

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 软件工程

学　　 号 1173710223

班　　 级 1737102

学 生 张峙岳

指 导 教 师 吴锐

**计算机科学与技术学院**

**2018年12月**

**摘 要**

“Great souls often reside in ordinary bodies.”是英国一句有名的谚语，其寓意为“伟大的灵魂常栖息于平凡的身躯中”，“Hello world!”是一个非常简单的程序，也是每个刚开始学习计算机的“菜鸟”所认识的第一个程序。随着我们学习课程的不断深入，我们所讨论的话题也逐渐开始，从“printf”，“scanf”这些基本的函数，过渡到“深度学习”，“区块链”这些高端前卫的概念。而“Hello world!”这个最简单也最平凡的程序，似乎也早已渐渐被我们遗忘。可是，”Hello world!”对于CS这门学科来说，真的仅仅只是一个普通的程序吗？本文将通过hello的一生，总结并归纳计算机程序的编译，运行，回收的具体步骤，深入总结和理解计算机系统组成原理。

hello是开始，hello亦是结束。

**关键词：平凡而伟大，hello，计算机系统，剖析**；

**（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”并非“Peer-to-Peer”，而是“From Program to Process”（从程序到进程）。事实上，在我们最初接触计算机的，老师往往会带领我们认识并使用1-2种的计算机语言编译器，而编译器帮助我们对我们所编写的程序进行了编译，执行（事实上一般编译运行都是一个按钮一起进行的orz），以及程序运行结束后“杀死”进程（回收）都由编译器代替我们进行了合并步骤并完成。所以对于我们在语言学习阶段，一个程序的使用只需要两步“编译运行”，“关闭”。

然而正如人作出一项决定需要经过反复的思考琢磨比较，最终才去决定执行一样，一个程序从程序到进程的历程也并非那么简单。将在在之后的论述中去描述。

## 1.2 环境与工具

## 1.2.1 硬件环境



## 1.2.2 软件环境

## Windows10 64位；Vmware Pro；Ubuntu 14.04 LTS 64位；

## 1.2.3 开发工具

## CodeBlocks；vi/vim/gpedit+gcc;

## 1.3 中间结果

## hello.i（预处理源程序）

## hello.s（汇编程序）

## hello.o（可重定位目标程序）

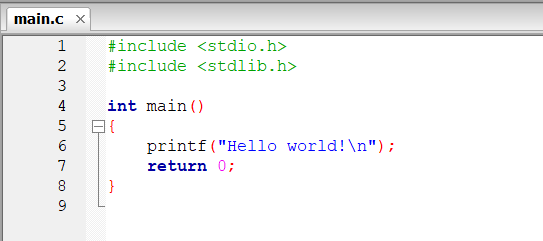
hello.elf(hello的elf头信息)

asm.txt（hello反汇编）

## 1.4 本章小结

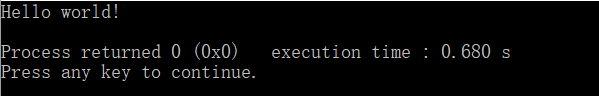
“Hello world!”是一个非常简单的程序，也是每个刚开始学习计算机的“菜鸟”所认识的第一个程序。随着我们学习课程的不断深入，我们所讨论的话题也逐渐开始，从“printf”，“scanf”这些基本的函数，过渡到“深度学习”，“区块链”这些高端前卫的概念。而“Hello world!”这个最简单也最平凡的程序，似乎也早已渐渐被我们遗忘。可是，”Hello world!”对于CS这门学科来说，真的仅仅只是一个普通的程序吗？

将“Hello”和“World”一起使用的程序最早出现于1972年，在贝尔实验室成员Brian Kernighan撰写的内部技术文件《Introduction to the Language B》之中，而后又因为在Brian Kernighan 和[Dennis M. Ritchie](https://baike.baidu.com/item/Dennis%20M.%20Ritchie)合著的《The C Programme Language》使用而广泛流行。因为它的简洁，实用，使得它作为最常用的第一个演示程序，而沿用至今。



(新建一个code blocks项目时自动生成的”Hello world!”程序)

而“Hello”也同样是第一个“P2P”程序，只不过，这个“P2P”并非“Peer-to-Peer”，而是“From Program to Process”（从程序到进程）。事实上，在我们最初接触计算机的，老师往往会带领我们认识并使用1-2种的计算机语言编译器，而编译器帮助我们对我们所编写的程序进行了编译，执行（事实上一般编译运行都是一个按钮一起进行的orz），以及程序运行结束后“杀死”进程（回收）都由编译器代替我们进行了合并步骤并完成。所以对于我们在语言学习阶段，一个程序的使用只需要两步“编译运行”，“关闭”。



然而正如人作出一项决定需要经过反复的思考琢磨比较，最终才去决定执行一样，一个程序的历程也并非那么简单。

**（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(cpp) 根据以字符#开头的命令，修改原始的C 程序。

作用：预编译过程主要处理那些源代码中以#开始的预编译指令，主要处理规则如下：

1）将所有的#define删除，并且展开所有的宏定义；

2）处理所有条件编译指令，如#if，#ifdef等；

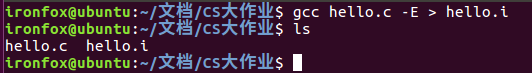
3）处理#include预编译指令，将被包含的文件插入到该预编译指令的位置。该过程递归进行，及被包含的文件可能还包含其他文件。

4）删除所有的注释//和 /\*\*/；

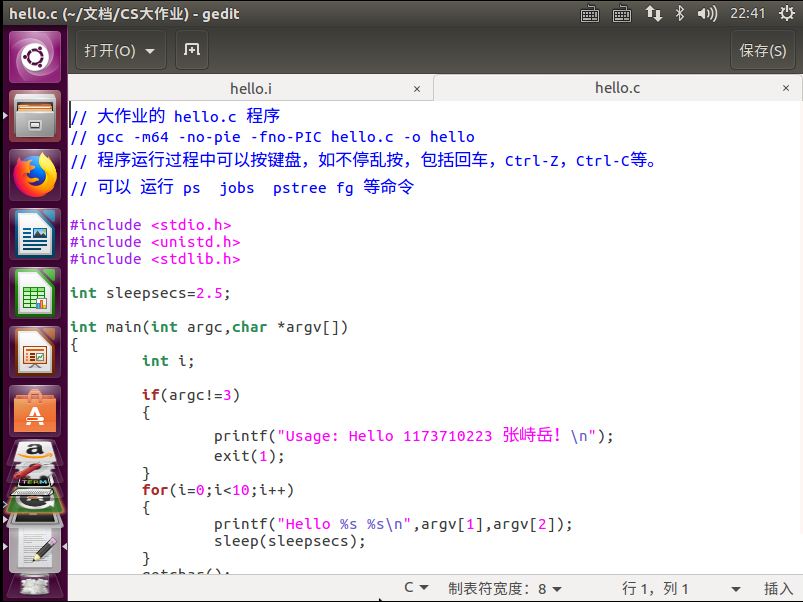
5）添加行号和文件标识，如#2 “hello.c” 2,以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号信息；

