

**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 1170300818

班　　 级 1703008

学 生 张时光

指 导 教 师 郑贵宾

**计算机科学与技术学院**

**2018年12月**

**摘 要**

深入理解计算机系统主要讲解了计算机系统对一个函数运行中，所起到的作用以及为程序稳定快速运行所作的贡献，本文将从hello程序的一生，从预处理到虚拟内存分配，再到系统级I/O，hello一步一步成为可执行程序，再到运行稳定，功能齐全，概述计算机系统所采用的技术。

**关键词：**预处理；编译；汇编；链接；进程管理；存储管理；io管理

**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过程：在linux中，Program hello.c经过预处理、编译、汇编、链接，最终成为可执行程序hello，之后运行执行该程序，shell会调用fork函数，为其产生子进程执行该程序，成为了进程process

020过程，在fork之后，执行execve，映射到虚拟内存，进入程序后分别为其映射，分配时间，并发，执行逻辑控制流，程序结束后，shell执行wait回收执行完的hello进程。

## 1.2 环境与工具

硬件环境：Intel Core i7-6700HQ x64CPU,16G RAM,256G SSD +1T HDD.

软件环境：Ubuntu18.04.1LTS

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

## 1.3 中间结果

|  |  |
| --- | --- |
| hello1.i | 预处理完的文件 |
| hello2.i | 预处理完的文件 |
| hello.s | 编译之后的文件 |
| hello.o | 汇编之后的可重定位目标执行 |
| hello | 链接之后的可执行目标文件 |
| hello.elf | hello.o反汇编之后的ELF格式文本文件 |
| hello\_out.elf | hello反汇编之后的ELF格式文本文件 |
| hello\_out.objdump | hello反汇编文件的文本文件 |
| hello.objdump | hello,o反汇编文件 |

## 1.4 本章小结

本次作业将从p2p过程入手，一步步跟踪hello的成长，再到020过程，一步步跟踪hello在内存进程中的处理

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

试图解释为预处理指令，其中ISO C/C++要求支持的包括#if/#ifdef/#ifndef/#else/#elif/#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的#（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

作用有三：

将源文件中#include后的源文件如stdlib中链接入该文件

执行#if之后由条件所决定的代码

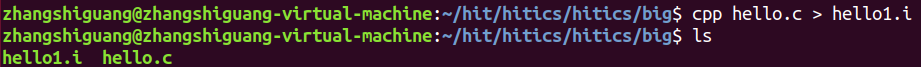
执行有关#define的宏定义

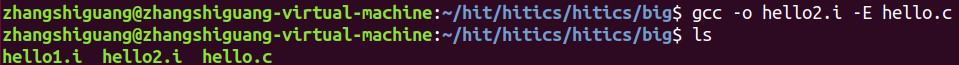
## 2.2在Ubuntu下预处理的命令

cpp hello.c > hello.i

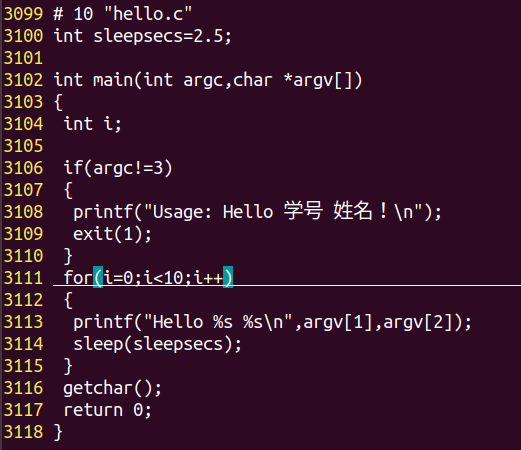
gcc hello.c -E-o hello.i

使用gcc和cpp指令生成hello.i文件





## 2.3 Hello的预处理结果解析



使用vim打开后发现main函数之前拓展了3000多行库函数文件。在展开过程中，cpp会到默认的环境变量中，也即默认的路径中去寻找原库函数，如果其中仍然使用过了#define，就将其递归展开。

## 2.4 本章小结

hello.c在他一生的第一步预处理将库文件扩展到了源文件中，为以后提供了方便

**（第2章0.5分）**

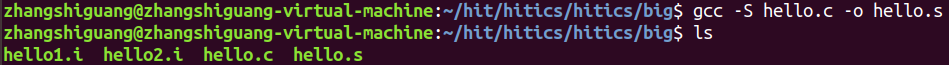
# 第3章 编译

## 3.1 编译的概念与作用

编译器将hello.i翻译成hello.s，一个源程序翻译成汇编语言的文件，而这个过程就是编译。如gcc指令指的即是GCC C编译器，linux上默认的编译器

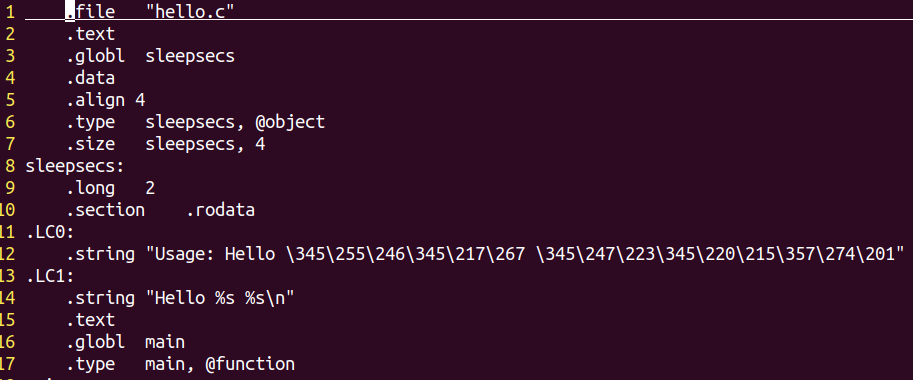
## 3.2 在Ubuntu下编译的命令

gcc -S hello.i -o hello.s



## 3.3 Hello的编译结果解析

3.3.1



所有以 “.”为开头的行都是指导汇编器和链接器工作的伪代码

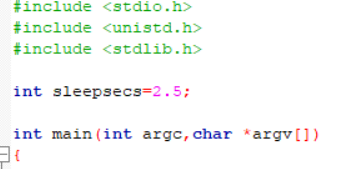
|  |  |
| --- | --- |
| 伪代码行 | 含义 |
| .file | 生命原文件名称 |
| .text | 声明一下为代码段 |
| .globl | 声明全局变量 |
| .type | 函数类型相关 |
| .size | 声明大小 |
| .long | 声明long类型变量 |
| .section .rodata | 声明以下为rodata段 |
| .string | 声明string类型 |

3.3.2数据。

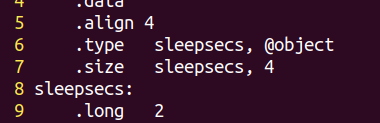
整数类型：

1. 全局变量int型sleepsecs;

源代码中将它赋值为2.5，实际上整型变量sleepsecs丢弃浮点后部分为2



在编译器翻译过的hello.C文件hello.s中，该变量在全局变量.data段被声明，并被设置为对齐方式为4，设置大小为4字节，并被设置为目标型，并且为long类型。



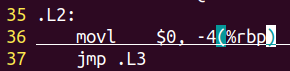
1. 局部变量（传参）argc

通过寄存器%edi传参，并压入栈内



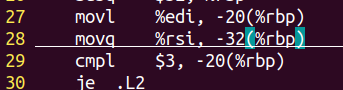
1. 局部循环变量i

存入到栈内



数组类型：

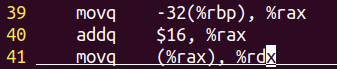
1. 局部指针变量argv，将其取出



通过寄存器将将指针变量argv传参，压入栈内

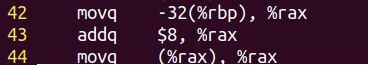
1. 数组其他元素

因为栈内数组元素是由大地址向小地址排列



将指针变量赋值给%rax

加上立即数16寻址，取出argv[1]，赋值给%rdx



将指针变量付给%rax

加上立即数8寻址，取出argv[2]，赋值给%rax

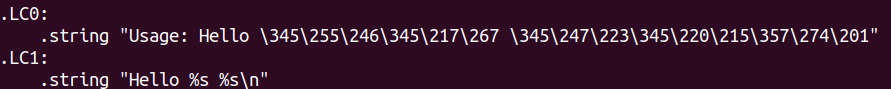
字符串类型：

1.“Usage: Hello 学号 姓名！\n”

第一个printf传入的输出格式化参数，在hello.s中声明。

（其中\345 ……为UTF-8编码汉字字符串）

2.“Hello %s %s\n”， printf传入的输出format参数，在hello.s中声明.

