

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 计算机科学与技术学院

学　　 号 1180300626

班　　 级 1836101

学 生 丛日东

指 导 教 师 刘宏伟

**计算机科学与技术学院**

**2019年12月**

**摘 要**

本文通过遍历hello.c在Linux下完整的一个生命周期，从预处理，编译，汇遍等初始过程到动态链接，内存管理，shell运行等较为高级的程序概念，到最后的更深层次的异常控制流，虚拟内存，IO管理等内容，将计算机系统中重要基础的概念结合实例具体的描述了一遍。本文旨在将课本知识更好的运用到实际操作过程中，使操作系统这一概念深入人心。

**关键词：**操作系统；动态链接；虚拟内存；内存管理；

**（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

P2P: From Program to Process

描述的是hello从一个程序变成一个进程的过程。

Gcc编译器读取helllo.c，经过预处理，编译，汇编分别形成hello.i文本文件，hello.s汇编语言文件，hello.o可重定位目标程序二进制文件。之后经过链接器链接形成可执行文件hello，也就是程序。通过执行可执行文件，操作系统使用fork函数形成一个子进程，分配相应的内存资源，通过execve加载该程序，从而完成从程序

到进程的转变过程。

O2O: From Zero-0 to Zero -0

描述的是hello进程由无到有再到无这一过程。

Shell在命令行中接收到hello指令，在内置指令中查找，由于不存在便通过fork产生子进程，通过execve在子进程中加载helllo程序。加载过程中，首先为hello创建代码段，数据段，用户栈等内存结构，然后执行映射共享区域，设置程序计数器等操作。进入main函数之后程序会使用到内存中的数据，通过各级存储，输出想要的信号。函数的连续运行离不开系统的信号处理，逻辑控制流等。Hello执行结束之后shell负责回收，实现了从0到0的过程。

## 1.2 环境与工具

## 硬件环境：X64 CPU ，2.50GHz ， 8G RAM

## 软件环境：Windows 10 64位 ，Vmware 14 ，Ubuntu 16.04 LTS 64 位

## 开发工具：gcc ,gedit , DevC++ , gdb，edb

## 1.3 中间结果

hello.c ：hello源代码

hello.i ：预处理后的文本文件

hello.s ：hello.i编译后的汇编文件

hello.o ：hello.s汇编后的可重定位目标文件

hello.elf : hello.o文件的ELF格式

hello\_out.elf : hello文件的ELF格式

hello\_objdump ：hello的反汇编代码

hello.0\_objdump ：hello.o的反汇编代码

hello ：链接后的可执行文件。

## 1.4 本章小结

本章介绍了P2P和O2O 的含义，交代了实验的运行环境以及工具等内容，概述了hello在系统中的生命周期，是接下来所有章节的开端。

**（第1章0.5分）**

# 第2章 预处理

## 2.1 预处理的概念与作用

## 概念：预处理是编译器在编译开始之前调用预处理器来执行以#开头的命令（读取头文件、执行宏替代等）、删除注释、包含其他文件、执行条件编译、布局控制等修改原始的C程序，生成以.i结尾的文本文件。

## 作用：

## ​ 1. 文件包含。例如hello.c中第一行的#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h文件的内容，并把它直接插入到程序文本中；

## ​ 2. 删除注释；

## ​ 3. 执行宏替代。宏是对一段重复文字的简短描写，例如#define MAX 2147483647在预处理中会把所有MAX替代为2147483647，#define MAX(x,y) ((x)>(y))?(x): (y)在预处理中会把所有MAX(x,y)替换为((x)>(y))?(x): (y)。