6）保留所有的#pragma编译器指令，因为编译器须要使用它们；

## 2.2在Ubuntu下预处理的命令



## 2.3 Hello的预处理结果解析



（源程序代码）



（保存得到的预处理文件hello.i，注意到int main之前有3000+行）

通过对比hello.c以及hello.i两个文档，我们不难发现在预处理阶段，我们的注释被删除掉且在主函数前插入了所使用到的头文件。

## 2.4 本章小结

预处理过程去除了人工助于理解的注释和引入了头文件，帮助程序由从便于人理解转化为便于机器理解，是编译中重要的一部分。

**（第2章0.5分）**

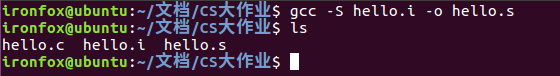
# 第3章 编译

## 3.1 编译的概念与作用

概念：这里的编译不是指程序从源文件到二进制程序的全部过程，而是指将经过预处理之后的程序转换成特定汇编代码(assembly code)的过程。

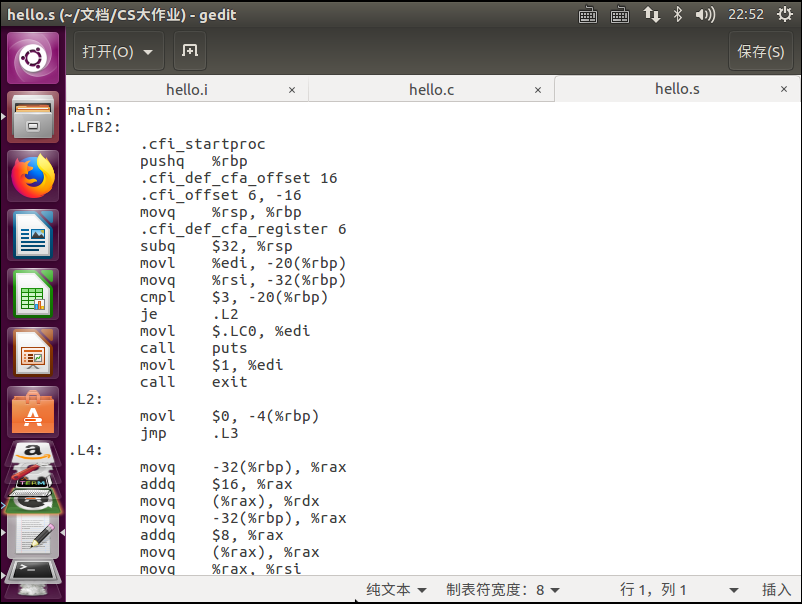
作用：经过预处理之后的程序转换成特定汇编代码

## 3.2 在Ubuntu下编译的命令



（编译的指令）

## 3.3 Hello的编译结果解析



（得到程序的汇编代码）

3.3.1 数据

有变量int sleepsecs，编译器将其编译成

.type sleepsecs, @object

.size sleepsecs, 4

3.3.2 赋值

赋值语句sleepsecs=2.5编译器将其编译成

.long 2

.section .rodata

赋值语句i=0编译器将其编译成

movl $0, -4(%rbp)

3.3.3 类型转换（显示或隐式）

由于sleepsecs是int型的而2.5是float类型的，这就有一个隐式的类型转换，编译器将2.5隐式地转换成了2存入sleepsecs。

3.3.4 算术操作

编译器将i++编译成

addl $1, -4(%rbp)

3.3.5 关系操作

编译器将i<10编译成

cmpl $9, -4(%rbp)

jle .L4

将argc!=3编译成

cmpl $3, -20(%rbp)

je .L2

3.3.6 数组/指针/结构操作

printf函数对指针和对数组的操作编译器编译为：

movq -32(%rbp), %rax

addq $16, %rax

movq (%rax), %rdx

movq -32(%rbp), %rax

addq $8, %rax

movq (%rax), %rax

movq %rax, %rsi

3.3.7 控制转移

编译器将if(argc!=3)编译成：

cmpl $3, -20(%rbp)

je .L2

将for循环里面的比较和转移编译成：

cmpl $9, -4(%rbp)

jle .L4

3.3.8 函数操作

编译器将printf("Usage: Hello 1173710223 张峙岳！\n");编译为：

movl $.LC0, %edi

call puts

将printf("Hello %s %s\n",argv[1],argv[2]);编译为：

movq -32(%rbp), %rax

addq $16, %rax

movq (%rax), %rdx

movq -32(%rbp), %rax

addq $8, %rax

movq (%rax), %rax

movq %rax, %rsi

movl $.LC1, %edi

movl $0, %eax

call printf

将sleep(sleepsecs);编译为：

movl sleepsecs(%rip), %eax

movl %eax, %edi

call sleep

## 3.4 本章小结

编译器（ccl）将Hello.i文件翻译成文本文件Hello.s，这是个汇编语言程序。注意，汇编语言是计算机底层世界通用通用的语言，不同的高级语言、不同的编译器，最终得到的还是一样的汇编语言。

**（第3章2分）**

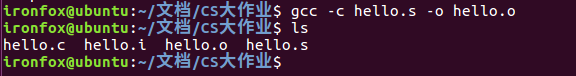
# 第4章 汇编

## 4.1 汇编的概念与作用

概念：汇编语言（assembly language）是一种用于电子计算机、微处理器、微控制器或其他可编程器件的低级语言，亦称为符号语言。在汇编语言中，用助记符（Mnemonics）代替机器指令的操作码，用地址符号（Symbol）或标号（Label）代替指令或操作数的地址。在不同的设备中，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。普遍地说，特定的汇编语言和特定的机器语言指令集是一一对应的,不同平台之间不可直接移植。可以说，汇编语言是与人与计算机沟通的一道桥梁。

作用：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(relocatable object program) 的格式，并将结果保存在目标文件hello.o 中。汇编器将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。汇编相对于编译过程比较简单，只需要根据汇编指令和机器指令的对照表一一进行翻译。

## 4.2 在Ubuntu下汇编的命令



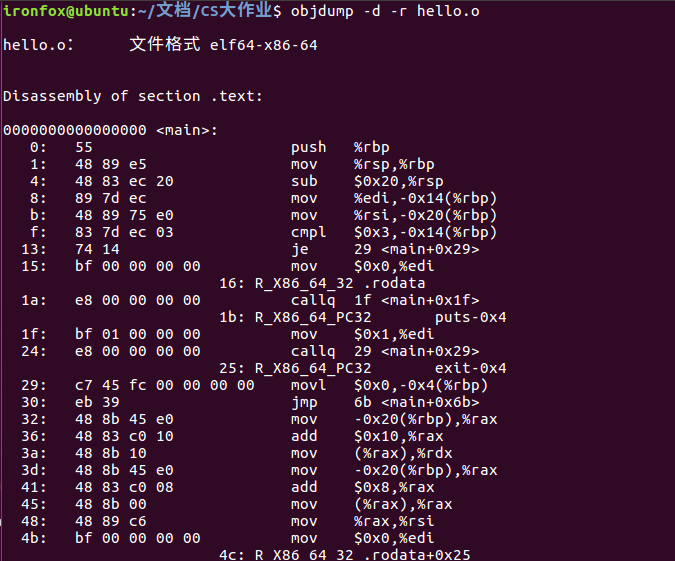
（汇编过程）

## 4.3 可重定位目标elf格式



（hello头部文件信息）

## 4.4 Hello.o的结果解析



与反汇编文件对比可知，在汇编过程中，程序为每条语句加上了具体的地址，全局变量和常量都被安排到了具体的地址里面，操作数在hello.s里面的十进制转为到到hello.o里面的机器级十六进制程序，跳转语句jx&jxx原来对应的符号都变成了相对偏移地址，函数调用时原来的函数名字也被替换成了函数的相对偏移地址。

