

**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 1170300225

班　　 级 11703002

学 生 于彤

指 导 教 师

**计算机科学与技术学院**

**2018年12月**

**摘 要**

本文详细描述了一个典型的Hello World C语言成语在计算机系统的完整运行流程。包括预处理、汇编、链接以及与hello.c相关的fork、缺页、资源回收等操作。一个hello.c程序从1KB人为编写的文件到汇编语言代码、机器语言、二进制代码然后进入内存、cache，被CPU加载并运行，知道结束，系统回收运行中所占用的系统资源。Hello.c完整的一生贯穿了计算机系统所有的关键技术和知识，让我们更加清楚的认识到每日触手可及的计算机到底是怎么改变我们的生活的。

**关键词：**预处理，汇编，链接，信号，进程，I/O操作，虚拟内存。

**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 - 7 -](#_Toc532238406)

[3.1 编译的概念与作用 - 7 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 7 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 7 -](#_Toc532238409)

[3.4 本章小结 - 10 -](#_Toc532238410)

[第4章 汇编 - 11 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 11 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 11 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 11 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 13 -](#_Toc532238415)

[4.5 本章小结 - 14 -](#_Toc532238416)

[第5章 链接 - 15 -](#_Toc532238417)

[5.1 链接的概念与作用 - 15 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 15 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 15 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 17 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 17 -](#_Toc532238422)

[5.6 hello的执行流程 - 18 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 19 -](#_Toc532238424)

[5.8 本章小结 - 20 -](#_Toc532238425)

[第6章 hello进程管理 - 21 -](#_Toc532238426)

[6.1 进程的概念与作用 - 21 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 21 -](#_Toc532238428)

[6.3 Hello的fork进程建过程 - 23 -](#_Toc532238429)

[6.4 Hello的execve过程 - 23 -](#_Toc532238430)

[6.5 Hello的进程执行 - 24 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 24 -](#_Toc532238432)

[6.7本章小结 - 24 -](#_Toc532238433)

[第7章 hello的存储管理 - 26 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 26 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 26 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 26 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 27 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 28 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 29 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 29 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 29 -](#_Toc532238442)

[7.9动态存储分配管理 - 30 -](#_Toc532238443)

[7.10本章小结 - 31 -](#_Toc532238444)

[第8章 hello的IO管理 - 32 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 32 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 32 -](#_Toc532238447)

[8.3 printf的实现分析 - 32 -](#_Toc532238448)

[8.4 getchar的实现分析 - 33 -](#_Toc532238449)

[8.5本章小结 - 33 -](#_Toc532238450)

[结论 - 34 -](#_Toc532238451)

[附件 - 35 -](#_Toc532238452)

[参考文献 - 37 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

Hello.c 经过预处理，编译，汇编，链接然后生成操作系统的可执行文件。操作系统会为hello.c的运行专门fork出一个子进程来调用系统函数加载hello的可执行文件，并利用mmap函数将在磁盘中的文件与虚拟内存建立映射。接下来，CPU从第一条命令开始逐一执行每条指令，直到程序中的每一条命令执行结束。子进程终止后，发送信号给父进程，让父进程回收它。至此，整个过程结束。

## 1.2 环境与工具

软硬件环境：X64 CPU;2G RAM;256GHD Disk;Windows 10 64位；Ubuntu18.04 64位；

开发和调试工具：gcc编译器，codeblocks，EDB，GDB，OBJDUMP等

## 1.3 中间结果

hello.i :hello.c预处理产生的文件，将hello.c中头文件替换为对应内容

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

hello.o: 汇编器将hello.s翻译成机器语言指令，它是一种可重定位目标文件。

hello：由hello.o和其他库链接成的可执行文件，会被加载到内存中执行。

anshelloo.txt：保存着hello.o反汇编生成的文件，用于结果分析。

readelfhelloo.txt：保存着hello.o文件的elf格式，通过readelf得到。

anshello.txt：保存着hello的反汇编生成的文件，用于结果分析。

readelfhello.txt:保存着hello文件的elf格式，通过命令readelf得到。

## 1.4 本章小结

本章简要介绍了hello’s p2p的整体过程，描述了hello.c在计算机系统的整体运行流程和使用到的操作系统资源，明确了hello.c的生命周期中需要生成的各种文件、占用的各种资源等等。在第二小节中还描述了在跟踪hello.c运行过程时使用的软硬件环境。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

C语言的预处理主要有三个方面的内容： 1.宏定义； 2.文件包含； 3.条件编译。 预处理命令以符号“#”开头。#号必须是该行除了任何空白字符外的第一个字符。#后是指令关键字，在关键字和#号之间允许存在任意个数的空白字符。整行语句构成了一条预处理指令，该指令将在编译器进行编译之前对源代码做某些转换。下面是部分预处理指令：

## 2.2在Ubuntu下预处理的命令

gcc –E hello.c –o hello.i

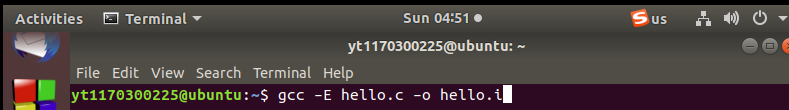


图 1Ubuntu下预处理的命令

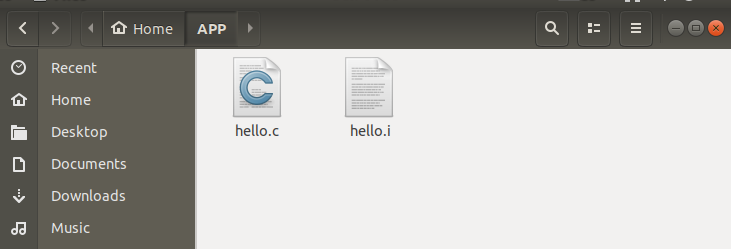


图 2Ubuntu下预处理产生的文件

## 2.3 Hello的预处理结果解析

对比hello.c和hello.i文件，发现计算机将.c程序中的头文件进行了解析，把以#开头的头文件的内容进行了替换，将头文件中定义的内容直接插入到文件中，其他内容保持不变，如此一来，在接下来的过程中，就知道各文件的位置，为接下来的过程奠定了基础。所以，预处理是整个生成可执行文件过程的基础。

## 2.4 本章小结

本章介绍了hello.c预处理的过程，即将程序中的头文件替换为相应的内容，并保留到文件扩展名为.i的文件中。除此之外，还介绍了linux系统下的预处理命令：gcc –E hello.c –o hello.i等知识。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

编译可以指利用编译程序从源语言编写的源程序产生目标程序的过程。 也可以指把高级语言变成计算机可以识别的2进制语言，计算机只认识1和0，编译程序把人们熟悉的语言换成2进制的。 编译程序把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析；语法分析；语义检查和中间代码生成；代码优化；目标代码生成。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误，给出提示信息。