## ​ 4. 条件编译。是根据实际定义宏（某类条件）进行代码静态编译的手段。可根据表达式的值或某个特定宏是否被定义来确定编译条件。

5. 布局控制。布局控制的主要功能是为编译程序提供非常规的控制流信息。

## 2.2在Ubuntu下预处理的命令

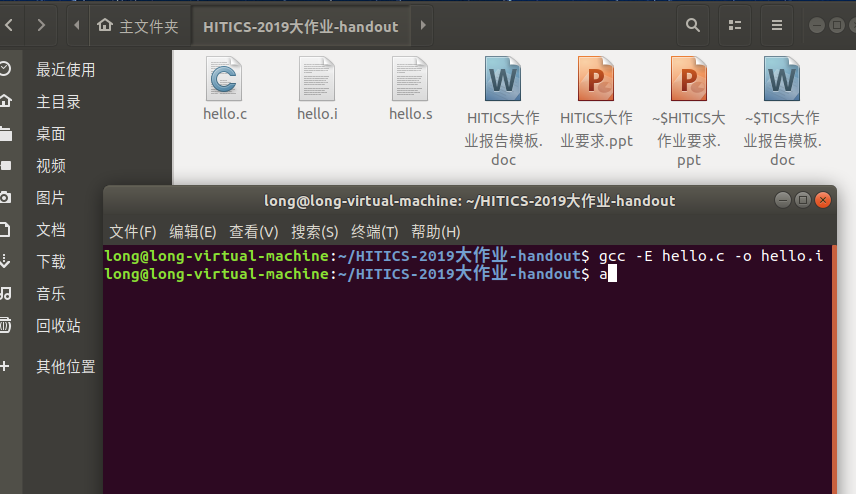


图2.2.1 对hello.c进行预处理

## 2.3 Hello的预处理结果解析

经过预处理，系统生成hello.i文件。其中首先包含了对头文件的解析以及对宏进行展开，引入了typedef关键字，结构体等特殊的数据结构类型。文件的内容变得更加丰富。以下是部分截图。

图2.3.1hello.i文件部分内容



图2.3.2hello.i文件部分内容

## 

## 2.4 本章小结

本章对hello.c源文件进行预处理，并通过查看预处理之后的文件内容，更加深入的了解到预处理的概念和作用。完成该阶段转换后可以进行下一阶段的汇编。

**（第2章0.5分）**

# 第3章 编译

## 3.1 编译的概念与作用

概念： 编译是利用编译程序从预处理文本文件产生汇编程序（文本）的过程。主要包含五个阶段：词法分析；语法分析；语义检查、中间代码生成、目标代码生成。

作用：编译作用主要是将文本文件hello.i翻译成文本文件hello.s，并在出现语法错误时给出提示信息，执行过程主要从其中三个阶段进行分析：

1. 词法分析。词法分析的任务是对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号，把作为字符串的源程序改造成为单词符号串的中间程序；

2. 语法分析。语法分析器以单词符号作为输入，分析单词符号串是否形成符合语法规则的语法单位，如表达式、赋值、循环等，最后看是否构成一个符合要求的程序，按该语言使用的语法规则分析检查每条语句是否有正确的逻辑结构，程序是最终的一个语法单位；