4.5 本章小结

汇编语言（assembly language）是一种用于电子计算机、微处理器、微控制器或其他可编程器件的低级语言，亦称为符号语言。在汇编语言中，用助记符（Mnemonics）代替机器指令的操作码，用地址符号（Symbol）或标号（Label）代替指令或操作数的地址。在不同的设备中，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。普遍地说，特定的汇编语言和特定的机器语言指令集是一一对应的,不同平台之间不可直接移植。

许多汇编程序为程序开发、汇编控制、辅助调试提供了额外的支持机制。有的汇编语言编程工具经常会提供宏，它们也被称为宏汇编器。

汇编语言不像其他大多数的程序设计语言一样被广泛用于程序设计。在今天的实际应用中，它通常被应用在底层，硬件操作和高要求的程序优化的场合。驱动程序、嵌入式操作系统和实时运行程序都需要汇编语言。

**（第4章1分）**

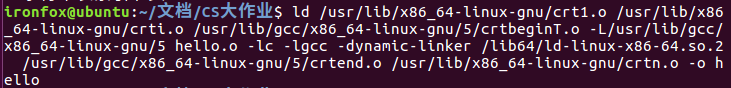
# 第5章 链接

## 5.1 链接的概念与作用

概念： 在hello程序中调用了printf函数，它是每个C编译器都会提供的标准C库中的一个函数。printf函数存在于一个名为printf.o的单独的预编译好了的目标文件中，而这个文件必须通过链接合并到我们的Hello.o文件中。

作用：链接过程将多个目标文以及所需的库文件(.so等)链接成最终的可执行文件(executable file)。链接器（ld）就是做这个工作的，它最终会输出一个可执行目标文件hello，这个可执行目标文件可以被加载到内存中，由系统执行。

## 5.2 在Ubuntu下链接的命令



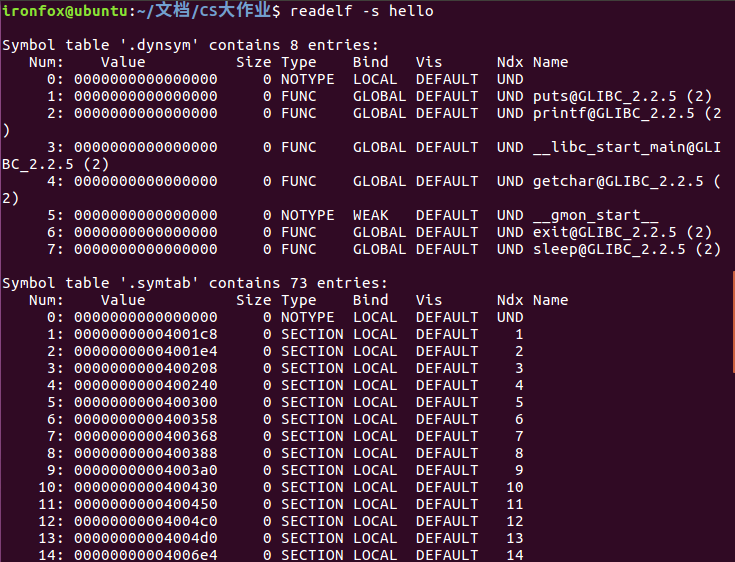
（链接指令）

至此为止，我们终于得到了我们的可执行文件hello，真是可喜可贺~

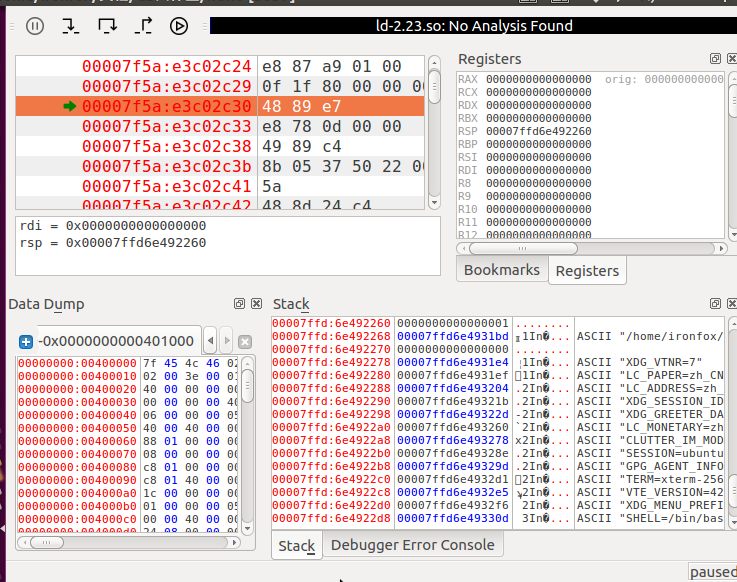
C:\Users\dell\AppData\Roaming\Tencent\Users\1693426820\TIM\WinTemp\RichOle\`9E5DDNSD0Q}@D{R3SM8`1C.png

## 5.3 可执行目标文件hello的格式

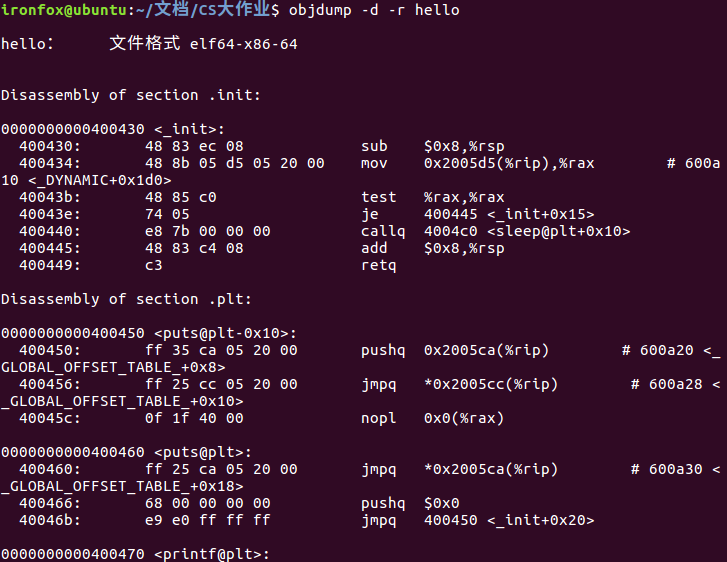
![](B$LK0H`QY]51B8H7JYN$F](data:image/png;base64,)



## 5.4 hello的虚拟地址空间



## 5.5 链接的重定位过程分析



## 5.6 hello的执行流程

## 0x00000000004004d0 in \_start ()

## 0x0000000000400480 in \_\_libc\_start\_main@plt ()

## 0x0000000000400670 in \_\_libc\_csu\_init ()

## 0x0000000000400430 in \_init ()

## 0x00000000004005b0 in frame\_dummy ()

## 0x0000000000400540 in register\_tm\_clones ()

## 0x00000000004005f2 in main ()

## 0x0000000000400460 in puts@plt ()

## 0x00000000004004a0 in exit@plt ()

## 0x0000000000400580 in \_\_do\_global\_dtors\_aux ()

## 0x0000000000400500 in deregister\_tm\_clones ()

## 0x00000000004006e4 in \_fini ()

## 0x00000000004004d0 in \_start ()

## 0x0000000000400480 in \_\_libc\_start\_main@plt ()

## 0x0000000000400670 in \_\_libc\_csu\_init ()

## 0x0000000000400430 in \_init ()

## 0x00000000004005b0 in frame\_dummy ()

## 0x0000000000400540 in register\_tm\_clones ()

## 0x00000000004005f2 in main ()

## 0x0000000000400460 in puts@plt ()

## 0x00000000004004a0 in exit@plt ()

## 0x0000000000400580 in \_\_do\_global\_dtors\_aux ()

## 0x0000000000400500 in deregister\_tm\_clones ()

## 0x00000000004006e4 in \_fini ()

## 5.7 Hello的动态链接分析

\_GLOBAL\_OFFSET\_TABLE\_的变化



## 5.8 本章小结

我们的所编写的程序hello的一生到这里就……当然还没有完！我们通过预处理-编译-汇编-链接后得到的可执行程序hello，可是计算机如何执行这个程序呢？这是计算机运行的核心问题。即使已经编写好程序，但程序是死的。只有活的进程才能产出。我们的hello从程序到进程的漫漫征程才刚刚开始。