汇编器将hello.i翻译成hello.s，将c语言程序转化为汇编语言。便于计算机系统的识别和运行以及优化。

## 3.2 在Ubuntu下编译的命令

gcc –S hello.i –o hello.s



图 3 Ubuntu下编译的命令

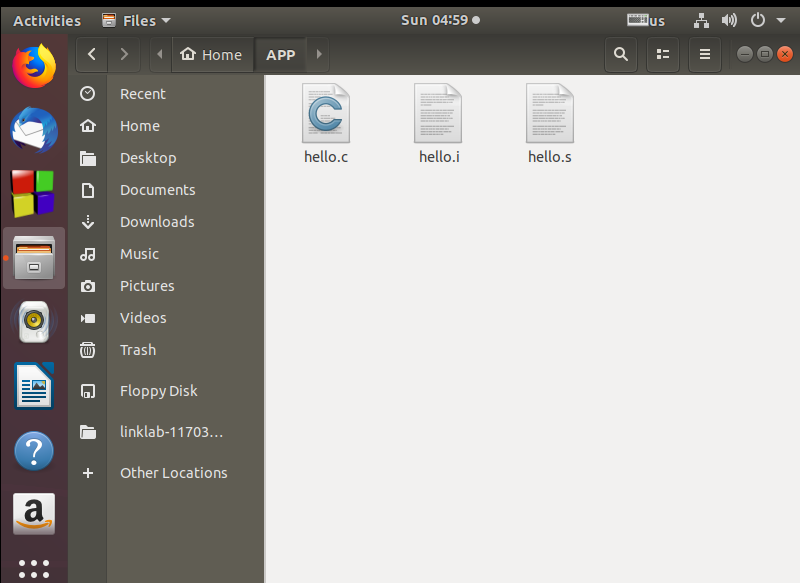


图 4 Ubuntu下编译产生的文件

## 3.3 Hello的编译结果解析

以下为汇编结果的内容：

*.file "hello.c"*

*.text*

*.globl sleepsecs*

*.data*

*.align 4*

*.type sleepsecs, @object*

*.size sleepsecs, 4*

*sleepsecs:*

*.long 2*

*.section .rodata*

*.LC0:*

*.string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"*

*.LC1:*

*.string "Hello %s %s\n"*

*.text*

*.globl main*

*.type main, @function*

*main:*

*.LFB5:*

*.cfi\_startproc*

*pushq %rbp*

*.cfi\_def\_cfa\_offset 16*

*.cfi\_offset 6, -16*

*movq %rsp, %rbp*

*.cfi\_def\_cfa\_register 6*

*subq $32, %rsp*

*movl %edi, -20(%rbp)*

*movq %rsi, -32(%rbp)*

*cmpl $3, -20(%rbp)*

*je .L2*

*leaq .LC0(%rip), %rdi*

*call puts@PLT*

*movl $1, %edi*

*call exit@PLT*

*.L2:*

*movl $0, -4(%rbp)*

*jmp .L3*

*.L4:*

*movq -32(%rbp), %rax*

*addq $16, %rax*

*movq (%rax), %rdx*

*movq -32(%rbp), %rax*

*addq $8, %rax*

*movq (%rax), %rax*

*movq %rax, %rsi*

*leaq .LC1(%rip), %rdi*

*movl $0, %eax*

*call printf@PLT*

*movl sleepsecs(%rip), %eax*

*movl %eax, %edi*

*call sleep@PLT*

*addl $1, -4(%rbp)*

*.L3:*

*cmpl $9, -4(%rbp)*

*jle .L4*

*call getchar@PLT*

*movl $0, %eax*

*leave*

*.cfi\_def\_cfa 7, 8*

*ret*

*.cfi\_endproc*

*.LFE5:*

*.size main, .-main*

*.ident "GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0"*

*.section .note.GNU-stack,"",@progbits*

3.3.1 sleepsecs：为全局整型变量，在hello.s中存放在.data节中，四字节对齐，分配的大小为四个字节，存放的数据由2.5变为2.

3.3.2 main：为函数名称，全局变量，存放在.text节中，存放着main函数首地址。

3.3.3 printf：函数名称，未定义符号，从动态库中引用

3.3.4 sleep：函数名称，未定义符号，从动态链接库中引用

3.3.5 exit：函数名称，未定义符号，从动态链接库中引用

3.3.6 getchar：函数名称，未定义符号，从动态链接库中引用

3.3.7 if：控制转移，例如程序中的if（argc!=3）,被解析为cmpl $3 , -20(%rbp)

和je .L2。所以当参数数目为三时跳转到.L2对应的语句，否则，继续向下执行。

3.3.8 =：赋值符号，程序中的i=i+1,转化为addl $1 -4(%rbp),，这里i存放在“-4（%rbp）所对应的地址中。

3.3.9 !=：关系操作，程序中的if(argc!=3)转化为je .L2,判断是否相等，如果相等，则跳转，否则继续执行。

3.3.10 <：关系操作，程序中for(i=0;i<10;i++)中条件判断的部分被转化为cmpl $9 -4(%rbp) 和 jle .L4，当i不小于等于9时退出循环，和原程序相符

3.3.11 arg[]：数组操作，确定数组首地址，如果想寻找数组中某个元素，只要利用它的索引乘以一个常数，再加上首地址就可以找到该元素了。

3.3.12 函数调用：调用printf或puts等函数时，直接用call 函数名这种语句实现。

3.3.13 返回值：将函数的返回值存放在%rax中，当时用该返回值时，直接从rax寄存器中取出即可

3.3.14 参数传递：将程序的第一个参数存储到%rdi中，第二个参数存放在%rsi中，依次存放，这些操作均在调用函数之前完成。

3.3.15 循环操作：此处是解释为先在.L2中赋初始状态，在.L3中判断循环是否继续进行，在.L4中执行循环内容。执行过程为，先进入.L2，然后跳转到.L3，然后跳转到.L4，在.L3和.L4中来回跳转直到循环结束。

3.3.16 只读数据：在使用这部分数据时，跳转到相应地址，取出相应数据，如程序中printf函数打印的字符串。

## 3.4 本章小结

本章介绍了计算机系统对hello.c的编译的过程，计算机系统通过编译器从hello.i 文件生成hello.s文件的过程。并介绍了ubuntu提供的相关命令。然后详细分析了hello.s文件的内容，明确了C语言经过编译生成的汇编文件的内容。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

把汇编语言翻译成机器语言的过程称为汇编。在汇编语言中，用助记符(Memoni)代替操作码，用地址符号(Symbol)或标号(Label)代替地址码。这样用符号代替机器语言的二进制码，就把机器语言变成了汇编语言。于是汇编语言亦称为符号语言。

