

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机

学　　 号 1190202413

班　　 级 05

学 生 庄雨杰

指 导 教 师 史先俊

**计算机科学与技术学院**

**2021年5月**

**摘 要**

本文介绍了hello程序的一生，在linux下借助一些工具，对hello程序如何从一个文本文件hello.c经过预处理、编译、汇编及链接，再通过进程管理、异常与信号处理、内存管理及I/O管理一步步地在计算机中运行的过程做了详细地分析，下面让我们走进hello辉煌的程序人生。

**关键词：**hello;预处理；编译；汇编；链接；进程管理；存储管理；I/O管理

**（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

Hello的P2P：

P2P，即From Program to Process。使用vim、CodeBlocks等工具进行编程，最后得到一个hello.c文件的过程叫做program。

如图1.1所示，hello.c经过预处理、编译、汇编、链接这四个阶段后，生成可执行文件hello。然后在linux下的命令行输入./hello 1190202413 庄雨杰 1，Shell会会调用fork函数创建一个子进程，并调用execve函数子进程的上下文中运行这个程序。execve函数加载并运行可执行文件hello，且带参数列表argv（这里的参数列表就是hello、1190202413、庄雨杰、1）和环境变量envp，通过将虚拟内存映射到物理内存以及内核的进程调度与其他程序并发地运行。这就是hello的process。以上就是hello的P2P。

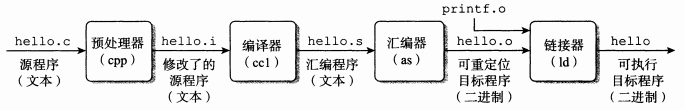


图1.1 hello.c编译过程

Hello的020：

020，即从无到有，再从有到无。一开始的无是hello运行之前，不占任何内存资源。而之后在linux终端下运行hello，shell为其fork一个子进程，子进程通过execve系统调用启动加载器，加载器删除子进程现有的虚拟内存段，并创建一组新的代码、数据、堆和栈段。通过将虚拟地址空间中的页映射到可执行文件hello的页大小的片，新的代码和数据段被初始化为可执行文件的内容。然后开始执行hello程序，通过段式管理、页式管理、多级cache、I/O设备等联动，CPU为hello程序分配时间片，对每条指令在流水线上进行取值、译码、执行、访存、写回、更新PC。最后当程序运行结束时，父进程shell将子进程hello回收，释放其内存和上下文信息，此时hello不占任何内存资源。这就是hello的020（从无到有，再从有到无）的过程。

## 1.2 环境与工具

硬件环境：笔记本电脑（x64 Intel Core i5-9300H CPU；2.40GHz；8G RAM；256GHD Disk）

软件环境：Windows10 64位；Vmware 15.5.0 build-14665864；Ubuntu 20.04.2 LTS 64位

开发工具：Visual Studio 2019 64位；CodeBlocks；Winhex; 记事本；vi/vim/gpedit+gcc；EDB；GDB/OBJDUMP；TestStudio；Gprof；Valgrind

## 1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

hello.c：源程序

hello.i：预处理后的文本文件

hello.s：编译后汇编程序文本文件

hello.o：汇编后的可重定位目标文件（二进制文件）

hello：链接后的可执行目标文件

## 1.4 本章小结

本章介绍了hello的P2P和020，列举了完成这次大作业的环境、工具以及过程中产生的文件。

**（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：

程序设计领域中，预处理一般是指在程序源代码被翻译为目标代码的过程中，生成[二进制代码](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6%E4%BB%A3%E7%A0%81/4879654" \t "_blank)之前的过程，其实就是预处理器将.c文件翻译成.i文件的过程。典型地，由预处理器(preprocessor) 对程序源代码文本进行处理，得到的结果再由编译器核心进一步编译。这个过程并不对程序的源代码进行解析，但它把源代码分割或处理成为特定的单位——（用C/C++的术语来说是）预处理记号(preprocessing token)用来支持语言特性（如C/C++的宏调用）。

常见的预处理是C语言和C++语言。ISO C和ISO C++都规定程序由源代码被翻译分为若干有序的阶段(phase) [1]  [2]  ，通常前几个阶段由预处理器实现。预处理中会展开以#起始的行，试图解释为[预处理指令](https://baike.baidu.com/item/%E9%A2%84%E5%A4%84%E7%90%86%E6%8C%87%E4%BB%A4" \t "_blank)(preprocessing directive) ，其中ISO C/C++要求支持的包括#if/#ifdef/#ifndef/#else/#elif/#endif（[条件编译](https://baike.baidu.com/item/%E6%9D%A1%E4%BB%B6%E7%BC%96%E8%AF%91)）、#define（[宏定义](https://baike.baidu.com/item/%E5%AE%8F%E5%AE%9A%E4%B9%89)）、#include（源[文件包含](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E5%8C%85%E5%90%AB" \t "_blank)）、#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：

C语言预处理程序的作用是根据源代码中的预处理指令修改你的源代码。预处理指令是一种命令语句(如#define)，它指示预处理程序如何修改源代码。在对程序进行通常的编译处理之前，编译程序会自动运行预处理程序，对程序进行编译预处理，这部分工作对程序员来说是不可见的。

例如#define指令定义了一个宏---用来代表其他东西的一个命令，通常是某一个类型的常量。预处理会通过将宏的名字和它的定义存储在一起来响应#define指令。当这个宏在后面的程序中使用到时，预处理器”扩展”了宏，将宏替换为它所定义的值。

#include指令告诉预处理器打开一个特定的文件，将它的内容作为正在编译的文件的一部分“包含”进来。例如：下面这行命令：

#include<stdio.h>指示预处理器打开一个名字为stdio.h的文件，并将它的内容加到当前的程序中。

预处理器的输入是一个C语言程序，即.c文件。程序可能包含预处理指令。预处理器会执行这些指令，并在处理过程中删除这些指令。预处理器的输出是另外一个程序：原程序的一个编辑后的版本，不再包含指令，即.i文件。预处理器的输出被直接交给编译器。

## 2.2在Ubuntu下预处理的命令

指令：gcc -m64 -no-pie -fno-PIC -E hello.c -o hello.i

截图：

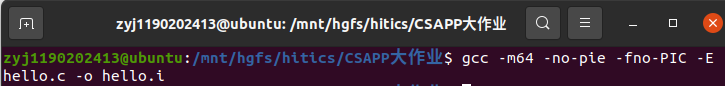


图2.1 Ubuntu下预处理指令

## 2.3 Hello的预处理结果解析

截图：



图2.2 预处理生成hello.i

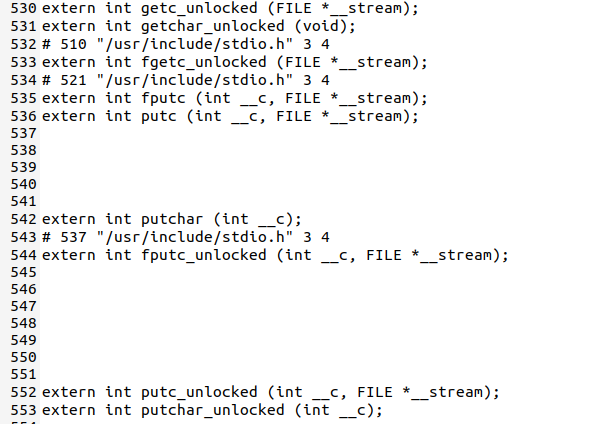


图2.3 hello.i部分内容1

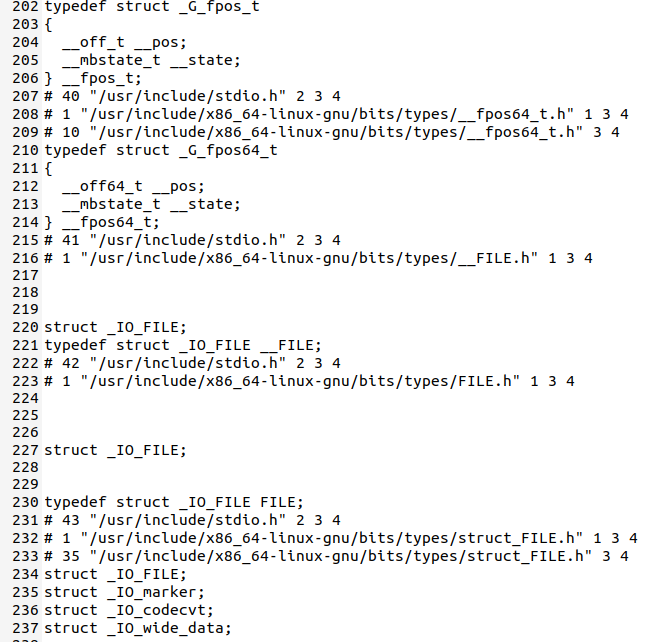


图2.4 hello.i部分内容2

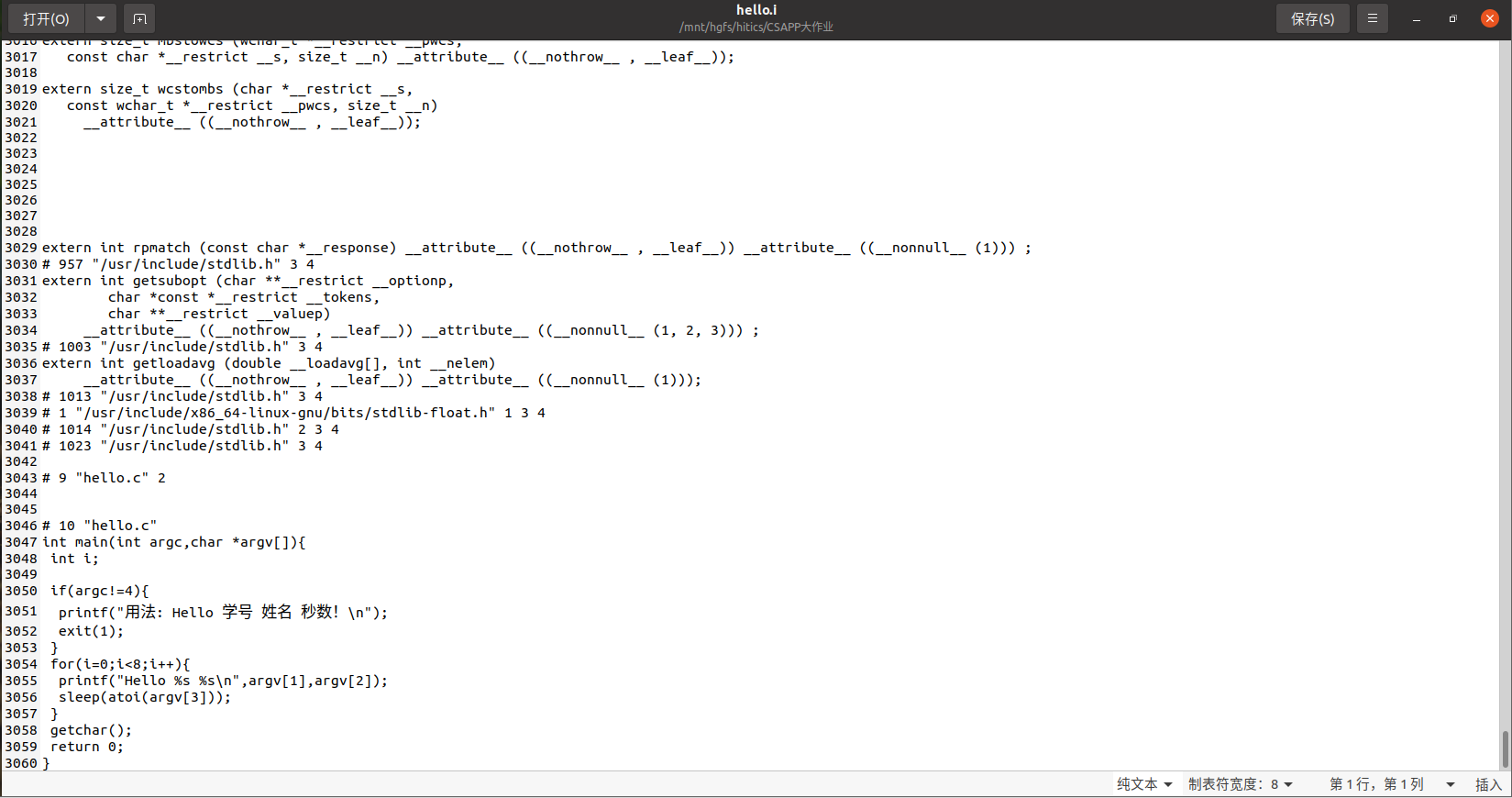


图2.5 hello.i部分内容3

解析：

如图2.3、2.4、2.5所示，我们可以看到源程序短短23行代码，经预处理成hello.i文件包含了3060行代码。这是因为源程序hello.c中有如下三条预处理指令：

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

这三条预处理指令告诉预处理器读取系统头文件stdio.h、unistd.h、stdlib.h的内容，并把它们插入到程序文本中，一般包括结构体的定义（typedef struct）、函数声明（extern）、对引用目录的标注（# 28 "/usr/include/stdio.h" 2 3 4）等内容。若源程序中有#define预处理指令还会对相应的符号进行替换，或者其他类型的预处理指令，预处理器也会执行对于的操作。

我们再看源程序的其他部分，预处理得到的hello.i中的main函数和源程序保持一致，因为预处理器不会对其他部分进行修改。

## 2.4 本章小结

本章介绍了预处理的概念与作用，以及在linux下源程序hello.c文件怎么经过预处理将头文件插入到程序文本中，最后得到hello.i文件的过程。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