**（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：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

进程的概念主要有两点：第一，进程是一个实体。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（text region）、数据区域（data region）和堆栈（stack region）。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；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为进程。

作用：程序是一个没有生命的实体，只有处理器赋予程序进程时（操作系统执行之），它才能成为一个活动的实体。

## 6.2 简述壳Shell-bash的作用与处理流程

概念：在计算机科学中，Shell俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（命令解析器）。它类似于DOS下的command.com和后来的cmd.exe。它接收用户命令，然后调用相应的应用程序。

它又是一种程序设计语言。作为命令语言，它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

文字操作系统与外部最主要的接口就叫做shell。shell是操作系统最外面的一层。shell管理你与操作系统之间的交互：等待你输入，向操作系统解释你的输入，并且处理各种各样的操作系统的输出结果。

shell提供了你与操作系统之间通讯的方式。这种通讯可以以交互方式（从键盘输入，并且可以立即得到响应），或者以shell script(非交互）方式执行。shell script是放在文件中的一串shell和操作系统命令，它们可以被重复使用。本质上，shell script是命令行命令简单的组合到一个文件里面。

shell基本上是一个命令解释器，类似于DOS下的command。它接收用户命令（如ls等），然后调用相应的应用程序。较为通用的shell有标准的Bourne shell (sh）和C shell (csh）。

基本上shell分两大类：

图形界面shell（Graphical User Interface shell 即 GUI shell），应用最为广泛的 Windows Explorer （微软的windows系列操作系统），还有也包括广为人知的 Linux shell，其中linux shell 包括 X window manager (BlackBox和FluxBox），以及功能更强大的CDE、GNOME、KDE、 XFCE。

命令行式shell（Command Line Interface shell ，即CLI shell），例如bash / sh / ksh / csh（Unix/linux 系统）

作用：人机交互

## 6.3 Hello的fork进程创建过程

概念：shell通过调用fork 函数创建一个新的运行的子进程。也就是hello程序，hello进程几乎但不完全与shell相同。Hello进程得到与shell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。hello进程还获得与shell任何打开文件描述符相同的副本，这就意味着当shell调用fork 时，hello可以读写shell中打开的任何文件。shell和hello进程之间最大的区别在于它们有不同的PID。

hello子进程

fork

shell

父进程

作用：

## 6.4 Hello的execve过程

execve 函数加载并运行可执行目标文件filename, 且带参数列表argv 和环境变量列表envp 。只有当出现错误时，例如找不到filename, execve 才会返回到调用程序。所以，与fork 一次调用返回两次不同， execve 调用一次并从不返回。

execve()

hello

shell

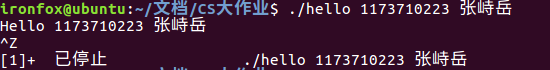
## 6.5 Hello的进程执行

## 当一个程序调用fork的时候，实际上就是将进程的内存空间，包括text, global data, heap和stack，又复制出来一个，构成一个新的进程，并在内核中为改进程创建新的附加信息 (比如新的PID，而PPID为原进程的PID)。此后，两个进程分别地继续运行下去。新的进程和原有进程有相同的运行状态(相同的变量值，相同的instructions...)。我们只能通过进程的附加信息来区分两者。

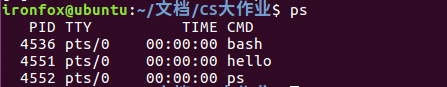
## 每个进程包括一些进程附加信息，包括PID，PPID，PGID等，用来说明进程的身份、进程关系以及其它统计信息。这些信息并不保存在进程的内存空间中。内核会为每个进程在内核自己的空间中分配一个变量(task\_struct结构体)以保存上述信息。内核可以通过查看自己空间中的各个进程的附加信息就能知道进程的概况，而不用进入到进程自身的空间 (就好像我们可以通过门牌就可以知道房间的主人是谁一样，而不用打开房门)。每个进程的附加信息中有位置专门用于保存接收到的信号(正如我们在Linux信号中所说的“信箱”)。

## 程序调用exec的时候，进程清空自身内存空间的text, global data, heap和stack，并根据新的程序文件重建text, global data, heap和stack (此时heap和stack大小都为0)，并开始运行。

## 6.6 hello的异常与信号处理



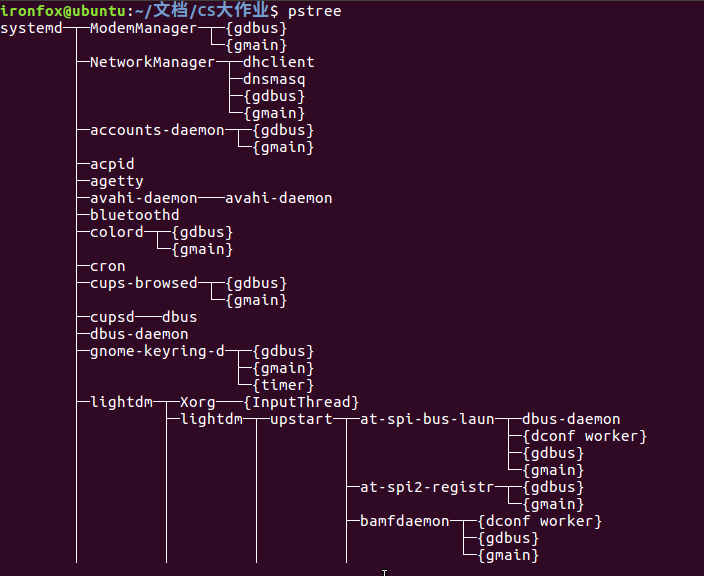
（将程序停止）



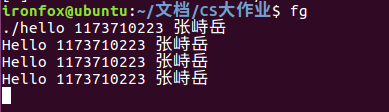
（查看ps）

C:\Users\dell\AppData\Roaming\Tencent\Users\1693426820\TIM\WinTemp\RichOle\JUJ~%DN9$X[1~$A3O7U~@21.png

(查看jobs)



（pstree）



（fg）

## 6.7本章小结

当我们运行一个程序时，这个程序实在bash中运行的。其实是bash用fork函数创建了一个子进程，子进程复制了bash的映像，然后在子进程的映像中调用了exec系列的函数，将我们写的程序的可执行文件（映像）替换为子进程从父进程那里复制来的映像。然后执行我们写的程序的可执行文件。

fork之后bash下建立了一个子进程，开始这个子进程和bash是共用一个PCB的，当子进程发生改变时，子进程复制了父进程的PCB并调用了exec系列函数，将你要运行的程序的进程的PCB替换掉你从父进程那里复制来的PCB。然后执行你的程序。

**（第6章1分）**

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

逻辑地址（Logical Address） 是指由程式产生的和段相关的偏移地址部分。例如，你在进行C语言指针编程中，能读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，他是相对于你当前进程数据段的地址，不和绝对物理地址相干。只有在Intel实模式下，逻辑地址才和物理地址相等（因为实模式没有分段或分页机制,Cpu不进行自动地址转换）；