在hello.c中，汇编器将hello.s翻译成机器语言指令，并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。Hello.o是一个二进制文件，不能通过文本编辑器直接查看，但可以通过反汇编查看其对应的汇编语言。程序通过汇编将汇编语言转化为机器语言，转化成为了计算机可以直接读懂的语言，从而使程序的执行更进一步。

## 4.2 在Ubuntu下汇编的命令

命令：gcc –c hello.s –o hello.o

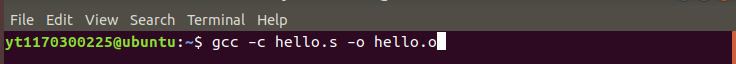


图 5 hello.s汇编的过程

## 4.3 可重定位目标elf格式

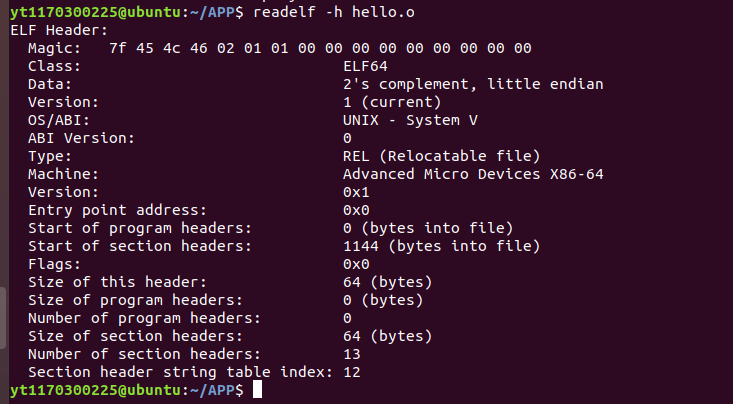


图 6 elf 头

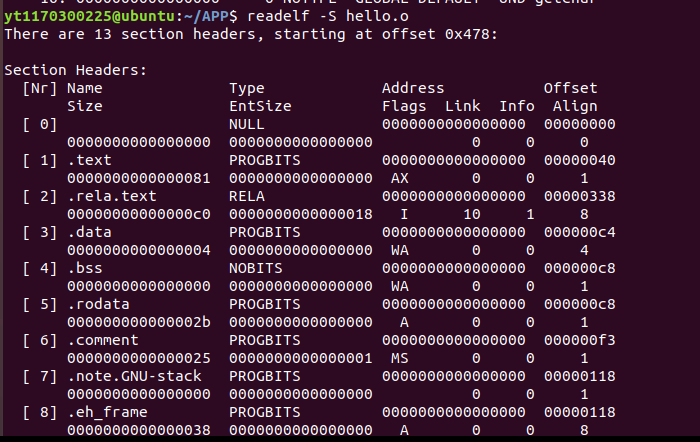


图 7 节头信息

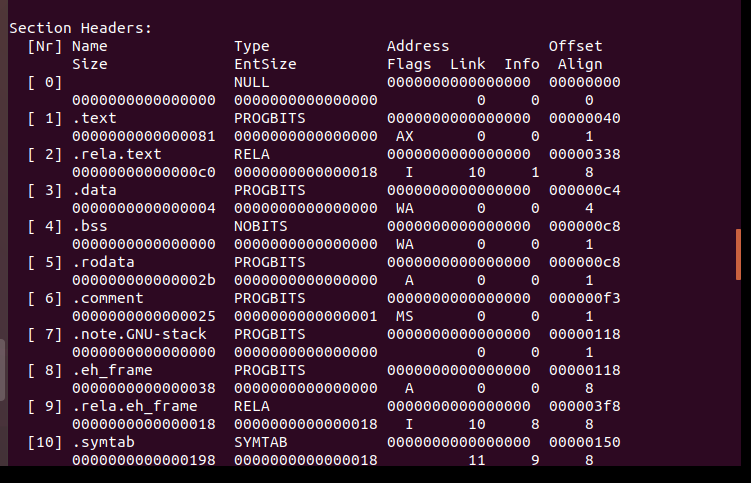


图 8 节头信息

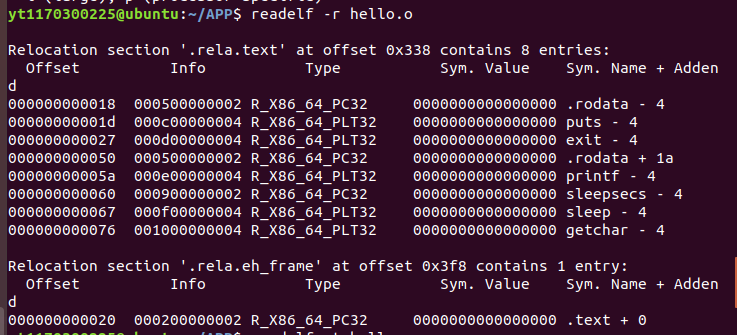


图 9 重定义节的信息

重定位目标elf格式

ELF文件由4部分组成，分别是ELF头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。实际上，一个文件中不一定包含全部内容，而且他们的位置也未必如同所示这样安排，只有ELF头的位置是固定的，其余各部分的位置、大小等信息由ELF头中的各项值来决定。其中ELF头包括了ELF头的大小，目标文件的类型，机器类型，及头部表的文件偏移，节头部表中条目的大小和数量，还有.text节存放着代码，.rodata节存放着只读数据，比如printf要打印出的语句，.data节存放全局和静态C变量，这里存放sleepsecs，.symtab节，存放着符号的内容，有17个条目，还有一些重定位节对于一些符号的重定义。重定位条目存放在.rela.text节中，里面有八个重定位条目，分别为.rodata,puts,printf,exit,.rodata,sleepsecs,sleep,getchar,值得注意的是，有两个相同的.rodata符号，但它们对应的内容却不同，即对应两个只读数据。这几个重定位条目中，在hello.c中定义的函数重定位类型为R\_X86\_64\_PC32,即使用32位PC相对地址的引用。对于hello.c中定义的全局变量，如sleepsecs，重定位类型位R\_X86\_64\_32,即使用32位绝对地址的引用。对于其他函数，根据共享库的知识，是引用在动态库中的内容及地址，如sleep，getchar等函数。重定位条目还记录了偏移量，以及引用所指向的符号等，用于以后的链接及重定位。