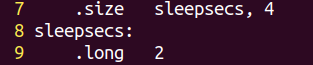


3.2.3赋值

|  |  |
| --- | --- |
| 指令 | 字节长 |
| b | 1Byte |
| w | 2Byte |
| l | 4Byte |
| q | 8Byte |

C程序汇总由两个赋值操作:1.int sleepsecs=2.5; 2. i=0

1.sleepsecs虽然被设置成long类型但，其字节长度依然为4



2.int i=0；



操作字节数为l，四字节

3.2.4类型转换

C程序中int sleepecs=2.5; 发生了类型转换，2.5默认的类型为double类型，赋值给int 类型的sleepecs，发生了类型转换，double 🡪int 类型转换，去掉浮点后的部分，即遵从向零取整。

3.2.5算术操作

C程序中设计的算术操作有：

1. i++；
2. leaq .L1（%rip）,%rdi



即%rdi=.LC1+%rip

3.2.6关系操作

C程序中的关系运算有：

1. argc!=3



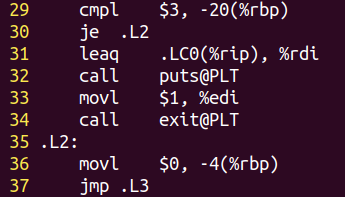
通过比较3和argc来判断，实际上是argc-3，然后设置条件码，作为下一步是否跳转的依据

1. i<10

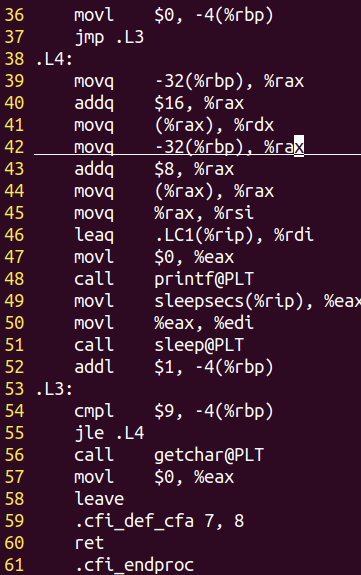


判断i小于10.计算i-9然后设置条件码

3.2.7控制跳转



这里控制着当argv!=3是执行.L2代码段内中汇编指令，如上述关系操作中je检查ZF（是否为零）标志位，如果argv-3为0，则体哦暗转，如果不等则执行下一条汇编指令。



使用循环变量i循环10次，循环终止条件为i>9，即i<=9则继续循环，当i>9，在53行则直接跳出循环，按顺序执行吓一跳汇编代码

3.2.8函数操作

C程序中设计的函数操作的有：

1. main函数：通过外部向main函数传递参数argc和\*argv[]，分别使用%rdi和%rsi储存，返回值在%eax中，为0；
2. printf函数，第一次传递字符串“Usage: Hello 学号 姓名！\n”字符串的首地址，第二次传递“Hello %s %s\n”的首地址，以及变量argv[1],argv[2]。
3. exit函数，

将%edi=1，传入



1. sleep函数

sleepsecs为%rdi，传入

5．getchar函数

无参数传入

## 3.4 本章小结

这章主要针对汇编翻译，汇编指令做了解释

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

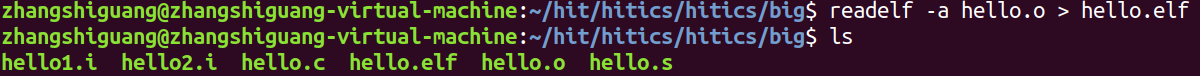
汇编器（as）将.s汇编程序翻译成机器语言指令，把这些指令打包成可重定位目标程序的格式，并把结果保存在.o二进制目标文件中。这个过程称为汇编。

注意：这儿的汇编是指从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。

## 4.2 在Ubuntu下汇编的命令

gcc -c hello,s -o hello.o

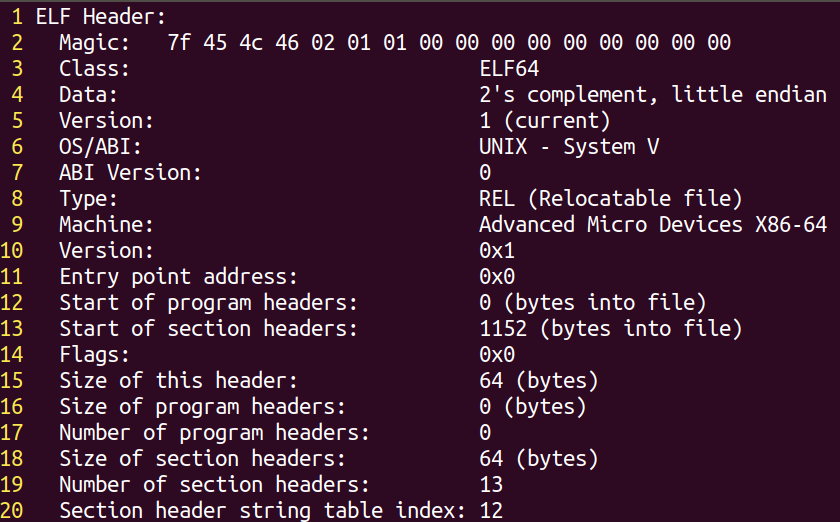
## 4.3 可重定位目标elf格式



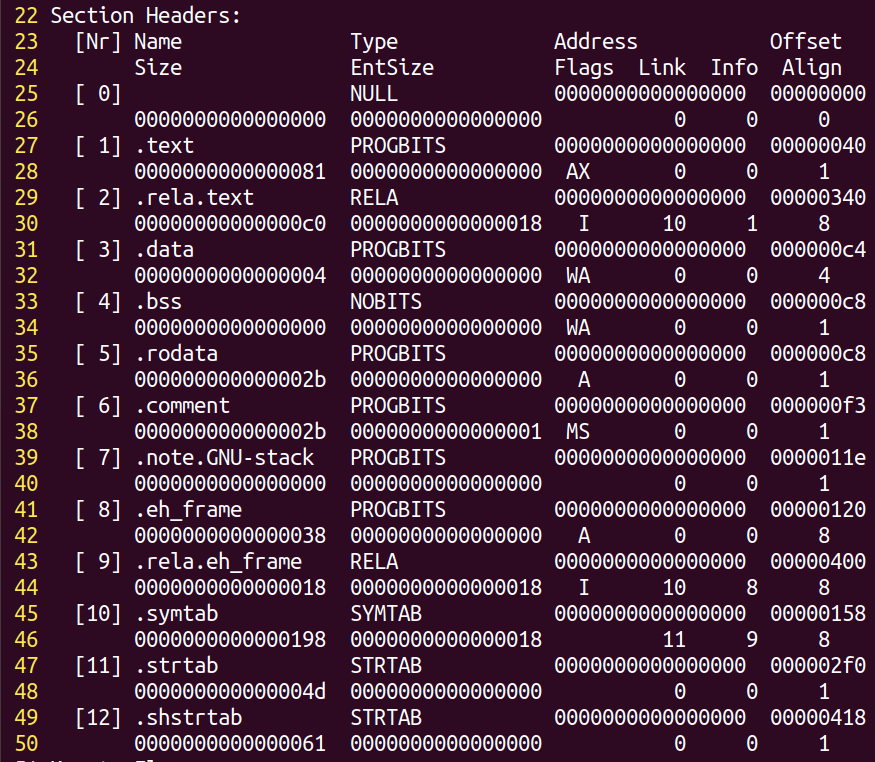
1. ELF header

Magic 开头是以16Byte开始，记录了生成该文件的系统的字大小和字节顺序

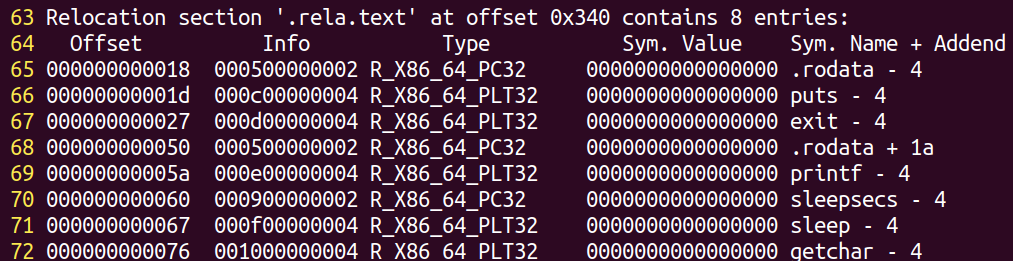
ELF头剩下的部分包含帮助链接器语法分析和解释目标文件的信息，其中包括ELF头的大小、目标文件的类型、机器类型、字节头部表的文件偏移，以及节头部表中条目的大小和数量等信息。



1. Section headers：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



1. .rela.text



重定位节.rela.text ,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。其中对puts函数，exit函数，printf函数，sleep函数，getchar()，以及字符串，sleepsecs全局变量进行函数重定位声明（offset）。

|  |  |
| --- | --- |
| Offset | 需要进行重定向文件在.text或者.data中的偏移量 |
| Info | 包括symbol和type两部分，其中symbol占前4个字节，type占后4个字节，symbol代表重定位到的目标在.symtab中的偏移量，type代表重定位的类型 |
| Type | 重定向位置的辅助信息，占8字节 |
| Addend | 重定向到目标类型 |
| Sym.Name | 重定向的目标名称 |

## 4.4 Hello.o的结果解析

指令：objdump -d -r hello.o > hello.objdump

对比hello.s和hello.objdump得出以下结论

1. 函数调用：在hello.s中，函数调用紧跟着函数名称后@PLT，声明延迟绑定，而在反汇编文件中，则指向当前指令的下一个字节（即地址）。原因是因为该函数迟绑定，该函数为共享库中函数，只有运行时，动态链接器作用后才能确定相应的PLT条目地址。

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

1. 分支转移：反汇编代码跳转指令的操作数使用的不是段名称如.L3，因为段名称只是汇编语言中的助记符，所以在汇编之后，反汇编后的hello.objdump，取而代之的是确定的偏移量。

|  |
| --- |
|  |
|  |

1. 调用全局变量：在hello.s中，访问rodata(printf中字符串)，使用段名称+%rip，而在hello.objdump中0+%rip，因为rodtata中数据地址是运行时动态链接器确定，访问时需要重定位。

|  |
| --- |
|  |
|  |

## 4.5 本章小结

本章总结了有关反汇编和汇编文件中的区别，以及ELF格式，以及汇编器为链接器做的贡献。

**（第4章1分）**

# 第5章 链接

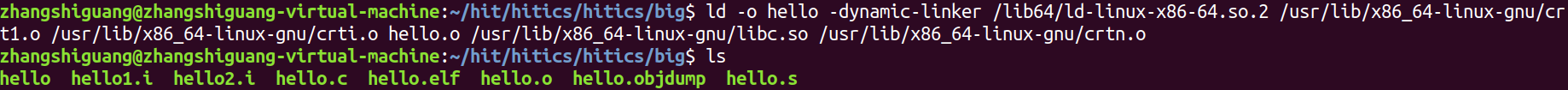
## 5.1 链接的概念与作用

链接时将各种代码和数据段连接到一个单一文件的过程，生成可执行目标文件。可被加载到内存中运行，可以在加载时执行（加载器作用），运行时执行，编译时执行。链接由链接器程序自动执行。