逻辑也就是在Intel保护模式下程式执行代码段限长内的偏移地址（假定代码段、数据段如果完全相同）。应用程式员仅需和逻辑地址打交道，而分段和分页机制对你来说是完全透明的，仅由系统编程人员涉及。应用程式员虽然自己能直接操作内存，那也只能在操作系统给你分配的内存段操作。

线性地址（Linear Address） 是逻辑地址到物理地址变换之间的中间层。程式代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386的线性地址空间容量为4G（2的32次方即32根地址总线寻址）。

物理地址（Physical Address） 是指出目前CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

虚拟地址（Virtual Memory）是指计算机呈现出要比实际拥有的内存大得多的内存量。因此他允许程式员编制并运行比实际系统拥有的内存大得多的程式。这使得许多大型项目也能够在具有有限内存资源的系统上实现。一个非常恰当的比喻是：你不必非常长的轨道就能让一列火车从上海开到北京。你只需要足够长的铁轨（比如说3公里）就能完成这个任务。采取的方法是把后面的铁轨即时铺到火车的前面，只要你的操作足够快并能满足需求，列车就能象在一条完整的轨道上运行。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

## Intel要求两次转换，这样虽说是兼容了，但是却是很冗余。

## 另一方面，其它某些硬件平台，没有二次转换的概念，Linux也需要提供一个高层抽像，来提供一个统一的界面。所以，Linux的段式管理，事实上只是“哄骗”了一下硬件而已。

## 按照Intel的本意，全局的用GDT，每个进程自己的用LDT——不过Linux则对所有的进程都使用了相同的段来对指令和数据寻址。即用户数据段，用户代码段，对应的，内核中的是内核数据段和内核代码段。

## include/asm-i386/segment.h

## #define GDT\_ENTRY\_DEFAULT\_USER\_CS 14

## #define \_\_USER\_CS (GDT\_ENTRY\_DEFAULT\_USER\_CS \* 8 + 3)

## #define GDT\_ENTRY\_DEFAULT\_USER\_DS 15

## #define \_\_USER\_DS (GDT\_ENTRY\_DEFAULT\_USER\_DS \* 8 + 3)

## #define GDT\_ENTRY\_KERNEL\_BASE 12

## #define GDT\_ENTRY\_KERNEL\_CS (GDT\_ENTRY\_KERNEL\_BASE + 0)

## #define \_\_KERNEL\_CS (GDT\_ENTRY\_KERNEL\_CS \* 8)

## #define GDT\_ENTRY\_KERNEL\_DS (GDT\_ENTRY\_KERNEL\_BASE + 1)

## #define \_\_KERNEL\_DS (GDT\_ENTRY\_KERNEL\_DS \* 8)

## 复制代码

## 把其中的宏替换成数值，则为：

## #define \_\_USER\_CS 115 [00000000 1110 0 11]

## #define \_\_USER\_DS 123 [00000000 1111 0 11]

## #define \_\_KERNEL\_CS 96 [00000000 1100 0 00]

## #define \_\_KERNEL\_DS 104 [00000000 1101 0 00]

## 复制代码

## 方括号后是这四个段选择符的16位二制表示，它们的索引号和T1字段值也可以算出来了

## \_\_USER\_CS index= 14 T1=0

## \_\_USER\_DS index= 15 T1=0

## \_\_KERNEL\_CS index= 12 T1=0

## \_\_KERNEL\_DS index= 13 T1=0

## 复制代码

## T1均为0，则表示都使用了GDT，再来看初始化GDT的内容中相应的12-15项(arch/i386/head.S)：

## .quad 0x00cf9a000000ffff /\* 0x60 kernel 4GB code at 0x00000000 \*/

## .quad 0x00cf92000000ffff /\* 0x68 kernel 4GB data at 0x00000000 \*/

## .quad 0x00cffa000000ffff /\* 0x73 user 4GB code at 0x00000000 \*/

## .quad 0x00cff2000000ffff /\* 0x7b user 4GB data at 0x00000000 \*/

## 按照前面段描述符表中的描述，可以把它们展开，发现其16-31位全为0，即四个段的基地址全为0。

## 这样，给定一个段内偏移地址，按照前面转换公式，0 + 段内偏移，转换为线性地址，可以得出重要的结论，“在Linux下，逻辑地址与线性地址总是一致（是一致，不是有些人说的相同）的，即逻辑地址的偏移量字段的值与线性地址的值总是相同的。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

## 原理上来讲，Linux只需要为每个进程分配好所需数据结构，放到内存中，然后在调度进程的时候，切换寄存器cr3，剩下的就交给硬件来完成了）。

## 前面说了i386的二级页管理架构，不过有些CPU，还有三级，甚至四级架构，Linux为了在更高层次提供抽像，为每个CPU提供统一的界面。提供了一个四层页管理架构，来兼容这些二级、三级、四级管理架构的CPU。这四级分别为：

## 页全局目录PGD（对应刚才的页目录）

## 页上级目录PUD（新引进的）

## 页中间目录PMD（也就新引进的）

## 页表PT（对应刚才的页表）。

## 那么，对于使用二级管理架构32位的硬件，现在又是四级转换了，它们怎么能够协调地工作起来呢？嗯，来看这种情况下，怎么来划分线性地址吧！

## 从硬件的角度，32位地址被分成了三部份——也就是说，不管理软件怎么做，最终落实到硬件，也只认识这三位老大。

## 从软件的角度，由于多引入了两部份，，也就是说，共有五部份。——要让二层架构的硬件认识五部份也很容易，在地址划分的时候，将页上级目录和页中间目录的长度设置为0就可以了。

## 这样，操作系统见到的是五部份，硬件还是按它死板的三部份划分，也不会出错，也就是说大家共建了和谐计算机系统。

## 这样，虽说是多此一举，但是考虑到64位地址，使用四层转换架构的CPU，我们就不再把中间两个设为0了，这样，软件与硬件再次和谐——抽像就是强大呀！！！

## 例如，一个逻辑地址已经被转换成了线性地址，0x08147258，换成二制进，也就是：

## 0000100000 0101000111 001001011000

## 内核对这个地址进行划分

## PGD = 0000100000

## PUD = 0

## PMD = 0

## PT = 0101000111

## offset = 001001011000

## 现在来理解Linux针对硬件的花招，因为硬件根本看不到所谓PUD,PMD，所以，本质上要求PGD索引，直接就对应了PT的地址。而不是再到PUD和PMD中去查数组（虽然它们两个在线性地址中，长度为0，2^0 =1，也就是说，它们都是有一个数组元素的数组），那么，内核如何合理安排地址呢？

## 从软件的角度上来讲，因为它的项只有一个，32位，刚好可以存放与PGD中长度一样的地址指针。那么所谓先到PUD，到到PMD中做映射转换，就变成了保持原值不变，一一转手就可以了。这样，就实现了“逻辑上指向一个PUD，再指向一个PDM，但在物理上是直接指向相应的PT的这个抽像，因为硬件根本不知道有PUD、PMD这个东西”。