## 4.4 Hello.o的结果解析

每条机器语言由指令操作符，还可能有寄存器指示符，还可能有一个八字节的常数字组成。对于汇编语言中的每一条指令，都被翻译成一条机器指令。机器语言通过指令集于汇编语言建立起映射。例如main函数中的push %rbp这条指令被翻译为机器语言时为55，这个数为十六进制数，代表操作为将一个数压进栈，所取的数为寄存器%rbp中的数。当然，hello.o的反汇编中有些地方和hello.s中内容有所不同。比如操作数在hello.s中很多是十进制的，但在hello.o中是十六进制的；还有，在hello.s中作为跳转符号的的.L2 , .L3 .LCO等在hello.o中不见了，并被零所取代，并且在后面注释了其对应内容和在下方标上了重定位信息，用于以后的重定位。 对于分支和跳转部分，后面跳转的不再是.L2之类的符号，而是对应的真真实实的地址，和相对于main函数的偏移，利于程序运行和阅读。对于函数的调用，在hello.s中只是函数的名字而没有其他信息，在hello.o中，函数的地址暂时是指令地址相对于main函数的偏移地址，并在下方标注重定位信息，使其于定义的符号关联起来。

## 4.5 本章小结

本章主要介绍了汇编的概念及作用，并给出了对程序汇编的命令“gcc –c hello.s –hello.o”。其次，通过readelf命令观察hello.o的elf文件格式，对其中的各个节进行解析，着重解析了其中包含可重定位条目的.rel.text节，对这些可重定位条目进行了解析。除此之外，还将hello.o反汇编生成的内容和hello.s中的内容进行对比，总结了它们数据表示，分支转移，函数调用等方面的不同，加深了对汇编过程的理解。

**（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为一个单一文件的过程，这个文件可被复制到内存中并执行。链接可以执行与编译时，也就是在源代码被翻译成机器代码时；也可以执行与加载时，也就是在程序被加载器加载到内存并执行时；甚至执行与进行时，也就是由应用程序来执行。在现代系统中，链接是由链接器自动执行的。

链接在软件开发中扮演着一个关键的角色，因为他们是的分离编译成为可能。我们不用将一个大型的应用程序组织成为一个巨大的源文件，而是可以把它分成更小，更好管理的模块，可以独立的修改和编译这些模块。当我们改变这些模块中的一个时，只需简单的重新编译它，并重新链接应用，而不必重新编译其他文件。对于hello的链接过程，是将hello.o和其他所需要的库连接到一块生成可执行文件。

## 5.2 在Ubuntu下链接的命令

使用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命令进行动态链接。

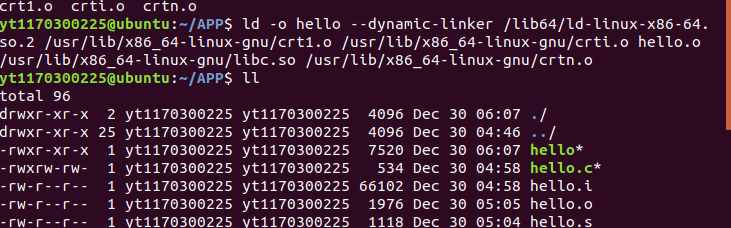


图 10 Ubuntu下链接的命令

## 5.3 可执行目标文件hello的格式

分析hello的ELF格式，用readelf等列出其各段的基本信息，包括各段的起始地址，大小等信息。

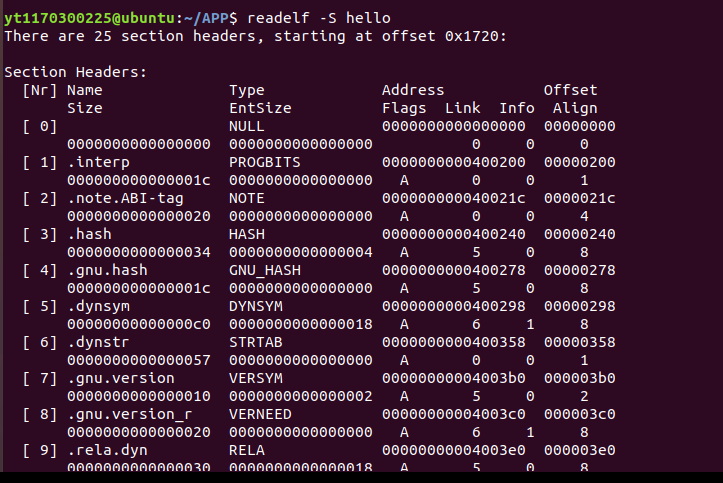


图 12 hello节头信息

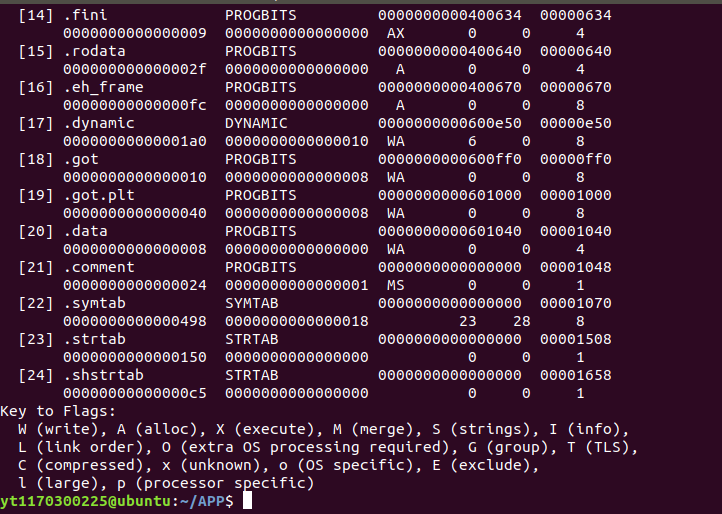


图 13 hello节头信息

可执行目标文件主要由ELF头,段头部表，.init，.text，.rodata，.data，.bss，.symtab等组成，其中ELF头描述文件的总体格式，它还包括程序的入口点，也就是当程序运行时要执行的第一条指令的地址。.text,.data，.rodata节和可重定位文件是类似的，除了这些节已经重定位到它们最终运行时内存地址以外，.init节定义了一个小函数，叫做.init，程序初始化代码会调用它。因为可执行文件是完全链接的，所以它不再需要.rel节。可执行文件被设计得很容易加载到内存，可执行文件的连续的片被映射到连续的内存段。程序头部表描述了这种映射关系。

## 5.4 hello的虚拟地址空间

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。

通过edb加载hello进程，发现里面只能观察到.init，.plt，.text，.data，.fini这几个节，而不能观察到其他几个节。通过与5.3中各节对照发现，各节的首地址与elf文件中每个节的首地址相同，并且每个节的字节数和elf文件中所记录的字节数确实相同。但每个节所紧挨着的节的地址并不都是连续的，再次观察elf文件中节头部表的信息，发现还有字对齐，有两个字对齐，也有四字对齐。将每个节根据这个规则对齐后，发现每个节是基本连续的。除此之外，虚拟地址空间代码段的部分都是从0x400000开始的，而数据段都是从0x600000开始的。还有，虚拟地址空间大部分是未分配的，只有一小部分被利用，这是为了让程序更好的加载到内存。