注意：这儿的链接是指从 hello.o 到hello生成过程。

## 5.2 在Ubuntu下链接的命令

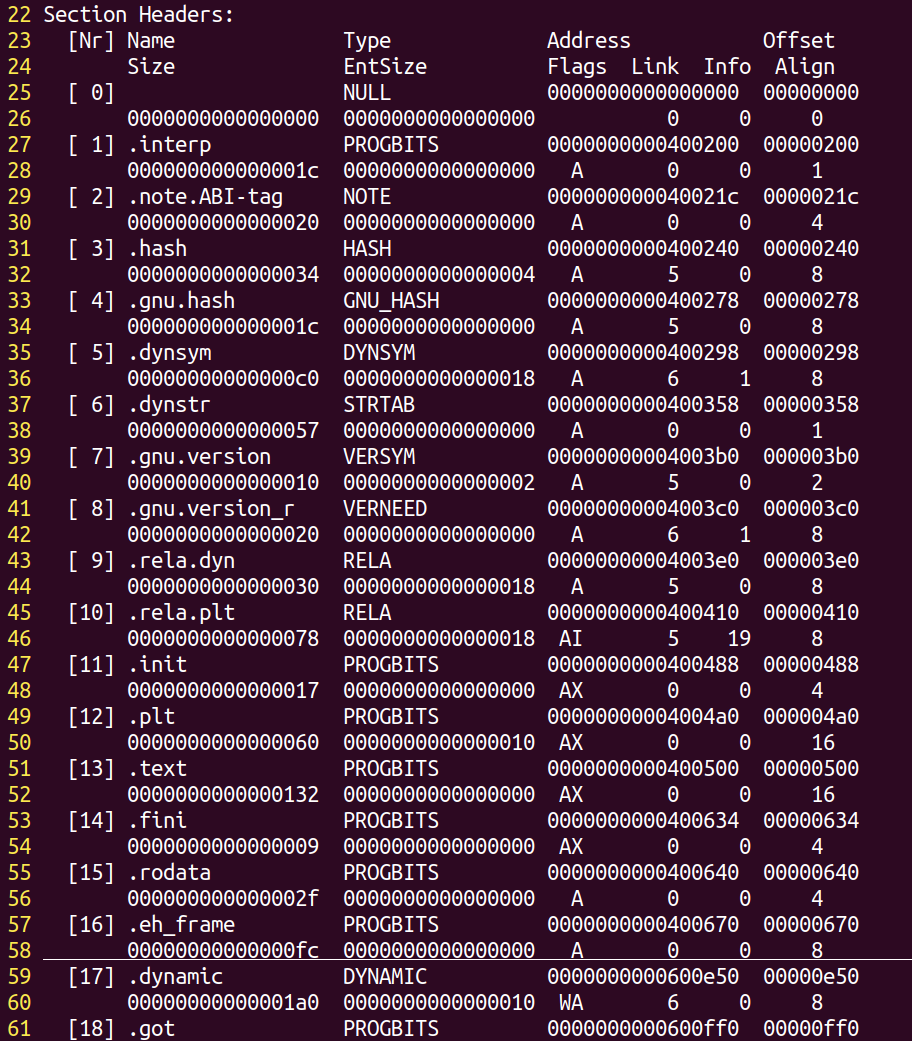
ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86\_64-linux-gnu/crt1.o /usr/lib/x86\_64-linux-gnu/crti.o hello.o /usr/lib/x86\_64-linux-gnu/libc.so /usr/lib/x86\_64-linux-gnu/crtn.o

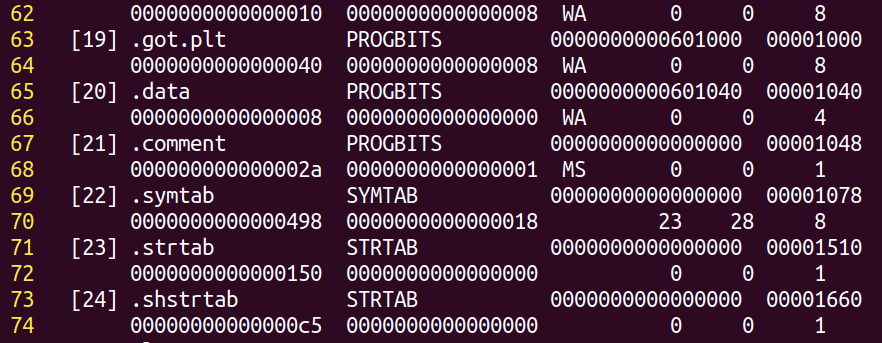


## 5.3 可执行目标文件hello的格式

shell指令：readelf -a hello > hello\_out.elf

在hello的ELF格式文件中，sectionheader声明了每个节的信息（链接过后的），由该elf格式文件信息，我们就可以清除的看到代码段，数据段的信息（偏移量，字对齐，权限，字节长，起始地址）

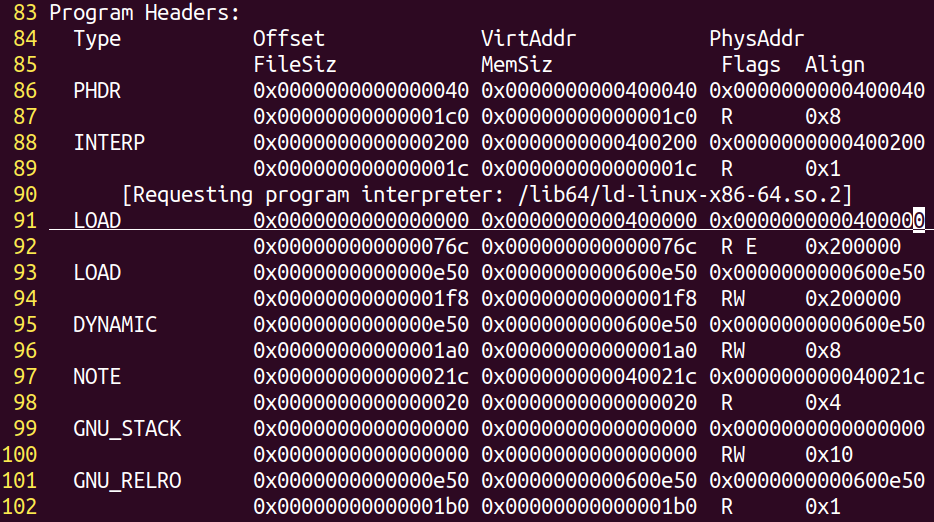




## 5.4 hello的虚拟地址空间

使用edb打开hello程序，在hello\_out.edf中虚拟内存0x400000~0x401000段，程序载入，从开始到.eh\_frame的排列如programheader中声明

程序头部表：



LOAD部分声明了两个区域，从0x400000~0x40076b是第一个内存段，有读、执行权限的代码段。，总共大小为0x76c，包含ELF头部、段头部表、.init、.text、rodata段。

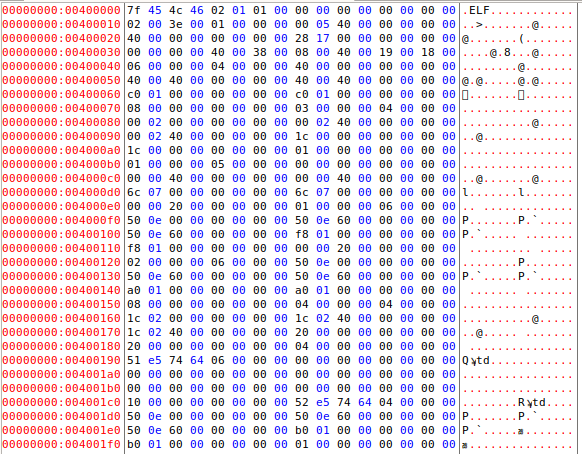
从0x600e50开始，长为0x1f8字节为第二个内存段，由读写权限的数据段。

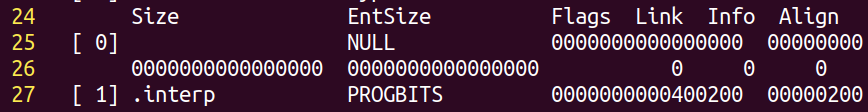
各节信息：

|  |  |
| --- | --- |
| 节名称 | 描述 |
| .interp | 保存ld.so的路径 |
| .note.ABI-tag | Linux下特有的section |
| .hash | 符号的哈希表 |
| .gnu.hash | GNU拓展的符号的哈希表 |
| dynsym | 运行时/动态符号表 |
| .dynstr | 存放.dynsym节中的符号名称 |
| .gnu.version | 符号版本 |
| dynsym | 运行时/动态符号表 |
| .dynstr | 存放.dynsym节中的符号名称 |
| .gnu.version | 符号版本 |
| init | 程序初始化需要执行的代码 |
| .plt | 动态链接-过程链接表 |
| .fini | 当程序正常终止时需要执行的代码 |
| .eh\_frame | contains exception unwinding and source language information. |
| .dynamic | 存放被ld.so使用的动态链接信息 |
| got | 动态链接-全局偏移量表-存放变量 |
| .got.plt | 动态链接-全局偏移量表-存放函数 |
| .data | 初始化了的数据 |
| .comment | 一串包含编译器的NULL-terminated字符串 |