概念：

编译，就是编译器（cll）将.i文本文件翻译成.s文本文件的过程，即把预处理后的代码转化为汇编指令的过程。汇编指令只是CPU相关的，也就是说C代码和python代码，代码逻辑如果相同，编译完的结果其实是一样的。

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

编译过程中会检查语法是否存在错误。编译后生成的.s文件，即汇编程序比预处理文件更容易让机器理解、比.o可重定位目标文件更容易让程序员理解。生成的汇编程序可以由汇编器（as）汇编后进一步形成可重定位目标程序（二进制）的.o文件。

## 3.2 在Ubuntu下编译的命令

指令：gcc -m64 -no-pie -fno-PIC -S hello.c -o hello.s

截图：

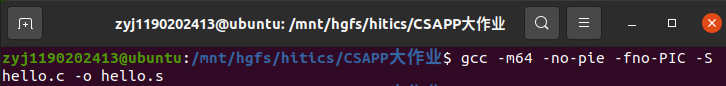


图3.1 Ubuntu下编译指令

## 3.3 Hello的编译结果解析

截图：

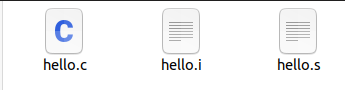


图3.2 编译生成hello.s

### 3.3.1数据

hello.s中的数据有常量、

<1>常量

如图3.3所示，源程序hello.c中有两个字符串常量，都是作为printf的格式参数，一个是"用法: Hello 学号 姓名 秒数！\n"，另一个是"Hello %s %s\n"。

如图3.4所示，在hello.s文件中，编译器把它们放在标号LC0中和标号LC1中。至于为何如此表示，根据UTF-8的编码原则，汉字被编码为三个字节，在hello.s中以编码形式保存；而英文、\n、%、空格等字符串，则被编码为一个字节，与ASCII码的规则兼容，这些字符在hello.s中以原来的形式保存。



图3.3 hello.c中的main函数

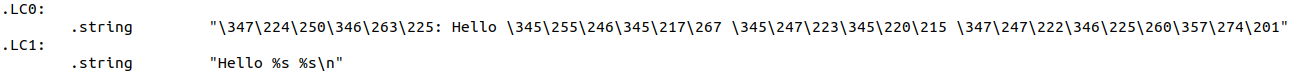


图3.4 hello.s中的字符串常量

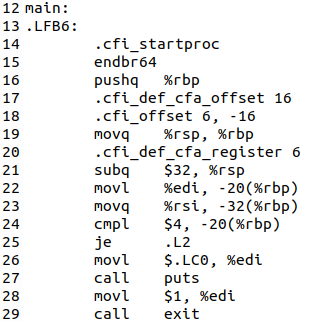


图3.5 hello.s中main的部分内容

<2>局部变量

如图3.3所示，在源程序hello.c中包含一个int类型的局部变量i，和传入的参数int argc、char \* argv[]。

由图3.5的22、23行可知，在hello.s中，传入的参数argc被放在寄存器%edi中和-20（%rbp）中，而传入的参数argv被放在寄存器%rsi中和-32（%rbp）中。由3.6图第31行可知，局部变量i，作为循环的计数器被放在-4（%rbp）中，初值为0，当它小于等于7时，函数执行循环部分，即标号.L4处的汇编指令。

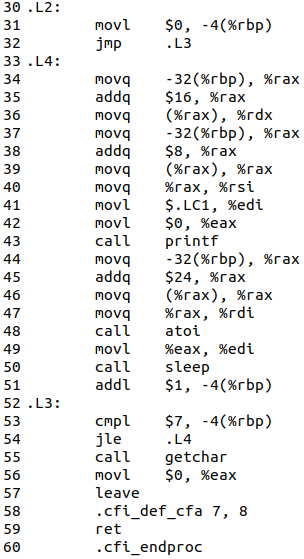


图3.6 hello.s中main部分内容

### 3.3.2赋值

如图3.6的第31行，这条语句使用了movl指令，对局部变量i赋了初值0，即图3.3中for循环中的i=0编译后的结果。

### 3.3.3算术操作

如图3.6的第51行，这条语句使用了addl指令，将立即数$1加到-4（%rbp）中，即图3.3中for循环中的i++操作编译后的结果。

### 3.3.4关系操作

<1>!=操作

如图3.7所示，将立即数$4与-20（%rbp）中的内容进行比较，若相等，则跳转到标号.L2处，即图3.3中if语句的判断条件argc!=4编译后的结果。



图3.7 hello.s中的!=

<2> <操作

如图3.8所示，将立即数$7与-4（%rbp）中的内容进行比较，若小于等于，则跳转到标号.L4处，.L4就是for循环的内容，故这就是图3.3中for循环条件判断语句i<8编译后的结果。



图3.8 hello.s中的<

### 3.3.5数组/指针/结构体操作

<1>argv[1]和argv[2]

如图3.9所示，-32（%rbp）放的是类型为char\*数组argv的首地址。当其调用printf前，要进行传参时，首先将首地址移到%rax中，再对其addq $16，即取出了&agrv[2]，再用（%rax）即（&argv[2]）=argv[2]赋值到%rax作为printf的第三个参数；argv[1]同理，不过这时候是addq $8，也赋值到%rsi中，作为printf的第二个参数。

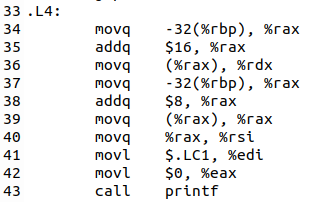


图3.9 hello.s中的数组操作1

<2>argv[3]

如图3.10所示，要向atoi中传入参数argv[3]，同上理，不过这时候是addq $24，赋值到%rax中，作为atoi的第一个参数。

以上就是hello.c中数组操作编译后的结果。

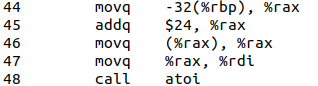


图3.10 hello.s中的数组操作2

### 3.3.6控制转移

如图3.3所示，共有两处控制转移：1.if(argc!=4) 2. for(i=0;i<8;i++)。

<1>if(argc!=4)编译后的结果

如图3.11所示，-20（%rbp）里面放的是argc的值，cmpl指令和je指令执行操作：若argc等于4，则由jmp指令跳转到标号.L2处，否则继续执行下一条指令。

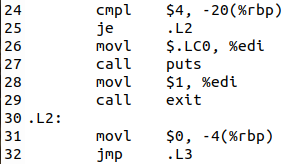


图3.11 hello.s中的控制转移1

<2> for(i=0;i<8;i++)编译后的结果

如图3.12所示，for循环i=0，在.L2处初始化，之后由jmp指令跳转至.L3，与7进行比较，cmpl指令和jle指令执行操作：若i小于等于7，则跳转至标号.L4处，即循环体；否则继续执行下一条指令，即循环结束后的第一条语句。

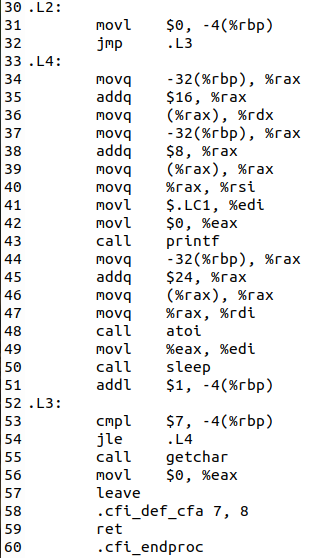


图3.12 hello.s中的控制转移2

### 3.3.7函数操作

<1>printf函数

如图3.3所示，print函数在main中出现过两次。

第一次调用如图3.13所示，调用前先将参数$LC0传入到%edi中作为第一个参数，然后调用printf函数，返回值被存放在寄存器%rax中。

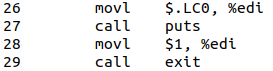


图3.13 hello.s中的printf函数1

第二次调用如图3.14所示，调用前先将参数argv[2]传入%rdx中作为第三个参数，argv[1]传入%rsi中作为第二个参数，$.LC1即前文提到的格式参数，传入到%edi中作为第一个参数，然后调用printf函数，返回值被存放在寄存器%rax中。

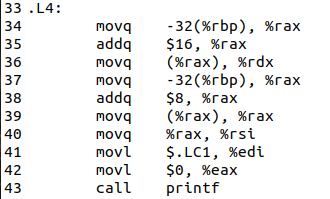


图3.14 hello.s中的printf函数2

<2>exit函数

如图3.15所示，调用exit(1)前，先将立即数$1传入%edi中作为第一个参数，然后调用exit函数。



图3.15 hello.s中exit函数

<3>atoi函数和sleep函数

如图3.16所示，调用atoi（argv[3]）之前，先将argv[3]传入%edi中作为第一个参数，然后调用atoi函数，返回值被存放在寄存器%rax中。然后将atoi的返回值传给%edi作为sleep的第一个参数，然后调用sleep函数，返回值被存放在寄存器%rax中。

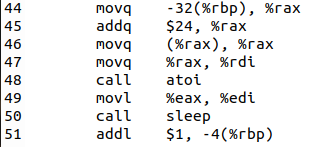


图3.16 hello.s中的atoi和sleep函数

<4>getchar函数

如图3.17所示，该函数不需要传递参数，所以直接使用call指令调用getchar函数，返回值被存放在寄存器%rax中。



图3.17 hello.s中getchar函数

## 3.4 本章小结

本章介绍了编译的概念与作用，以及hello.i文件转化为hello.s文件的过程。同时还对其中的汇编代码中C语言的数据和操作进行了解析。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

概念：

汇编器(as)将hello.s文件翻译成二进制机器语言指令，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，并将结果保存到目标文件hello.o中。hello.o是一个二进制文件，包含着程序的指令编码，如果用文本编辑器打开，将看到一堆乱码。

作用：

汇编器将汇编代码转变成计算机可以理解和执行的指令，生成目标文件。

## 4.2 在Ubuntu下汇编的命令

指令：gcc -m64 -no-pie -fno-PIC -c hello.s -o hello.o

截图：

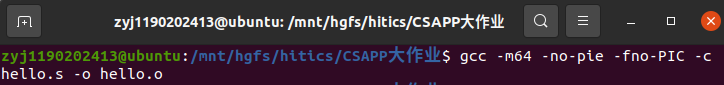


图4.1 Ubuntu下汇编命令

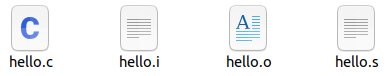


图4.2 汇编生成hello.o



图4.3 用vim打开hello.o显示乱码

## 4.3 可重定位目标elf格式

指令：readelf -a hello.o

截图：



图4.4 readelf指令查看hello.o

分析hello.o的ELF格式，用readelf等列出其各节的基本信息，特别是重定位项目分析。

### 4.3.1ELF头

ELF信息如图4.5所示，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。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（如可重定位、可执行或者共享的）、机器类型（如x86-64）、节头部表的文件偏移，节头部表中条目的大小和数量。

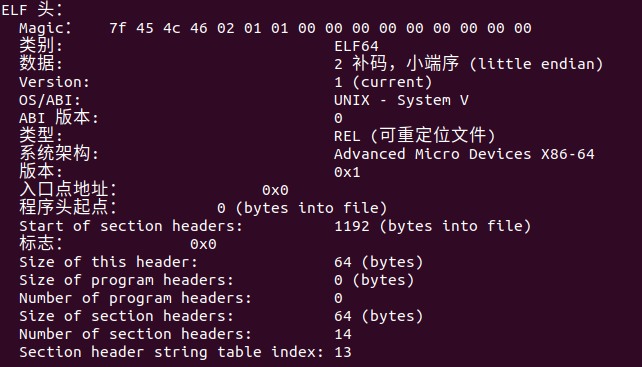


图4.5 ELF头信息

### 4.3.2节头部表

节头部表的信息如图4.6所示，不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定的条目。夹在ELF头和节头部表之间的都是节：

1.ELF头：描述文件的总体格式

2..text节：已编译程序的机器代码。

3..rodata节：只读数据，比如printf语句中的格式串和开关语句的跳转表

4..data节：已初始化的全局和静态C变量。局部C变量在运行时被保存在栈中，既不出现在.data节中，也不出现在.bss节中。

5..bss节：未初始化的全局和静态C变量。

6..symtab节：一个符号表，它存放在程序中定义和引用的函数和全局变量的信息。

7..rel.text节：一个.text节中的列表，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。

8..rel.data节：被模块引用或定义的所有全局变量的重定位信息。

9..debug节：一个调试符号表，其条目是程序中定义的全局变量和类型定义，程序中定义和引用的全局变量，以及原始的C源文件。

10..line节：原始C源程序的行号和.text节中机器指令之间的映射

11..strtab节：一个字符串表，其内容包括.symtab和.debug节中的符号表，以及节头部中的节名字。

12.节头部表：描述目标文件的节。

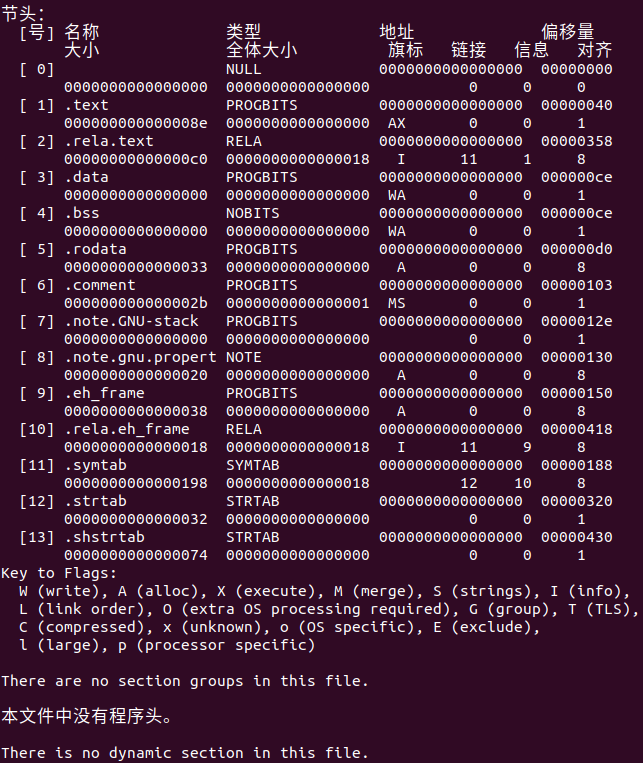


图4.6 节头部表信息

### 4.3.3重定位节

当目标文件进行链接时，重定位节内的信息告诉链接器在将目标文件合并成可执行目标文件时如何修改其节内容的信息。通过此信息，可执行文件和共享目标文件可包含进程的程序映像的正确信息。 代码的重定位条目放在.rel.text 中，已初始化数据的重定位条目放在.rel.data 中。

