页异常) 

缺页异常处理程序选择一个牺牲页 (VP 4) 

导致缺页的指令重新启动: 页面命中





缺页中断处理：缺页处理程序是系统内核中的代码，选择一个牺牲页面，如果 这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表。当缺 页处理程序返回时，CPU重新启动引起缺页的指令，这条指令再次发送虚拟地址到 内存管理单元，这次内存管理单元就能正常翻译虚拟地址了

## 7.9动态存储分配管理

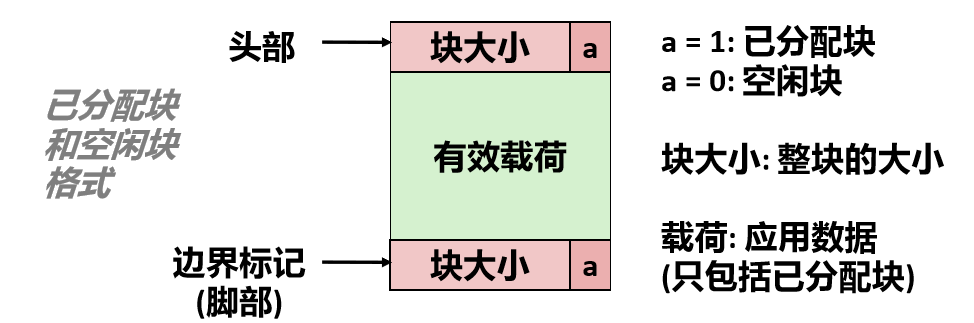
动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为 一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已 分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用 来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器、隐式分配器。

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。

**带边界标签的隐式空闲链表**

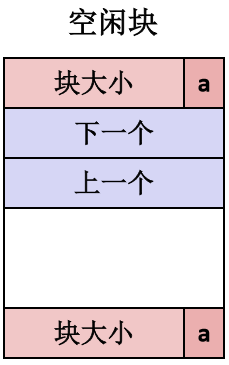


一个块是有一个字的头部、有效载荷，以及可能的一些额外的填充组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。

再为每个块添加一个脚部，脚部就是头部的一个副本。如果每个块包括一个脚部，那么分配器可以通过检查它的脚部，判断前面一个块的起始位置和状态。这个脚部总是在距当前块开始位置一个字的距离。

**显示空间链表**

将空闲块组织委员某种形式的显式数据结构，堆组织成一个双向空闲链表，在每个空闲块中，都包含一个前驱和后继指针，如图



使用两种排序策略：

一种方法使用后进先出的顺序维护链表，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。

## 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， VA 到PA 的变换、物理内存访问， ello 进程 fork 时的内存映射、execve 时的内存映射、缺页故障与缺页中断处理、动态 存储分配管理。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

## 所有的 IO 设备都被模型化为文件，而所有的输入和输出都被 当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O

## 8.2 简述Unix IO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

printf的函数本身：

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;   
    }

其中arg 获得第二个参数，即输出的时候格式化串。

而其中的vsprintf为：

int vsprintf(char \*buf, const char \*fmt, va\_list args)

{

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);

}

vsprintf 程序按照格式 fmt 结合参数 args 生成字符串，并返回字串的长度。

write 函数：

write:   
     mov eax, \_NR\_write   
     mov ebx, [esp + 4]   
     mov ecx, [esp + 8]   
     int INT\_VECTOR\_SYS\_CALL   
其中在 printf 中调用write(buf,i)将长度为 i 的 buf 输出到屏幕上。

在 write 函数中，将栈中参数放入寄存器，ecx 是字符个数，ebx 存放第一个 字符地址，int INT\_VECTOR\_SYS\_CALLA 代表通过系统调用 syscall，查看 syscall 的实现：

sys\_call:   
       
     ;ecx中是要打印出的元素个数   
     ;ebx中的是要打印的buf字符数组中的第一个元素   
     ;这个函数的功能就是不断的打印出字符，直到遇到：'\0'   
     ;[gs:edi]对应的是0x80000h：0采用直接写显存的方法显示字符串   
     xor si,si   
     mov ah,0Fh   
     mov al,[ebx+si]   
     cmp al,'\0'   
     je .end   
     mov [gs:edi],ax   
     inc si   
    loop:   
     sys\_call   
      
    .end:   
     ret

从寄存器中通过总线复制到显卡的显存中，存储的是字符的 ASCII 码。

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）。

就此字符串就打印在了屏幕上

## 8.4 getchar的实现分析

## **异步异常-键盘中断的处理**：当用户按键时，键盘接口会得到一个代表该按键 的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子 程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码 转换成 ASCII 码，保存到系统的键盘缓冲区之中。

## **getchar 函数**落实到底层调用了系统函数 read，通过系统调用 read 读取存储在 键盘缓冲区中的 ASCII 码直到读到回车符然后返回整个字串，getchar 进行封装， 大体逻辑是读取字符串的第一个字符然后返回。

## 8.5本章小结

本章介绍了 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

**过程：**

1. 预处理，将hello.c调用的外部库归入到hello.i中
2. 编译，将hello.i编译为汇编文件hello.s
3. 汇编，将汇编文件hello.s变为可重定位目标文件hello.o
4. 链接，将hello.o和其他的可重定位目标文件和动态链接库链接成可执行目标文件hello
5. 运行，输入./hello 1170300720 taofeiyu
6. fork，调用fork，创建子进程
7. execve，调用execve函数，加载映射虚拟内存，进入程序入口后再加载物理内存，最后进入main函数。
8. 执行，CPU为hello分配资源，享有控制逻辑流
9. 访存，内存管理单元将VA翻译为PA
10. 申请动态内存，printf调用malloc，申请堆中内存
11. 信号，如果有Ctrl+c，Ctrl+z键入，停止，挂起hello
12. 结束，子进程被父进程或者init回收，其享有的资源被回收

**感悟：**

计算机系统是一个复杂又简单的整体，深入理解它为我们的计算机学习打下了坚而厚实的基础。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件



**（附件0分，缺失 -1分）**

# 参考文献

[1] Bryant,R.E. 《深入理解计算机系统》. 北京：机械工业出版社，2016-11-15

[2] Pianistx， [转]printf 函数实现的深入剖析

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

**（参考文献0分，缺失 -1分）**