## 5.5 链接的重定位过程分析

链接首先是对hello.o中引用的符号进行解析。对于在hello.c中定义的符号，例如main函数中引用的sleepsecs，编译器会将它与符号表中已定义的符号关联起来，生成一个可重定位条目。对于hello.c中未定义的符号，例如puts, printf, exit, main, sleep等，编译器会假定它是在其他链接的模块中定义的，并将这个交给链接器处理，然后链接器会将它与一个定义关联起来。接下来开始进行重定位节和符号定义。相同类型的节合并为同一类型的新的聚合节。然后，链接器将运行时地址赋给新的聚合节，付给输入模块定义的每个节，以及赋给输入模块定义的每个符号。此时，程序中的每条指令和全局变量都有唯一的运行时地址了。

接下来，对hello.o中引用的符号进行重定位，这一步依靠重定位条目。我们依次分析下程序中需要重定位的地方。首先，是对于printf函数中字符串的重定位。它们在hello.o中如下所示：

15: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 1c <main+0x1c>

18: R\_X86\_64\_PC32 .rodata-0x4

4d: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 54 <main+0x54>

50: R\_X86\_64\_PC32 .rodata+0x1a

它们使用的是PC 32位相对地址。这两个分析的过程是完全相同的，所以我们只需分析一个即可，那我们来看第一个。对于这个符号，它的重定位条目是r.offset = 0x15, r.symbol = .rodata, r.type = R\_X86\_64\_PC32,r.addend = -0x4.这本该是PC相对地址引用，但观察hello的反汇编，发现这虽然是PC32位地址的相对引用，但最后修改的时候却是32位地址的绝对引用。所以，我们根据32位地址绝对引用来讲解.ADDR(r.symbol)= ADDR(.rodata), \*reptr = ADDR(r.symbol) + r.addend = 0x400714,修改对应代码段如下所示：

40061c: bf 14 07 40 00 mov $0x400714,%edi

我们再看另一种重定位，对于puts, sleep, exit, printf, 这一类函数的重定位，是引用动态库中的函数，但是它们几个远离是完全相同的，所以我们只需要分析一个即可。那我们就以puts函数为例。

1c: e8 00 00 00 00 callq 21 <main+0x21>

1d: R\_X86\_64\_PLT32 puts-0x4

这类重定位与其他不同，它是在运行过程中进行重定位的。当程序第一次执行到callq 4004d0<pus@plt>这行语句时，跳转到对应的语句如下：

00000000004004d0 <puts@plt>:

4004d0: ff 25 42 0b 20 00 jmpq \*0x200b42(%rip) # 601018 <puts@GLIBC\_2.2.5>

4004d6: 68 00 00 00 00 pushq $0x0

4004db: e9 e0 ff ff ff jmpq 4004c0 <.plt>

程序直接执行这个模块，执行pushq $0x0语句，将puts的ID压入栈中，然后跳转到.plt，然后将动态链接器的一个参数压入栈中，再跳转到动态链接器。动态链接器根据两个参数来确定puts的运行时地址，然后全局变量偏移表中puts的部分。再把控制传递给puts函数，重启调用puts的控制流。不过这次跳转是直接跳转到真正的puts函数，而不是pushq $0x0这行命令了。

除了上面两种重定位以外，还有另一种重定位在hello.o中出现，那就是对全局变量sleepsecs的重定位，语句如下：

5e: 8b 05 00 00 00 00 mov 0x0(%rip),%eax # 64 <main+0x64>

60: R\_X86\_64\_PC32 sleepsecs-0x4

这是重定位绝对引用，其中r.offset = 0x60 ,r.symbol = sleepsecs ,r.type = R\_X86\_64\_PC32,r.addend = -0x4，接下来开始进行重定位，ADDR(s)=ADDR(.text); ADDR(r.symbol)=ADDR(sleepsecs); refaddr=ADDR(s) + r.offset;\*reptr = (unsigned)(ADDR(r.symbol) + r.addend – refaddr),最后计算出它的运行时地址并修改代码段。为下图所示：

400661: 8b 05 e9 09 20 00 mov 0x2009e9(%rip),%eax

至此，重定位和链接基本全部完成。

## 5.6 hello的执行流程

以下为hello的执行流程：第一项代表程序名称，第二项代表程序地址。

表格 1 hello 的执行流程

|  |  |
| --- | --- |
| ld-2.27.so!\_start | 0x7fd7:8dfda090 |
| ld-2.27.so!\_dl\_start | 0x7fd7:8dfdaea0 |
| ld-2.27.so!\_dl\_start\_user | 0x7fd7:8dfda09b |
| ld-2.27.so!\_dl\_init | 0x7fd7:8dfe9630 |
| hello!\_start | 0x400500 |
| libc-2.27.so!\_\_libc\_start\_main | 0x7fd7:8dc09ab0 |
| -libc-2.27.so!\_\_cxa\_atexit | 0x7fd7:8dc2b430 |
| -libc-2.27.so!\_\_libc\_csu\_init | 0x4005c0 |
| hello!\_init | 0x400488 |
| libc-2.27.so!\_setjmp | 0x7fd7:8dc26c10 |
| -libc-2.27.so!\_sigsetjmp | 0x7fd7:8dc28e2b |
| –libc-2.27.so!\_\_sigjmp\_save | 0x7fd7:8dc2db30 |
| hello!main | 0x400532 |
| hello!puts@plt | 0x4004b0 |
| hello!exit@plt | 0x4004e0 |
| \*hello!printf@plt | – |
| \*hello!sleep@plt | – |
| \*hello!getchar@plt | – |
| ld-2.27.so!\_dl\_runtime\_resolve\_xsave | 0x7fd7:8dc324d0 |
| -ld-2.27.so!\_dl\_fixup | 0x7fd7:8dc34520 |
| –ld-2.27.so!\_dl\_lookup\_symbol\_x | 0x7fd7:8dd24dc0 |
| libc-2.27.so!exit | 0x7fd7:8dd32fd0 |
|  |  |

## 5.7 Hello的动态链接分析

在调用共享库函数时，编译器没有办法预测这个函数的运行时地址，因为定义它的共享模块在运行时可以加载到任意位置。正常的方法是为该引用生成一条重定位记录，然后动态链接器在程序加载的时候再解析它。GNU编译系统使用延迟绑定(lazybinding),将过程地址的绑定推迟到第一次调用该过程时。

延迟绑定是通过GOT和PLT实现的。GOT是数据段的一部分，而PLT是代码段的一部分。两表内容分别为：

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。

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[O]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。