如图4.7所示，重定位节中包含.rodata、puts、exit、printf、sleep、getchar等符号的偏移的相关信息。在链接时，链接器会依据这些信息对目标文件进行重定位，最终得到一个可执行文件。

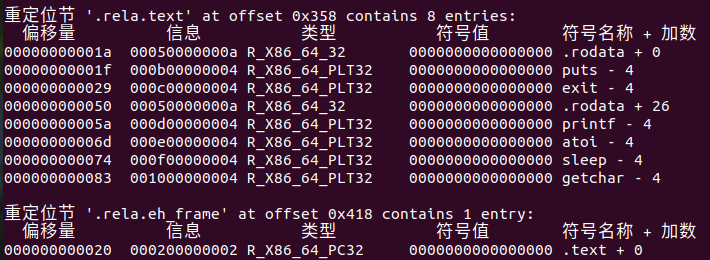


图4.7 重定位节信息

### 4.3.5符号表

符号表时由汇编器构造的，使用编译器输出到汇编语言.s文件中的符号。这张符号表包含一个条目的数组，放在.symtab节中，其信息如图4.8所示。Name是字符串表中的字节的偏移，指向符号的以null结尾的字符串名字。Value是符号的地址，对于可重定位的模块来说，value是距定义目标的节的起始位置的偏移。对于可执行目标文件来说，该值是一个绝对运行时地址。Size是目标的大小（以字节为单位）。Type通常要么是数据，要么是函数。符号表还可以包含各个节的条目，以及对应原始源文件的路径名的条目。Bind字段表示符号是本地的还是全局的。Ndx为ABS代表不该被重定位的符号；ndx为UNDEF代表未定义的符号，也就是在本目标模块中引用，但是却在其他地方定义的符号；ndx=数字，例如ndx=1，表示.text节，而ndx=3，表示.data节。

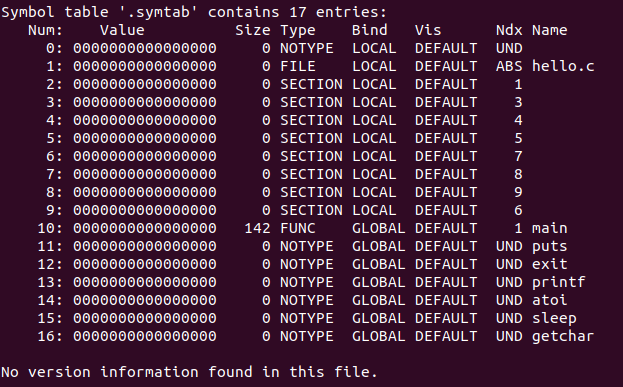


图4.8 符号表信息

## 4.4 Hello.o的结果解析

objdump -d -r hello.o 分析hello.o的反汇编，并请与第3章的 hello.s进行对照分析。

指令：objdump -d -r hello.o

指令截图：



图4.9 Ubuntu下反汇编指令

内容截图：

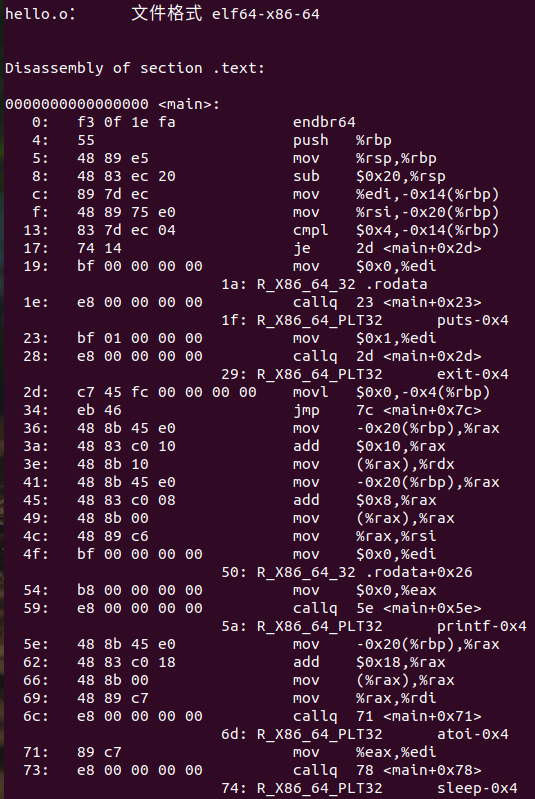


图4.10 反汇编内容1

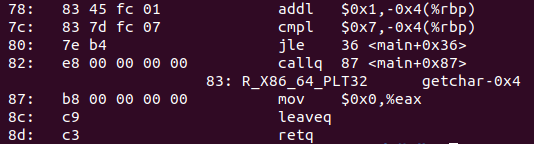


图4.11 反汇编内容2

对比：

<1>首先，如图4.10和4.11所示，反汇编内容有对应的机器指令，而hello.s文件中没有。

而且反汇编器使用的指令命名规则与GCC生成的汇编代码使用的有些细微的差别：反汇编代码中省略了汇编代码中很多指令结尾的“q”等后缀，例如movq、pushq等。相反，反汇编代码中部分指令例如call、leave和ret指令添加了“q”后缀，而这些指令的后缀在汇编代码中是省略的。

<2>如图4.12和4.13所示，hello.s中数字以十进制表示，而反汇编文件中以十六进制表示。



图4.12 hello.s的十进制表示



图4.13 反汇编的十六进制表示

<3>如图4.14和4.15所示，hello.s的jmp指令借助一个标号.L3来跳转至指定的位置，这种表示对程序员阅读来说更友好。而反汇编文件中的jmp指令采用相对地址偏移的跳转方式。



图4.14 hello.s的跳转



图4.15 反汇编的跳转

<4>重定位条目符号的使用上的不同。如图4.16和4.17所示，hello.s在调用atoi函数时，之间使用call atoi进行调用。而反汇编文件中，使用call 加上相对地址偏移的方式进行调用，即把atoi的重定位信息写入到机器指令中。



图4.16 hello.s重定位条目符号的使用



图4.17 反汇编的重定位条目符号的使用

由此可见，对于操作数，反汇编文件中通常以十六进制的形式表示，而.s文件中通常以十进制的形式表示。对于分支转移函数调用等，反汇编文件大部分采取相对寻址，通常为下一条指令的地址加偏移量，计算得绝对地址。

## 4.5 本章小结

本章介绍了汇编的概念与作用，用readelf查看了hello.o中ELF头、节头部表等信息，使用objump对hello.o文件进行反汇编，与hello.s的内容进行比对，并分析机器语言和汇编语言之间的映射关系。

**（第4章1分）**

# 第5章 链接

## 5.1 链接的概念与作用

概念：

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，这个文件可以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。在早期的计算机系统中，链接是手动执行的。在现代系统中，链接是由叫做链接器的程序自动执行的。

作用：

链接在软件开发中扮演着一个关键的角色，因为它们使得分离编译成为可能。链接能将各种代码和数据片段收集并组合成为一个单一文件，即链接多个目标文件成一个可执行文件。

## 5.2 在Ubuntu下链接的命令

按照实验报告眼球使用如下指令：

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

截图：

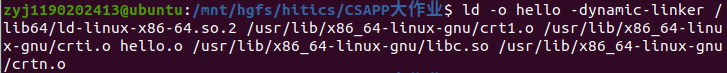


图5.1 Ubuntu下链接指令

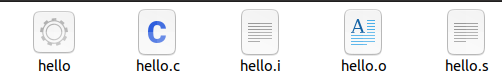


图5.2 链接后生成hello

## 5.3 可执行目标文件hello的格式

指令：readelf -a hello

截图：



图5.3 使用readelf查看hello

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。

### 5.3.1ELF头

ELF信息如图5.4所示，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。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（如可重定位、可执行或者共享的）、机器类型（如x86-64）、节头部表的文件偏移，节头部表中条目的大小和数量。



图5.4 ELF头信息

### 5.3.2节头部表

节头部表的信息如图5.5和5.6所示，不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定的条目。夹在ELF头和节头部表之间的都是节：

1.ELF头：描述文件的总体格式

2.段头部表：将连续的文件映射到运行时的内存段。

3..init节：定义了一个\_init函数，程序的初始化代码会调用它。

4..text节：已编译程序的机器代码。

5..rodata节：只读数据，比如printf语句中的格式串和开关语句的跳转表

6..data节：已初始化的全局和静态C变量。局部C变量在运行时被保存在栈中，既不出现在.data节中，也不出现在.bss节中。

7..bss节：未初始化的全局和静态C变量，以及所有被初始化为0的全局或静态变量。

8..symtab节：一个符号表，它存放在程序中定义和引用的函数和全局变量的信息。

9..debug节：一个调试符号表，其条目是程序中定义的全局变量和类型定义，程序中定义和引用的全局变量，以及原始的C源文件。

10..line节：原始C源程序的行号和.text节中机器指令之间的映射

11..strtab节：一个字符串表，其内容包括.symtab和.debug节中的符号表，以及节头部中的节名字。

12.节头部表：描述目标文件的节。

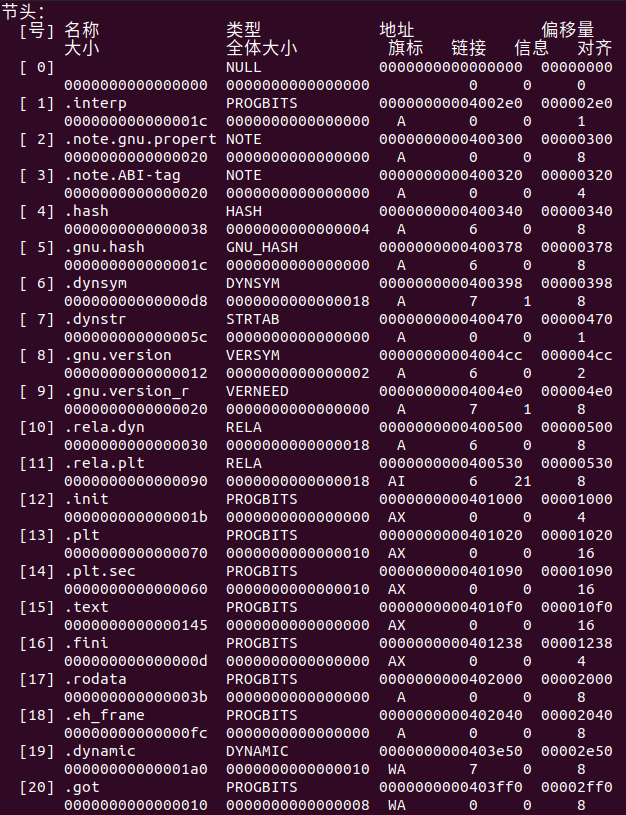


图5.5 节头部表信息1

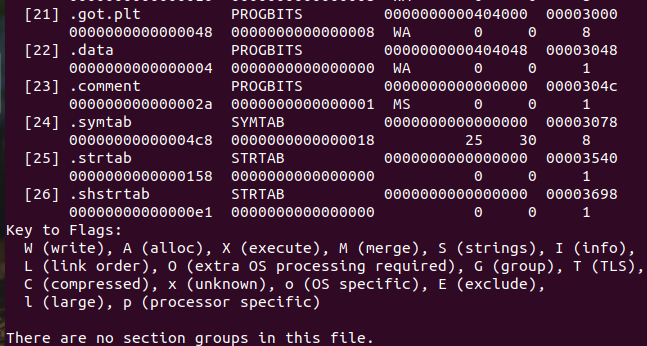


图5.6 节头部表信息2

### 5.3.3程序头表

ELF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。如图5.7所示，程序头表包含目标文件中的段大小、内存中的段大小、段的读/写访问权限、开始于内存的哪个位置、偏移量、对齐要求等信息。