## 然后交给硬件，硬件对这个地址进行划分，看到的是：

## 页目录 = 0000100000

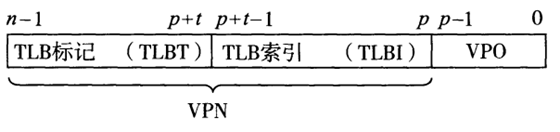
## PT = 0101000111

## offset = 001001011000

## 嗯，先根据0000100000(32)，在页目录数组中索引，找到其元素中的地址，取其高20位，找到页表的地址，页表的地址是由内核动态分配的，接着，再加一个offset，就是最终的物理地址了。

## 7.4 TLB与四级页表支持下的VA到PA的变换

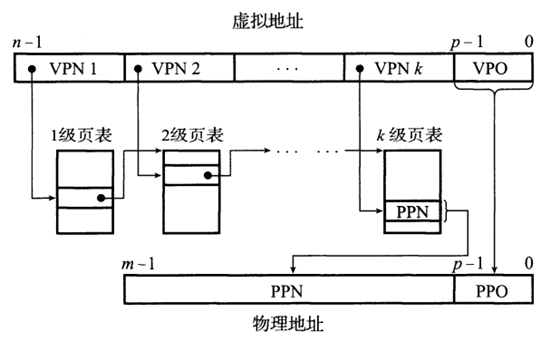
## 每次CPU产生一个虚拟地址，MMU就必须查阅相应的PTE，这显然造成了巨大的时间开销，为了消除这样的开销，MMU中存在一个关于PTE的小的缓存，称为翻译后备缓冲器（TLB）。



## 虚拟地址中用以访问TLB的组成部分

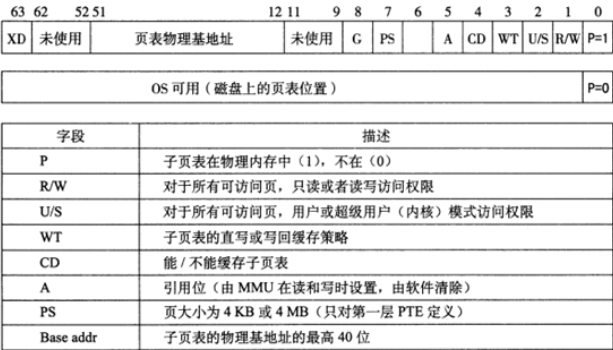
## TLB通过虚拟地址VPN部分进行索引，分为索引（TLBI）与标记（TLBT）两个部分。这样，MMU在读取PTE时会直接通过TLB，如果不命中再从内存中将PTE复制到TLB。

## TLB命中与不命中的操作图



## 在以上机制的基础上，如果所使用的仅仅是虚拟地址空间中很小的一部分，那么仍然需要一个与使用较多空间相同的页表，造成了内存的浪费。所以虚拟地址到物理地址的转换过程中还存在多级页表的机制：上一级的页表映射到下一级也表，直到页表映射到虚拟内存，如果下一级内容都未分配，那么页表项则为空，不映射到下一级，也不存在下一级页表，当分配时再创建相应页表，从而节约内存空间。

## 使用k级页表的地址翻译，具体来讲，页表条目的格式如下：



## 7.5 三级Cache支持下的物理内存访问

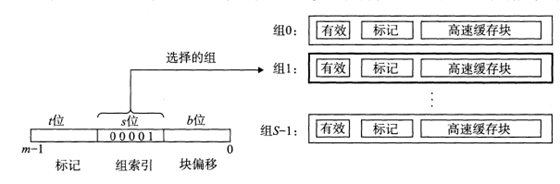
## 处理器对物理内存中数据的访问，同样需要经过缓存，即Cache，主流的处理器通常采用三级Cache。层与层之间按照以下原则进行读与写：

## 读取数据时，首先在高速缓存中查找所需字w的副本。如果命中，立即返回字w给CPU。如果不命中，从存储器层次结构中较低层次中取出包含字w的块，将这个块存储到某个高速缓存行中（可能会驱逐一个有效的行），然后返回字w。

## 下面具体三类Cache进行分析：

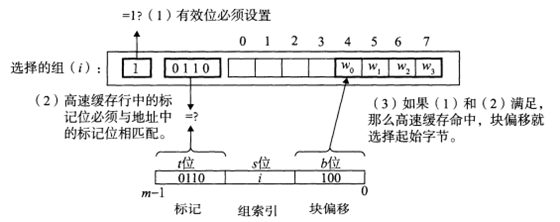
## （1）直接映射高速缓存

## 直接映射高速缓存每个组只有一行，当CPU执行一条读内存字w的指令，它会向L1高速缓存请求这个字。如果L1高速缓存中有w的一个缓存副本，那么就会得到L1高速缓存命中，高速缓存会很快抽取出w，并将它返回给CPU。否则就是缓存不命中，当L1高速缓存向主存请求包含w的块的一个副本时，CPU必须等待。当被请求块最终从内存到达时，L1高速缓存将这个块存放在它的一个高速缓存行里，从被存储的块中抽取出字w，然后将它返回给CPU。确定是否命中然后抽取的过程分为三步：1）组选择；2）行匹配；3）字抽取。



直接映射高速缓存中的组选择

组选择即从w的地址中间抽取出s个索引位，将其解释为一个对应组号的无符号整数，从而找到对应的组；行匹配即对组内的唯一一行进行判断，当有效位为1且标记位与从地址中抽取出的标记位相同则成功匹配，否则就得到不命中；而字选择即在行匹配的基础上通过地址的后几位得到块偏移，从而在高速缓存块中索引到数据。



直接映射高速缓存中的行匹配

（2）组相联高速缓存

组相联高速缓存每个组内可以多于一个缓存行，总体逻辑类似于直接映射高速缓存，不同之处在于行匹配时每组有更多的行可以尝试匹配，遍历每一行。如果不命中，有空行时也就是冷不命中则直接存储在空行；如果没有空行也就是冲突不命中，则替换已有行，通常有LFU（最不常使用）、LRU（最近最少使用）两者替换策略。

（3）全相联高速缓存

全相联高速缓存只有一个组，且这个组包含所有的高速缓存行（即E =

C/B）。对于全相联高速缓存，因为只有一个组，组选择变的十分简单。地址中不存在索引位，地址只被划分为一个标记位和一个块偏移。行匹配和字选择同组相联高速缓存。

写入数据时，假设我们要写一个已经缓存了的字w，在高速缓存中更新了它的w的副本之后，有两种方法来更新w在层次结构中紧接着低一层中的副本。分别是直写和写回，在这里分别介绍：

（1）直写

立即将w的高速缓存块写回到紧挨着的低一层中。优点是简单，缺点则是每次写都会引起总线流量。其处理不命中的方法是非写分配，即避开高速缓存，直接将这个字写到低一层去。

（2）写回

尽可能地推迟更新，只有当替换算法要驱逐这个更新过的块时，才把它写到紧接着的低一层中。优点是能显著地减少总线流量，缺点是增加了复杂性，必须为每个高速缓存行增加一个额外的修改位，表明是否被修改过。写回处理不命中的方法是写分配，加载相应低一层中的块到高速缓存中，然后更新这个高速缓存块，利用了写的空间局部性，但会导致每次不命中都会有一个块从低一层传到高速缓存。

通过这样的Cache读写机制，实现了从CPU寄存器到L1高速缓存，再到L2高速缓存，再到L3高速缓存，再到物理内存的访问，有效的提高了CPU访问物理内存的速度。

## 7.6 hello进程fork时的内存映射

## shell通过fork为hello创建新进程。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hello进程唯一的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、区域结构和样表的原样副本。它将两个进程中的每个页面都标记为只读，并将每个进程中的每个区域结构都标记为写时复制。