可以看出在调用dl\_init前后全局偏倚表的变化如下

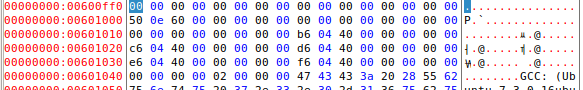


图 14 调用dl\_init之前

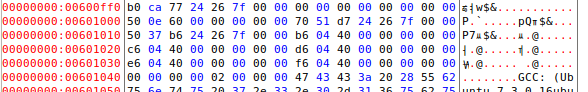


图 15 调用dl\_init之后

## 5.8 本章小结

本章主要介绍了计算机从编译到执行的过程中，有关链接部分的内容。从hello的链接和可重定位文件的分析来介绍了虚拟地址空间的知识。最后描述了hello的链接过程和重定位过程的详细内容，并使用edb对这些过程进行了分析。

**（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。进程的作用：

1. 一个独立的逻辑控制流，它提供一个假象，好像我们的程序独立地使用处理器。
2. 一个私有的地址空间，它提供一个假象，好像我们的程序独立地使用内存系统。

## 6.2 简述壳Shell-bash的作用与处理流程

Shell是一个命令行解释器，为用户提供了一个向Linux内核发送请求以便运行程序的界面系统级程序，用户可以用shell来启动、挂起、停止甚至是编写一些程序。

处理流程如下图所示：

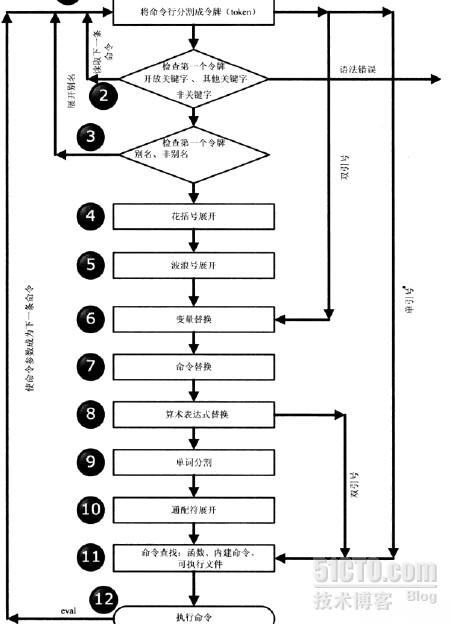


图 16 shell 的处理流程

1．Shell首先从命令行中找出特殊字符（元字符），在将元字符翻译成间隔符号。元字符将命令行划分成小块tokens。Shell中的元字符如下所示：

SPACE , TAB , NEWLINE , & , ; , ( , ) ,< , > , |

2． 程序块tokens被处理，检查看他们是否是shell中所引用到的关键字。

3． 当程序块tokens被确定以后，shell根据aliases文件中的列表来检查命令的第一个单词。如果这个单词出现在aliases表中，执行替换操作并且处理过程回到第一步重新分割程序块tokens。

4．Shell对~符号进行替换。

5．Shell对所有前面带有$符号的变量进行替换。

6．Shell将命令行中的内嵌命令表达式替换成命令；他们一般都采用$(command)标记法。

7．Shell计算采用$(expression)标记的算术表达式。

8．Shell将命令字符串重新划分为新的块tokens。这次划分的依据是栏位分割符号，称为IFS。缺省的IFS变量包含有：SPACE , TAB 和换行符号。

9．Shell执行通配符\* ? [ ]的替换。

10．shell把所有從處理的結果中用到的注释删除，並且按照下面的顺序实行命令的检查：

A. 内建的命令

B. shell函数（由用户自己定义的）

C. 可执行的脚本文件（需要寻找文件和PATH路径）

11．在执行前的最后一步是初始化所有的输入输出重定向。

12．最后，执行命令。

## 6.3 Hello的fork进程创建过程

shell通过调用fork函数来创建一个子进程。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但是独立的一个副本，包括代码和数据段，对，共享库以及用户栈。子进程获得与父进程任何 打开文件描述符相同的副本，这就意味着当父进程调用fork是，子进程可以读写父进程中打开的任何文件。父进程和子进程最大的区别是他们有不同的PID。Fork函数被调用一次，却会返回两次；一次返回到父进程中，返回的是子进程的PID，一次返回到子进程中，返回值为零。并且子进程和父进程是并发执行的。

## 6.4 Hello的execve过程

execve函数加载并运行可执行目标文件hello，且开始设置参数列表argv和环境变量列表envp。并且execve调用了之后不再返回。Argv变量指向一个以null结尾的指针数组，其中每个指针都指向一个参数字符串。Argv[0]中是hello。环境变量的列表是由一个类似的数据结构组成的。在execve加载了hello之后，它调用启动代码。启动代码libc\_start\_main设置栈，并将控制传递给新程序的主函数，hello开始执行。

## 6.5 Hello的进程执行

在程序头部表的引导下，加载器将hello的片复制到代码段和数据段。接下来，加载器跳转到程序的入口点，也就是\_start函数的地址。\_start函数调用系统启动函数\_\_libc\_start\_main，它初始化环境，调用用户层的main函数。如果参数不是三个，那么系统调用printf函数，将控制返回给内核，打印出“Usage: Hello 学号 姓名”，再将控制从内核转移到用户，继续系统调用，控制返回给内核，内核发送信号给该进程，此进程结束，被父进程回收。如果参数是三个，那么进入循环，系统调用函数printf，将控制返回内核，打印出“Hello 1170300521 张亚博”，控制返回给用户，用户进程执行睡眠2.5s，依次循环十次，最后跳出循环，系统调用getchar函数，等待用户输入字符，输入回车，main函数执行结束。进程结束，被父进程回收。

## 6.6 hello的异常与信号处理

hello执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

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

回车：系统中断，shell中打印换行，并将控制返回给进程，执行进程下一条命令。

Ctrl-Z：系统中断，进入内核态判断内容，内核发送SIGSTOP信号给进程，进程停止。

Ctrl-C：系统中断，进入内核态判断内容，内核发送信号SIGINT给进程，进程终止。

ps：打印出现在运行的进程的pid及进程名

jobs：打印出shell现在的所有作业

pstree：以图形化的形式打印出所有进程

fg：发送信号给hello进程，hello进程继续执行

kill：shell发送信号给其他进程，这里发送终止信号给hello进程，hello进程接收信号并终止

## 6.7本章小结

本章主要介绍了进程的概念与作用，并且以shell和hello进程为例详细介绍了子进程的创建，hello程序的加载过程，hello程序的执行过程，异常和信号的处理等。将hello从出生到执行这两个过程联系起来，加深了对程序加载和执行过程的认识。

**（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

结合hello说明逻辑地址、线性地址、虚拟地址、物理地址的概念。

逻辑地址：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：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。

