

**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 1190201421

班　　 级 1936603

学 生 张 瑞

指 导 教 师 刘宏伟

**计算机科学与技术学院**

**2021年6月**

**摘 要**

本文从一个hello程序入手，漫步计算机系统世界。文章将解析hello程序从hello.c经过预处理、编译、汇编和链接生成可执行文件的全过程，阐述计算机系统对hello进行的进程管理、存储管理和IO管理，展示hello程序在Linux系统里的完整生命周期。在对hello程序生命周期探索的过程中，将会使用到Ubuntu虚拟机、edb和gcc等实用的工具，融合CS:APP中各章的重点知识，进一步提升读者对Linux系统操作的熟练度，加深读者对计算机系统的认识。

**关键词：**hello程序；预处理；编译；汇编；链接；进程；存储；IO

**目 录**

[第1章 概述 - 4 -](#_Toc75364149)

[1.1 hello简介 - 4 -](#_Toc75364150)

[1.2 环境与工具 - 4 -](#_Toc75364151)

[1.3 中间结果 - 4 -](#_Toc75364152)

[1.4 本章小结 - 5 -](#_Toc75364153)

[第2章 预处理 - 6 -](#_Toc75364154)

[2.1 预处理的概念与作用 - 6 -](#_Toc75364155)

[2.2在Ubuntu下预处理的命令 - 6 -](#_Toc75364156)

[2.3 hello的预处理结果解析 - 6 -](#_Toc75364157)

[2.4 本章小结 - 8 -](#_Toc75364158)

[第3章 编译 - 9 -](#_Toc75364159)

[3.1 编译的概念与作用 - 9 -](#_Toc75364160)

[3.2 在Ubuntu下编译的命令 - 9 -](#_Toc75364161)

[3.3 hello的编译结果解析 - 9 -](#_Toc75364162)

[3.3.1 数据类型 - 9 -](#_Toc75364163)

[3.3.2 赋值操作 - 11 -](#_Toc75364164)

[3.3.3 算术操作 - 11 -](#_Toc75364165)

[3.3.4 关系操作 - 11 -](#_Toc75364166)

[3.3.5 数组操作 - 11 -](#_Toc75364167)

[3.3.6 控制转移 - 12 -](#_Toc75364168)

[3.3.7 函数操作 - 13 -](#_Toc75364169)

[3.4 本章小结 - 14 -](#_Toc75364170)

[第4章 汇编 - 15 -](#_Toc75364171)

[4.1 汇编的概念与作用 - 15 -](#_Toc75364172)

[4.2 在Ubuntu下汇编的命令 - 15 -](#_Toc75364173)

[4.3 可重定位目标elf格式 - 15 -](#_Toc75364174)

[4.4 hello.o的结果解析 - 18 -](#_Toc75364175)

[4.5 本章小结 - 20 -](#_Toc75364176)

[第5章 链接 - 21 -](#_Toc75364177)

[5.1 链接的概念与作用 - 21 -](#_Toc75364178)

[5.2 在Ubuntu下链接的命令 - 21 -](#_Toc75364179)

[5.3 可执行目标文件hello的格式 - 21 -](#_Toc75364180)

[5.4 hello的虚拟地址空间 - 24 -](#_Toc75364181)

[5.5 链接的重定位过程分析 - 25 -](#_Toc75364182)

[5.6 hello的执行流程 - 27 -](#_Toc75364183)

[5.7 hello的动态链接分析 - 28 -](#_Toc75364184)

[5.8 本章小结 - 28 -](#_Toc75364185)

[第6章 hello的进程管理 - 30 -](#_Toc75364186)

[6.1 进程的概念与作用 - 30 -](#_Toc75364187)

[6.2 简述壳shell-bash的作用与处理流程 - 30 -](#_Toc75364188)

[6.3 hello的fork进程创建过程 - 31 -](#_Toc75364189)

[6.4 hello的execve过程 - 31 -](#_Toc75364190)

[6.5 hello的进程执行 - 31 -](#_Toc75364191)

[6.6 hello的异常与信号处理 - 32 -](#_Toc75364192)

[6.7本章小结 - 35 -](#_Toc75364193)

[第7章 hello的存储管理 - 36 -](#_Toc75364194)

[7.1 hello的存储器地址空间 - 36 -](#_Toc75364195)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 36 -](#_Toc75364196)

[7.3 hello的线性地址到物理地址的变换-页式管理 - 36 -](#_Toc75364197)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 37 -](#_Toc75364198)

[7.5 三级Cache支持下的物理内存访问 - 38 -](#_Toc75364199)

[7.6 hello进程fork时的内存映射 - 39 -](#_Toc75364200)

[7.7 hello进程execve时的内存映射 - 39 -](#_Toc75364201)

[7.8 缺页故障与缺页中断处理 - 40 -](#_Toc75364202)

[7.9动态存储分配管理 - 41 -](#_Toc75364203)

[7.10本章小结 - 44 -](#_Toc75364204)

[第8章 hello的IO管理 - 45 -](#_Toc75364205)

[8.1 Linux的IO设备管理方法 - 45 -](#_Toc75364206)

[8.2 简述Unix IO接口及其函数 - 45 -](#_Toc75364207)

[8.3 printf的实现分析 - 46 -](#_Toc75364208)

[8.4 getchar的实现分析 - 47 -](#_Toc75364209)

[8.5本章小结 - 48 -](#_Toc75364210)

[结论 - 49 -](#_Toc75364211)

[附件 - 51 -](#_Toc75364212)

[参考文献 - 52 -](#_Toc75364213)

# 第1章 概述

## 1.1 hello简介

P2P:From Program to Process

用C语言编写得到的hello.c经过cpp的预处理变为hello.i，然后经过cc1的编译变为hello.s，再经过as的汇编变为hello.o，最后经过ld的链接成为可执行目标文件hello，如图1-1所示。当我们在shell中键入执行命令后，shell创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。

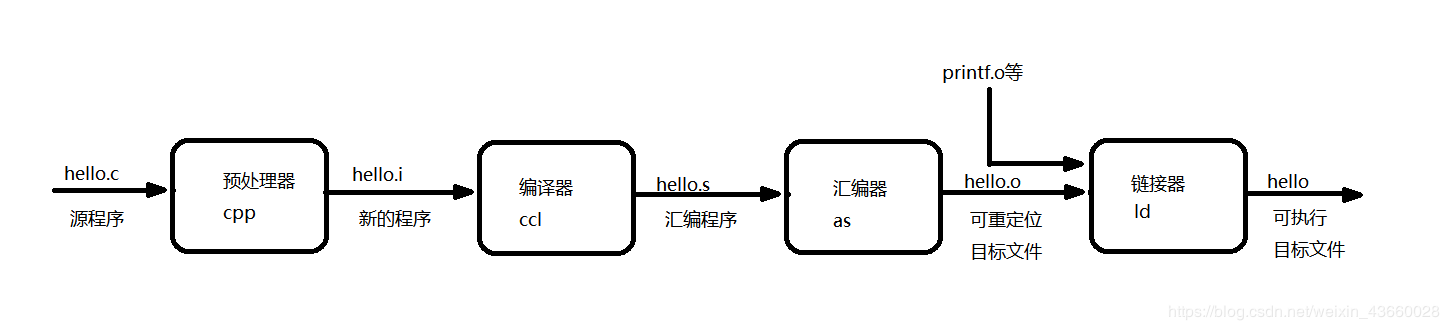


图1-1 编译系统

020:From Zero to Zero

Shell生成的子进程通过execve加载并运行hello，操作系统为其分配虚拟内存空间，再将虚拟内存空间映射到物理内存空间，然后进入 main函数执行目标代码。CPU为运行的hello分配时间片，执行逻辑控制流。IO和信号处理使得hello中的内容能顺利显示到屏幕上。程序运行结束后，shell接收信号，回收hello进程，内核将会清除与hello相关的数据。

## 1.2 环境与工具

硬件环境：X64 CPU；2.5GHz；8G RAM；256GHD Disk

软件环境：Windows10 64位；VMware16；Ubuntu 20.04 LTS 64位

工具：gcc；edb；objdump；readelf等

## 1.3 中间结果

|  |  |
| --- | --- |
| 文件名 | 文件作用 |
| hello.i | hello.c预处理后的文本文件，用于分析预处理结果 |
| hello.s | hello.i编译后的汇编文件，用于分析编译结果 |
| hello.o | hello.s汇编后的可重定位目标文件，用于分析汇编结果 |
| elf.txt | hello.o的ELF格式，用于分析hello.o |
| objdump.txt | hello.o的反汇编代码，用于分析hello.o |
| hello | hello.o链接后的可执行目标文件 |
| elf2.txt | hello的ELF格式，用于分析hello |
| objdump2.txt | hello的反汇编代码，用于分析hello |

## 1.4 本章小结

本章简述了hello程序P2P和020的整个过程，介绍了实验所用到的环境和工具，并给出了hello程序走完整个生命周期将会经历的中间文件。

# 第2章 预处理

## 2.1 预处理的概念与作用

概念：

预处理是指预处理器根据以字符#开头的命令（包括宏定义、条件编译和源文件包含等），修改原始的C程序，插入指定文件，扩展指定的宏，得到一个完整的文本文件。

作用：

1.对宏定义进行宏替换。

2.处理条件编译命令，过滤不需要编译的代码。

3.将需包含的源文件的内容插入程序文本中。

4.过滤掉所有的注释。

5.使得源代码在不同的执行环境中被方便的修改或编译。

## 2.2在Ubuntu下预处理的命令

预处理命令：gcc hello.c -E -o hello.i

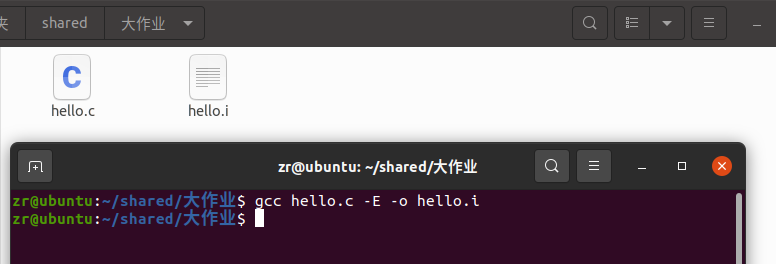


图2-1 预处理命令

## 2.3 hello的预处理结果解析

将得到的hello.i以文本文件形式打开，会发现原本几十行的hello.c（如图2-2）变成了数千行。文件的最后末端部分是hello.c的主体内容（如图2-3），前面部分（如图2-4）将指定的源文件包含了进来，说明已经将hello.c进行了预处理。