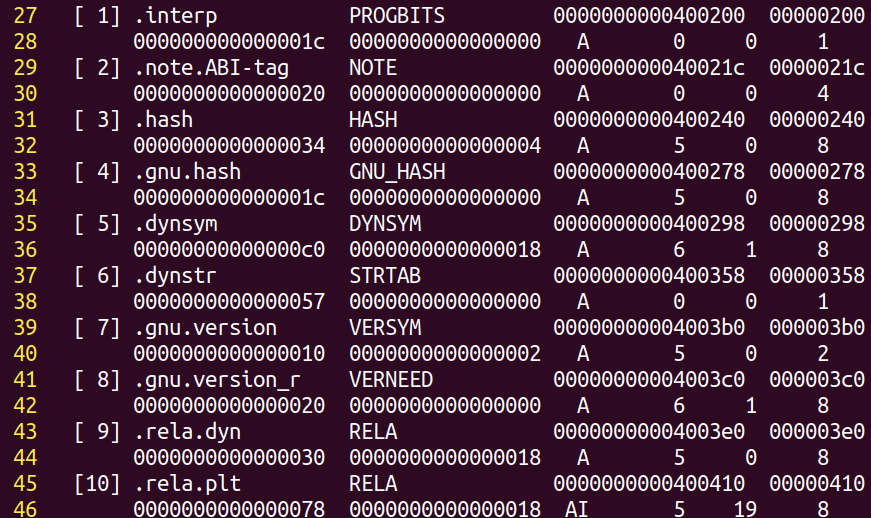
**代码段（只读）：起始地址0x400000**

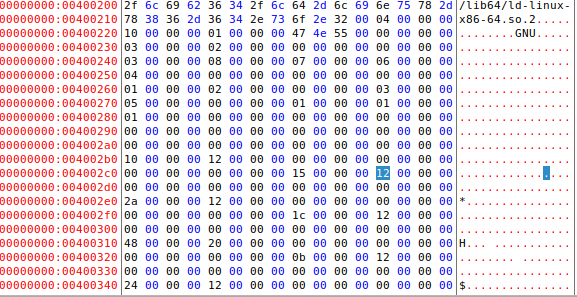
1. 0x00400000~0z4001ff都是ELF头

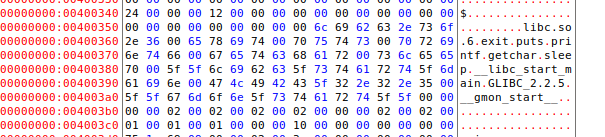




1. 0x400200~0x400487是段头部表，由连续的文件节映射到运行时内存得到的

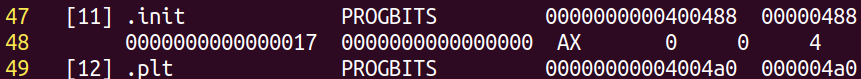






1. .init

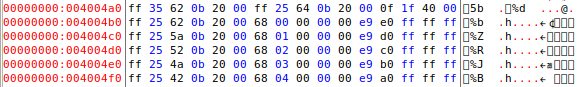
0x400488~0x4004bf是.init段，它定义了一个小函数\_init，程序的初始代码会调用它 。





1. .plt



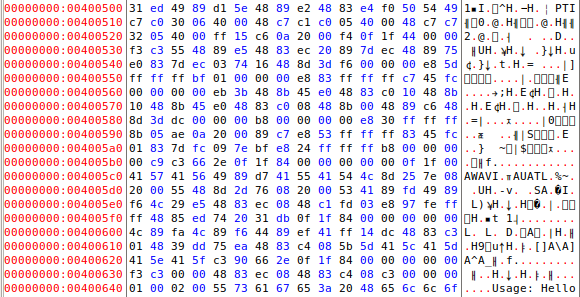


1. .text

已编译程序的机器代码

从0x400500到0x400633





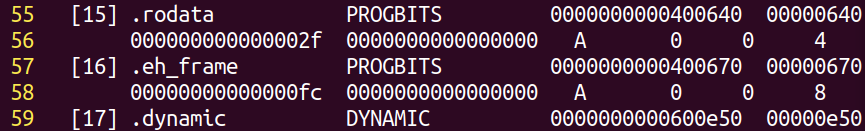
1. .fini

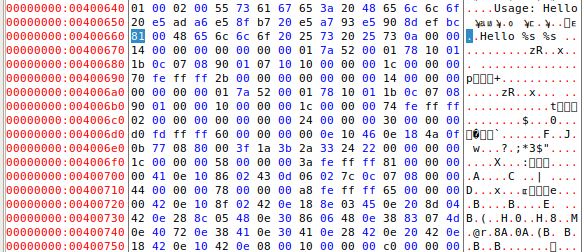




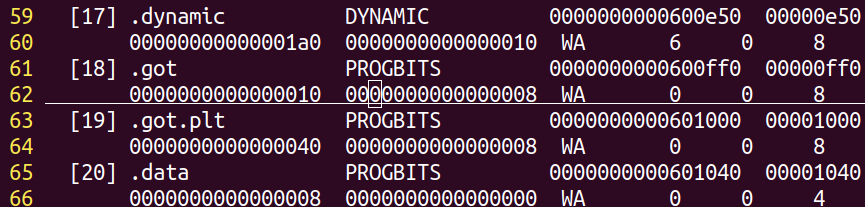
1. .rodata~.en\_frame

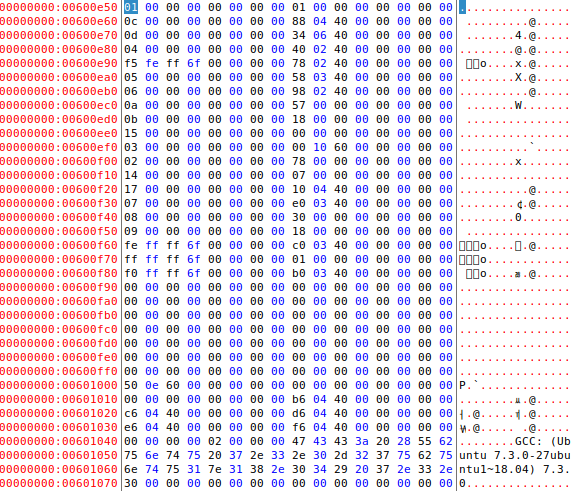
.rodata为只读数据，右侧可看到hello.c中声明的字符串





**数据段（读写）：起始地址：0x600e50**





## 5.5 链接的重定位过程分析

Shell命令：objdump -d -r hello > hello\_out.objdump

对比hello\_out.objdump 和 hello.objdump

1.函数数量：链接成为可执行目标文件后，定义了函入口\_start函数，\_init初始函数，plt条目中的printf函数，sleep函数，getchar函数。

2．.rodata的引用

链接器链接之后，确定.rodata与.text节之间相对距离，因此链接器直接修改call之后的偏移量为目标地址与下一条指令的地址之差，计算如下

refptr = s + r.offset = Pointer to 0x40054A

refaddr = ADDR(s) + r.offset= ADDR(main)+r.offset=0x400532+0x18=0x40054A

(偏移量存入地址) 



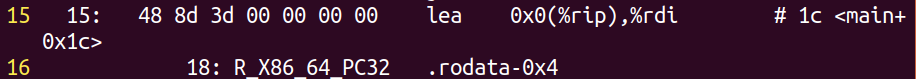
\*refptr=(unsigned)(ADDR(r.symbol)+r.addend-refaddr)

=ADDR(str1)+r.addend-refaddr

=0x400644+(-0x4)-0x40054A

=(unsigned) 0xF6





## 5.6 hello的执行流程

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

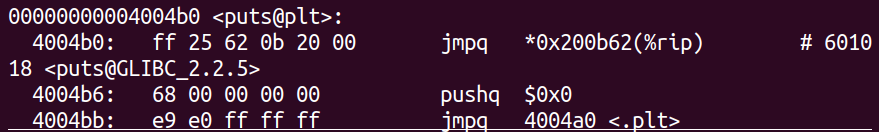
libc-2.27.so!exit

使用edb执行hello，说明从加载hello到\_start，到call main,以及程序终止的所有过程。请列出其调用与跳转的各个子程序名或程序地址。

## 5.7 Hello的动态链接分析

编译器程序调用一个由共享库定义的函数，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为值，正常的方法时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在解析它。不过**这种方法**不是PIC，因为它需要链接器修改调用模块的代码段。因而GNU编译系统就采用了延迟绑定的技术。即通过过程连接表PLT和全局偏移量表GOT两个数据结构共同实现的。GOT是数据段的一部分，PLT是代码段的一部分。他们之间存在一些对应关系。

例如puts函数的PLT表条目：



0x0为函数ID

0x601018为GOT表内内容。

调用之前的GOT.plt表





GOT[0], GOT[1] GOT[2] 和动态链接器有关。之后每个条目都是PLT调用命令的下一条指令，如GOT[3]的值就为0x4004b6即恰好为puts@plt函数中的第二条指令。

调用之后



地址被动态链接器更改，即下一次抵用PLT 直接调用共享内存库内的puts函数。



。

## 5.8 本章小结

本章针对链接的全部过程做了总结，从静态链接，到链接重定位，再到动态链接PLT和GOT表，对链接做了大致回顾。至此hello可执行目标文件，正式诞生，之后的就都是如何运行的故事了

**（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。它提供给应用程序两个假象：

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

## 6.2 简述壳Shell-bash的作用与处理流程

Shell-bash的作用：Shell字面理解就是个“壳”,是操作系统(内核)与用户之间的桥梁,充当命令解释器的作用,将用户输入的命令翻译给系统执行

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

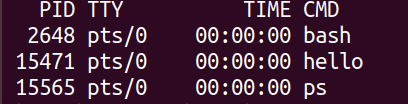
## 6.3 Hello的fork进程创建过程

Shell命令：./hello 1170300818 张时光

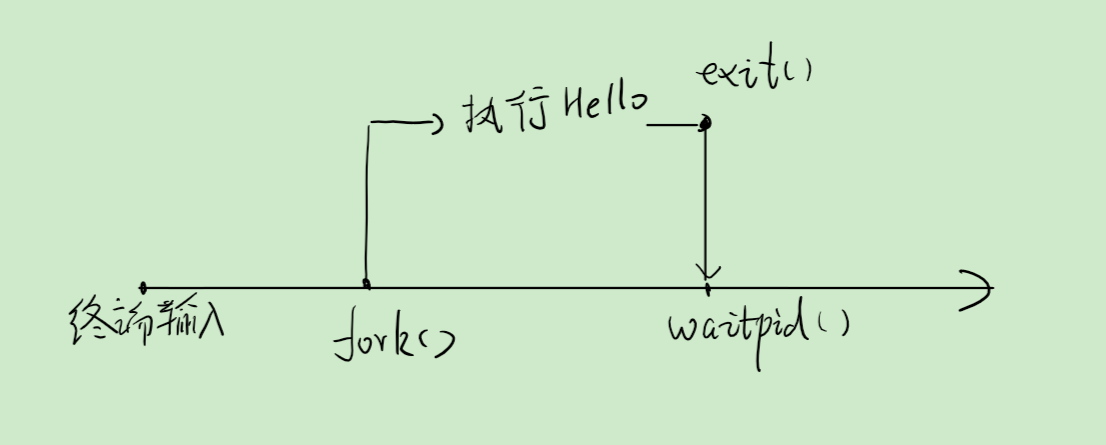
./hello 1170300818 张时光 &

因为./hello不是内部命令（可用whereis判断命令是否在bin内），因而运行该程序的时候会创建一个子进程，，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的一份独立副本，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。他们是并发运行的，内核以任意的方式交替执行。