图5.7 程序头表信息

### 5.3.4Section to Segment mapping

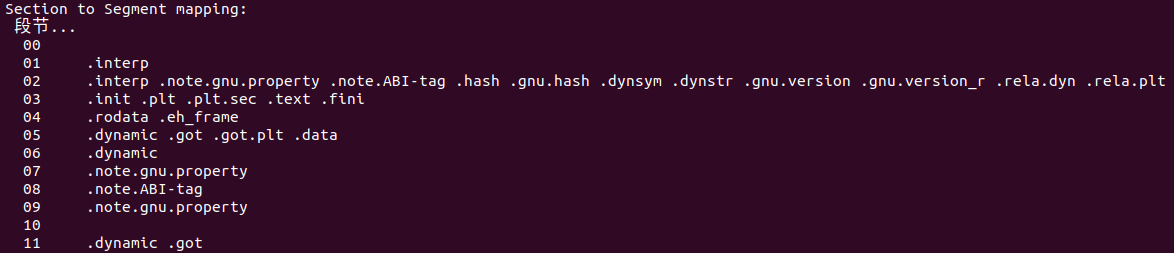


图5.8 Section to Segment mapping信息

### 5.3.5Dynamic section

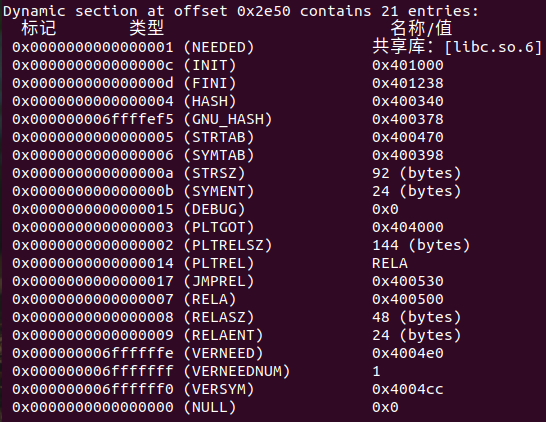


图5.9 Dynamic section信息

### 5.3.6重定位节

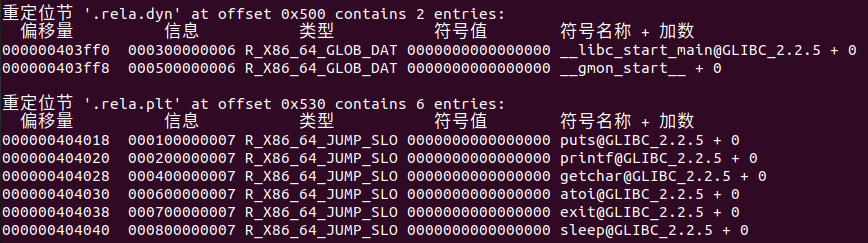


图5.10 重定位节信息

### 5.3.7符号表

符号表时由汇编器构造的，使用编译器输出到汇编语言.s文件中的符号。这张符号表包含一个条目的数组，放在.symtab节中，其信息如图5.11、5.12和5.13所示。

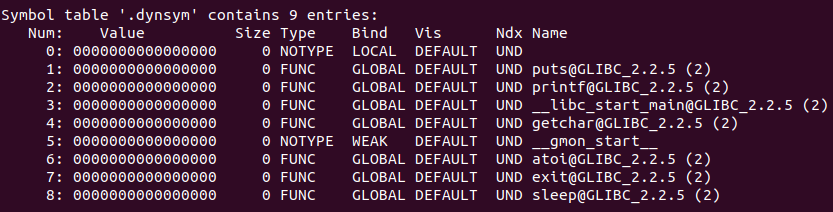


图5.11 符号表信息1

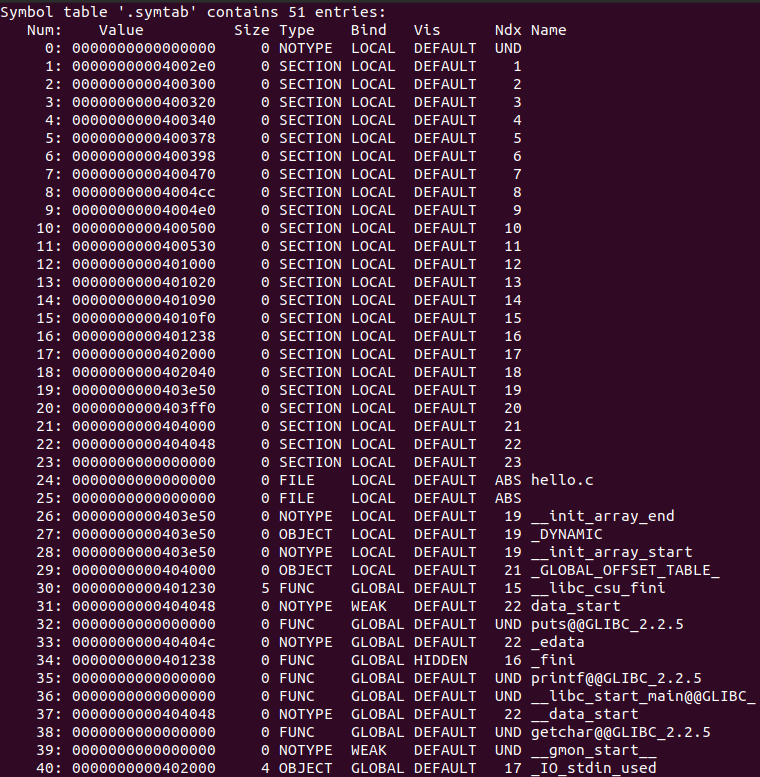


图5.12 符号表信息2

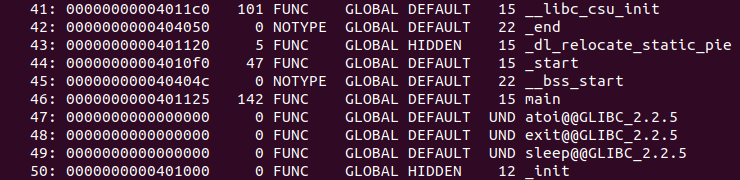


图5.13 符号表信息3

### 5.3.8版本信息



图5.14 版本信息

## 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。

由图5.15可知，hello的虚拟地址空间起始与0x400000处。

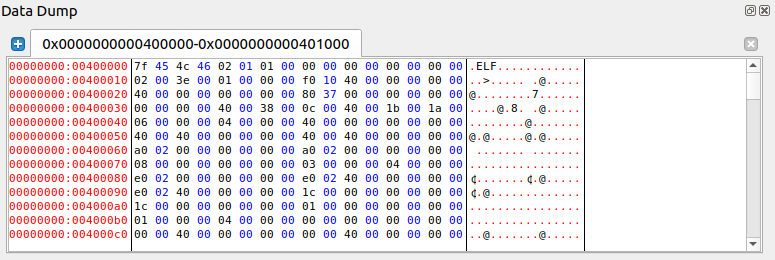


图5.15 edb查看hello的虚拟地址空间

使用edb点击Plugins->SymbolViewer,查看本进程的虚拟地址空间各段信息如图5.16和5.17所示，其信息与5.3中的信息一致。

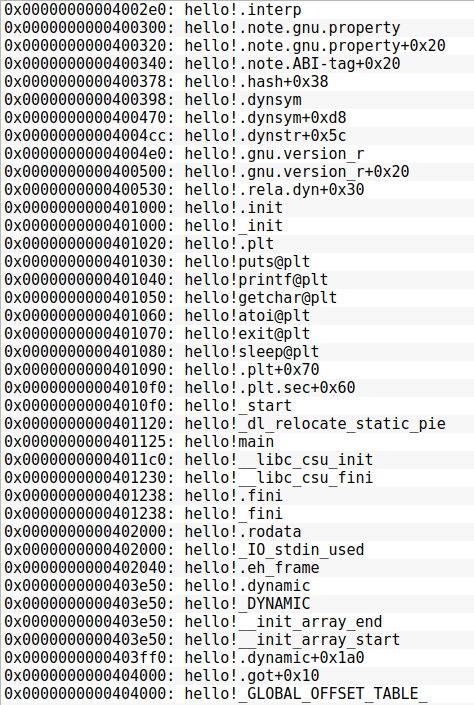


图5.16 虚拟地址空间各段信息1

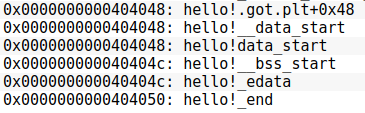


图5.17 虚拟空间各段信息2

## 5.5 链接的重定位过程分析

objdump -d -r hello 分析hello与hello.o的不同，说明链接的过程。

指令：objdump -d -r hello

截图：



图5.18 objdump分析hello的指令

不同：

<1>如图5.19所示，hello的反汇编文件代码的虚拟地址空间是从0x400000开始的，而由图4.10可知，hello.o的反汇编文件代码的虚拟地址空间是从0x0开始的。左边列出的地址不同，且链接器填上了跳转指令例如je等调用函数需要使用的地址。



图5.19 hello的反汇编1

<2>如图5.20所示，hello的反汇编代码中加入了printf等函数的反汇编代码，这是因为链接过程中执行重定位后加入的，而这是在hello.o的反汇编代码中所没有的。

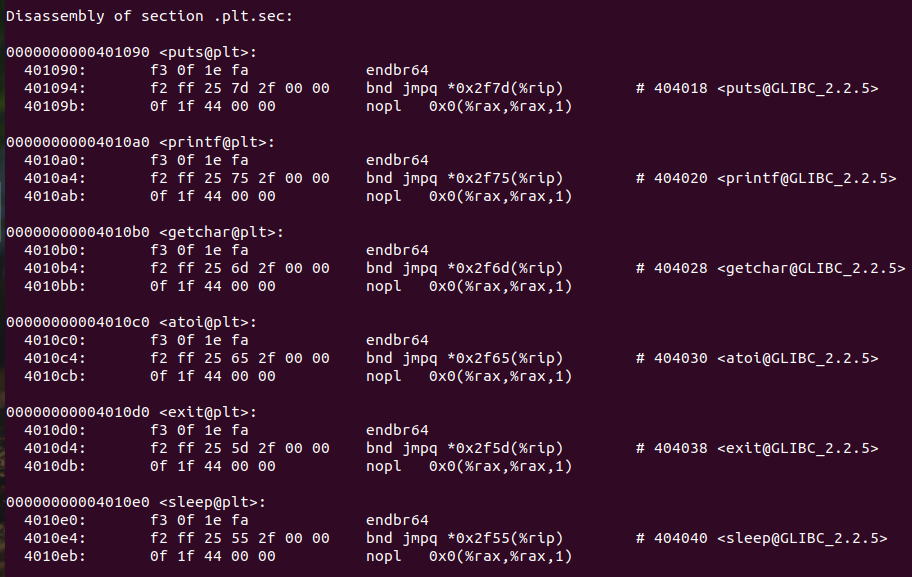


图5.20 hello的反汇编2

hello.o中的重定位节信息如图5.21所示。

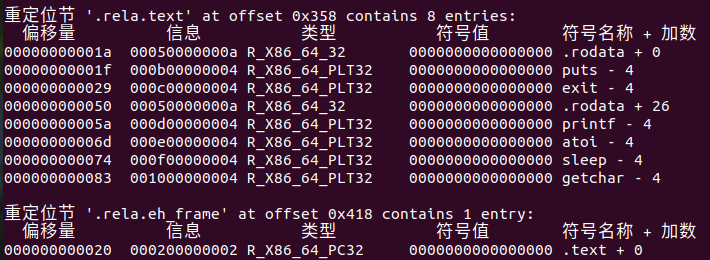


图5.21 hello.o的重定位节信息

hello中如何重定位：

链接器完成了符号解析这一步后，就把代码中的每个符号引用和正好一个符号定义关联起来。接下来就开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时地址。重定位由两步组成：

1. 重定位节和符号定义。在这一步中，链接器将所有相同类型的节合并为同一类型的聚合节。例如，所有输入模块的.data节被全部合并成hello的.data节。然后，链接器将运行时内存地址赋给新的聚合节，赋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。
2. 重定位节中的符号引用。在这一步中，链接器修改代码节和数据节中对每个符号的引用，使得它们指向正确的运行时地址。要执行这一步，链接器依赖于可重定位目标模块中称为重定位条目的数据结构。由于hello.o中的重定位条目较多，这里就举一个例子。例如atoi函数，如图5.22所示，调用atoi函数时，机器指令e8后面并没有跟上一个正确的地址，而是用一个重定位条目来告诉链接器这个函数在链接时需要进行重定位。如图5.23所示，重定位后链接器使atoi这个符号指向了0x4010c0这个地址，并且修改了代码节中的代码:e8 2a ff ff ff，如此完成了重定位。

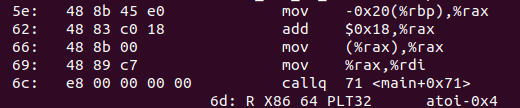


图5.22 hello.o中atoi函数的调用



图5.23 hello中atoi函数的重定位

## 5.6 hello的执行流程

|  |  |
| --- | --- |
| 程序名 | 程序地址 |
| \_start | 0x00000000004010f0 |
| \_\_libc\_csu\_init | 0x00000000004011c0 |
| \_init | 0x0000000000401000 |
| main | 0x0000000000401125 |
| printf@plt | 0x00000000004010a0 |
| atoi@plt | 0x00000000004010c0 |
| sleep@plt | 0x00000000004010e0 |
| getchar@plt | 0x00000000004010b0 |
| \_fini | 0x0000000000401238 |

## 5.7 Hello的动态链接分析

编译器通过运用以下这个有趣的事实来生成对全局变量的PIC引用：无论我们在内存中的何处加载一个目标模块（包括共享目标模块），数据段与代码段的距离总是保持不变。因此，代码段中的任何指令和数据段中的任何变量之间的距离都是一个运行时常量，与代码段和数据段的绝对内存位置是无关的。