图2-2 hello.c代码

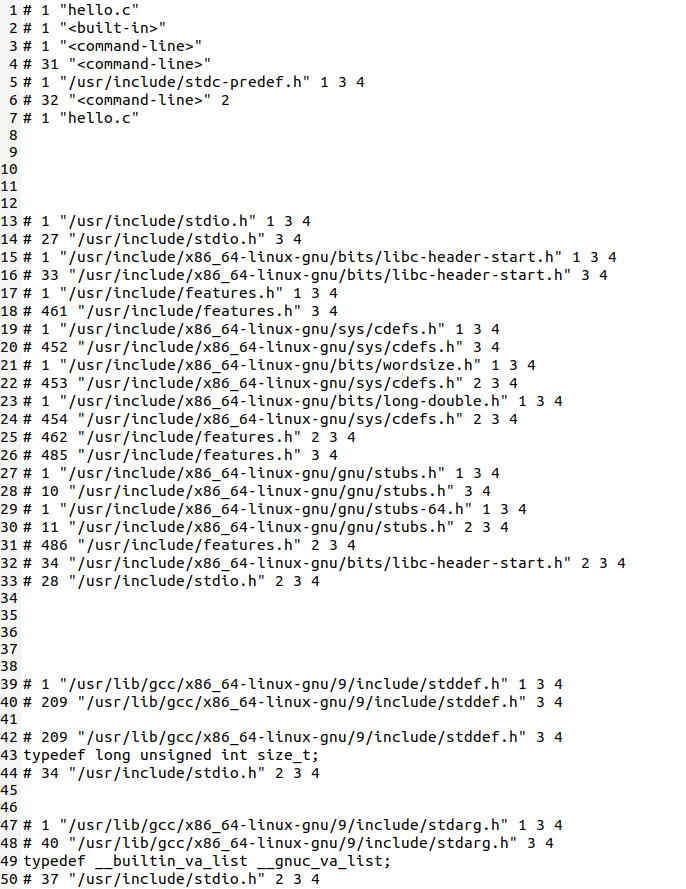


图2-3 hello.i前面部分代码（仅展示了前50行）



图2-4 hello.i末端部分代码

## 2.4 本章小结

本章介绍了预处理的概念及作用，并演示了在Ubuntu虚拟机下对hello.c进行预处理的过程，分析了预处理结果hello.i。

# 第3章 编译

## 3.1 编译的概念与作用

概念：

编译是指编译器将预处理后得到的.i文件（文本文件）处理成为.s文件（文本文件），它包含一个汇编语言程序。

作用：

编译器能将不同的高级语言写出的源程序转换为使用通用的汇编语言的汇编语言程序，为后续生成机器语言的代码做好准备。值得一提的是，除了实现上述基本功能，编译器还可以实现语法检查、程序优化等功能。

注意：这儿的编译是指从 .i 到 .s 即预处理后的文件到生成汇编语言程序。

## 3.2 在Ubuntu下编译的命令

编译命令：gcc hello.i -S -o hello.s

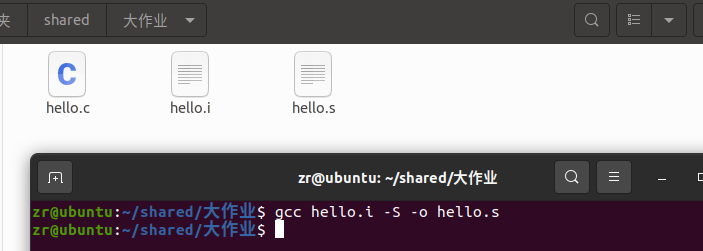


图3-1 编译命令

## 3.3 hello的编译结果解析

## 3.3.1 数据类型

1.全局变量 int sleepsecs

全局变量sleepsecs被存储在了数据段（如图3-2），其值大小不是2.5，而是2，这是因为sleepsecs的类型被定义为了int，这里进行了一个隐式的转换。

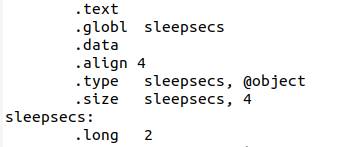


图3-2 全局变量sleepsecs的编译结果

2.main函数参数 int argc

argc作为第一个参数，被存储在寄存器%rdi中。



图3-3 main函数参数argc的编译结果

3. main函数参数 char \*argv[]

argv作为第二个参数，被存储在寄存器%rsi中。注意argv是作为一个char型数组的首地址被存在%rsi中的。



图3-4 main函数参数argv的编译结果

4.局部变量 int i

局部变量通常储存在栈上，在hello.s中，我们能发现局部变量i存储在了栈上-4(%rbp)处。



图3-5 局部变量i的编译结果

5.常量

代码中出现的常量，均以立即数的形式出现在hello.s中（如图3-6）。



图3-6 常量的编译结果

6.字符串

字符串被放在只读数据段，如图3-7。

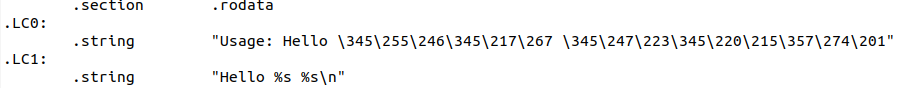


图3-7 字符串的编译结果

## 3.3.2 赋值操作

程序中的赋值操作i = 0，由movl实现（如图3-5）。其中l表示传送的是双字大小，即4个字节。

## 3.3.3 算术操作

程序中算术操作i++，由addl实现（如图3-8）。



图3-8 i++的编译结果

## 3.3.4 关系操作

1.argc != 3

判断main函数参数argc与3的大小时，用到了cmpl（如图3-9），然后设置条件码，为后续跳转做准备。



图3-9 argc != 3的编译结果

2.i < 10

判断局部变量i和10的大小时，用到了cmpl（如图3-10），然后设置条件码，为后续跳转做准备。



图3-10 i < 10的编译结果

## 3.3.5 数组操作

程序中涉及到读取argv[1]和argv[2]的操作（如图3-11），可以看到先是将argv[0]的地址传入%rax，然后将%rax加16，即可得到argv[2]的地址，再将%rax中的值作为地址，将内存中对应地址处的值传给%rdx，至此便成功将argv[2]中的值读取出来存储到%rdx中了。读取argv[1]的值方法类似，不过在将%rax加16处改为了将%rax加8，使得%rax中存储的是argv[1]的地址，而不是argv[2]的地址。

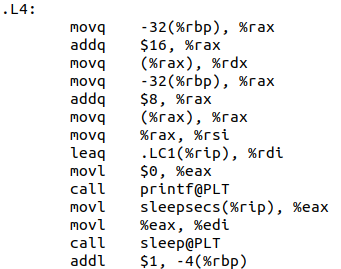


图3-11 argv[1]和argv[2]的编译结果

## 3.3.6 控制转移

1.if (argc != 3)

3.3.4中的关系操作会使得条件码被设置，当条件满足时（ZF被设置，即两数相等），指令会跳转到指定处（.L2），如图3-12。



图3-12 if (argc != 3)的编译结果

2.for ( i = 0; i < 10; i++)

第一次循环开始前，先给i赋初值，然后无条件跳转到循环条件判断处（.L3），之后每次循环开始前都会比较i与10的大小，设置条件码，从而根据条件码判断是否需要继续跳转到循环体开始处（.L4），如图3-13。

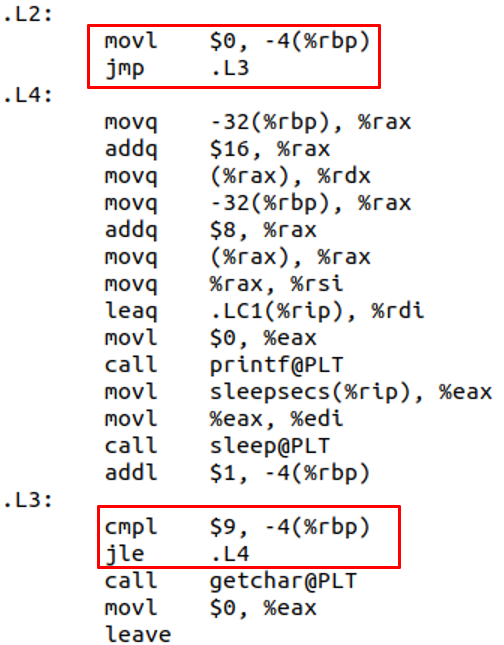


图3-13 if (argc != 3)的编译结果

## 3.3.7 函数操作

函数被调用之前，需要先在寄存器中存好要传的参数（存储器使用顺序为%rdi、%rsi、%rdx、%rcx、%r8、%r9），然后用call来调用函数，函数的返回值会被存放到寄存器%rax中。

1.main函数

参数传递：main函数的传入参数argc和argv，分别用寄存器%rdi和%rsi存储，前面已经详细叙述过。

函数调用：main函数的调用是由\_start函数调用的系统启动函数\_\_libc\_start\_main来完成的，main的返回值也将由它处理，并且在需要的时候会将控制返回给内核。

函数返回：设置%eax为0并且leave，如图3-14，对应return 0。



图3-14 main函数返回的编译结果

2.printf函数

参数传递：第一次调用时，将字符串“Usage: Hello 学号 姓名！\n”的有效地址传入%rdi作为参数，调用puts，如图3-15。第二次调用时，将字符串“Hello %s %s\n”的有效地址传入%rdi，将argv[1]和argc[2]的值分别传入%rsi和%rdx作为参数，调用puts，如图3-16。