Ps指令下的的hello：



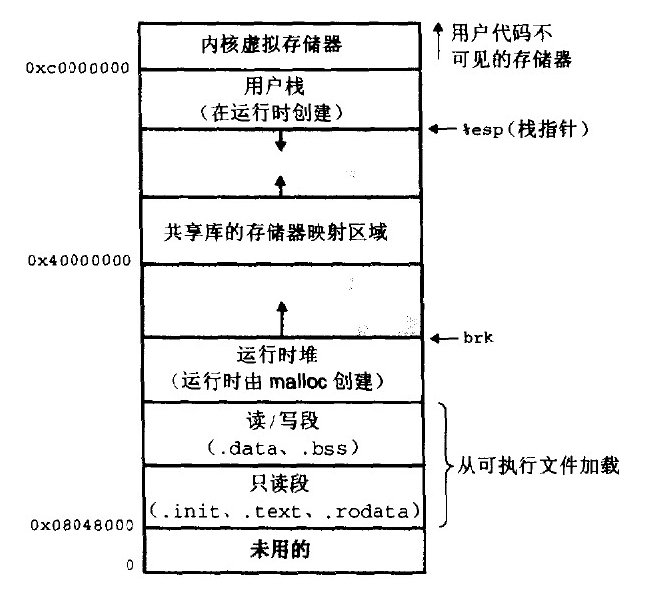
流程：



## 6.4 Hello的execve过程

execve函数在当前进程的上下文中加载并运行一个新程序

execve过程：execve在当前进程的上下文加载运行hello程序，删除原有的虚拟内存段，并创建一组新的数据段，代码段，用户栈，运行时堆，并被初始化为0，再通过虚拟内存缺页处理，将该段映射到虚拟内存，而后加载器执行将pc设置为\_start地址，运行hello程序。



## 6.5 Hello的进程执行

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。

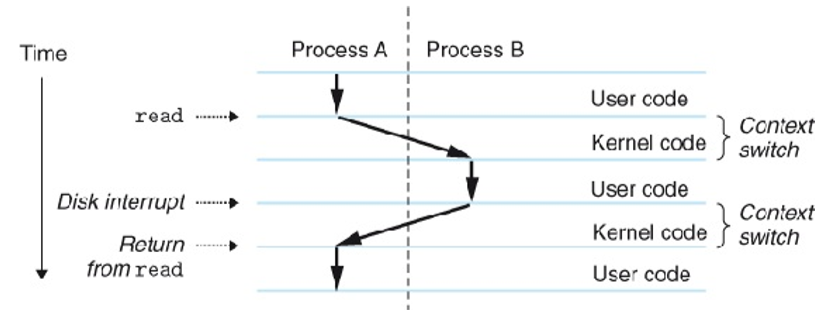
进程时间片：一个进程执行它的控制流的一部分的每一时间段叫做时间片。

进程调度：在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占的进程，这种决策就叫做调度。

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。

在hello中，如果在运行中被抢占，则进行上下切换，会保存当前hello的上下文 ,并恢复之前被抢占的某个进程被保存的上下文，并把控制传递给新进程。

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。



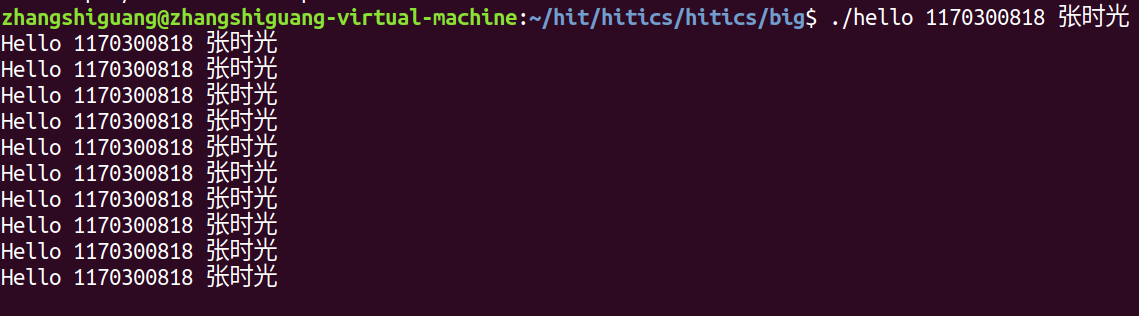
Hello运行在用户模式，调用sleep函数后，进入了内核模式，内核处理完后后释放hello进程，将hello进程放入到等待队列，程序开始sleep休眠，当休眠达到2s后发送中断信号，此时执行中断处理，将hello进程从等待队列加入运行队列，

Hello运行调用到getchar的时候，调用内核函数read，从用户模式进入到内核模式，等待键盘输入到缓冲区。此时发生调度，内核执行上下文切换，当输入结束后，缓冲区数据传输到内存完毕后，发出一个中断信号

## 6.6 hello的异常与信号处理

Hello执行过程中会出现

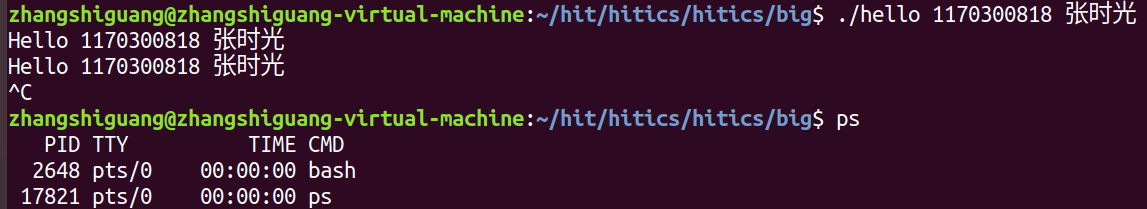
正常执行hello程序的结果：



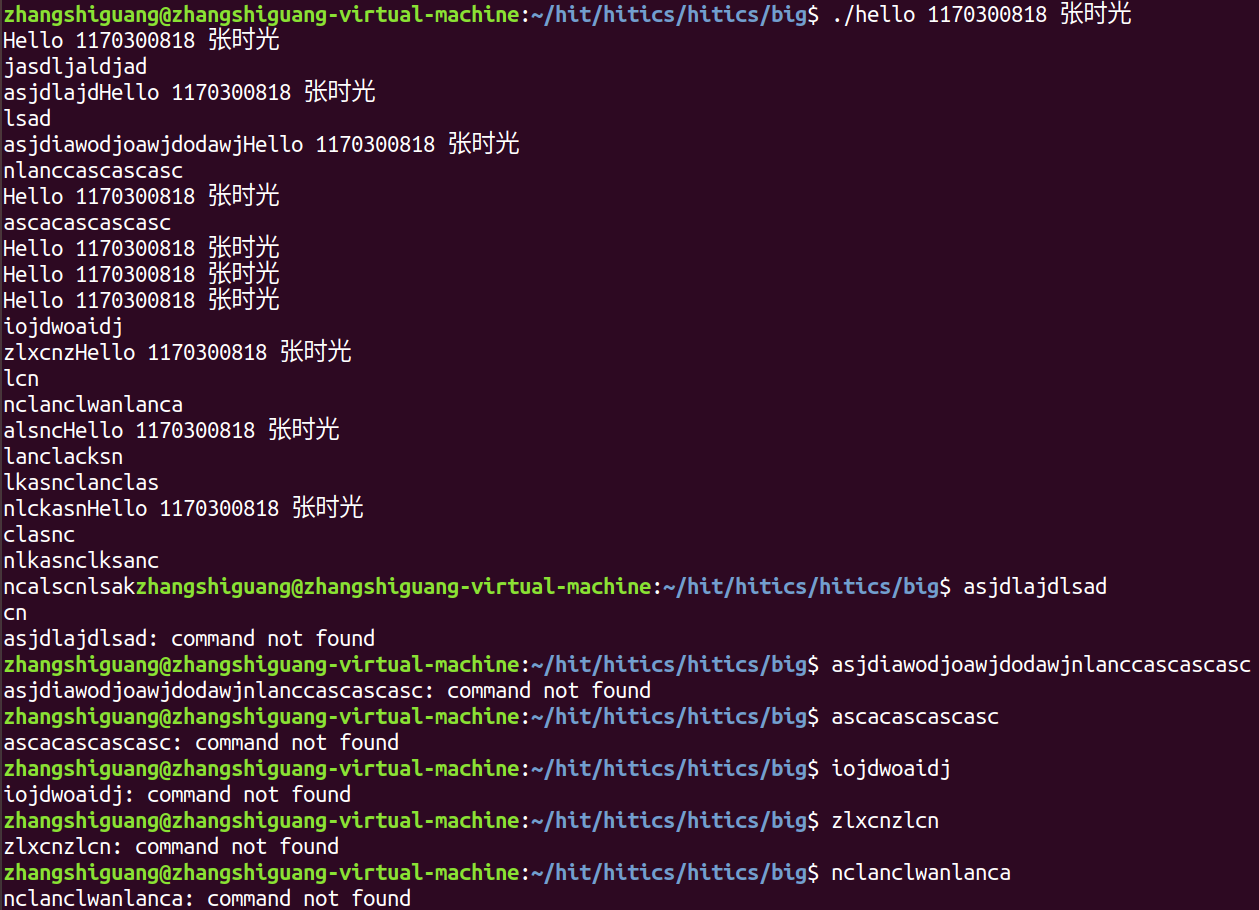
按下ctrl+z后shell父进程收到SIGSTP信号，将hello进程挂起，job为1，可用fg 1将他调到前台