3. 目标代码生成。目标代码生成器把语法分析后或优化后的中间代码经汇编程序汇编生成汇编语言代码，成为可执行的机器语言代码。

## 3.2 在Ubuntu下编译的命令

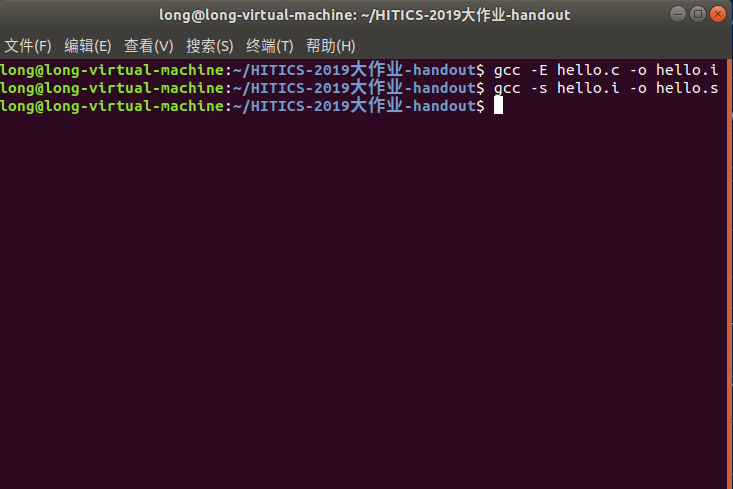


图3.2.1对hello.i文件进行编译

## 3.3 Hello的编译结果解析

3.31数据

1.全局变量

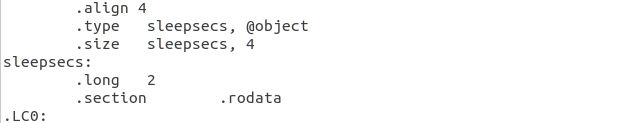


图3.31.1全局变量sleepsecs

全局变量sleepsecs在这里被声明为初始化的全局变量，放在.rodata节中

2.局部变量

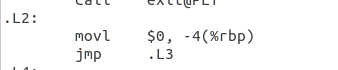


图3.31.2局部变量i

局部变量i被初始成0

3.常量

立即数表示

4.字符串常量

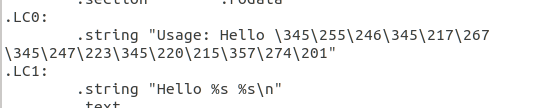
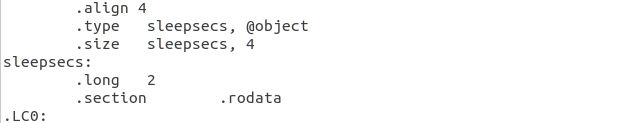


图3.31.4字符串常量

在.LC0和.LC1中出现的两个.string字符串常量

3.32赋值

全局变量sleepsecs被赋值为2



3.32.1全局变量的赋值

3.33类型转换

在.c源程序中sleepsecs的声明中默认将2.5转换为2，赋初值的时候可以看出来。

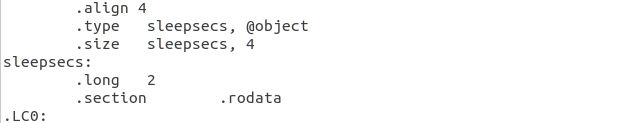


图3.33.1类型转换

3.34算数操作

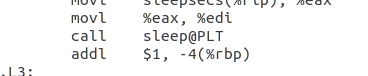
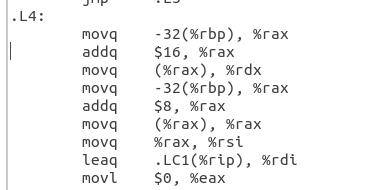


图3.34.1++操作

adl执行的是++操作。

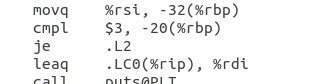
3.35数组/指针/结构操作



3.35.1数组操作

(%rax)执行之前已经将rbp-32的值赋给了rax,所以此时的取地址是为了得到相邻的数组的值argv[2].

3.36控制转移



3.36.2控制转移

可以明确看出je有控制转移动作。

3.37函数操作

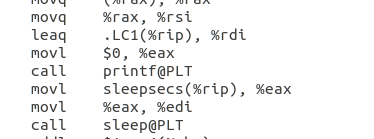


图3.37sleep函数

调用printf函数和sleep函数

## 3.4 本章小结

本章通过gcc编译器译生成hello.s文件，hello.s文件使用汇编语言，对C语言的各个数据类型以及各类操作进行处理，包括控制转移，函数操作，类型转换等。经过文件的解析，我们进一步了C语言与汇编语言的对应关系，可以进行下一步的汇编操作。