想要生成对全局变量PIC引用的编译器利用了这个事实，它在数据段开始的地方创建了一个全局偏移量表GOT。在加载时，动态链接器会重定位GOT中的每个条目，使得它包含目标的正确的绝对地址。每个引用全局目标的目标模块都有自己的GOT。

GNU编译系统使用了一种很有趣的技术--延迟绑定来解决动态库函数模块调用的问题，将过程地址的绑定推迟到第一次调用该过程时。

延迟绑定通过全局偏移量表（GOT）和过程链接表（PLT）实现。如果一个目标模块调用定义在共享库中的任何函数，那么就有自己的GOT和PLT。前者是数据段的一部分，后者是代码段的一部分。

通过在dl\_init前后设置断点，我们可以看到调用前后动态链接项目的变化。

在hello程序调用dl\_init前，如图5.24使用EDB进行调试发现，全局偏移量表的地址为0x601000。

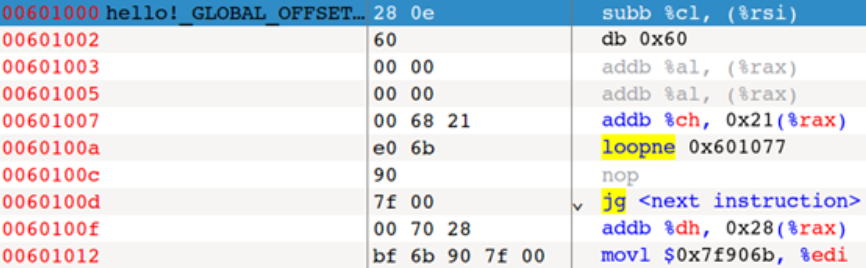


图5.24 dl\_init调用前的GOT

在hello程序调用dl\_init后，如图5.25使用EDB进行调试发现，全局偏移量表的地址为0x7f906bbf2870，而这个地址技术动态链接器的入口点。

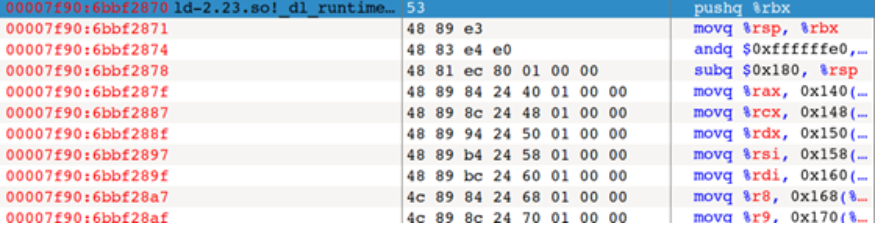


图5.25 dl\_init调用后的GOT

## 5.8 本章小结

本章介绍了链接的概念与作用，对hello.o文件进行了链接，分析了hello文件的格式信息、虚拟地址信息和重定位的过程，对比了hello和hello.o反汇编结果的不同，并用edb查看了hello的执行流程对其动态链接进行了分析。

**（第5章1分）**

# 第6章 hello进程管理

## 6.1 进程的概念与作用

概念：

进程的经典定义是一个执行中程序的实例，系统的每个程序都运行在某个进程的上下文（context）中。上下文是由程序正确运行所需的状态组成的，这个状态包括存放在内存里的程序的代码和数据，它的栈、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。

作用：

每次运行程序时，shell 创建一新进程，在这个进程的上下文切换中运行这个 可执行目标文件。应用程序也能够创建新进程，并且在新进程的上下文中运行它 们自己的代码或其他应用程序。

进程还提供给应用程序关键的抽象：

1.一个独立的逻辑控制流，它提供一个假象，好像我们的程序独占地使用处理器。

2.一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统。

## 6.2 简述壳Shell-bash的作用与处理流程

作用：

Shell是一个交互型的应用级程序，它是系统的用户界面，提供了用户与内核进行交互操作的一种接口，它代表用户运行其他程序。Shell是一个命令行解释器，它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、停止甚至时编写一些程序。Shell还是一个功能相当强大的编程语言，易编写，易调试，灵活性较强。Shell是解释执行的脚本语言，在Shell中可以直接调用Linux系统命令。

Bash是一个满足POSIX规范的shell，是一个命令处理器，通常运行于文本窗口中，并能执行用户直接输入的命令。

处理流程：

Shell是一个命令行解释器，它输出一个提示符，等待输入一个命令行，然后执行这个命令。用户输入命令行后，Shell-bash首先会调用相关函数，该函数会解析以空格分隔的命令行参数。在解析了命令行之后，第一个参数被假设为要么是一个内置的shell命令名，马上就会解释这个命令；要么是一个可执行目标文件，会在一个新的子进程的上下文中加载并运行这个文件。如果该程序需要在前台运行，则shell会等待直到前台运行结束。如果输入的命令行的第一个参数不是一个内置的shell命令名且也没在路径里找到该可执行文件，shell会输出一条错误提示信息。Shell随后输出一个提示符，等待下一个输入的命令行。

## 6.3 Hello的fork进程创建过程

创建过程：

首先我们输入命令行./hello 1190202413 庄雨杰 1。Shell会调用相关函数解析该命令行，由于我们输入的不是一个内置命令，shell会认为第一个参数是一个可执行文件并在相关路径下寻找，由于我们输入的是./hello，所以shell在当前目录下找到了hello文件。此时shell作为父进程会调用fork函数创建一个新的运行的子进程，该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，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本。但子进程会获得一个与父进程不同的PID。

## 6.4 Hello的execve过程

过程：

execve函数在当前进程的上下文中加载并运行一个新进程，这个当前进程即上文fork的子进程。

execve函数加载并运行可执行文件hello，且带参数列表argv（这里的参数列表就是hello、1190202413、庄雨杰、1）和环境变量envp。只有当出现错误时，例如找不到文件名，execve才会返回到调用程序。

至于execve是如何加载hello的，它会调用某个驻留在存储器中称为加载器的操作系统代码来运行它。当加载器运行时，它创建类似于图6.1所示的内存映像。在程序头部表的引导下，加载器将可执行文件的片复制到代码段和数据段。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，该函数定义在libc.so中。它初始化执行环境，调用用户层的main函数，处理main函数的返回值，并且在需要的时候把控制返回给内核。

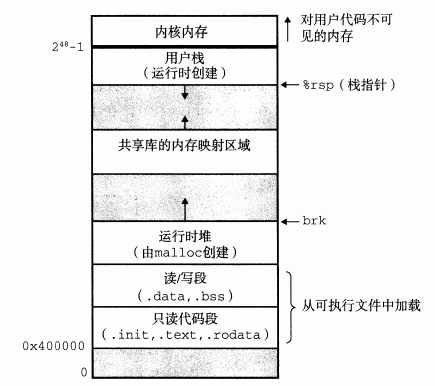


图6.1 Linux x86-64运行时内存映像

## 6.5 Hello的进程执行

进程上下文信息：

内核位每个进程维持一个上下文。上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值组成，这些对象包括通用目的寄存器、浮点器、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。

进程时间片：

多个流并发地执行的一般现象被称为并发。一个进程和其他进轮流运行的概念称为多任务。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。因此，多任务也叫做时间分片。

Hello的进程执行：

在执行过程中，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，这个决策称为调度，是由内核中称为调度器的代码处理的。如图6.2所示，我们不妨借用书上的图来描述hello的进程执行，将进程A看成运行hello的进程，进程B看成某一个即将抢占进程A的进程。一开始内核选择了进程A来运行hello程序，随后由于hello程序中调用了sleep函数，此时由用户模式进入了内核模式，发生了上下文切换，内核让进程A休眠，定时器开始计时并切换到进程B，回到用户模式。随后进程B执行一段时间后，定时器计时结束发送了一个中断的信号，此时又由用户模式进入了内核模式，发生了上下文切换，重新开始了原先被抢占的进程A，回到用户模式，从中断处继续执行。在hello调用getchar函数的时候也是如此，只是此时内核等待的是一个来自输入流的信号。

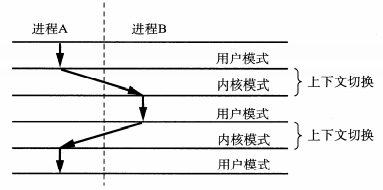


图6.2 进程上下文切换

## 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。

### 6.6.1不停乱按，包括回车

如图6.3所示，不停乱按，包括回车的操作只是将输入的字符放在输入缓冲区中，等到hello执行完for循环运行到getchar函数时，从缓冲区中读入了一个字符，程序运行结束。

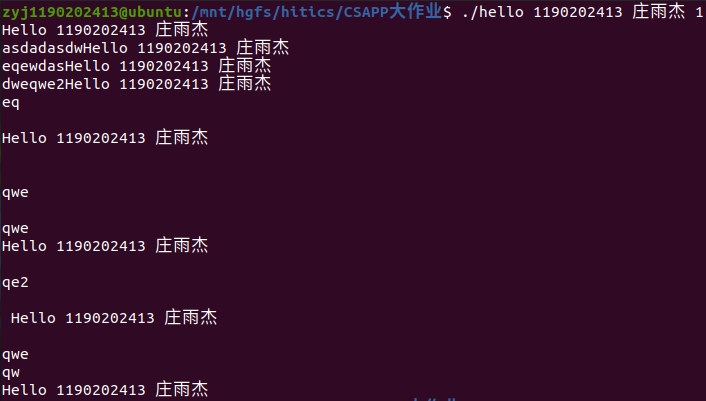


图6.3 不停乱按，包括回车的结果

### 6.6.2Ctrl-z

如图6.4所示，当hello运行时按下ctrl-z，系统内核会发送一个SIGTSTP信号给前台进程组的每一个进行，然后回调用相应的信号处理程序，最终会停止前台进程组的所有进程。

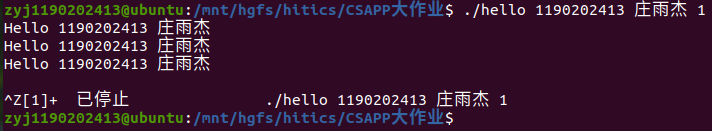


图6.4 输入ctrl-z的结果

### 6.6.3Ctrl-c

如图6.5所示，当hello运行时按下ctrl-c,系统内核会发送一个SIGINT信号给前台进程组的每一个进程，然后调用相应的信号处理程序，最终会终止前台进程组的所有进程。

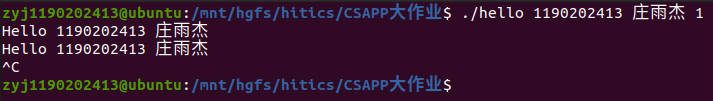


图6.5 输入ctrl-c的结果

### 6.6.4 Ctrl-z运行ps /jobs/ pstree/fg/kill命令

<1>ps命令

如图6.6所示，ps命令列出了所有的进程，我们可以看到hello进程还在。

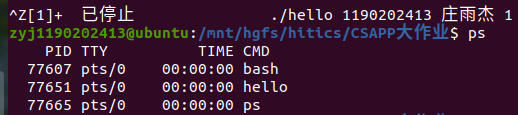


图6.6 ctrl-z后运行ps命令的结果

<2>jobs命令

如图6.7所示，jobs命令列出了所有的作业，而hello这个作业显示停止。

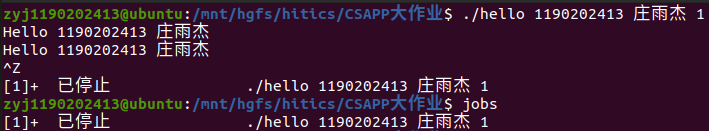


图6.7 ctrl-z后运行jobs命令的结果

<3>pstree命令

如题6.8所示，pstree命令以树状显示进程间的关系，由于内容太多，这里只截取一部分。

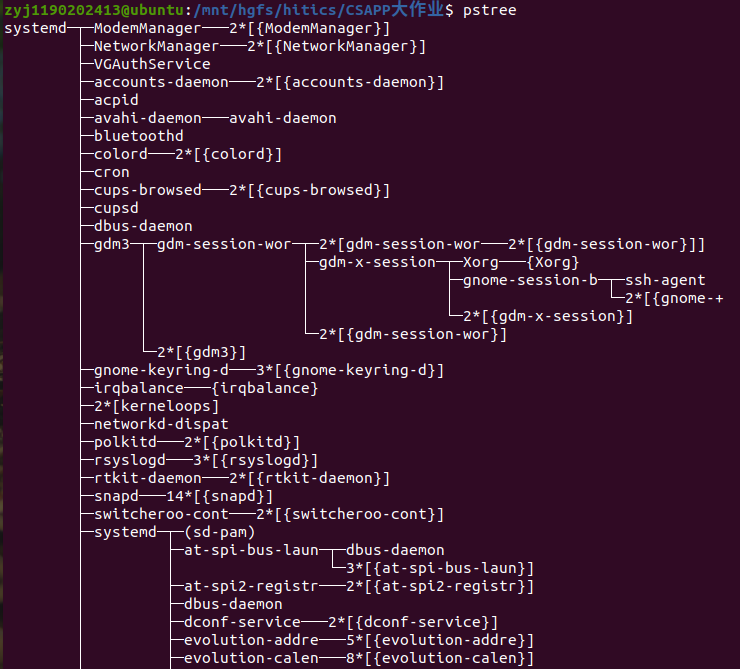


图6.8 ctrl-z后运行pstree命令的结果

<4>fg命令

如图6.9所示，hello在收到一个SIGINT信号后停止运行，转为后台作业，随后输入命令行fg，内核发送一个SIGCONT信号给后台作业hello，然后调用相应的信号处理程序，最终hello程序继续运行。