## 当fork在新进程中返回时，新进程现在的虚拟内存刚好的和调用fork时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就是为每个进程保持了私有地址空间的概念。

## 7.7 hello进程execve时的内存映射

## execve函数在shell中加载并运行包含在可执行文件hello中的程序，用hello程序有效地替代了当前程序。加载hello的过程主要步骤如下：

## 首先删除已存在的用户区域，也就是将shell与hello都有的区域结构删除。然后映射私有区域，即为新程序的代码、数据、bss和栈区域创建新的区域结构，均为私有的、写时复制的。下一步是映射共享区域，将一些动态链接库映射到hello的虚拟地址空间，最后设置程序计数器，使之指向hello程序的代码入口。

## 经过这个内存映射的过程，在下一次调度hello进程时，就能够从hello的入口点开始执行了。

## 7.8 缺页故障与缺页中断处理

缺页中断就是要访问的页不在主存，需要操作系统将其调入主存后再进行访问。在这个时候，被内存映射的文件实际上成了一个分页交换文件。

页缺失（英语：Page fault，又名硬错误、硬中断、分页错误、寻页缺失、缺页中断、页故障等）指的是当软件试图访问已映射在虚拟地址空间中，但是目前并未被加载在物理内存中的一个分页时，由中央处理器的内存管理单元所发出的中断。

通常情况下，用于处理此中断的程序是操作系统的一部分。如果操作系统判断此次访问是有效的，那么操作系统会尝试将相关的分页从硬盘上的虚拟内存文件中调入内存。而如果访问是不被允许的，那么操作系统通常会结束相关的进程。

虽然其名为“页缺失”错误，但实际上这并不一定是一种错误。而且这一机制对于利用虚拟内存来增加程序可用内存空间的操作系统（比如Microsoft Windows和各种类Unix系统）中都是常见且有必要的。

微软在较新版Windows的资源监视器中使用“硬错误”（Windows Vista及以上）、“硬中断”（Windows 8及以上）这一术语来指代“页缺失”。

缺页中断发生时的事件顺序如下：

1) 硬件陷入内核，在内核堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在特殊的CPU寄存器中。

2) 启动一个汇编代码例程保存通用寄存器和其他易失的信息，以免被操作系统破坏。这个例程将操作系统作为一个函数来调用。

3) 当操作系统发现一个缺页中断时，尝试发现需要哪个虚拟页面。通常一个硬件寄存器包含了这一信息，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析这条指令，看看它在缺页中断时正在做什么。

4) 一旦知道了发生缺页中断的虚拟地址，操作系统检查这个地址是否有效，并检查存取与保护是否一致。如果不一致，向进程发出一个信号或杀掉该进程。如果地址有效且没有保护错误发生，系统则检查是否有空闲页框。如果没有空闲页框，执行页面置换算法寻找一个页面来淘汰。

5) 如果选择的页框“脏”了，安排该页写回磁盘，并发生一次上下文切换，挂起产生缺页中断的进程，让其他进程运行直至磁盘传输结束。无论如何，该页框被标记为忙，以免因为其他原因而被其他进程占用。

6) 一旦页框“干净”后（无论是立刻还是在写回磁盘后），操作系统查找所需页面在磁盘上的地址，通过磁盘操作将其装入。该页面被装入后，产生缺页中断的进程仍然被挂起，并且如果有其他可运行的用户进程，则选择另一个用户进程运行。

7) 当磁盘中断发生时，表明该页已经被装入，页表已经更新可以反映它的位置，页框也被标记为正常状态。

8) 恢复发生缺页中断指令以前的状态，程序计数器重新指向这条指令。

9) 调度引发缺页中断的进程，操作系统返回调用它的汇编语言例程。

10) 该例程恢复寄存器和其他状态信息

## 7.9动态存储分配管理

## 动态存储分配管理由动态内存分配器完成。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。堆是一个请求二进制零的区域，它紧接在未初始化的数据区后开始，并向上生长（向更高的地址）。分配器将堆视为一组不同大小的块的集合来维护。

## 每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可以用来分配。空闲块保持空闲，直到它显示地被应用程序所分配。

## 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。

## 动态内存分配器从堆中获得空间，将对应的块标记为已分配，回收时将堆标记为未分配。而分配和回收的过程中，往往涉及到分割、合并等操作。

## 动态内存分配器的目标是在对齐块的基础上，尽可能地提高吞吐率及空间占用率，即减少因为内存分配造成的碎片。其实现常见的数据结构有隐式空闲链表、显式空闲链表、分离空闲链表，常见的放置策略有首次适配、下一次适配和最佳适配。

## 7.10本章小结

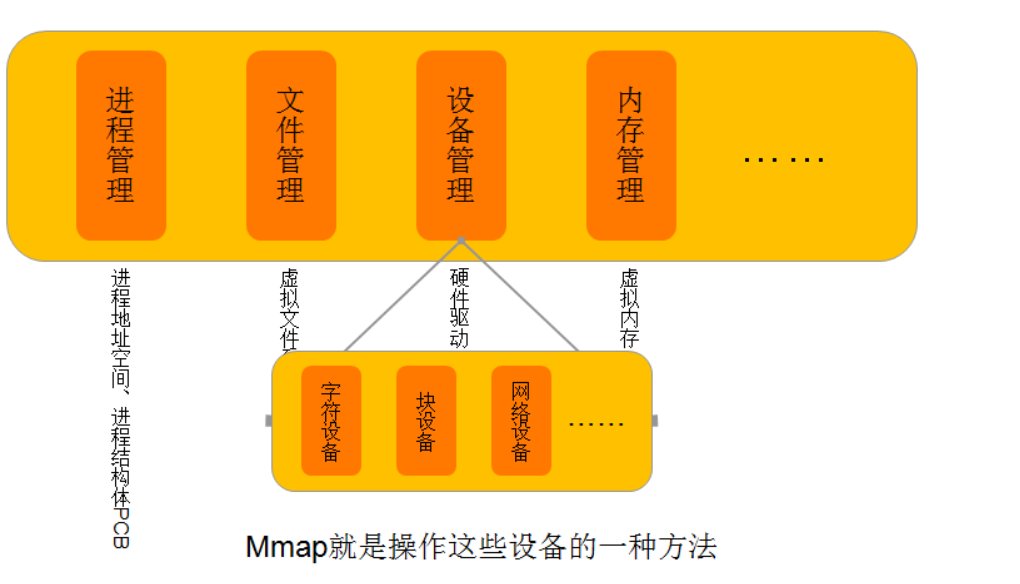
内存管理也起到了为hello提速的作用，当CPU执行机构收到应用程序发来的虚拟地址后，首先到TLB中查找相应的页表数据，如果TLB中正好存放着所需的页表，则称为TLB命中（TLB Hit）,接下来CPU再依次看TLB中页表所对应的物理内存地址中的数据是不是已经在一级、二级缓存里了，若没有则到内存中取相应地址所存放的数据。既然说TLB是内存里存放的页表的缓存，那么它里边存放的数据实际上和内存页表区的数据是一致的，在内存的页表区里，每一条记录虚拟页面和物理页框对应关系的记录称之为一个页表条目（Entry），同样地，在TLB里边也缓存了同样大小的页表条目（Entry）。

除了TLB以外，加速的方式还有很多，例如4级页表，3级cache等等。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法



上图说了，mmap是操作这些设备的一种方法，所谓操作设备，比如IO端口（点亮一个LED）、LCD控制器、磁盘控制器，实际上就是往设备的物理地址读写数据。

但是，由于应用程序不能直接操作设备硬件地址，所以操作系统提供了这样的一种机制—内存映射，把设备地址映射到进程虚拟地址，mmap就是实现内存映射的接口。

操作设备还有很多方法，如ioctl、ioremap

而mmap的好处是，mmap把设备内存映射到虚拟内存，则用户操作虚拟内存相当于直接操作设备了，省去了用户空间到内核空间的复制过程，相对IO操作来说，增加了数据的吞吐量。