图3-15 第一次调用printf函数

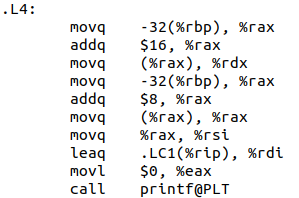


图3-16 第二次调用printf函数

函数调用：在if条件判断中被main函数调用一次（如图3-15），在for循环中被main函数调用一次（如图3-16）。

3.exit函数

参数传递：传入的参数为1，存储在%rdi中，如图3-17。

函数调用：在if条件判断中被main函数调用，如图3-17。



图3-17 调用exit函数

4.sleep函数

参数传递：传入的参数为sleepsecs，存储在%rdi中，如图3-18。

函数调用：在for循环中被main函数调用，如图3-18。



图3-18 调用sleep函数

5.getchar函数

函数调用：在main函数中，最后返回前被调用，如图3-19。



图3-19 调用getchar函数

## 3.4 本章小结

本章介绍了编译的概念及作用，演示了在Ubuntu虚拟机下对hello.i进行编译得到hello.s的过程，并分析了编译结果。在结果分析部分，对hello.c中存在的数据类型、赋值操作、算术操作、关系操作、数组操作、控制转移和函数操作均进行了具体的分析，详细解读了hello的编译结果。

# 第4章 汇编

## 4.1 汇编的概念与作用

概念：

汇编是指汇编器将编译后得到的.s文件（文本文件）翻译成机器语言指令，把这些指令打包成可重定位目标程序的格式，并将结果保存在目标文件.o（二进制文件）中。

作用：

将汇编代码转为机器语言指令，使代码真正能被机器识别并执行。

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

汇编命令：as hello.s -o hello.o

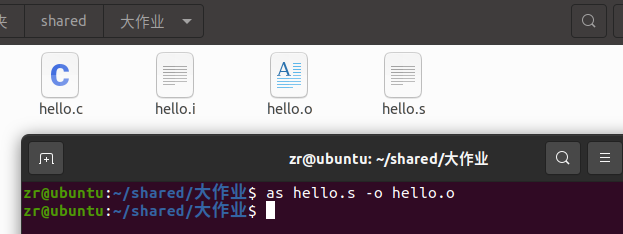


图4-1 汇编命令

## 4.3 可重定位目标elf格式

首先使用指令readelf -a hello.o > elf.txt获得hello.o的ELF可重定位目标文件，如图4-2。

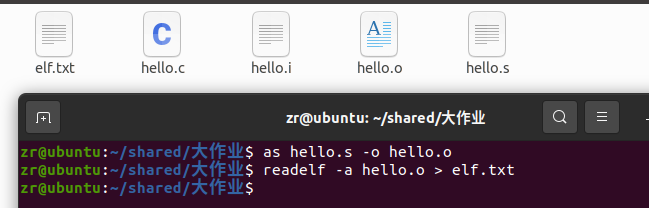


图4-2 生成elf.txt

1.ELF头

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，以及节头部表中条目的大小和数量。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大小的条目。

如图4-3，可以看出该系统字的大小为64位，采用小端序，机器类型为x86-64，这个ELF头的大小为64字节，有14个节头，文件是可重定位目标文件。



图4-3 ELF头

2.节头部表

节头部表是描述目标文件的节，它会描述目标文件中不同节的编号、名称、类型、地址、偏移量、大小、全体大小、旗标、链接、信息和对齐等，如图4-4。

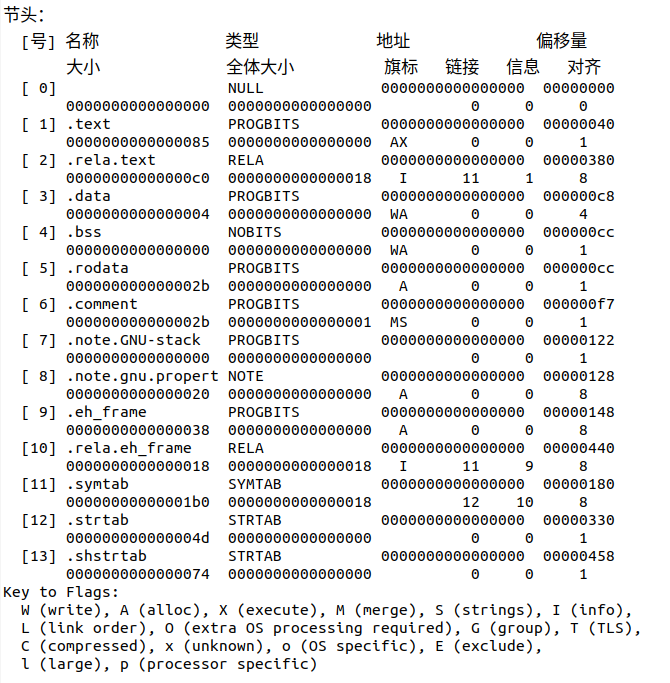


图4-4 节头部表

3.重定位节

重定位节中的偏移量是指需要被修改的引用的节偏移；类型告知链接器如何修改新的引用；符号名称标识被修改引用应该指向的符号；加数是一个有符号常数，一些重定位要使用它对被修改引用的值做偏移调整。具体的重定位过程将在第5章进行叙述，此处只介绍相关概念。

.rela.text（如图4-5）是一个.text 节中位置的列表，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。



图4-5 .rela.text节

类似地，.rela.eh\_frame（如图4-6）中包含.eh\_frame节中需要进行重定位的信息。

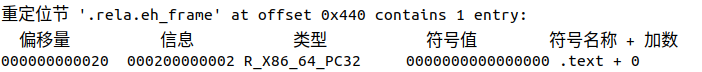
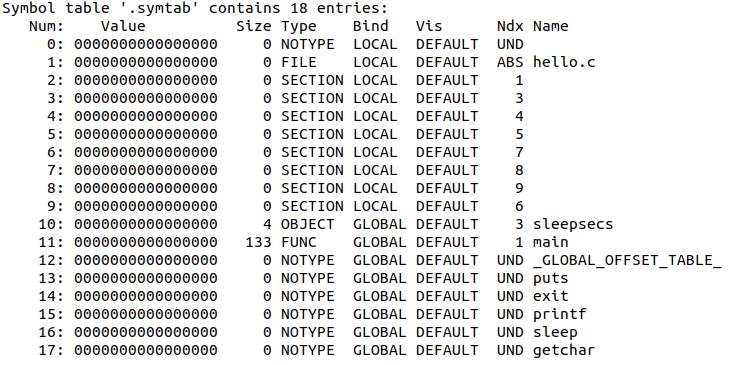


图4-6 .rela.eh\_frame节

4.符号表

.symtab是一个符号表（如图4-7），它存放在程序中定义和引用的函数和全局变量的信息。每个可重定位目标文件在.symtab中都有一张符号表（除非特意用STRIP命令去掉它）。然而，和编译器中的符号表不同，.symtab符号表不包含局部变量的条目。

图4-7 符号表

## 4.4 hello.o的结果解析

首先使用objdump -d -r hello.o > objdump.txt得到hello.o的反汇编代码，如图4-8。

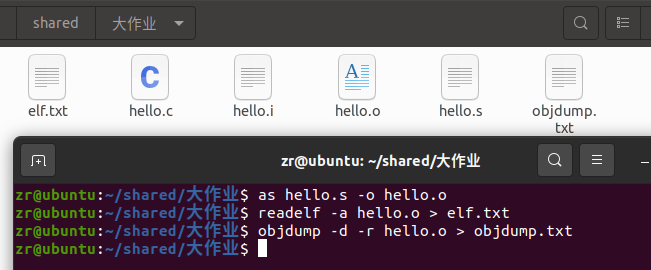


图4-8 生成objdump.txt

查看objdump.txt的内容，如图4-9。

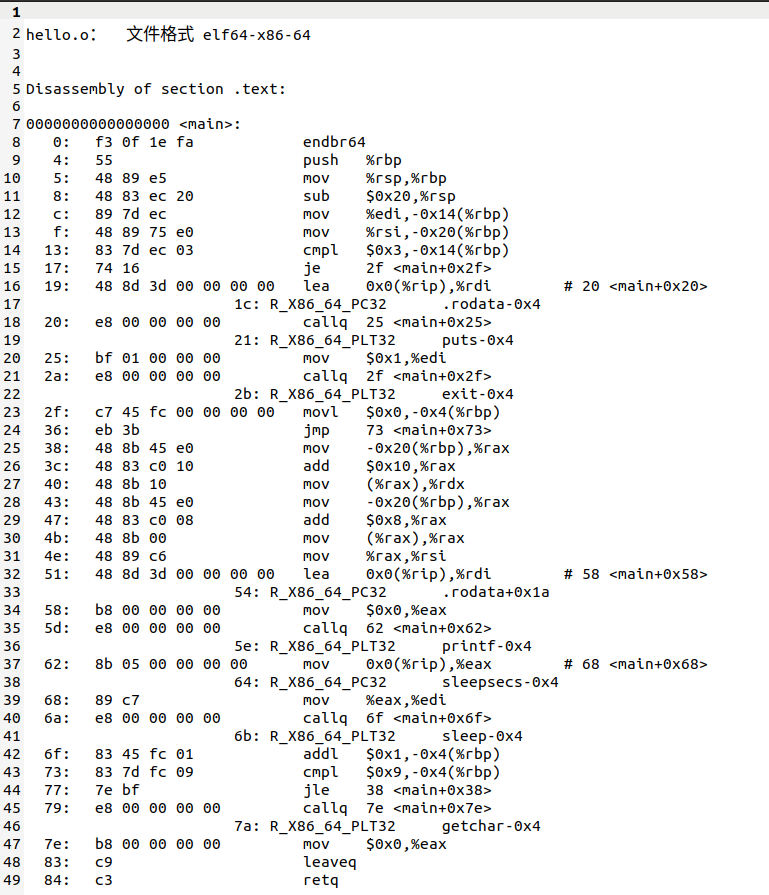


图4-9 objdump.txt

将objdump.txt与hello.s比较，会发现两者具有一定的相似性，即在汇编指令部分几乎完全一致，涉及到函数调用的部分汇编指令发生了变化，引入了重定位条目。而且objdump.txt中出现了hello.s中没有的机器语言。