图6.9 ctrl-z后运行fg命令的结果

<5>kill命令

如图6.10所示，hello程序停止时，输入kill -9 77855，内核发送一个SIGKILL信号给进程hello，然后调用相应的信号处理程序，最终杀死了进程hello。用ps查看进程信息，显示进程hello已杀死。



图6.10 ctrl-z后运行kill命令的结果

## 6.7本章小结

本章介绍了进程的概念与作用，描述了shell-bash在用户与系统交互之间扮演怎样的角色，并详细地阐述了shell是如何fork得到一个子进程并execve来执行hello程序的以及hello在执行中的上下文切换与异常和信号的处理。

**（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：

逻辑地址是指由程序产生的与段相关的偏移地址部分。也叫做相对地址。

线性地址：

线性地址是[逻辑地址](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E5%9C%B0%E5%9D%80/3283849)到[物理地址](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%9C%B0%E5%9D%80/2129)变换之间的中间层。在分段部件中[逻辑地址](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E5%9C%B0%E5%9D%80/3283849)是段中的[偏移地址](https://baike.baidu.com/item/%E5%81%8F%E7%A7%BB%E5%9C%B0%E5%9D%80/3108819)，然后加上基地址就是线性地址。

虚拟地址：

虚拟地址是CPU运行在保护模式下，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。

物理地址：

在存储器里以字节为单位存储信息，为正确地存放或取得信息，每一个字节单元给以一个唯一的存储器地址，称为物理地址，也叫做绝对地址。

以图7.1为例，此处的0x16就是偏移地址，加上基地址才能得到0x401012。



图7.1 hello反汇编的偏移地址

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

段式管理，是指把一个程序分成若干个段进行存储，每个段都是一个逻辑实体，程序员需要知道并使用它。它的产生是与程序的模块化直接有关的。段式管理是通过段表进行的，它包括段号、段起点、装入位、段的长度等。此外还需要主存占用区域表、主存可用区域表。

如图7.2所示，段寄存器用于存放段选择符，CS指向程序代码所在段、SS指向栈区所在段等等。可以通过段选择符访问对于的全局/局部描述符表，得到对应的段描述符。段描述符中包含存取权限、段限、段基址信息。通过段基址和偏移地址（即逻辑地址）可以计算得线性地址，完成逻辑地址到线性地址的变换。

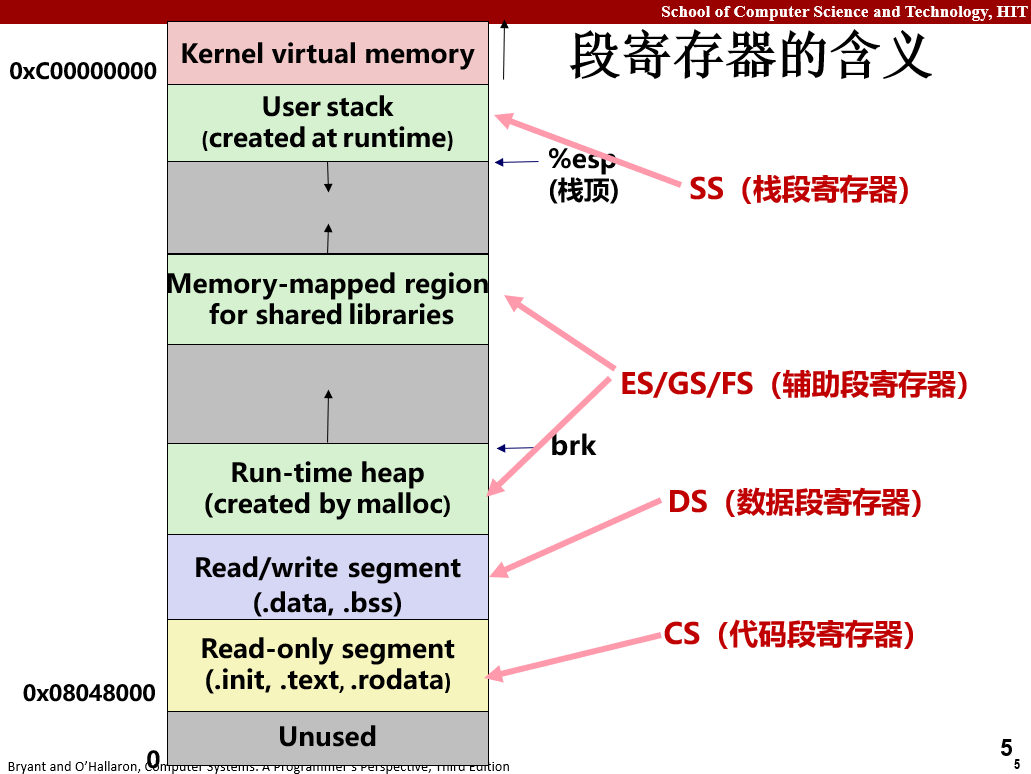


图7.2 段寄存器

## 7.3 Hello的线性地址到物理地址的变换-页式管理

线性地址也称为虚拟地址。

页式管理：

页式管理的基本原理将各进程的虚拟空间划分成若干个长度相等的页，页式管理把内存空间按页的大小划分成片或者页面，然后把页式虚拟地址与内存地址建立一一对应页表，并用相应的硬件地址变换机构，来解决离散地址变换问题。页式管理采用请求调页或预调页的技术实现了内外存存储器的统一管理。

虚拟页：

虚拟内存被组织位一个由存放在磁盘上的N个连续的字节大小的单元组成的数组。每字节都有一个唯一的虚拟地址，作为到数组的索引。VM系统通过将虚拟内存分割为称为虚拟页的大小固定块。

物理页：

与虚拟页类似，物理内存被分割为物理页。

页表：

页表存放在物理内存中，它将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表。操作系统负责维护页表的内容，以及在磁盘与DRAM之间来回传送页。

线性地址到物理地址的变换:

首先虚拟地址VA被分为两个部分：虚拟页号VPN和虚拟页偏移量VPO。CPU访问一个虚拟地址时，会先取出其VPN部分，MMU利用VPN来选择适当的页表条目PTE 。将页表条目中的物理页号和虚拟地址中的VPO串联起来，就得到相应的物理地址。以上就是线性地址到物理地址的变换。Q

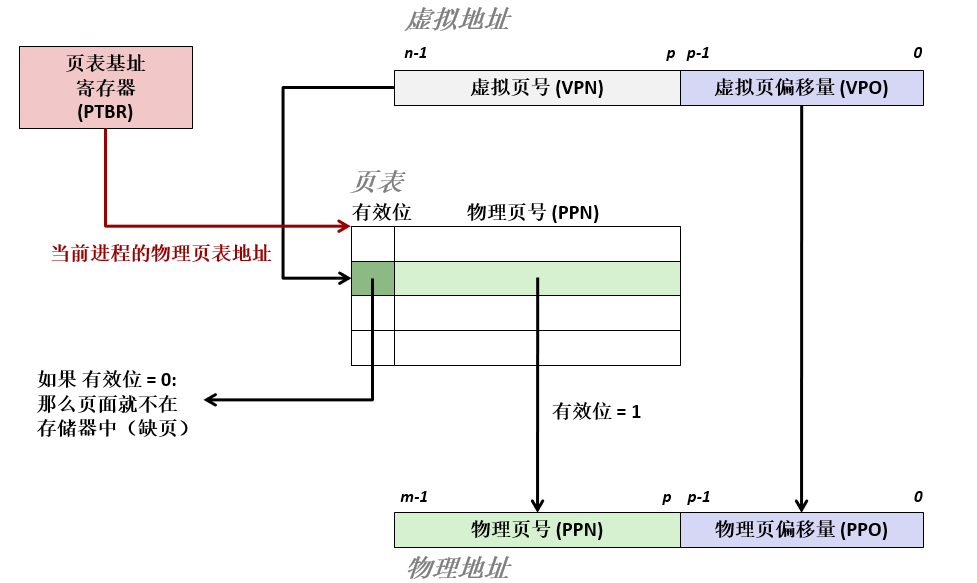


图7.3 基于页表的地址翻译

## 7.4 TLB与四级页表支持下的VA到PA的变换

TLB：

许多系统为了消除查阅PTE的开销，它们在MMU中包括了一个关于PTE的小的缓存，称为翻译后背缓冲器（TLB）。

TLB是一个小的、虚拟寻址的缓存，其中每一行都保存着一个由单个PTE组成的块。TLB通常有高度的相联度。如图7.4所示，用于组选择和行匹配的索引和标记字段是从虚拟地址中的虚拟页号中提取出来的。如果TLB有T=2t个组，那么TLB索引（TLBI）是由VPN的t个最低位组成的，而TLB标记（TLBT）是由VPN中剩余的位组成的。

CPU产生一个虚拟地址VA后，MMU根据TLBT和TLBI从TLB中取出相应的PTE，MMU将这个虚拟地址翻译成一个物理地址PA，并且将它发送到高速缓存/主存。高速缓存/主存将所请求的数据字返回给CPU。当TLB不命中时，MMU必须从L1缓存中取出相应的PTE，新取出的PTE存放在TLB中，可能会覆盖一个已经存在的条目。

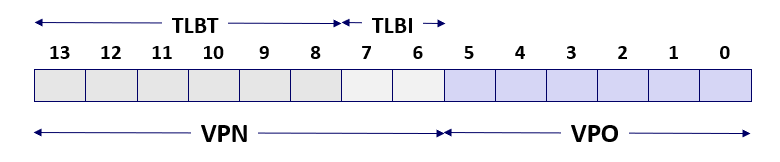


图7.4 虚拟地址中TLB的组成部分

多级页表：

单级页表过于浪费内存空间，因为对于一个典型的程序，4GB的虚拟地址空间的大部分都会是未分配的。故采用多级页表层次结构的地址翻译。如图7.5所示，虚拟地址VA被划分成为k个VPN和1个VPO。每个VPNi都是一个到第i+1级页表的所有，其中1<=i<=k。第j级页表中的每个PTE，1<=j<k-1，都指向第j+1级的某个页表基址。第k级页表中的每个PTE包含某个物理页表的PPN，或者一个磁盘块地址。为了构造物理地址，在能够确定PPN之前，MMU必须访问k个PTE。对于只有一级的页表结构，PPO和VPO是相同的。最后将得到的PPN和VPO串联起来就得到物理地址PA。

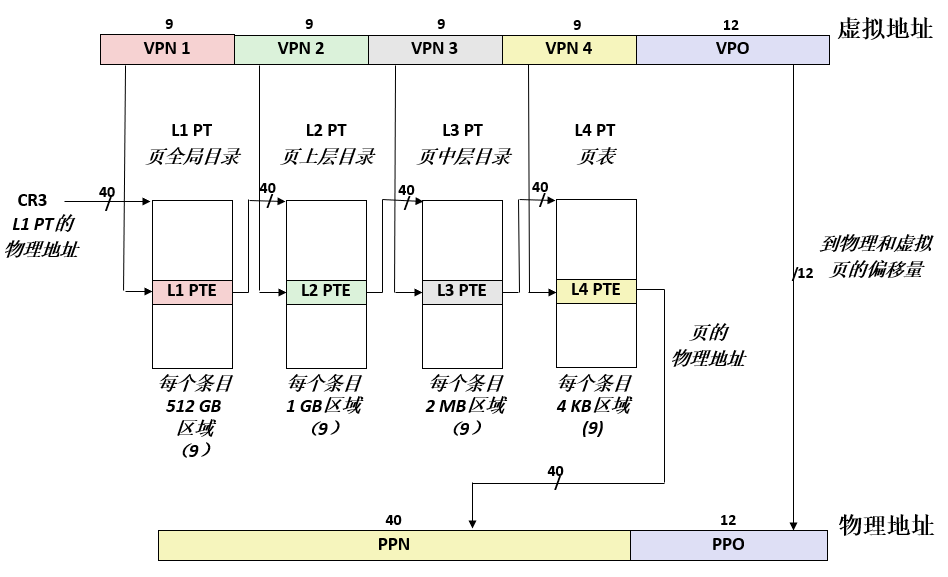


图7.5 使用k级页表的地址翻译

例如，Core i7采用四级页表层级结构，当一个Linux进程在运行时，虽然Core i7体系结构允许页表换进换出，但是与已分配了的页相关联的页表都是驻留在内存中的。CR3控制寄存器指向第一级页表（L1）的起始位置。如图7.5所示，VPN1提供到一个L1 PTE的偏移量，这个PTE包括L2页表的基地址。VPN2提供一个L2 PTE的偏移量，以此类推。

以上就是TLB与四级页表支持下的VA到PA的变换。

## 7.5 三级Cache支持下的物理内存访问

高速缓存Cache访问优先级从高到低为L1、L2、L3.当CPU访问一个虚拟地址VA时，首先MMU会根据TBLI和TBLT查找TLB中是否有相应的PTE。若命中，则直接返回给MMU。若不命中，则会依次访问L1、L2、L3，直到命中。否则，会访问主存中的多级页表，得到PTE返回给MMU。MMU将得到的PTE和剩下的VPO翻译成最后的物理地址PA，然后将PA发送到Cache。