在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。但在微观上：由于只有一个CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，让每个程序轮流执行。

时间片即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。而不会造成CPU资源浪费。

为了让我们的hello跑得更加迅速，计算机分配了指令流水线，从而提高处理器执行指令的效率，把一条指令的操作分成多个细小的步骤，每个步骤由专门的电路完成的方式。

## 8.2 简述Unix IO接口及其函数

首先是常用的几个函数  open , read,write,lseek, close

　　open函数  函数原型  int open(char \*  path,int  oflag,...)           返回值是一个文件描述符   path顾名思义就是文件名  oflage文件是打开方式   第三个形参应用于创建文件时使用  /\*创建文件其实还有一个create函数使用  以及openat由于还未使用过这个函数和open的差异 所以不在此处累赘\*/          open函数  使用if  判断的时候 注意小细节

　　read函数   函数原型   ssize\_t   read(int fd  ,   void\* buf , size\_t nbytes)    返回值是文件读取字节数  在好几种情况下会出现返回值不等于文件读取字节数 也就是第三个参数nbytes的情况  第二个形参buf读取到buf的内存  文件偏移量(current  file offset)受改变

　　write函数   函数原型  ssize\_t  write(int fd , const  void\* buf, size\_t nbytes)   返回值是文件写入字节数    fd是文件描述符  将buf内容写入nbytes个字节到文件  但这里需要注意默认情况是需要在系统队列中等待写入（打开方式不同也会不同）    //以上三个出错都返回-1

　　lseek函数off\_t lseek(int fd, off\_t offset , int whence)   返回值成功函数返回新的文件偏移量  失败-1   fd文件描述符   off\_t是有符号的整数  whence其实是和off\_t配套使用的  SEEK\_SET文件开始处   SEEK\_CUR当前值的相对位置  SEEK\_END文件长度+-

　　close函数原型  int close(int fd)    返回值 成功返回0  失败—1    关闭文件描述符

## 8.3 printf的实现分析

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

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

最终我们的hello程序将要显示在我们的屏幕上，这是通过cpu与我们计算机的外部设备（显示屏）进行IO交互，才最终将我们的hello程序展示在我们的眼前

那么IO是如何协调工作的那？

对于设备来说，其有两部分组成，一部分是机械部分，另一部分是电子控制部分，而电子控制部分是通过各种寄存器和cpu进行通信，被cpu所控制的，比如控制寄存器，数据寄存器，状态寄存器。他们分别和控制总线，数据总线，状态总线相连接。

操作系统对IO设备的管理主要分为三部分：逻辑IO，设备驱动程序，中断服务程序。

设备驱动程序：完成了对不同设备的各种各样的控制，对应用层提供接口。

中断服务程序：当设备结束的时候，向cpu发出中断信号。

设备按数据组织可以分为两类，以数据块作为传输存储的单位的块设备，以字符为单位存储传输信息的字符设备。

按照用户的请求，IO控制设备的各种操作。完成IO设备和内存的数据交换，最终完成IO请求。

最终，我们hello在bash中被回收，走完了它短暂但却精彩的一生。（没有变成孤儿进程或者僵尸进程真实太好啦）。

**（第8章1分）**

# 结论

Hello经过预处理-编译-链接生成可执行文件，通过shell fork子进程，excuve保护进程，通过cpu为其分配存储提高运行效率，通过IO管理并交互外部设备输出并打印到显示屏，最后由shell回收，结束hello的一生。

其实对于我来说，这几个月以来学习计算机系统的经历或许和hello的一生也是十分相似。

CSAPP的第一章是引子，通过任何编程书里都作为第一个程序的”hello world“来分析了整个软件硬件系统。高级语言通过编译链接后在计算机系统里是如何表示和执行的？这是开始，亦是终止。以前学过的课程和看过的书，要么是从讲某一种高级语言，要么是完全不涉及软件的计算机组成原理。至于怎么将两者联系起来，此前并没有接触过。

第二章回顾了整数浮点数的二进制表示，以前学过，但是再次复习巩固。这章感觉最深的是无符号和有符号数，在代码中尤其要注意两种数据计算时上溢下溢的问题。

第三章主要是讲汇编语言，对于我来说，这一章是一个巨大的挑战，第一次接触有关汇编语言的知识，摒弃了以外对于高级语言简单易懂的认知，第一次认识到了计算机的复杂性和如今语言发展历史的不容易。这章的独特之处是把高级语言写的程序和对应的汇编代码联系起来了，对照着理解，慢慢明白了机器是如何实现语言级别的设计。

第四章属于体系结构的知识，讲了CPU的指令集，流水线等等。甚至还重点描述了HCL做硬件设计，算是很深入的一个主题了。

第五章的名字特别吸引人---“优化程序性能”，在程序员的日常开发中，设计和优化部分可能占的比例最大。一个模块一旦设计好了之后，写代码是很快的。功能实现之后恐怕就一头栽进了性能优化方面的工作了。这章针对一个简单例子竟然从五六个方面进行了优化，尤其这里把指令流水线也考虑进去了。恐怕对我等普通程序员来说也是从来没有想到过的。

第六章讲存储器的层次结构，这部分是直接能感受到的。整天沉溺于分配内存释放内存之中的我们了解最多的是内存和硬盘，而这章中着重讲的是两级高速缓存的原理以及告诉缓存级别的代码优化。如果配以valgrind的machgrind来分析程序的话能加深对这部分的理解。

第七章讲链接，这也是hello编译过程中的最后一步，之后就进入了执行的过程。

接下来第八章讲异常控制流，不过这里的控制流不仅仅是程序中的exception。从底层硬件到高层语言，每一级别都有异常，如何处理这些异常，也是我们在日常编写程序需要时时注意的问题。

第九章分析了测量程序执行时间，这个分析程序瓶颈时经常需要做的事情。不过这章更深层的分析了如何能够得到准确的执行时间，尽管操作系统提供了获取时间的借口，但是我们并没有考虑过其准确度和精确度。

第十章是我最喜爱的一个章节，虚拟存储器，与程序运行直接相关的一个概念。这部分的学习对于每个程序员来说是必须的，否则的话总会迷茫于sizeof数组如何得到数组长度的疑问中。

后面三章讲了系统级I/O，网络编程和并发编程。这部分的内容与硬件关系少一些了，主要是进程和线程级的。内容虽然不多，但是每一部分都值得深入学习。

这本书能让我初步认知理解了整个计算机系统。在以后的编程学习中拓宽了我的思路，考虑的问题也会更全面。

Hello是开始，hello也是结束。

感谢与hello的相遇，带我走进了一个崭新的世界。感谢CSAPP这门课程，让我为今后的学习与工作，奠定了一份新的力量。

**（结论0分，缺少 -1分，根据内容酌情加分）**

# 附件

## hello.i（预处理源程序）

## hello.s（汇编程序）

## hello.o（可重定位目标程序）

hello.elf(hello的elf头信息)

asm.txt（hello反汇编）

hello.c（源文件）

hello（可执行文件）

CSAPP大作业学习总结.doc

**（附件0分，缺失 -1分）**

# 参考文献

**为完成本次大作业你翻阅的书籍与网站等**

[1] 《深入理解计算机系统》 作者：（美）布赖恩特（Bryant,R.E.）“Great souls often reside in ordinary bodies.”

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/ collection/anatmorp.

**（参考文献0分，确实 -1分）**