**（第3章2分）**

# 第4章 汇编

## 4.1 汇编的概念与作用

## 概念：把汇编语言翻译成机器语言的过程称为汇编。

## 作用：汇编器（as）将hello.s翻译成机器语言指令，并把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保存在二进制目标文件hello.o中。

## 4.2 在Ubuntu下汇编的命令

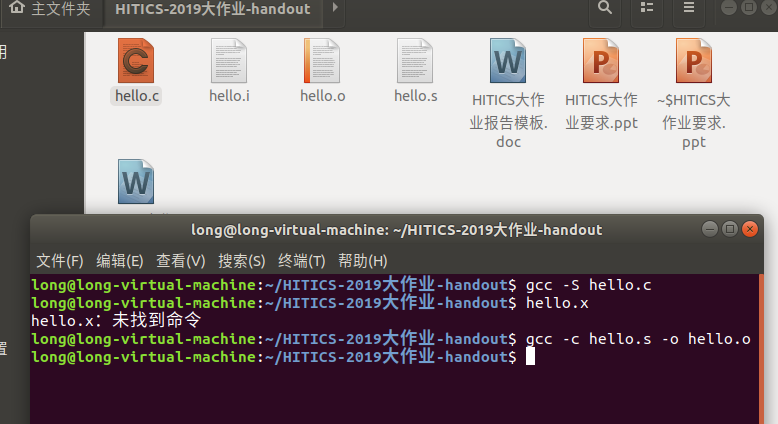


图4.2.1对hello.s文件进行汇编

## 4.3 可重定位目标elf格式

首先使用readelf –a hello.o > hello.elf 生成hello.o文件的ELF格式。分析其组成各部分。

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、可执行或共享的）、机器类型（如x86-64）、节头部表（section header table）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头目（entry）。

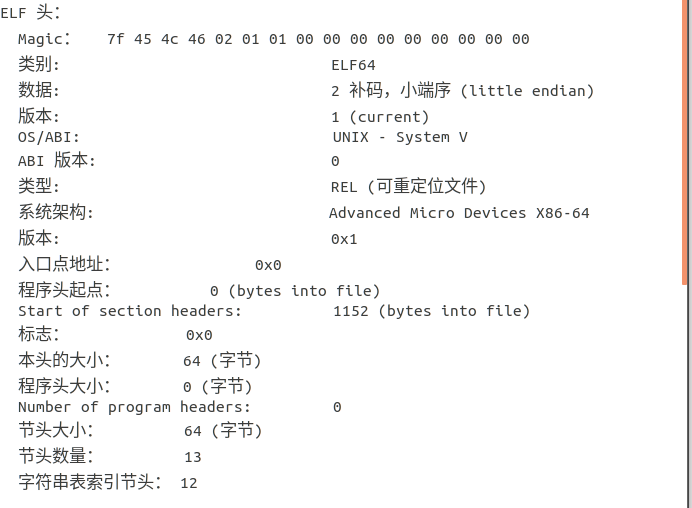
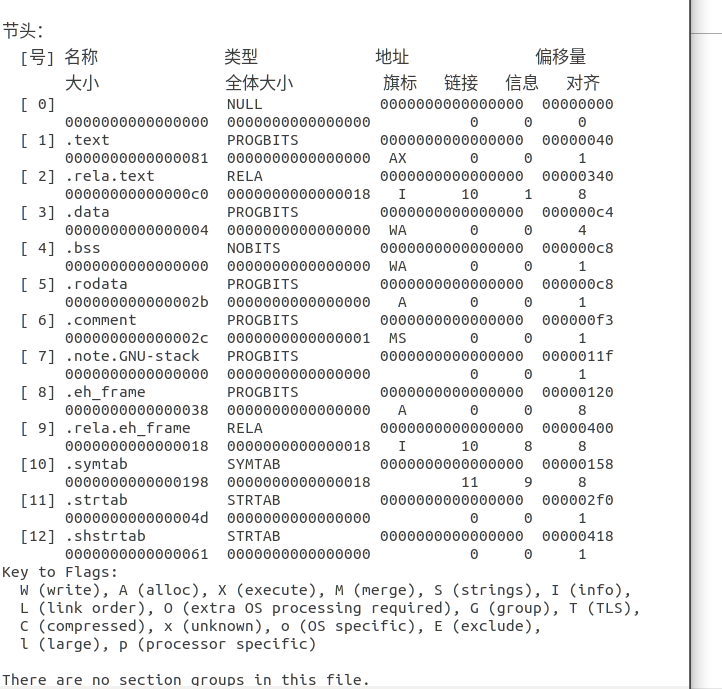


图4.3.1ELF头

节头部表记录了各节名称、类型、地址、偏移量、大小、全体大小、旗标、连接、信息、对齐信息。

 图4.3.2节头表

.rela.text记录了一个.text节中位置的列表，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。一般而言，任何调用外部函数或者引用全局变量的指令都需要修改。另一方面，调用本地函数的指令则不需要修改。

在hello.o的重定位节中包含puts、exit、printf、sleep、getchar函数以及全局变量sleepsecs，还有.rodata节（包含printf的格式串）的偏移量、信息、类型、符号值、符号名称及加数。rela.eh\_frame记录了.text的信息。