如图7.6所示，PA从左到右被分为三个部分：标记（CT）、组索引（CI）和块偏移（CO）。Cache根据这三个信息进行查找。若一级CacheL1命中，则返回数据给CPU，否则就到二级CacheL2中查找，若还不命中，则到三级CacheL3中查找。若Cache都不命中才到主存中查找。

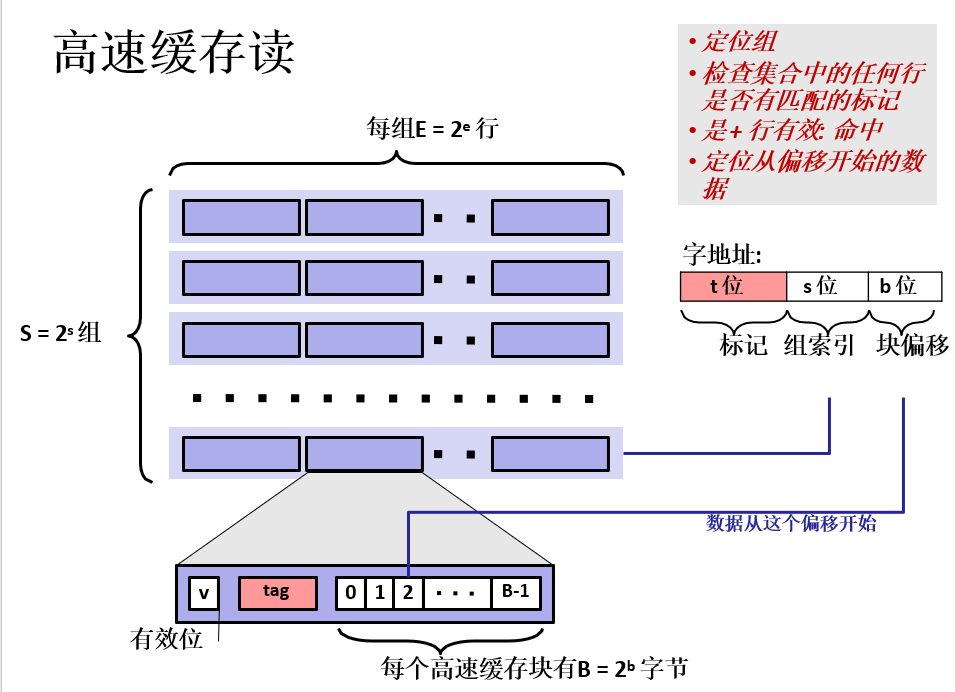


图7.6 PA的组成和访问

## 7.6 hello进程fork时的内存映射

如图7.6所示，不妨把进程1当成shell，进程2当成hello。当fork 函数被当前进程shell调用时，内核为新进程hello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hello创建虚拟内存，它创建了当前进程shell的mm\_struct、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

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

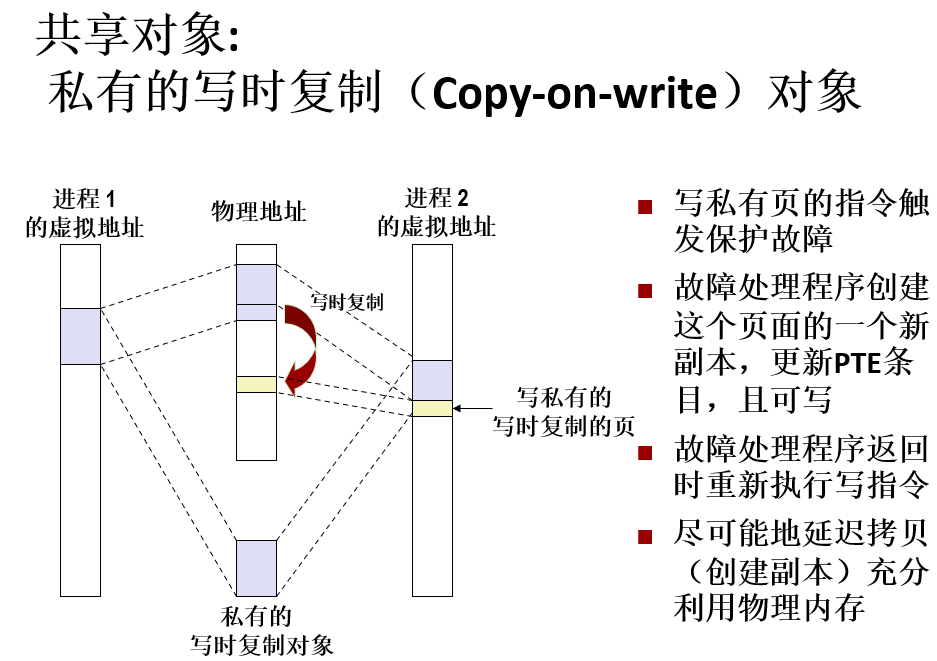


图7.7 hello进程fork时的内存映射

## 7.7 hello进程execve时的内存映射

execve 函数在当前进程hello中加载并运行包含在可执行目标文件hello中的程序，用hello程序有效地替代了当前程序。加载并运行hello需要以下几个步骤：

1.删除已存在的用户区域。删除当前进程shell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hello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。栈和堆区域也是请求二进制零的，初始长度为零。图7.8概括了私有区域的不同映射。

3.映射共享区域。如果hello程序与共享对象（或目标）链接，比如标准C 库libc. so，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。

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中的程序计数器，使之指向代码区域的入口点。

下一次调度hello进程时，它将从这个入口点开始执行。Linux将根据需要换入代码和数据页面。

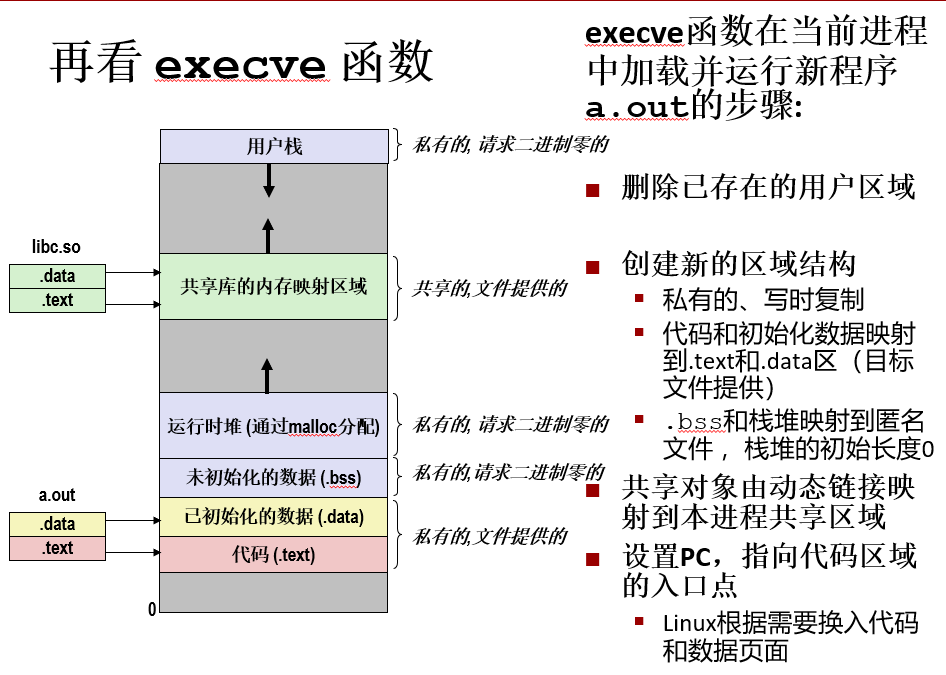


图7.8 hello进程execve时的内存映射

## 7.8 缺页故障与缺页中断处理

如图7.9所示，Linux将虚拟内存组织成段的集合。内核为每个进程维护一个单独的任务结构。任务结构中的元素包含或者指向内核运行该进程所需要的所有信息（例如，PID、指向用户栈的指针、可执行目标文件的名字、以及程序计数器）。这个任务结构的第一个条目指向mm\_struct，它描述了虚拟内存的当前状态。我们感兴趣的两个字段是pgd和mmap。其中pgd指向第一级页表（页全局目录）的基址，而mmap指向一个vm\_area\_structs（区域结构）的链表，其中每个vm\_area\_structs都描述了当前虚拟地址空间的一个区域。当内核运行这个进程时，就将pgd存放在CR3控制寄存器中。

假设MMU在试图翻译某个虚拟地址A时，触发了一个缺页。这个异常导致控制转移到内核的缺页处理程序，处理程序随后就执行下面的步骤：

1.虚拟地址A是否合法？缺页处理程序搜索区域结构的链表，把A和每个区域结构中的vm\_start和vm\_end做比较。如果指令是不合法的，那么缺页处理程序就触发一个段错误，从而终止这个进程。这个情况在图7.9中标识为“1”。

2.试图进行的内存访问是否合法？换句话说，进程是否有读写这个区域内页面的权限。例如，这个缺页是不是由一条试图对这个代码段里的只读页面进行写操作的存储指令造成的？这个缺页是不是因为一个运行在用户模式中的进程试图从内核虚拟内存中读取字造成的？如果试图进行的访问是不合法的，那么缺页处理程序会触发一个保护异常，从而终止这个进程。这种情况在图7.9种标识为“2”。

3.此刻，内核知道了这个缺页是由于对合法的虚拟地址进行合法的操作造成的。它是这样来处理这个缺页的：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，这条指令将再次发送A到MMU。这次，MMU就能正常地翻译A，而不会再产生缺页中断了。

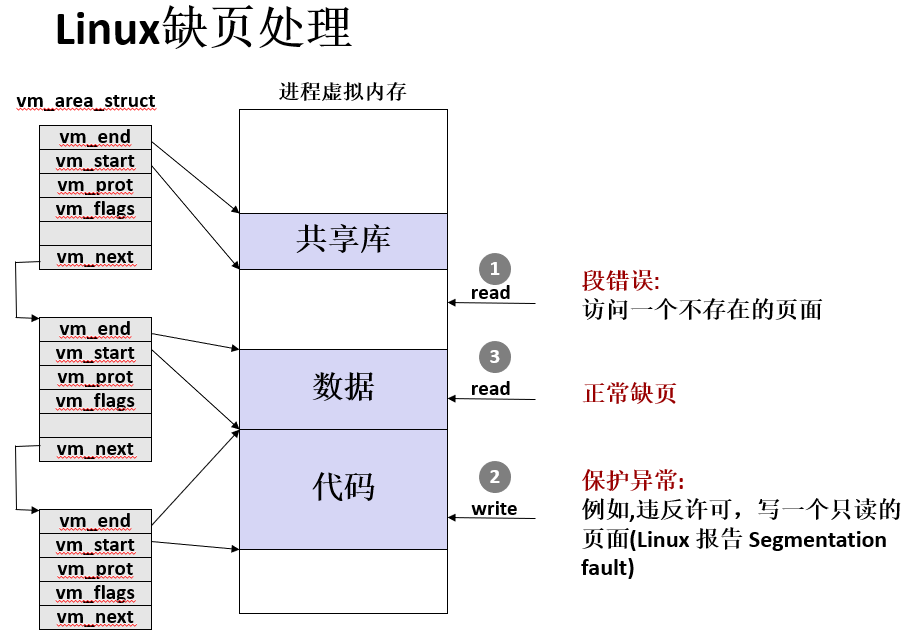


图7.9 缺页故障与缺页中断处理

## 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。系统之间细节不同，但是不失通用性，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（向更高的地址）。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。

分配器有两种基本风格。两种风格都要求应用显式地分配块。它们的不同之处在于由哪个实体来负责释放已分配的块。

显式分配器：要求应用显式地释放任何已分配的块。例如，C标准库提供一种叫做ma耳oc程序包的显式分配器。C程序通过调用malloc函数来

分配一个块，并通过调用free函数来释放一个块。C++中的new和delete操作符与C中的malloc和free相当。

隐式分配器：另一方面，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。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配的块的过程叫做垃圾收集。例如，诸如Lisp、ML以及Java之类的高级语言就依赖垃圾收集来释放已分配的块。

动态内存管理的基本方法与策略：

1. 带边界标签的隐式空闲链表分配器

原理：

在这种分配器中，一个块是由一个字的头部，有效载荷，以及可能的一些额外的填充和一个在块的结尾巴处的脚部组成的。其中头部编码了这个块的大小(包括头部和所以的填充)，以及这个块是已分配的还是空闲的。头部后面就是应用调用malloc时请求的有效载荷，有效载荷后面时一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

脚部就是Knuth提出的一种聪明而通用的技术，叫做边界标记，允许在常数时间内进行堆前面块的合并。其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就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。合并的时候访问前面的块的脚部和后面的块的头部，判断是否是空闲块进行合并并且修改相应块的头部和脚部信息。