机器语言是纯粹的二进制数据表示的语言，能真正被机器识别并执行；汇编语言是容易被人理解的直接描述CPU行为的语言，但机器并不能直接识别和执行。每一条汇编语言的指令都能被转换为唯一确定的机器语言指令，即两者间存在着一一对应的映射关系。

机器语言中的操作数是由寄存器指示符字节指明用到的寄存器，用常数字指明立即数数据、地址指示符的偏移量及分支和调用的目的地址；而汇编语言用立即数、寄存器和内存引用的方式指明操作数。

hello.s中，程序为了进行分支转移，进行了分段并命名（如.L3），这是在编译时候标记的助记符。在汇编成机器语言之后，为了明确下一条指令的位置，不能再使用这种段名称的标记方式，所以objdump.txt得到的反汇编代码中的跳转指令用的不再是段名称，而是确定的地址。

hello.s中，函数调用后面直接跟着被调用函数的名称，而objdump.txt中，call的目标地址是下一条指令。这是因为hello.c中调用的函数都是共享库中的函数，最终需要通过动态链接器来确定函数运行时的目标地址，在汇编成为机器语言的时候，对于这些不确定地址的函数调用，将其call指令后的相对地址设置为全0(即设为下一条指令），并在.rela.text节中添加重定位条目，等待后续的重定位。

## 4.5 本章小结

本章介绍了汇编的概念及作用，演示了在Ubuntu虚拟机下对hello.s进行汇编得到hello.o的过程，并分析了汇编结果。在分析结果时，先是用readelf查看了hello.o的ELF文件，从ELF头、节头部表、重定位节和符号表的角度分别进行了分析，介绍一些基本概念，为第5章做好准备。同时也使用objdump查看了hello.o的反汇编代码，从操作数、分支转移和函数调用等方面分析了机器语言和汇编语言的差异。

# 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，这个文件可被加载(复制)到内存并执行。

作用：

链接使得分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。当我们改变这些模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其它文件。

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

链接命令：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

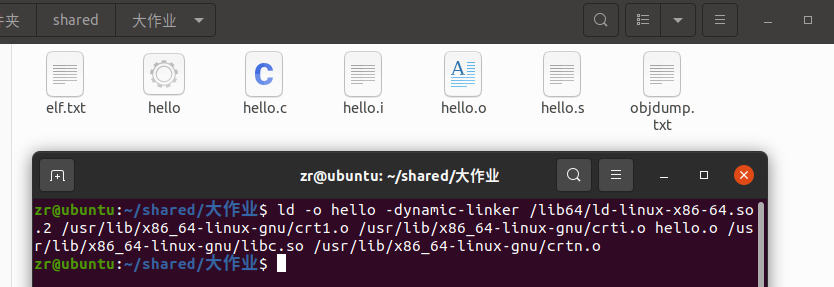


图5-1 链接命令

## 5.3 可执行目标文件hello的格式

首先使用指令readelf -a hello > elf2.txt获得hello的ELF可执行目标文件，如图5-2。

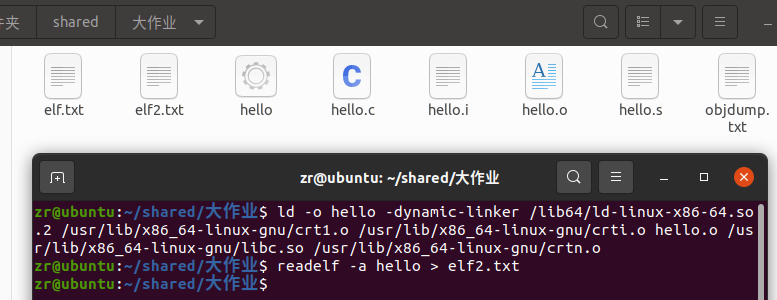


图5-2 生成elf2.txt

1.ELF头

如图5-3，可以看出ELF头中部分信息与链接前相比发生了变化，文件类型变成了可执行目标文件，节头和程序头均增加，还获得了程序的入口点地址。

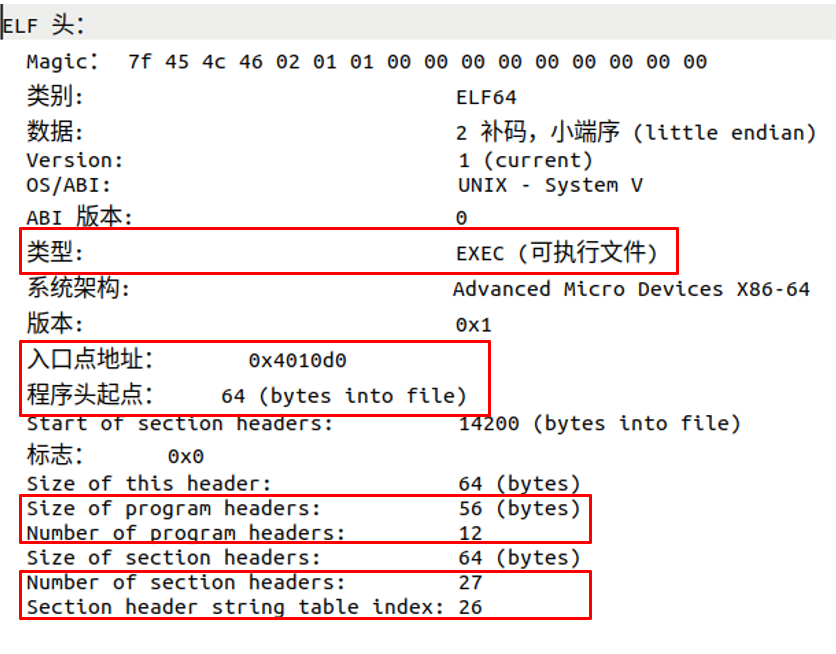


图5-3 ELF头

2.节头部表

如图5-4，此时节的数量增多，各节已被重定位到最终运行时的内存地址处，地址、偏移量、大小等数据发生了变化。

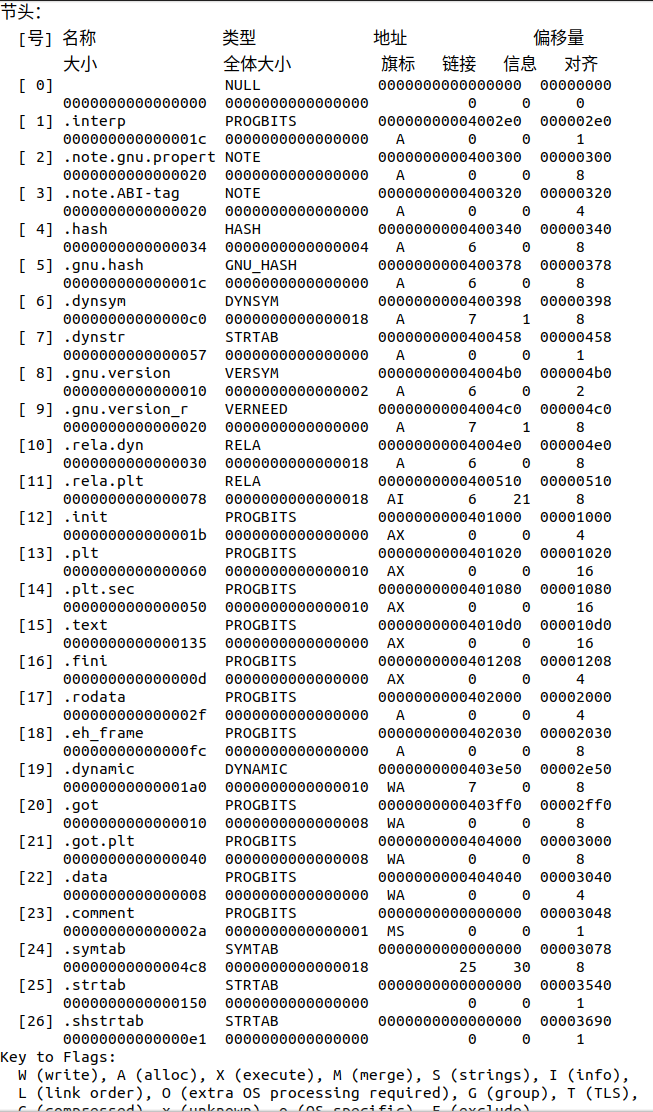


图5-4 节头部表

3.程序头部表

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5-5，offset表示目标文件中的偏移，viraddr是虚拟地址，phyaddr是内存地址，filesize是目标文件中段的大小，memsiz是内存中段的大小，flags是运行时的访问权限，align是对齐要求。

PHDR指保存了二进制的程序头表，INTERP指定程序已经从可执行文件映射到具体的内存区域后，必须调用的解释器（如动态链接器）的路径，LOAD表示可加载的程序段，DYNAMIN保存了由动态链接器使用的信息，NOTE保存辅助信息，GNU\_STACK标志栈是否可执行，GNU\_RELRO指定重定位后需被设置成只读的内存区域。

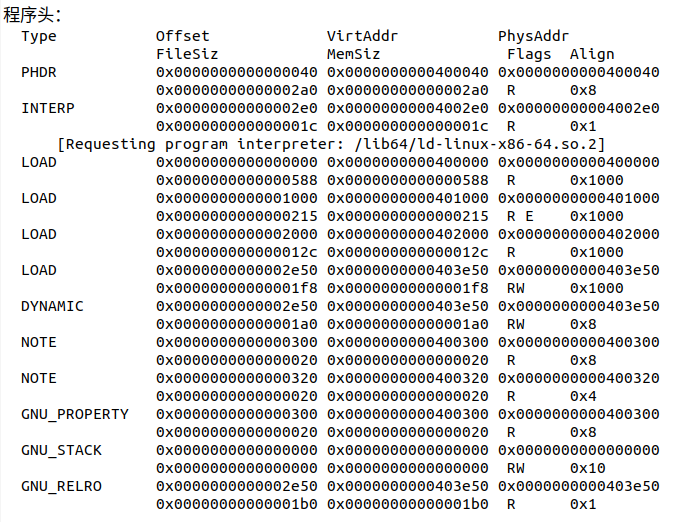


图5-5 程序头部表

## 5.4 hello的虚拟地址空间

下面在Data Dump中查看5.3中出现的段，以标为INTERP和第三个标为LOAD的段为例。



图5-6 INTERP段

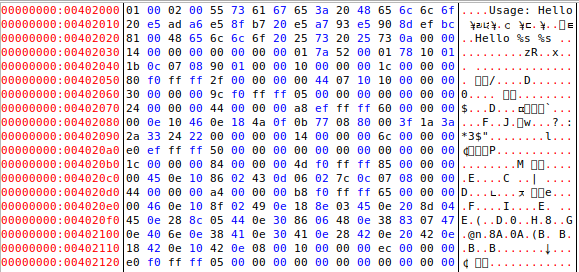


图5-7 LOAD段

以这两个段为例，能看到在程序头表中描述的段的各类信息（类型、起始地址和大小等）均和Data Dump查看结果一致。实际上，所有段的虚拟地址信息都被正确而完整地记录在了程序头表中。