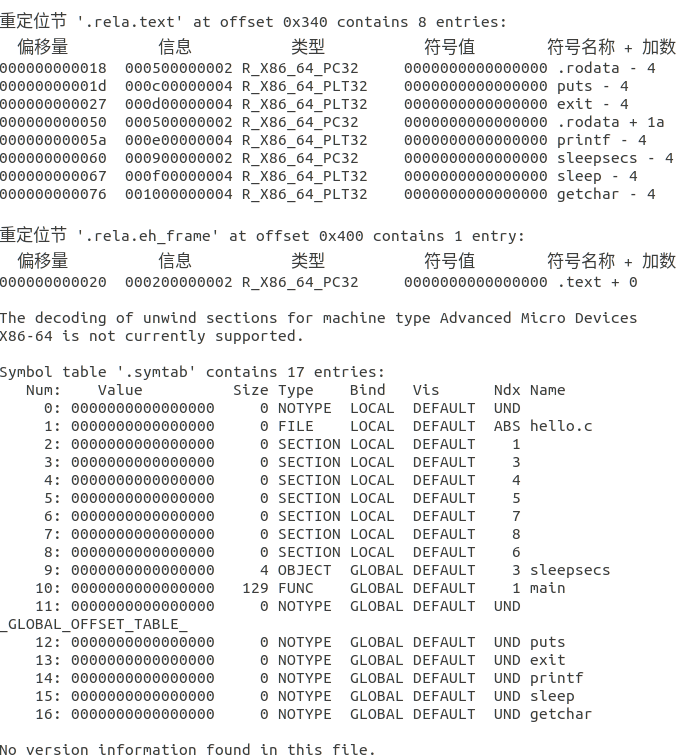
.symtab存放着程序中定义和引用函数和全局变量的信息。且不包含局部变量的条目。hello.elf的.symtab表和重定位节如下

图4.3.3重定位节

## 4.4 Hello.o的结果解析

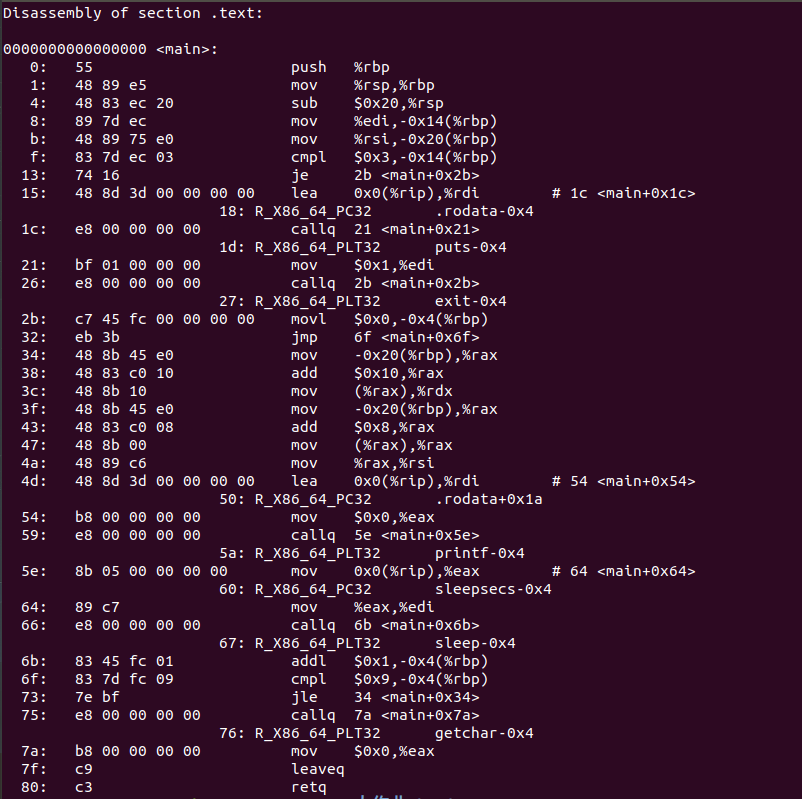
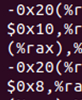