虚拟地址：虚拟地址是Window程序运行在386保护模式下，这样程序访问存储器所使用的逻辑地址称为虚拟地址，与实地址模式下的分段地址类似，虚拟地址也可以写为“段：偏移量”的形式，这里的段是指段选择器。CPU启动保护模式后，程序运行在虚拟地址空间中。注意，并不是所有的程序都是运行在虚拟地址中。

物理地址：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。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

## 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。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。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

## 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(pageframe)。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。

## 这种管理方式的优 点是，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，一个程序不必连续存放。这样就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)。缺点是仍旧要求程序全部装入内存，没有足够的内存，程序就不能执行。

在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中

实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，—个页式系统中，一般要采用如下的数据结构。

进程页表：完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间)的映射。

每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序。

物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况，其数据结构可采用位示图和空闲页链表。

请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换也可以结合到各进程的PCB(进程控制块)里。

## 7.4 TLB与四级页表支持下的VA到PA的变换

VA：virtual address称为虚拟地址，PA：physical address称为物理地址。CPU通过地址来访问内存中的单元，如果CPU没有MMU，或者有MMU但没有启动，那么CPU内核在取指令或者访问内存时发出的地址(此时必须是物理地址，假如是虚拟地址，那么当前的动作无效)将直接传到CPU芯片的外部地址引脚上，直接被内存芯片(物理内存)接收，这时候的地址就是物理地址。如果CPU启用了MMU(一般是在bootloader中的eboot阶段的进入main()函数的时候启用)，CPU内核发出的地址将被MMU截获，这时候从CPU到MMU的地址称为虚拟地址，而MMU将这个VA翻译成为PA发到CPU芯片的外部地址引脚上，也就是将VA映射到PA中。MMU将VA映射到PA是以页(page)为单位的，对于32位的CPU，通常一页为4k，物理内存中的一个物理页面称页为一个页框(page frame)。虚拟地址空间划分成称为页（page）的单位,而相应的物理地址空间也被进行划分，单位是页框(frame).页和页框的大小必须相同。

首先将CPU内核发送过来的32位VA[31:0]分成三段，前两段VA[31:20]和VA[19:12]作为两次查表的索引，第三段VA[11:0]作为页内的偏移，查表的步骤如下：

1. 从协处理器CP15的寄存器2(TTB寄存器，translation table base register)中取出保存在其中的第一级页表(translation table)的基地址，这个基地址指的是PA，也就是说页表是直接按照这个地址保存在物理内存中的。
2. 以TTB中的内容为基地址，以VA[31:20]为索引值在一级页表中查找出一项(2^12=4096项)，这个页表项(也称为一个描述符，descriptor)保存着第二级页表(coarse page table)的基地址，这同样是物理地址，也就是说第二级页表也是直接按这个地址存储在物理内存中的。
3. 以VA[19:12]为索引值在第二级页表中查出一项(2^8=256)，这个表项中就保存着物理页面的基地址，我们知道虚拟内存管理是以页为单位的，一个虚拟内存的页映射到一个物理内存的页框，从这里就可以得到印证，因为查表是以页为单位来查的。
4. 有了物理页面的基地址之后，加上VA[11:0]这个偏移量(2^12=4KB)就可以

## 7.5 三级Cache支持下的物理内存访问

根据MMU提供的物理地址，先在L1缓存中寻找数据。物理地址一共有52位，将前四十位作为CT标记位，接下来六位当作CI组索引，最后六位当作块中偏移量。先根据CI找到对应的组，然后再该组中寻找是否有对应的CT标记并且有效位是否显示有效。如果存在对应CT，那么缓存命中，取出相应数据返回给CPU。否则缓存不命中，如果该组中有空的块，那么是冷不命中，否则为冲突不命中。如果L1缓存不命中，那么接下来去L2缓存中继续寻找对应数据。寻找方式和再L1中寻找方式相同。如果缓存命中，那么将缓存块中的内容返回给CPU以及更新L1缓存。在更新L1缓存时，对应组如果有未分配的块，那么将内容加载到该块中；否则，根据LRU原则，找到一个牺牲快，用该块内容替换它。如果L2缓存不命中，那么与前面相同，去L3缓存中根据相同的方法寻找对应数据，如果缓存命中，那么直接将数据返回给CPU，并更新L2，L1缓存（更新规则与前面相同）。如果缓存不命中，那么根据物理地址在主存中寻找对应数据，找到数据后，将数据返回给CPU，以及更新各级缓存。

## 7.6 hello进程fork时的内存映射

父子进程的代码段是相同的，所以代码段是没必要复制的，因此内核将代码段标记为只读，这样父子进程就可以安全的共享此代码段了。fork之后在进程创建代码段时，新子进程的进程级页表项都指向和父进程相同的物理页帧。

对于父进程的数据段，堆段，栈段中的各页，由于父子进程要相互独立，所以我们采用写实复制的技术，来最大化的提高内存以及内核的利用率。刚开始，内核做了一些设置，令这些段的页表项指向父进程相同的物理内存页。调用fork之后，内核会捕获所有父进程或子进程针对这些页面的修改企图(说明此时还没修改)并为将要修改的页面创建拷贝。系统将新的页面拷贝分配给被内核捕获的进程，还会对子进程的相应页表项做适当的调整，现在父子进程就可以分别修改各自的上述段，不再互相影响了。

## 7.7 hello进程execve时的内存映射

首先，删除已存在的用户区域。为新程序的代码，数据，和数据区域创建新的区域结构。然后，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。接下来，映射共享区域。如果hello程序和共享对象链接，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。最后，设置程序计数器。ｅｘｅｃｖｅ做的最后一件事就是设置当前进程上下文中的程序计数器，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

## 当进程执行过程中发生缺页中断时，需要进行页面换入，步骤如下:

## 首先硬件会陷入内核，在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CPU中特殊的寄存器中。

## 启动一个汇编代码例程保存通用寄存器及其它易失性信息，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。

## 当操作系统发现是一个页面中断时，查找出来发生页面中断的虚拟页面（进程地址空间中的页面）。这个虚拟页面的信息通常会保存在一个硬件寄存器中，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，用软件分析该指令，通过分析找出发生页面中断的虚拟页面。

## 检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程。

## 操作系统查找一个空闲的页框(物理内存中的页面)，如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框。

## 如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上，此时会引起一个写磁盘调用，发生上下文切换（在等待磁盘写的过程中让其它进程运行）。

## 页框干净后，操作系统根据虚拟地址对应磁盘上的位置，将保持在磁盘上的页面内容复制到“干净”的页框中，此时会引起一个读磁盘调用，发生上下文切换。

## 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态。

## 恢复缺页中断发生前的状态，将程序指令器重新指向引起缺页中断的指令。

## 调度引起页面中断的进程，操作系统返回汇编代码例程。

## 汇编代码例程恢复现场，将之前保存在通用寄存器中的信息恢复。

## 7.9动态存储分配管理

动态内存分配由动态内存分配器来实现，十分方便，而且有很好的移植性。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但不失通用性，假设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域后开始，并向上生长。对于每个进程，内核维护着一个变量ｂｒｋ，它指向堆的顶部。