## 5.5 链接的重定位过程分析

首先使用objdump -d -r hello > objdump2.txt得到hello的反汇编代码，如图5-8。

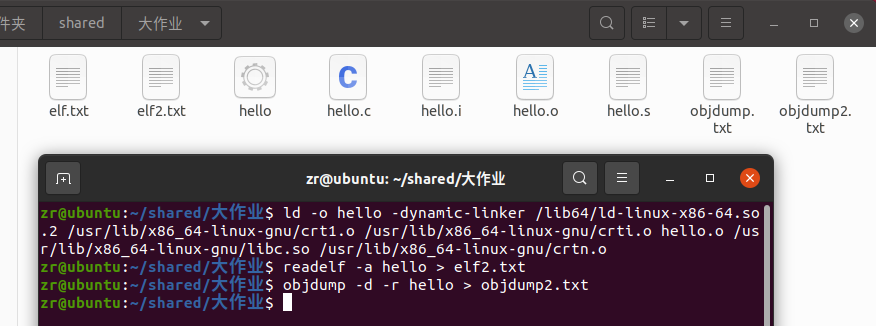


图5-8 生成objdump2.txt

查看objdump2.txt的内容，如图5-9。

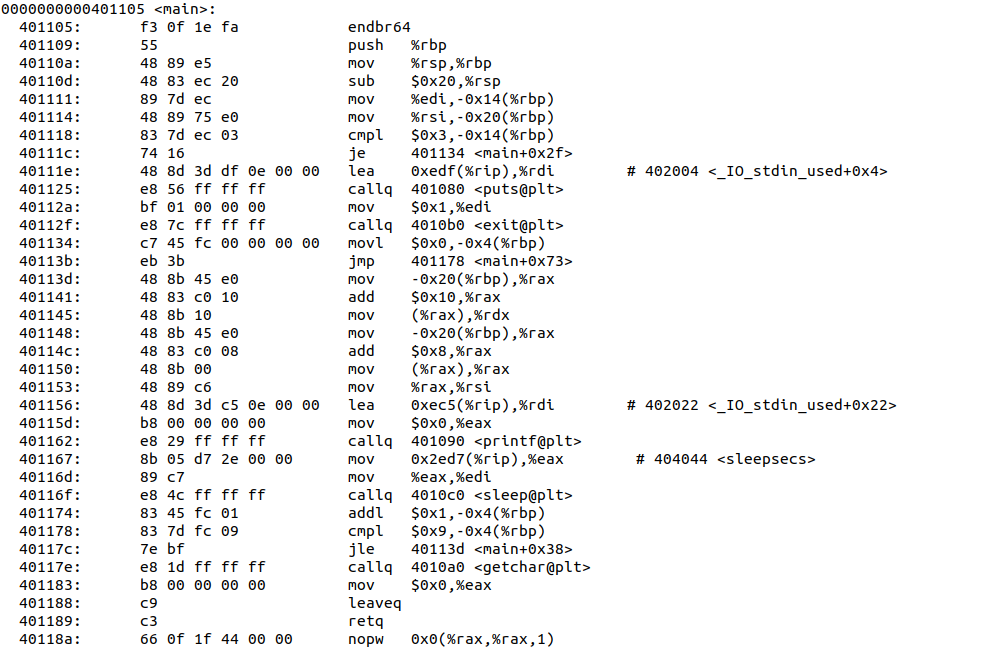


图5-9 objdump2.txt（仅包含main函数部分）

和上次反汇编的结果进行比较，有一些不同之处：

（1）hello反汇编得到的汇编代码中，已经有了确定的地址，即完成了重定位；而hello.o反汇编的结果尚未进行重定位，仅仅标注出了需要重定位的部分。

（2）hello反汇编的结果中出现了更多函数的汇编代码，节也增多了，这是因为链接使得多个文件合并，更多的节以及更多的函数都被引入了。

ELF有多种重定位类型，其中最基本的两种是R\_X86\_64\_PC32和R\_X86\_64\_32，这两种重定位算法的伪代码如图5-10所示，其中r表示重定位条目，s表示节。

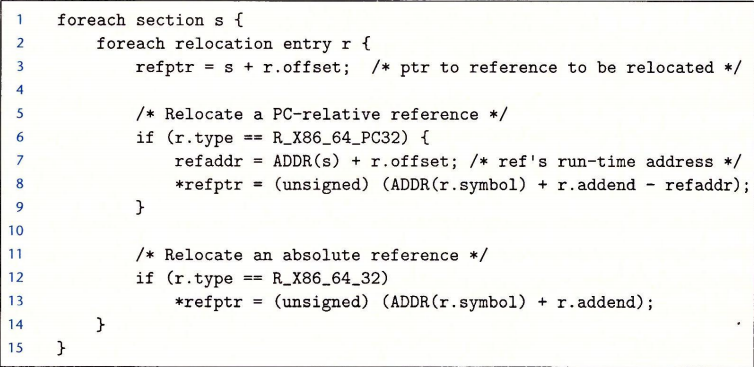


图5-10 重定位算法

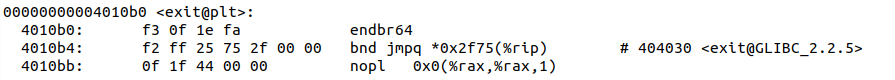


图5-11 exit重定位后的地址

现在以exit为例，分析其重定位的实现过程：查看图4-9中的代码和重定位条目，再结合图4-5中的重定位节的信息，可知main函数引用了exit，且需使用32位PC相对地址进行重定位（R\_X86\_64\_PLT32同R\_X86\_64\_PC32一样是相对寻址）。

若exit的重定位条目为r，则由图4-9知，有：

r.offset = 0x2b

r.symbol = exit

r.type = R\_X86\_64\_PLT32

r.addend = -4

且s满足：

ADDR(s) = 0x401105（如图5-9）

此外：

ADDR(r.symbol) = ADDR(exit) = 0x4010b0（如图5-11）

由图5-10中的公式知：

refaddr = ADDR(s) + r.offset = 0x401105 + 0x2b = 0x401130

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

= (unsigned)(0x4010b0 + (-4) -0x401130)

=(unsigned)(0xffffff7c)

如图5-12，重定位结果正确。



图5-12 hello中调用exit处代码

## 5.6 hello的执行流程

ld-2.31.so!\_dl\_start

ld-2.31.so!\_dl\_init

hello!\_start

call main

ld-2.31.so!\_\_libc\_start\_main

hello!\_init

hello!main

hello!puts@plt

hello!exit@plt

## 5.7 hello的动态链接分析

共享库是一个目标模块，在运行或加载时，可以加载到任意的内存地址，并和一个在内存中的程序链接起来。这个过程称为动态链接，是由一个叫做动态链接器的程序来执行的。共享库也称为共享目标，在Linux系统中通常用.so后缀来表示。

动态链接器使用过程链接表PLT+全局偏移量表GOT实现函数的动态链接，GOT中存放函数目标地址，PLT使用GOT中地址跳转到目标函数。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。

过程链接表（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。

全局偏移表（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[0]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ld-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。初始时，每个GOT条目都指向对应PLT条目的第二条指令。

.got.plt起始地址是0x404000（如图5-4），在dl\_init前，内容如图5-13所示。

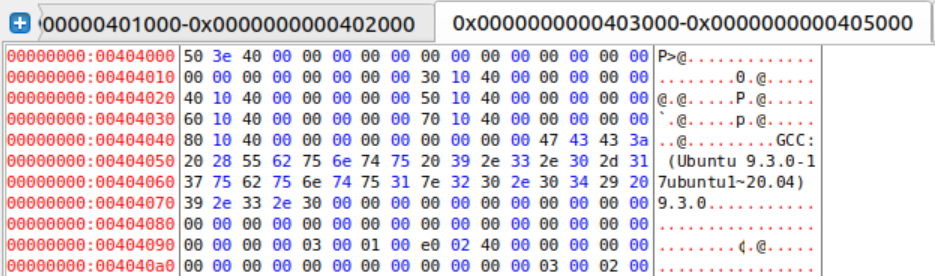


图5-13 dl\_init前.got.plt内容

在dl\_init后，内容如图5-14所示。

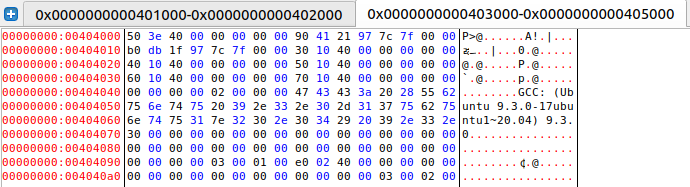


图5-14 dl\_init后.got.plt内容

## 5.8 本章小结

本章介绍了链接的概念及作用，演示了在Ubuntu虚拟机下对hello.o进行链接得到hello的过程，并分析了链接结果。在分析结果时，先是用readelf查看了hello的ELF文件，从ELF头、节头部表和程序头部表的角度分别进行了分析，验证了段在虚拟地址空间的位置。同时也使用objdump查看了hello的反汇编代码，发现链接后的目标文件函数和节都增加了，重定位也完成了，还以exit为例解释了重定位算法的具体实现。除了静态的链接，链接器还会动态链接共享库，在最后还展示了hello的执行流程，分析了hello的动态链接。