按下ctrl+c后，shell 父进程收到SIGINT信号，终止hello，并被回收。



乱按之后发现，这些乱按输入的字符串，被输入到键盘缓冲区，加入运行队列，当进程结束，会被当成shell指令输入



## 6.7本章小结

这章主要总结了hello程序在linux的shell进程中运行过程，显示fork进入子进程，后是execve加载新虚拟内存，再是进程调度，内核用户切换，之后又是信号处理，至此完成了hello P2P的（艰苦）过程，撒花\*★,°\*:.☆(￣▽￣)/$:\*.°★\* 。

**（第6章1分）**

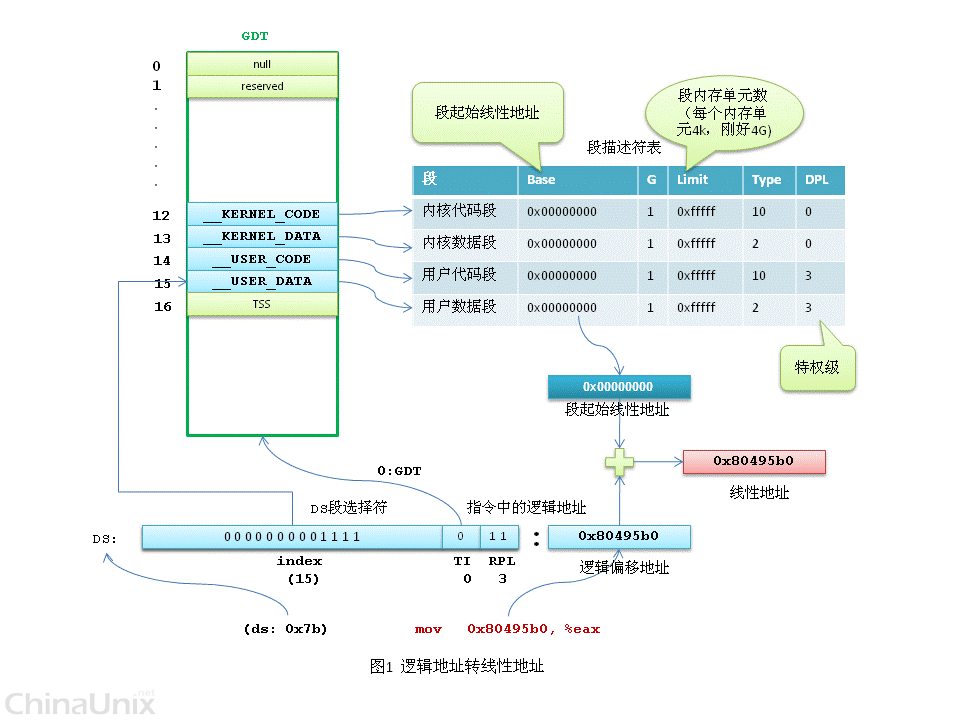
# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

逻辑地址：程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏移部分）组成。在汇编或者反汇编指令中出现的地址，如hello.s中出现的



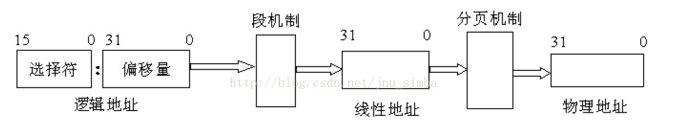
0x4005A1是一个逻辑地址，因为每个程序，代码段的逻辑地址都是从0X400000开始的，因而必须加上隐含的DS 数据段的基地址，才能构成线性地址。也就是说 0x80495b0 是当前任务的DS数据段内的**偏移**。



具体如图线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。应该是CSAPP中说的虚拟地址。

虚拟地址：现代操作系统都提供了一种内存管理的抽像，即虚拟内存。进程使用虚拟内存中的地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。

物理地址：用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相相应。



## 7.2 Intel逻辑地址到线性地址的变换-段式管理

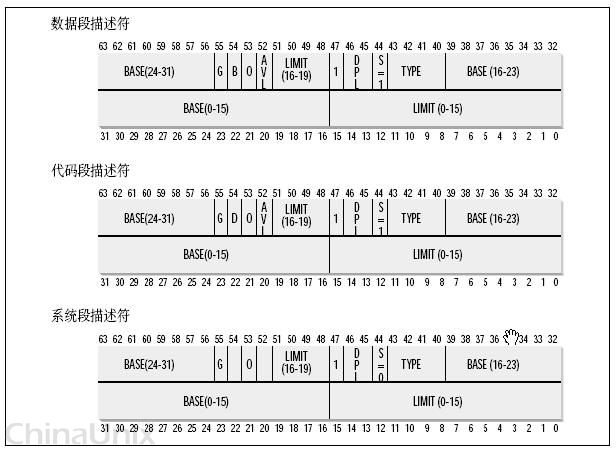
逻辑地址由两部份组成，**段标识符**: **段内偏移量**。段标识符字长16位。称位段选择符。前13位一个所以号，后3位包含硬件细节（最后两位用于权限检查）RPL。



索引号，也称作是段描述符，

因为逻辑地址转换为线性地址的过程较为复杂，就只说说linux下的段式管理吧

通过实例说明：



Base字段描述了一个段的开始位置的线性地址。而我们要得到的线性地址就是offset+base得到的。

Intel设计了GDT 全局段描述符表和LDT局部段描述符表，由T1字段决定访问GDT还是LDT。而对于linux来说，它实际上只是**哄骗了硬件**。因为linux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、内核数据段、内核代码段。

#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) 13

将宏替换成数值后

#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]

方括号后是这四个段选择符的16位二制表示、

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

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

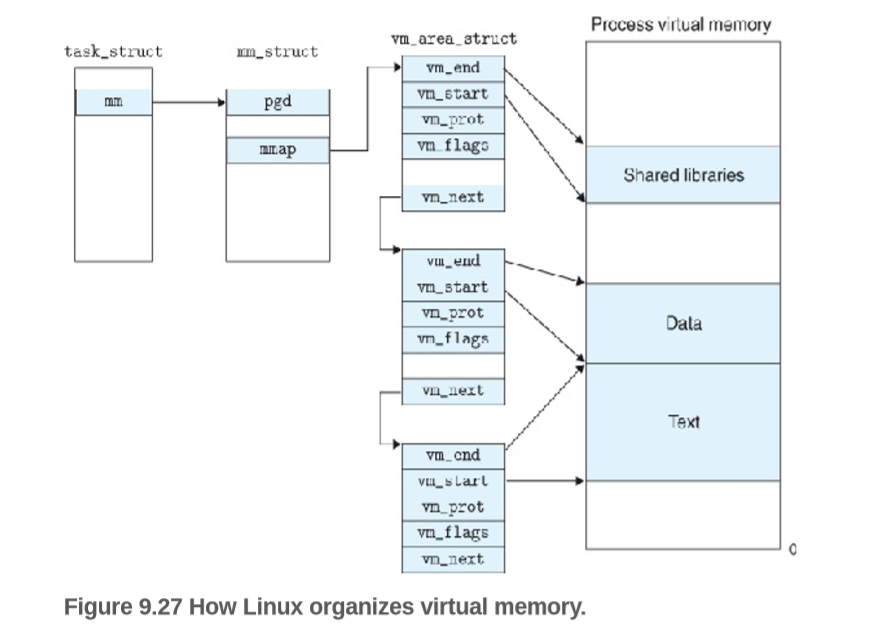
.quad 0x00cffa000000ffff /\* 0x73 user 4GB code at 0x00000000 \*/

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

最终发现，发现他们16~31位全为0，即在linux环境下逻辑地址与线性地址总是一致的。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

Linux下如何组织虚拟内存：



Linux位每个进程维护了一个单独的虚拟地址空间。Linux系统拥有一个进程中虚拟内存的内核数据结构，图中的task\_strcut。mm指向一个mm\_struct，它描述了虚拟内存的当前状态，pgd指向第一级页表的基地址，mmap指向一个vm\_area\_struct的链表，其中