分配器将堆视为一组不同大小块的集合来维护。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。已分配的块显式的保留为供应应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式的被应用所分配。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己隐式执行的。分配器有两种风格，分别是显式分配器和隐式分配器。前者要求应用显式的释放任何已分配的块，后者要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。

对于显式分配器，它有多种实现模式分别为隐式空闲链表和显式空闲链表以及分离的空闲链表。寻找空闲块有首次适配，最佳适配等。分配块被释放后有可能进行空闲块的合并，对于隐式空闲链表，分为四种情况：

1. 前面的块是空闲的，后面的块也是空闲的。将前一个块的头部保存的数目改为三个块的总和，后一个块的脚部保存的数目改为三个块数的总和。

1. 前面的块是空闲的，后面的块是已分配的。将前一个块的头部保存的数目改为两个块的和。
2. 前面的块是已分配的，后面的块是空闲的。将被释放的块的头部改为两个块数目的总和，后一个块的脚部也改为两个块的总和。
3. 前面的块是已分配的，后面的块是已分配的。保持不变。

对于显式空闲分配链表来说，有不同的排序策略，比如后进先出的顺序维护链表，还有按照地址的顺序来维护链表。对于前者，如果使用了边界标记，合并可以在常数时间内完成，而后者需要线性时间。但后者的空间利用率较高。

## 7.10本章小结

本章详细介绍了hello在执行过程中，操作系统关于内存方面的操作，如逻辑地址，线性地址，虚拟地址，物理地址相关概念；接下来介绍了Intel从逻辑地址到线性地址的变换-段式管理，然后是hello从线性地址到物理地址的转变-页式管理、TLB与四级页表支持下的VA到PA的变换、三级Cache支持下的物理内存访问、hello进程fork时的内存映射、hello进程execve时的内存映射、缺页故障与缺页中断处理以及动态存储分配管理等内容。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

一个Linux文件就是一个m个字节的序列，所有的I/O设备都被模式化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。这种将设备优雅的映射为文件的方式，允许Linux内核引出一个简单的低级的应用接口，称为UnixI/O，这使得所有的输入输出设备都能以一种统一且一致的方式来进行。

## 8.2 简述Unix IO接口及其函数

linux 提供如下 IO 接口函数：

read 和 write – 最简单的读写函数；

readn 和 writen – 原子性读写操作；

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

recv 和 send – 允许从进程到内核传递标志；

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

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

## 8.3 printf的实现分析

我们首先查看printf函数的定义：

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;

}

va\_list是个字符指针，（char\*）(&fmt)+4)表示的是第一个参数的地址。接下来执行vsprintf函数，它的作用就是将字符串格式化，从而产生格式化输出，并返回字符串的长度给i；然后通过陷阱系统调用write（0x80），文件描述符默认指向显示屏，将i个字符写到终端。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）。连续执行这个过程，最终将所有的内容都打印到显示器上。

## 8.4 getchar的实现分析

getchar函数的定义如下：

int getchar(void)

{

static char buf[BUFSIZ];

static char \*bb=buf;

static int n=0;

if(n==0)

{

n=read(0,buf,BUFSIZ);

bb =buf;

}

return (--n>=0)?(unsigned char)\*bb++:EOF;

}

## 调用getchar函数读取输入时，只有当输入字符为换行符'/n'或文件结束符EOF时，getchar才会停止执行，整个程序将会往下执行。并且，如果输入行是以EOF结束的（EOF之前不是换行符），则EOF会被“吃掉”（即不会被getchar读取到），getchar()的返回值一般情况下是非负 值，但也可能是负值，即返回EOF。这个EOF在函数库里一般定义为-1。

## 8.5本章小结

本章主要介绍了与hello有关的I/O部分的相关内容。首先介绍了I/O设备的管理方法，将一切I/O设备模式化为文件，并为应用程序提供接口称为Unix I/O。然后，对Unix I/O接口和其对应的函数进行了较为详细的描述，有读写文件，打开和关闭文件，文件描述符，获取文件位置等功能。接下来，对printf和getchar函数的实现机制进行了概述，在描述过程中利用了异常控制流中I/O引起中断和文件I/O操作的知识，算是所学知识的融合。

**（第8章1分）**

# 结论

用计算机系统的语言，逐条总结hello所经历的过程。

你对计算机系统的设计与实现的深切感悟，你的创新理念，如新的设计与实现方法。

Hello的一生完整贯穿了整个计算机系统的组成部分，从预处理到编译、汇编、链接到可执行文件。然后操作系统开始执行这个可执行文件，首先为该程序创建一个子进程并调用execve函数加载hello的可执行文件并建立内存映射。

Cpu更具PC中的指令开始执行程序，cpu获得的地址经过MMU转换成物理地址然后循环进行取值、分析、执行、中断的步骤，直到程序执行完毕。在外界IO设备或者其它部件发出中断请求的时候，系统会保存程序断点并处理中断事务，待到事务处理完毕又会恢复现场继续执行程序。最后程序执行完毕，子进程发信号给父进程并销毁子进程。

一个简单程序在计算机系统中的旅程让我们惊诧于计算机系统的强大与复杂，hello用它自己的身体为我们解开了计算机神秘的面纱，我们这些懵懂的人类才得以一睹它的真容，待到看穿之后，不得不叹，妙哉妙哉！

**（结论0分，缺少 -1分，根据内容酌情加分）**

# 附件

hello.i :预处理产生的文件



hello.s：编译产生的文件



hello.o: 汇编产生的文件



link.txt:链接产生的文件



hello：可执行文件



hellopoans.txt：由hello.o反汇编生成的文本文件



hellopoelf.txt：readelf得到的hello.o的信息



helloans.txt：hello的反汇编文件



helloelf.txt:readelf得到的hello的信息。



**（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] edb官方文档 https://github.com/eteran/edb-debugger/wiki/Compiling-(Ubuntu)

[2] edb操作<https://bbs.csdn.net/topics/280044538?list=lz>

[3] makefile与动态链接库案例分析——动态库链接动态库 <https://blog.csdn.net/huqinwei987/article/details/50517780>

[4] ELF文件查看利器之readelf用法<https://blog.csdn.net/roland_sun/article/details/37581715>

[5] Linux下进程的睡眠唤醒：<https://blog.csdn.net/shengin/article/details/21530337>

[6] 虚拟地址、逻辑地址、线性地址、物理地址：<https://blog.csdn.net/rabbit_in_android/article/details/49976101>

[7] 深入理解计算机系统

[8] printf函数实现的深入剖析：https://www.cnblogs.com/pianist/p/3315801.html