# 第6章 hello的进程管理

## 6.1 进程的概念与作用

概念：

进程的经典定义就是一个执行中的程序的实例。系统的每一个程序都是运行在某一个进程上下文中。上下文是由程序正确运行所需要的状态构成的。这个状态包括存放在内存中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、环境变量以及上下文描述符的集合。

作用：

1.每次用户通过向shell输入一个可执行目标文件的名字，运行程序时，shell就会创建一个新的进程，然后在这个进程的上下文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。

2.进程提供给应用程序的关键抽象：一个独立的逻辑控制流，好像我们的程序独占地使用处理器；一个私有的地址空间，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是一个应用程序，是操作系统中用户与系统内核进行交互的界面。

处理流程：

1.读取用户由键盘输入的命令行。

2.分析命令，以命令名作为文件名，并将其它参数改造为系统调用execve( )内部处理所要求的形式。

3.终端进程调用fork( )建立一个子进程。

4.终端进程本身调用wait( )来等待子进程完成（如果是后台命令，则不等待）。当子进程运行时调用execve( )，子进程根据文件名到目录中查找有关文件，调入内存，执行这个程序。

5.如果命令末尾有&,则终端进程不用执行系统调用wait( )，立即发提示符，让用户输入下一条命令；否则终端进程会一直等待，当子进程完成工作后，向父进程报告，此时中断进程醒来，作必要的判别工作后，终端发出命令提示符，重复上述处理过程。

## 6.3 hello的fork进程创建过程

父进程通过调用fork函数创建一个新的运行的子进程。新创建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同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。父进程和子进程最大的区别在于他们有不同的PID。父进程与子进程是并发运行的独立进程，内核能够以任意方式交替执行它们的逻辑控制流中的指令。

在这里，shell作为父进程通过fork函数为hello创建一个新的子进程，在这个新进程的上下文中运行hello。

## 6.4 hello的execve过程

子进程调用exceve函数在当前子进程的上下文加载并运行一个新的程序，此处即hello程序。exceve函数加载并运行可执行目标文件hello，且带参数列表argv和环境变量argc。加载器会删除现有的用户区域；映射私有空间，创建新的代码、数据、堆和栈区域，将代码段和数据段初始化为hello的代码和数据，堆和栈被置空；映射共享区域，实现动态链接；设置PC，将其指向hello程序的起始位置，即从下条指令开始执行hello程序。

## 6.5 hello的进程执行

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。

内核为每个进程维持了一个上下文。上下文就是内核重新启动一个被抢占的进程所需的状态。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了的进程，这种决策叫做调度。在内核调度了一个新的进程运行后，它就抢占当前进程，使用一种称为上下文切换的机制来将控制转移到新的进程，上下文切换1）保存当前进程的上下文，2）恢复某个先前被抢占的进程被保存的上下文，3）将控制传递给这个新恢复的进程。

最初hello运行在用户模式下，内核将控制给hello，为hello分时间片，上下文为hello的上下文。hello正常运行，当遇到sleep函数时，内核会因显式的请求抢占hello，进行上下文切换，将控制转移给sleep，直到sleep结束，内核再进行上下文切换，将控制传递给hello，过程如图6-1。其他的函数调度过程类似。

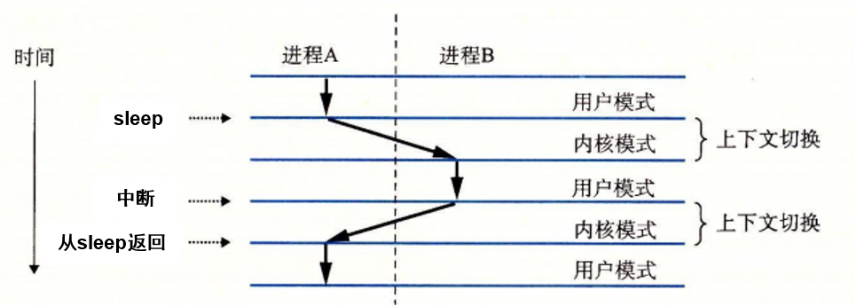


图6-1 进程调度（以sleep为例）

## 6.6 hello的异常与信号处理

可能出现的异常一共有四种：中断、陷阱、故障、终止。它们的产生原因及处理如图6-2。

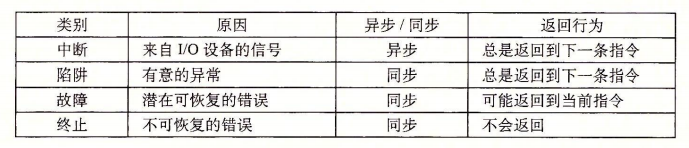


图6-2 异常类别及处理

在发生异常时会发出信号，常见信号种类如图6-3所示。



图6-3 信号种类及处理

图6-4展示了正常运行hello至结束的情况。



图6-4 正常运行

若在运行过程中乱按键盘，如图6-5。可以看到只是将字符缓存到stdin，直到按下’\n’，被当作命令读入。

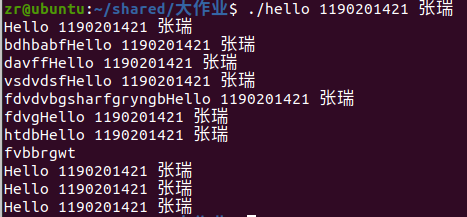


图6-5 乱按键盘

若在运行过程中按下Ctrl+C，内核发送一个SIGINT信号到前台进程组的每个进程，默认情况是终止前台作业，hello将会被立即终止，用ps查看发现无hello，如图6-6。

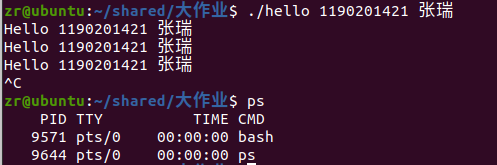


图6-6 按下Ctrl+C

若在运行过程中按下Ctrl+Z，内核发送一个SIGTSTP信号到前台进程组的每个进程，默认情况是停止前台作业，此时hello未被回收，而运行在后台，用ps查看发现有hello，如图6-7。

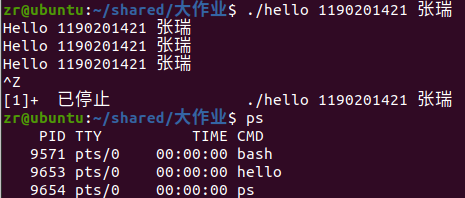


图6-7 按下Ctrl+Z

用jobs查看当前进程情况，用pstree查看相关进程关系，如图6-8。



图6-8 jobs和pstree查看相关信息

用 fg 1 将hello调回前台，此时会先打印执行 hello 的命令行命令，然后 hello 继续运行打印剩下的 7个输出，最后输入字串，程序结束，同时进程被回收，如图6-9。



图6-9 用fg调回hello

## 6.7本章小结

本章阐述了进程的概念与作用，介绍了 Shell 的作用和一般处理流程，分析了调用 fork 创建新进程，调用 execve加载并执行hello以及hello的进程调度。还以乱按键盘、按下Ctrl+C和按下Ctrl+Z为例，着重分析了hello 的异常与信号处理。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

逻辑地址：程序产生的和段相关的偏移地址，由一个标识符加上一个指定段内的相对地址的偏移量构成。

线性地址（虚拟地址）：逻辑地址到物理地址变换之间的中间层。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。

物理地址：指出目前CPU外部地址总线上的寻址物理内存的地址信号，用于内存级芯片的单元寻址，是地址变换的最终结果地址。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

一个逻辑地址由段标识符和段内偏移量两部分组成。段标识符是由一个16位长的字段组成，其中前13位是一个索引号，后3位是T1字段，索引号是段描述符的索引，很多个段描述符组成了一个段描述符表。先判断T1字段，看看这个段描述符究竟是在局部段描述符表(ldt)中还是全局段描述符表(gdt)中，通过索引号在描述符表内找到一个具体的段描述符。找到的段描述符加上偏移量即为线性地址。

## 7.3 hello的线性地址到物理地址的变换-页式管理

系统将虚拟页作为进行数据传输的单元。虚拟内存分割被成为虚拟页，物理内存也被分割为物理页，大小和虚拟页相同。任意时刻虚拟页都被分为三个不相交的子集：未分配的（VM系统还未分配的页）、缓存的（当前已经缓存在物理内存的已分配页）、未缓存的（当前未缓存在物理内存的已分配页）。

每次将虚拟地址转换为物理地址，都会查询页表来判断一个虚拟页是否缓存在DRAM的某个地方，如果不在DRAM的某个地方，通过查询页表条目可以知道虚拟页在磁盘的位置。页表将虚拟页映射到物理页，如图7-1。

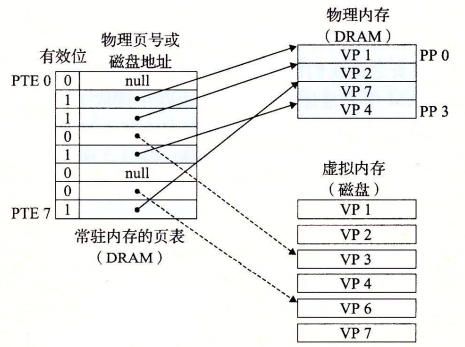


图7-1 页表

## 7.4 TLB与四级页表支持下的VA到PA的变换

先将虚拟地址分为虚拟页号（VPN），虚拟页偏移量（VPO），依据VPN（TLBT+TLBI）先在TLB中寻找，若找不到，则在高速缓存/内存中寻找，若找到对应的物理页号（PPN），再将PPN与VPO组合成物理地址，若还是查找不到，则需要缺页处理，在磁盘中查找，并将新的页更新入内存中，如图7-2。