vm\_start:指向这个区域的起始处

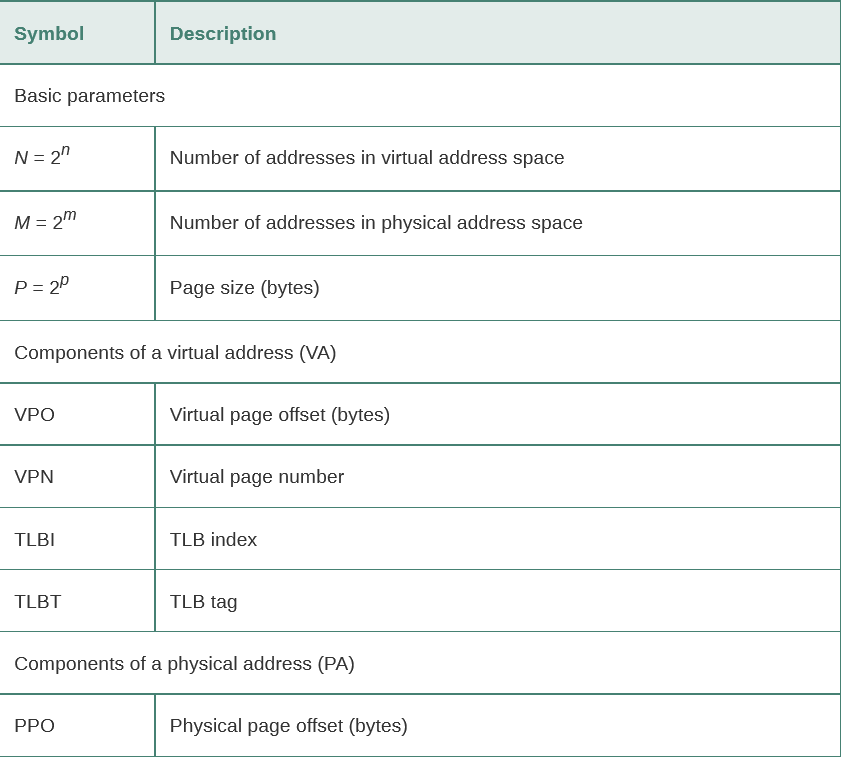
vm\_end指向这个取与的结束处。

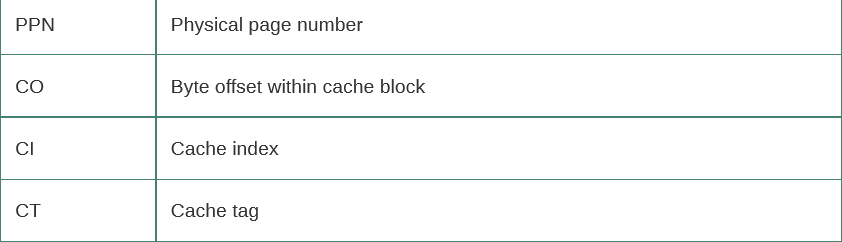
Vm\_prot：指向权限

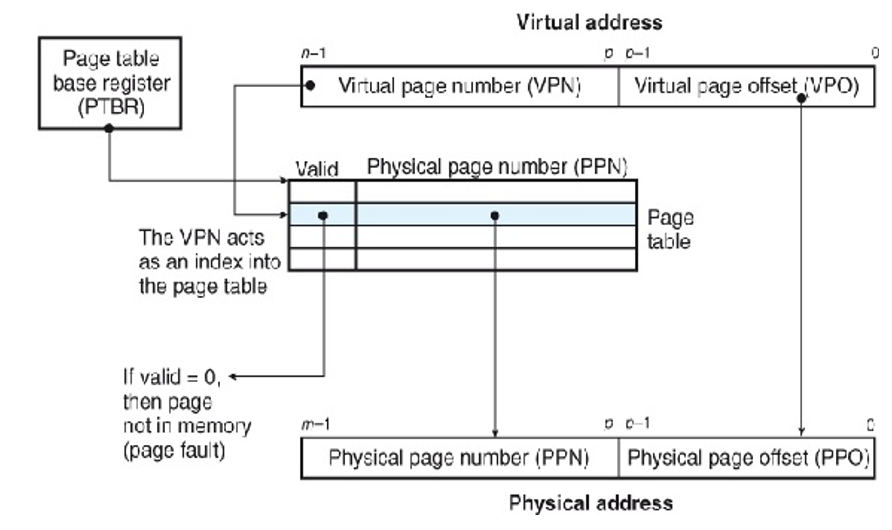
。。。。。。

vm\_next：指向下一个区域结构（如图共享库（shared libraries）数据段（Data）、代码段(Text)）

而对于每个段来说，系统又将他们分成了大小固定的块称作虚拟页，linux



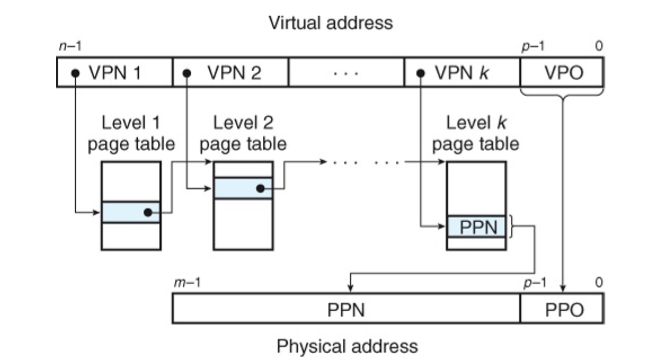


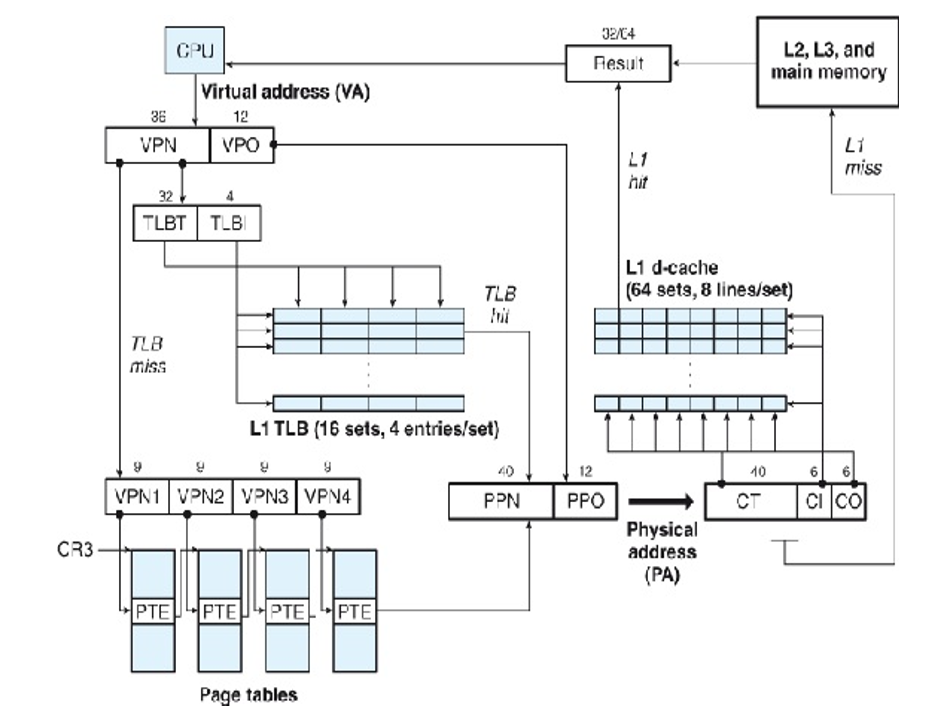


由页表基址寄存器(Page table base register)指向当前页表。n位的虚拟地址包含两个部分一个是p位的虚拟页面偏移（VPO），一个是（n-p）位的虚拟页号（VPN）。

MMU利用VPN来选择适当的PTE（用VPN作为索引），如果找到的物理页号有效位（valid）为0，则发生缺页（page fault）

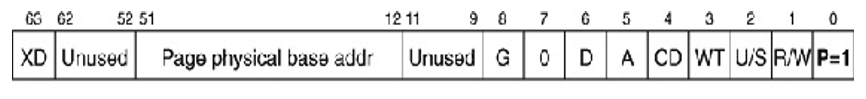
## 7.4 TLB与四级页表支持下的VA到PA的变换





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

页表大小为4，每一级PTE共9位索引，即512倍，则第四级PTE条目大小为4KB，L3的为2MB，L2的1G，L1的512G（对于个人计算机来说大的恐怖！），



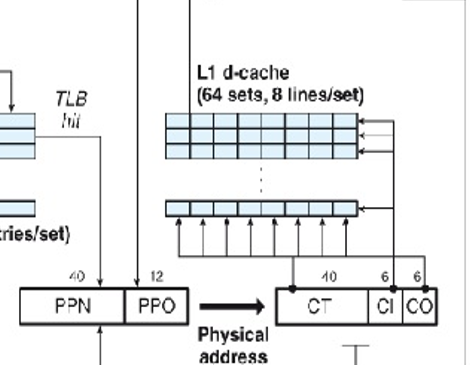
CPU产生虚拟地址VA， VA传送给MMU，MMU使用前36位VPN作为TLBT在TLB中匹配，如果命中，则得到PPN（40bit）与VPO（12bit）组合成PA（52bit）。

如果不幸没有命中，则会从第一级页表开始查询，VPN1为第一级页表偏移量（9Bytes），只知道第四页中查询PPN，如查到，与VPO组成PA，添加到TLB中，重新执行该指令。

如果查询时发现该地址为无效的，则发生缺页。

## 7.5 三级Cache支持下的物理内存访问

L1、L2、L3原理相似，只讨论L1



和PLT中的命中相类似

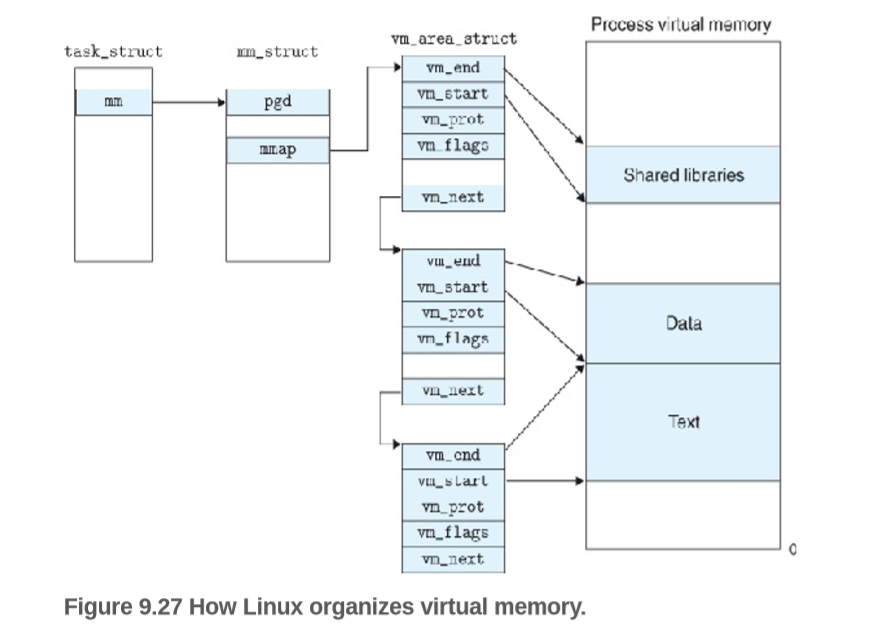
CO为块偏移，CI为块索引，CT为标记tag，因为一组八路，通过CI匹配，再通过CT查找，如果命中，则根据CO取出相应的块