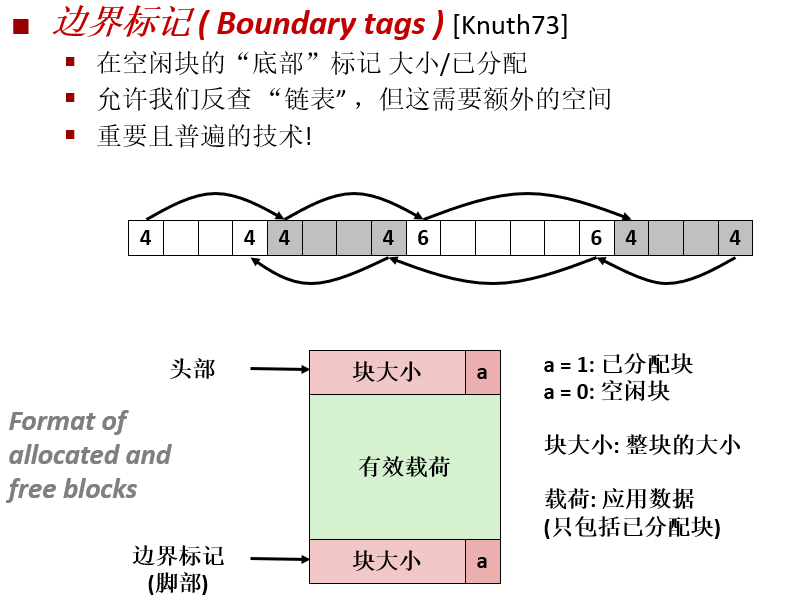


图7.10带边界标签的隐式空闲链表分配器

2.显式空闲链表

原理：

如图7.11所示，显示空闲链表就是将空闲块组织为某种形式的显示数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个pred（前驱）和succ（后继）指针。

使用双向链表而不是隐式空闲链表，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。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）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果的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比LIFO排序的首次适配由更高的内存利用率，接近最佳适配的利用率。

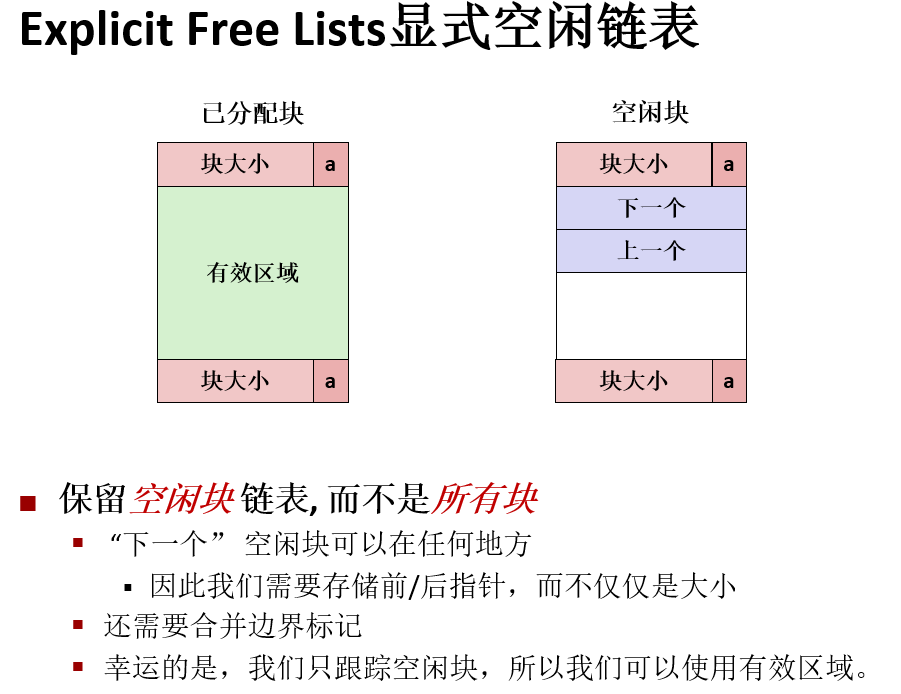


图7.11 显式空闲链表

3.分离的空闲链表

原理：

一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块。一个流行的减少分配时间的方法，通常称为分离存储，就是维护多个空闲链表，其中每个链表中的块有大致相等的大小。有关动态内存分配的文献描述了几十种分离存储方法，主要的区别在于它们如何定义大小类，何时进行合并，何时向操作系统请求额外的堆内存，是否允许分割等等。下面列举两种基本的方法：

<1>简单分离存储

使用简单分离存储，每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。

为了分配一个给定大小的块，我们检查相应的空闲链表。如果链表非空，我们简单地分配其中第一块的全部。空闲块是不会分割以满足分配请求的。如果链表为空，分配器就向操作系统请求一个固定大小的额外内存片，将这个片分成大小相等的块，并将这些块链接起来形成新的空闲链表。要释放一个块，分配器只要简单地将这个块插入到相应的空闲链表的前部。

<2>分离适配

使用这种方法，分配器维护一个空闲链表的数组。每个空闲链表是和一个大小类关联的，并且被组织成某种类型的显示或隐式链表。这里我们描述一种简单的版本。

为了分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到一个，那么就分割它，将剩余的部分插入到适当的空闲链表中，如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表，如此重复，知道找到一个合适的块。如果空闲链表中没有合适的块，那么就向操作系统请求额外的堆内存，从这个新的堆内存中分配一个块，将剩余部分放置再适当的大小类中。要释放一个块，我们执行合并，并将结果放置到相应的空闲链表中。

## 7.10本章小结

本章介绍了段式管理和页式管理以及如何依据页表和缓存完成虚拟地址到物理地址的变换和物理内存的访问，讲述了hello进程在fork和execve时的内存映射，以及发生缺页故障和缺页中断处理的步骤，最后描述了动态存储分配的基本方法与策略。

**（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：文件。

一个Linux文件就是一个m个字节的序列。所有的I/O设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。

每个Linux文件都有一个类型来表明它在系统中的角色：

1.普通文件：包含任意数据。

2.目录：是包含一组链接的文件，其中每个链接都将一个文件名映射到一个文件，这个文件可能是另一个目录。

3套接字：是用来与另一个进程进行跨网络通信的文件。

4.命名通道

5.符号链接

6.字符和块设备

设备管理：unix io接口。

这种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，这使得输入和输出都能以一种统一且一致的方式的来执行：

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，而文件的相关信息由内核记录，应用程序只需要记录这个描述符。

2. Linux shell创建的每个进程开始时都有三个打开的文件：标准输入（描述符为0）、标准输出（描述符为1）和标准错误（描述符为2）。

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，即从文件开头起始的字节偏移量。应用程序能够通过执行seek操作，显式地设置文件的当前位置为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字节的文件，当k大于等于m时执行读奥做会触发一个称为EOF的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的EOF符号。

5.关闭文件。当应用完成了对文件的访问之后，它就通知内核关闭这个文件。作为响应，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种。无论一个进程因为何种原因终止时，内核都会关闭所有打开的文件并释放它们的内存资源。

## 8.2 简述Unix IO接口及其函数

Linux以文件的方式对I/O设备进行读写，将设备均映射为文件。对文件的操作，内核提供了一种简单、低级的应用接口，即Unix I/O接口。

Unix I/O接口提供了以下函数：

1. int open(char \*filename, int flags, mode\_t mode)。进程通过调用open函数来打开一个已存在的文件或者创建一个新文件。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。

2. int close(int fd)。关闭一个打开的文件。

3. ssize\_t read(int fd, void \*buf, size\_t n)。读取文件，从当前文件位置复制最多n个字节到内存位置buf。返回值-1表示一个错误，而返回值0表示EOF。否则，返回值表示的是实际传送的字节数量。

4. ssize\_t write(int fd, const void \*buf, size\_t n)。写入文件，从内存位置buf复制至多n个字节到描述符fd的当前文件位置。

## 8.3 printf的实现分析

首先附上printf的源代码，如图8.1所示：

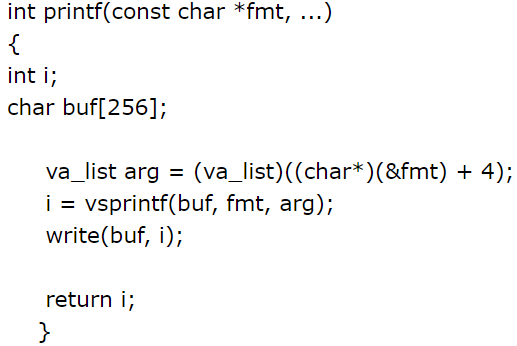


图8.1 printf函数的源代码

由图可知，printf的形参列表中fmt指向的是用来表示格式的字符串，…是可变形参的一种写法，当传递参数不确定时，就可以用这种方式表达。

Printf函数调用了vsprintf和write。其中vsprintf的源代码如图8.2所示，它的功能是接受格式串fmt。用fmt对可变参数进行格式化并输出。而write函数的功能是从内存位置buf复制至多i个字节到终端。

所以printf函数首先调用了vsprintf函数生成显示信息，然后调用write函数，write通过执行syscall指令调用了系统服务，来让内核执行打印操作。

接下来内核通过调用字符显示子程序，这个程序根据输入的ASCII码值在字模库中找到该字符对应的点阵，然后通过vram（存储每一个点的RGB颜色信息）输出。显示芯片将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）来实现字符串的输出。

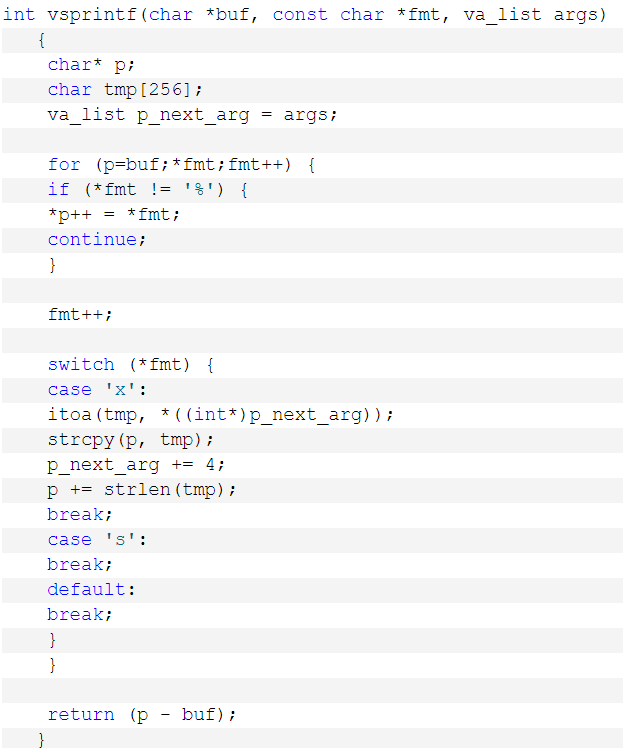


图8.2 vsprintf函数的源代码

## 8.4 getchar的实现分析

getchar函数的源代码如图8.3所示：

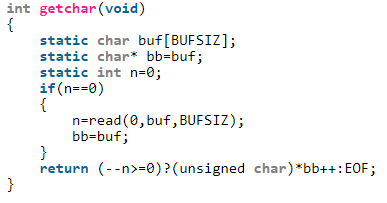


图8.3 getchar函数的源代码

由图8.3可知，getchar函数调用了read函数，read函数通过调用syscall中断来调用内核中的系统函数。进而调用键盘中断处理子程序，来接受按键扫描码转换成ascii码，保存到系统的键盘缓冲区。接下来read函数通过系统调用读取缓冲区的ascii码，直到接受到回车键才返回。如此来读取一个字符。

## 8.5本章小结

本章介绍了Linux下IO设备的管理方法，简述了Unix IO接口及其函数，分析了printf函数和getchar函数具体的实现过程。

**（第8章1分）**

# 结论

Hello所经历的过程：

1.hello本是一无所有。程序员借助编程工具，编写hello程序，生成hello.c。

2.hello.c经过预处理，根据以字符#开头的命令，修改原来的C程序，生成hello.i。

3.hello.i经过编译，生成一个包含汇编语言的hello.s文件。

4.hello.s经过汇编，将其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，生成hello.o。

5.hello.o经过链接生成可执行文件hello。

6.在linux终端下运行hello，shell调用fork创建子进程，子进程拥有一个和父进程一样的副本，但PID不同。

7.子进程调用execve函数在这个子进程的上下文中加载并运行hello，hello看上去像是一个人独占了CPU。

8.在CPU、TLB、各级Cache、主存、多级页表等的帮助下，hello度过了一个个难关（异常信号处理，例如缺页异常），一步步加载到内存中运行。

9.程序运行到了头，hello跑不动了。内核发来了SIGINT信号，hello进程听到后安详地结束了它的一生（终止）。Hello的父进程shell收到了SIGCHLD信号，为hello收了尸（回收子进程）。Hello彻底地走了，不带走一片云彩。

感悟：

学习这门课，受益匪浅。刚开始学这门课的时候，疑惑的地方其实很多，听得一知半解，因为它前面介绍的东西有些地方需要后面的知识来解释。随着课程的推进，我才慢慢体会到整个计算机系统逻辑的严密性和设计的巧妙之处。得益于计算机系统设计者的奇思妙想，硬件上的精巧设计，才使得整个程序的运行速度大大提高，内存占用率减少，能够流水线化地、并行地处理程序。进程和虚拟内存的概念使得程序之间互不干扰，安全并且正确地运行着。刚开始接触编程时，谁又能想到一个简简单单的hello world背后竟然有一段这么复杂曲折的过程呢？

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

hello.c：源程序



hello.i：预处理后的文本文件



hello.s：编译后汇编程序文本文件



hello.o：汇编后的可重定位目标文件（二进制文件）



hello：链接后的可执行目标文件



**（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] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[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.

[7] Randal E. Bryant. 机械工业出版社. 深入理解计算机系统.

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

**（参考文献0分，缺失 -1分）**

注：老师要求附上自媒体链接以便检查：

csdn: [CSAPP大作业程序人生-Hello‘s P2P\_HIT\_CY的博客-CSDN博客](https://blog.csdn.net/HIT_CY/article/details/118281334)

github: [HIT-CY/CSAPP (github.com)](https://github.com/HIT-CY/CSAPP)