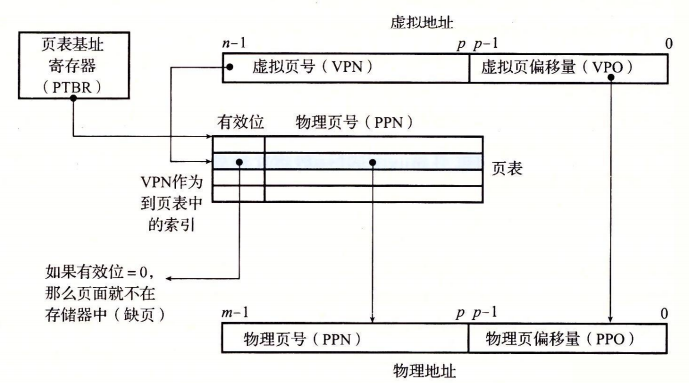


图7-2 使用页表的地址翻译

如图7-3，36位的VPN被分成4个9位的片，每个片被用做到一个页表的偏移量。CR3寄存器包含一级页表的基地址。VPN1提供一个一级PTE i的偏移量，它包含二级页表的基地址，VPN2再提供一个偏移量，以此类推。

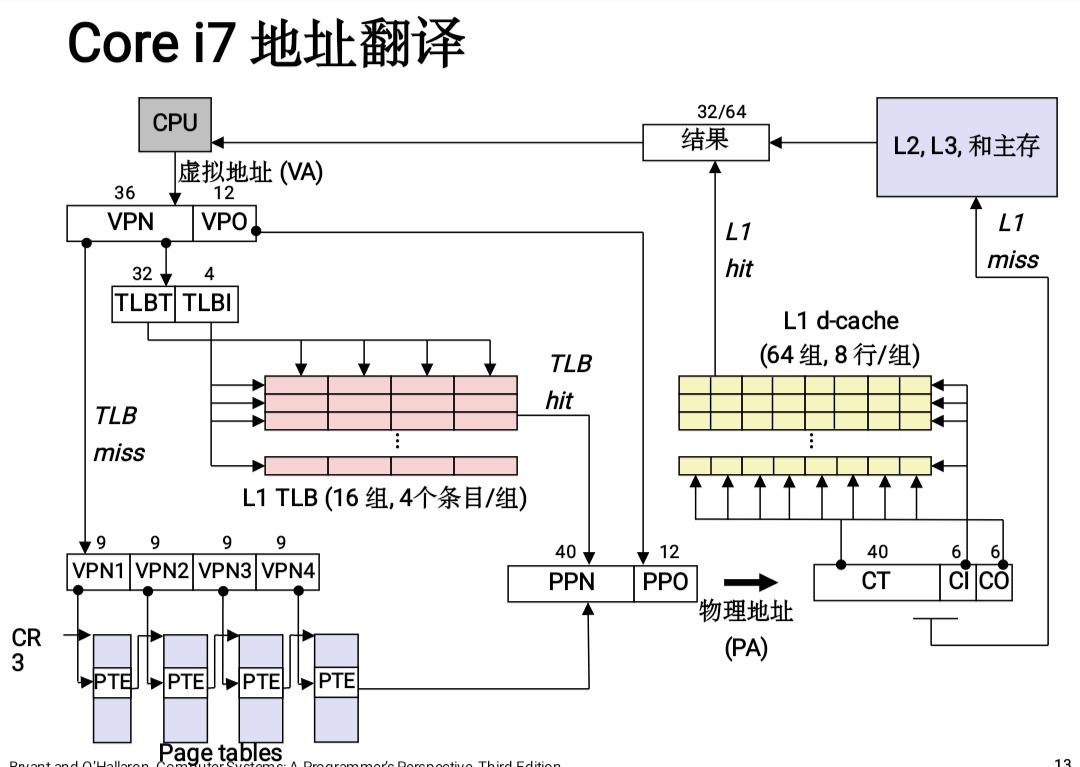


图7-3 TLB与四级页表支持下的VA到PA的变换

## 7.5 三级Cache支持下的物理内存访问

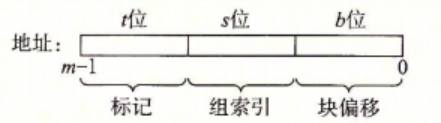


图7-4 物理地址的划分

如图7-4，将已得到的物理地址拆分为CT（标记），CI（组索引），CO(块偏移)，在L1 Cache中依据组索引，找到对应的组，再依据标记查找是否存在并判断是否有效，最后根据块偏移找到块，如图7-5。如果上述条件均满足则命中，否则按顺序对L2 Cache、L3 Cache、内存进行相同操作，直到命中，然后向上级返回。如果有空闲块则将目标块放置到空闲块中，否则将缓存中的某个块驱逐，将目标块放到被驱逐块的位置。

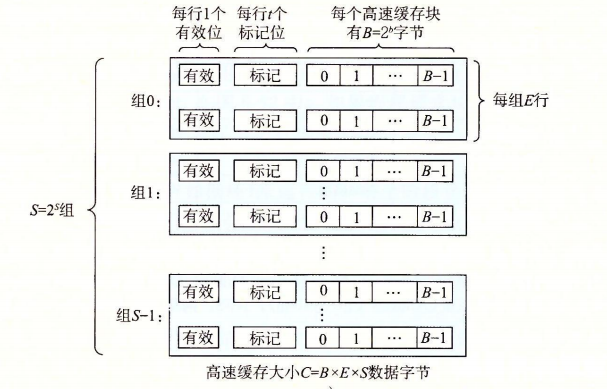


图7-5 高速缓存组织结构

## 7.6 hello进程fork时的内存映射

当fork 函数被shell调用时，内核为hello进程创建各种数据结构，并分配给它一个唯一的PID 。为了给hello进程创建虚拟内存，它创建了hello进程的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hello进程中返回时，hello进程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。

## 7.7 hello进程execve时的内存映射

execve 函数调用驻留在内核区域的启动加载器代码，在当前进程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，如图7-6。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，删除当前进程虚拟地址的用户部分中的已存在的区域结构。

(2)映射私有区域，为新程序的代码、数据、bss和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为hello文件中的.text和.data区，bss区域是请求二进制零的，映射到匿名文件，其大小包含在hello中，栈和堆地址也是请求二进制零的，初始长度为零。

(3)映射共享区域，hello程序与共享对象libc.so链接，libc.so是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

(4)设置程序计数器(PC)，execve做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

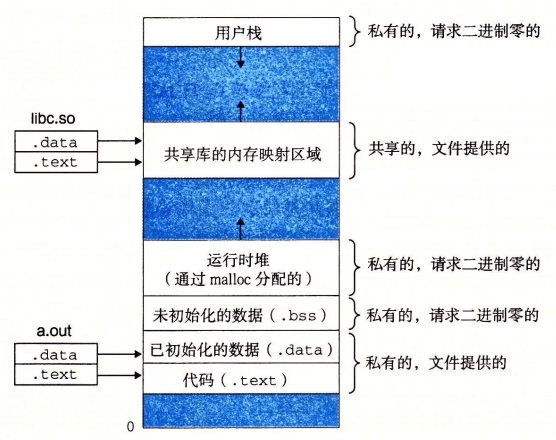


图7-6 加载器映射用户地址空间的区域

## 7.8 缺页故障与缺页中断处理

当指令引用一个虚拟地址，而与改地址相对应的物理页面不在内存中，因此必须从磁盘中取出时，就会发生缺页故障。缺页处理程序从磁盘加载适当的页面，然后将控制返回给引起故障的指令。当指令再次执行时，相应的物理页面已经驻留在内存中了，指令就可以没有故障地运行完成了。上述过程如图7-7所示。

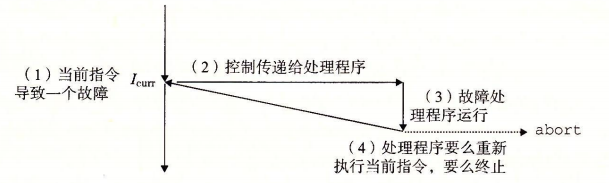


图7-7 缺页故障的处理

在发生缺页中断之后，系统会调用内核中的一个缺页处理程序。处理缺页要求硬件和操作系统内核协作完成，具体的操作过程如图7-8所示：

(1)处理器生成一个虚拟地址，并把它传送给MMU。

(2)MMU生成PTE地址，并从高速缓存/主存请求得到它。

(3)高速缓存/主存向MMU返回PTE。

(4)PTE中的有效位是零，所以MMU触发了一次异常，传递CPU中的控制到操作系统内核中的缺页异常处理程序。

(5)缺页处理程序确定出物理内存中的牺牲页，如果这个页面己经被修改了，则把它换出到磁盘。

(6)缺页处理程序页面调入新的页面，并更新内存中的PTE。

(7)缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU将引起缺页的虚拟地址重新发送给MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中，主存将所请求字返回给处理器。

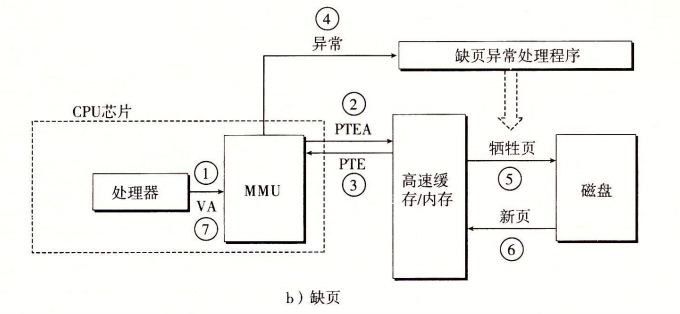


图7-8 缺页处理流程

## 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。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格。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malloc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。

下面介绍两种常见的分配器：

1.带边界标签的隐式空闲链表

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组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是8的倍数，且块大小的最低3位总是0。因此，我们只需要内存大小的29个高位，释放剩余的3位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明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。有效载荷后面是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。

我们称这种结构称为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。注意：此时我们需要某种特殊标记的结束块，可以是一个设置了已分配位而大小为零的终止头部。

Knuth提出了边界标记的技术，是在每个块的结尾处添加一个脚部，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，判断前面的一个块的起始位置和状态，这个脚部总是在据当前块开始位置一个字的距离。