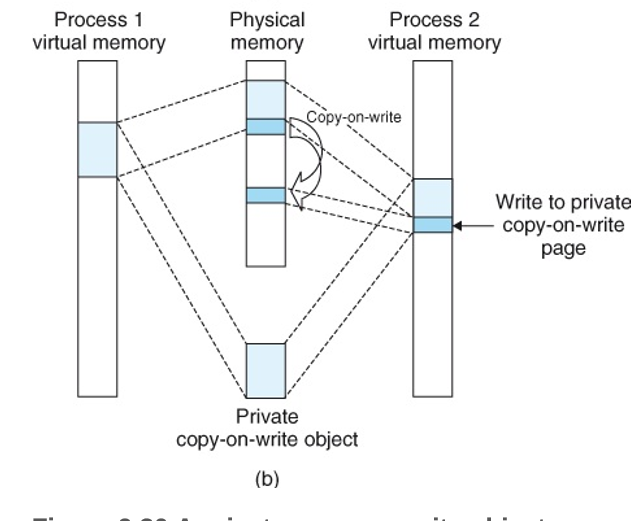
如果miss，就向下一级缓存查寻数据并取回。

如果miss，组内数据全部有效，则会发生驱逐（evict）根据相应算法（lru）行替换

（以下格式自行编排，编辑时删除）

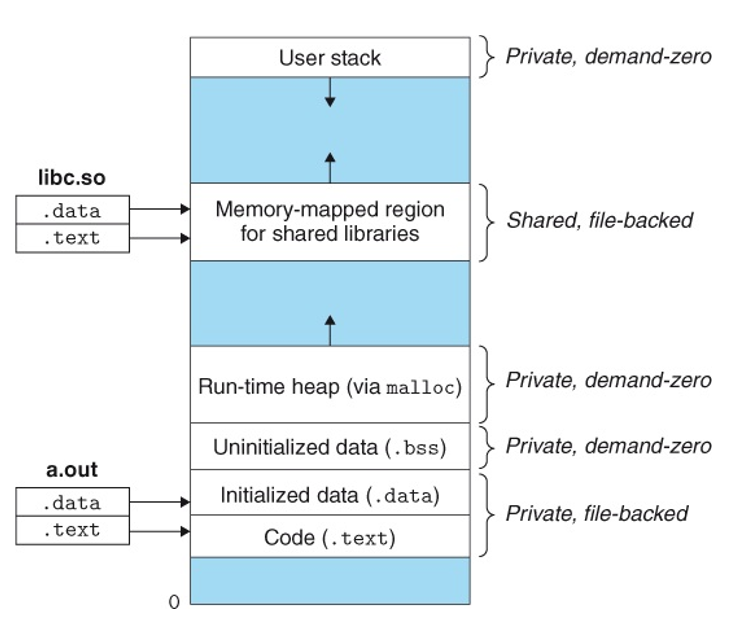
## 7.6 hello进程fork时的内存映射





Fork在传一个子进程后，为该新进程创建数据结构，创建虚拟内存，内核为新进程维护了一个单独的任务结构task\_struct，并保存在数结构中，并且将两个进程区域结构为私有的写时复制，两个进程的灭个页面都标记为已读。

## 7.7 hello进程execve时的内存映射



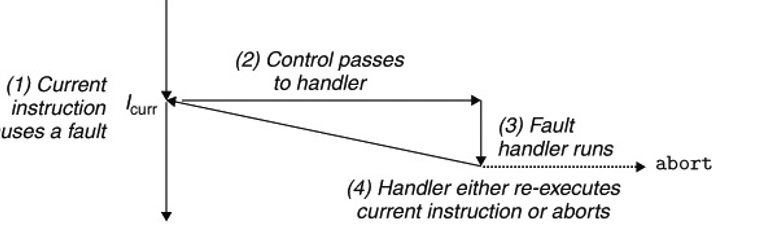
如上述于第六章所述：

Execve会：

1. 删掉已存在的用户区域
2. 映射到私有区域：为新程序的代码、程序、.bss、栈区域创建新的区域结构，新的区域都是**私有的、写时复制的，**
3. 映射共享区域
4. 设置程序计数器，将PC设置到\_start函数程序入口。

## 7.8 缺页故障与缺页中断处理

缺页故障是一种故障，当发生缺页时，该指令必须转到内核中去处理，去磁盘中查找，如果查找成功反回当前指令（重新执行），不成功则不返回，终止该指令。其处理流程遵循故障处理流程。

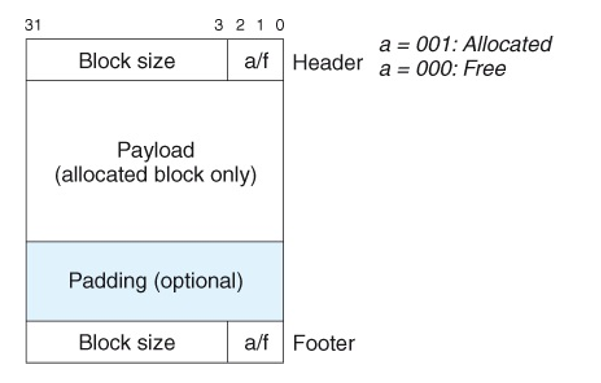


## 7.9动态存储分配管理

分配器分为两种基本风格：显式分配器、隐式分配器。

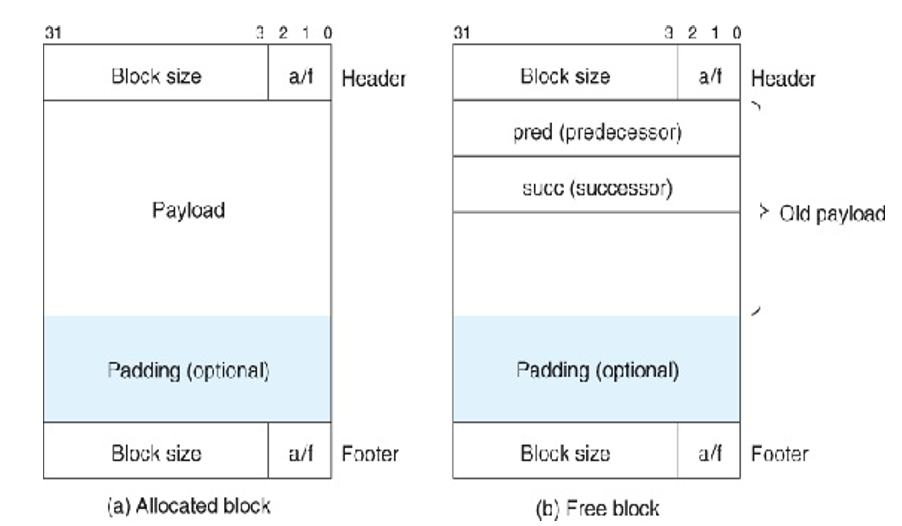
显式分配器：要求应用显式地释放任何已分配的块。

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。



Header占用4Byte，footer占用4Byte，header在隐式空间链表中起到了指针一样的作用，而footer是为了寻找空闲块而设计的，如果优化得当，可省略已分配块中的footer。

显示空间链表：



使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。

## 7.10本章小结

本章讲述了hello运行时，在内存中安家入户的过程，先是fork，创建当前进程的数据结构，并且与父进程被标记为私有的写时复制，之后通过execve加载器，初始化申请的数据结构，并将PC置于函数入口，之后再通过至少三次缺页将虚拟内存中的代码段，数据段，栈等申请物理内存，这样hello才能一路运行下去。

**（第7章 2分）**

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

设备的模型化：所有的IO设备都被模型化为文件，输入输出都作为相应文件的读和写的执行

设备管理：unix io接口，是linux内核简单的低级的接口。

## 8.2 简述Unix IO接口及其函数

1.打开和关闭文件。进程是通过调用open函数来打开一个已存在的文件或者创建一个新文件的。会调用close函数关闭一个打开的文件。Open函数将文件名转换为一个文件描述符，并且反汇描述符数字，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擦拭农户指明了进程打算如何访问这个文件：O\_RDONLY:只读，O\_WRONLY只写。O\_RDWR可读可写。访问权限由unmask来实现

2.读和写文件：引用程序通过分别调用read和write函数来执行输入和输出的。

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表示一个错我，返回值0表示EOF，否则返回值表示的是实际传送的字节数量。

Write函数从内存位置buf赋值之多n个字节到描述符fd的当前文件位置。

3.查找文件，同年过调用lseek函数，应用程序能够显示地修改当前文件的位置，这部分内容不再我们的讲述范围之内。

## 8.3 printf的实现分析

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;

}

(char\*)(&fmt) + 4) 表示的是...中的第一个参数。因为栈逆向生长。且传压栈参数是从右向左压栈。

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的作用就是格式化。它接受确定输出格式的格式字符串fmt。用格式字符串对个数变化的参数进行格式化，产生格式化输出。

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

sys\_call实现很麻烦，但他能实现显示格式化了的字符串，于是上述sys\_call就可以解释为%ecx中为打印的变量个数，%ebx为打印出buf字符数组中的第一个字符，[gs:edi]对应的是0x80000h：0采用直接写显存的方法显示字符串

即

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

本章主要总结了系统级的I/O，printf的实现、getchar的实现、至此，hello的一生彻底完结

**（第8章1分）**

# 结论

hello完成了精彩的一生

它完成了预处理，将自己调用的所有库函数展开合并在了一起

它完成了编译，将自己的C程序翻译成了汇编语言

它完成了汇编，把自己变成了一个可重定向的二进制目标文件

它完成了链接，它通过重定位和动态链接，成为了可执行目标程序

它完成了运行，它在shell程序里，在fork创建的子进程里运行，在execve加载后运行，在内核指令控制下与其他进程并发的执行，并在结束后，被回收了。

它完成了虚拟内存的申请，和父进程一同被设置为私有的写时复制，它通过execve初始化，并在PLT TLB MMU 中完成它内存的访问申请

它完成了堆内空间的申请分配。

# 附件

|  |  |
| --- | --- |
| hello1.i | 预处理完的文件 |
| hello2.i | 预处理完的文件 |
| hello.s | 编译之后的文件 |
| hello.o | 汇编之后的可重定位目标执行 |
| hello | 链接之后的可执行目标文件 |
| hello.elf | hello.o反汇编之后的ELF格式文本文件 |
| hello\_out.elf | hello反汇编之后的ELF格式文本文件 |
| hello\_out.objdump | hello反汇编文件的文本文件 |
| hello.objdump | hello,o反汇编文件 |

**（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] Randal E. Bryant. Computer Systems A Programmer's Perspective. Carnegie Mellon University

[2] <https://blog.csdn.net/freeelinux/article/details/54136688>

[3] http://bbs.chinaunix.net/thread-2083672-1-1.html

**（参考文献0分，缺失 -1分）**