考虑当分配器释放当前块时可能存在的所有情况：

1）前面的块和后面的块都是已分配的。

2）前面的块是已分配的，后面的块是空闲的。

3）前面的块是空闲的，而后面的块是已分配的。

4）前面的和后面的块都是空闲的。

按照图7-9分类处理即可：

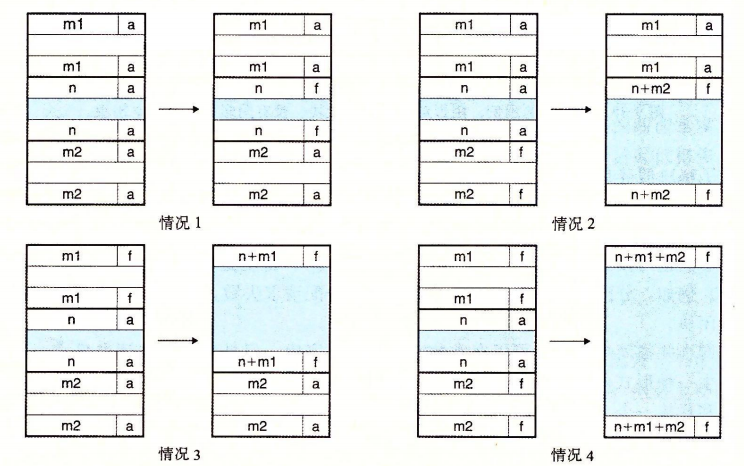


图7-9 使用边界标记的4种合并情况

然而这种方法也存在一个潜在的缺陷。它要求每个块都保持一个头部和一个脚部，在应用程序操作许多个小块时，会产生显著的内存开销。

幸运的是，有一种非常聪明的边界标记的优化方法，能够使得在已分配块中不再需要脚部。把前面块的已分配位/空闲位存放在当前块中多出来的低位中，那么已分配的块就不需要脚部了，这样我们就可以将这个多出来的空间用作有效载荷了。不过空闲块仍然需要脚部。

2.显式空间链表

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。例如，堆可以组织成一个双向空闲链表。在每个空闲块中，都包含一个前驱和后继指针，如图7-10。

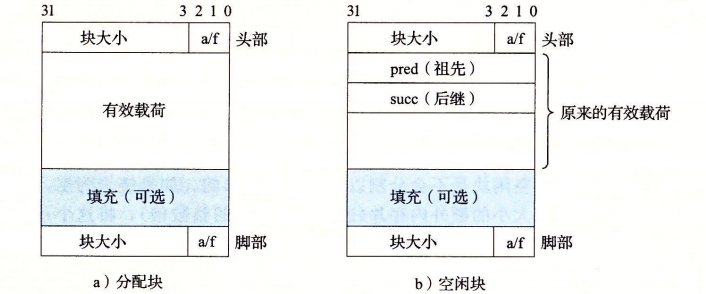


图7-10 使用双向空闲链表的堆块的格式

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的线性时间。不过，释放一个块的时间也可以是线性的，也可能是某个常数，这取决于我们所选择的空闲链表中块的排序策略。

一种方法是使用后进先出（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过的块。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。

另一种方式是按照地址顺序来维护链表，其中链表上每一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搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比LIFO排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

## 7.10本章小结

本章主要介绍了hello的存储管理，讲解了存储器地址空间，段式管理和页式管理的机制，TLB与四级页表支持下的VA到PA的变换和三级Cache支持下的物理内存访问，并且回顾了hello进程fork时和execve时的内存映射，分析了缺页故障与缺页中断处理的流程，最后介绍了动态存储分配管理的方式。这一章的知识有助于编写高速缓存友好代码，优化程序。

# 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（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。这种将设备映射为文件的方式，允许Linux内核引出一个简单、低级的应用接口，称为Unix I/O。

## 8.2 简述Unix IO接口及其函数

Unix I/O 接口使得所有输入和输出都能以一种统一且一致的方式来执行：

(1)打开文件。一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个I/O设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件。内核记录有关这个打开文件的所有信息。

(2)Linux shell创建的每个进程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）。头文件<unistd.h>定义了常量STDIN\_FILENO、STDOUT\_FILENO和STDERR\_FILENO，它们可用来代替显式的描述符值。

(3)改变当前的文件位置。对于每个打开的文件，内核保持着一个文件位置k，初始为 0。这个文件位置是从文件开头起始的字节偏移量。应用程序能够通过执行seek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≥m时，执行读操作会触发EOF条件，应用程序能检测到这个条件。类似地，写操作就是从内存复制n>0个字节到一个文件，从当前文件位置k开始，然后更新k。

(5)关闭文件。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池中。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。

Unix I/O函数：

(1)int open(char \*filename, int flags, mode\_t mode)：进程通过调用open函数来打开一个已存在的文件或是创建一个新文件。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。flags参数指明了进程打算如何访 问这个文件。mode参数指定了新文件的访问权限位。

(2)int close(int fd)：进程通过调用close函数来关闭一个打开的文件。

(3)ssize\_t read(int fd, void \*buf, size\_t n)：read函数从描述符为fd的当前文件位置复制最多n个字节到内存位置buf。返回值-1表示一个错误，0表示EOF。否则返回值表示的是实际传送的字节数量。

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

## 8.3 printf的实现分析

首先查看printf 的代码：

int printf(const char \*fmt, …)

{

int i;

char buf[256];

va\_list arg = (va\_list)((char\*)(&fmt) + 4 );

i = vsprintf(buf, fmt, arg);

write(buf, i);

return i;

}

其中va\_list 的定义为：typedef char \*va\_list；这说明它是一个字符指针。

(char\*)((&fmt) + 4)表示的是…中的第一个参数。

vsprintf返回的是要打印出来的字符串的长度，它的作用就是格式化。它接受确定输出格式的格式字符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。

再看看write:

     mov eax, \_NR\_write

     mov ebx, [esp + 4]

     mov ecx, [esp + 8]

     int INT\_VECTOR\_SYS\_CALL

其中INT\_VECTOR\_SYS\_CALL的实现为：init\_idt\_desc(INT\_VECTOR\_SYS\_CALL,DA\_3861Gate,sys\_caII,PRIVILEGE\_USER);

其中int INT\_VECTOR\_SYS\_CALL表示要通过系统来调用sys\_call这个函数。

sys\_call函数如下：

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

call save是为了保存中断前进程的状态。

sys\_call最终实现显示格式化的字符串的功能。它将字符串中的字节从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的ASCII码。字符显示驱动子程序将通过ASCII码在字模库中找到点阵信息将点阵信息存储到vram中。显示芯片会按照一定的刷新频率逐行读取vram，并通过信号线向液晶显示器传输每一个点（RGB分量），于是打印字符串就显示在了屏幕上。

## 8.4 getchar的实现分析

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

查看getchar的代码如下：

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;

}

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

## 8.5本章小结

本章主要介绍了 Linux 的 IO 设备管理方法，引出了Unix I/O 接口的概念，又进一步介绍了Unix I/O的函数，最后简单分析了printf函数和getchar函数的实现。

# 结论

至此，hello已经走完它短暂的一生。我们在这里对它的人生历程再进行一次回顾。

Hello出生于名为hello.c的文本文件里，程序员通过对键盘的一次次敲击赋予了它生命。

刚出生不久的hello来到了预处理器的怀抱里，它在这里被修改——插入指定文件，扩展指定的宏，变成了一个更完整的文本文件hello.i。

然后，在编译器的帮助下，它把自己从原本的高级语言源程序变成了汇编语言程序hello.s，努力向着被机器识别并执行的方向前进着。

接下来，汇编器再助它一臂之力，将它翻译成机器语言指令，并把这些指令打包成可重定位目标程序的格式保存在二进制文件hello.o中。

虽然已经能被机器识别并运行了，hello却发现自身还需要进一步完善。它又去寻求了链接器的帮助，经过静态链接和动态链接的它终于成为了可执行目标文件hello。

当我们在shell中输入指令执行hello的那一刻，hello变身成为进程。这一刻意义巨大，这标志着hello完成了人生中的第一个阶段——P2P(From Program to Process)。

Shell调用fork函数生成一个新的子进程，这个子进程通过调用execve函数启动加载器加载hello，为其分配虚拟内存空间，再将虚拟内存空间映射到物理内存空间，然后进入 main函数执行目标代码。CPU为运行的hello分配时间片，执行逻辑控制流。

在运行过程中，OS与MMU为hello的VA到PA操碎了心，TLB、四级页表和三级Cache等等各显神通为hello加速。

IO帮助hello中的内容顺利显示到屏幕上。

程序运行结束后，shell接收信号，回收hello进程，内核将会清除与hello相关的数据。Hello走完人生最后一个阶段——020(From Zero to Zero)，完美谢幕！

通过对hello程序人生的分析，我感受到计算机系统的设计是环环相扣、精妙无比的。这次的计算机系统漫游使我对计算机的底层实现有了更深的理解，还学习到了避免由计算机表示数字方式引起数字错误的实践技巧、避免诸如缓冲区溢出等安全漏洞的方法、优化代码以充分利用现代处理器和存储器系统的设计。有了软件和硬件的相互配合，才有了高效的计算机系统。深入了解这些组件是如何工作的以及这些组件如何影响程序的正确性和性能，能大大提升计算机从业者的技能。

# 附件

|  |  |
| --- | --- |
| 文件名 | 文件作用 |
| hello.i | hello.c预处理后的文本文件，用于分析预处理结果 |
| hello.s | hello.i编译后的汇编文件，用于分析编译结果 |
| hello.o | hello.s汇编后的可重定位目标文件，用于分析汇编结果 |
| elf.txt | hello.o的ELF格式，用于分析hello.o |
| objdump.txt | hello.o的反汇编代码，用于分析hello.o |
| hello | hello.o链接后的可执行目标文件 |
| elf2.txt | hello的ELF格式，用于分析hello |
| objdump2.txt | hello的反汇编代码，用于分析hello |

# 参考文献

[1] Randal E. Bryant, David R. O’Hallaron. 深入理解计算机系统（原书第3版）[M].机械工业出版社，2016.7

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

[3] <https://blog.csdn.net/weixin_30438795/article/details/117124493>

[4] <https://blog.csdn.net/wohenfanjian/article/details/105869692>