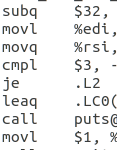
图4.4.1hello.o反汇编代码

不同之处：

1. 机器语言中的操作数：



反汇编如图所示，立即数是十六进制的



.s文件如图所示，立即数是十进制的

2.分支转移



反汇编如图所示，分支转移用地址表示



.s文件如图所示，分支转移用助记符表示

3.函数调用



反汇编如图所示，call后面跟着的是函数的地址



.s文件如图所示，call后面跟着的是函数的名称

## 4.5 本章小结

本章完成了对hello.s的汇编工作，同时通过生成elf格式具体的分析了hello.o的各个组成部分。通过反汇编生成的结果与hello.s的汇编程序代码进行对照分析，更加深刻地了解到二者的差别以及汇编的作用。通过完成本阶段工作，可以进行下一阶段的链接工作。

**（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称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：链接可以执行于编译时，也就是在源代码被翻译成机器代码时；也可以执行于加载时，也就是在程序被加载到内存并执行时；甚至执行于运行时，也就是由应用程序来执行。早期计算机系统中链接时手动执行的，在现代系统中，链接器由链接器自动执行。链接器使得分离编译成为可能。开发过程中无需将大型的应用程序组织委员一个巨大的源文件，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。

## 5.2 在Ubuntu下链接的命令

## ld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

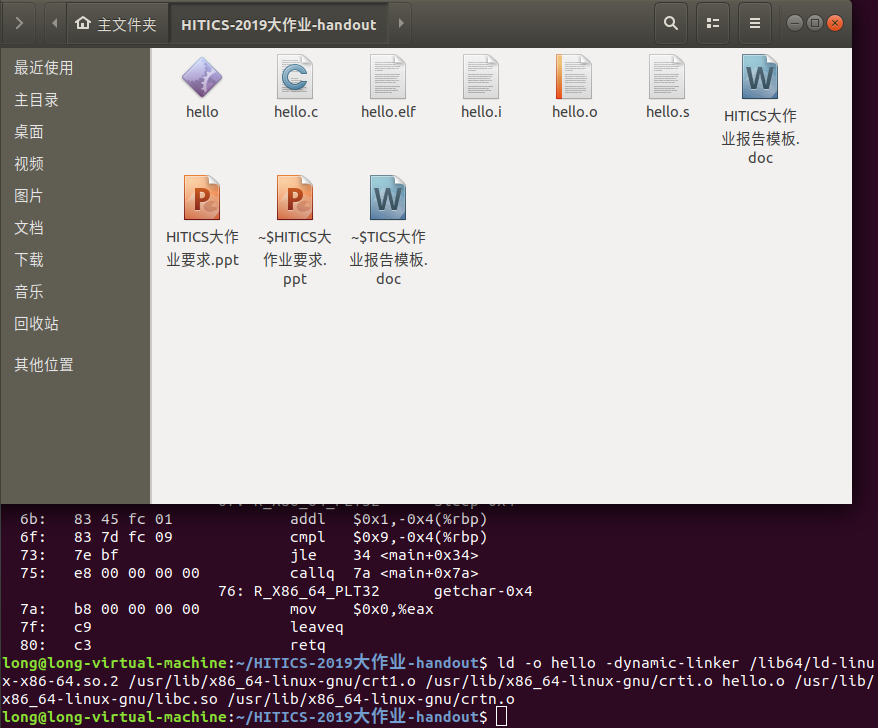


图5.2.1链接

## 5.3 可执行目标文件hello的格式

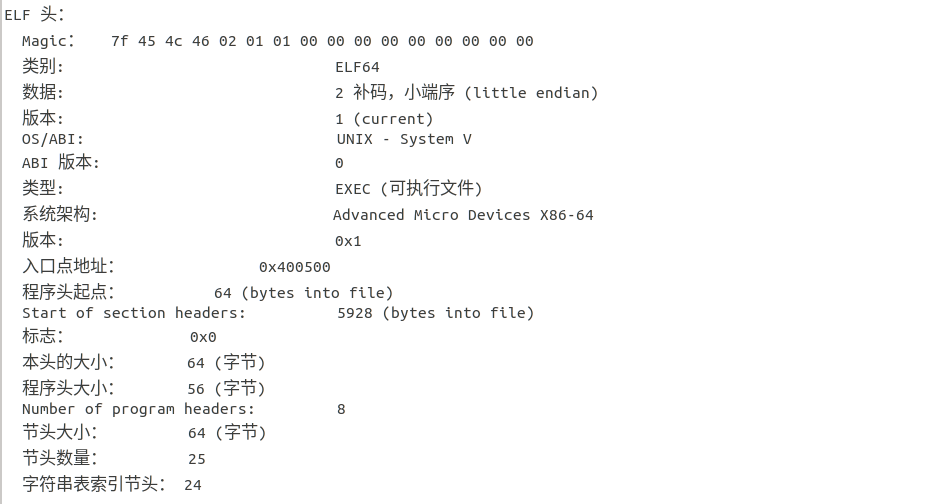


图5.3.1 hello的ELF头

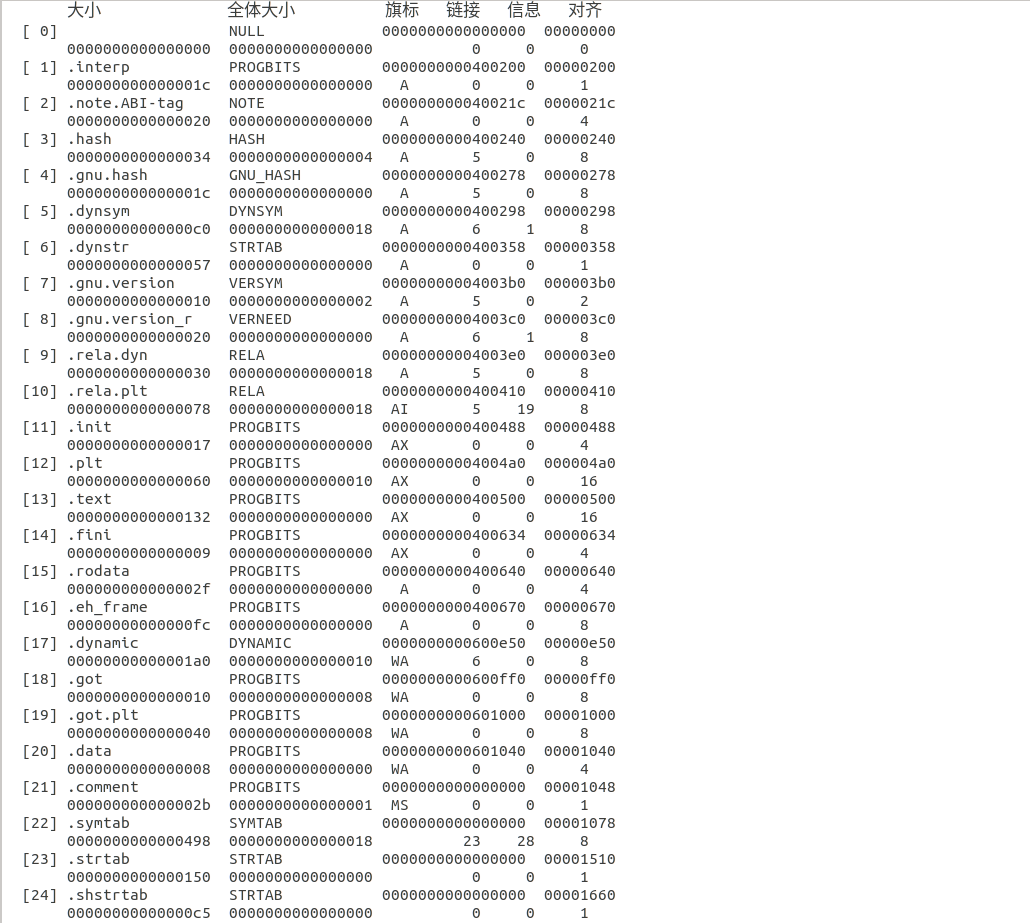


图5.3.2 hello的节头表

hello的ELF头和hello.o的ELF头相近，但其他部分内容前者更多。

## 5.4 hello的虚拟地址空间

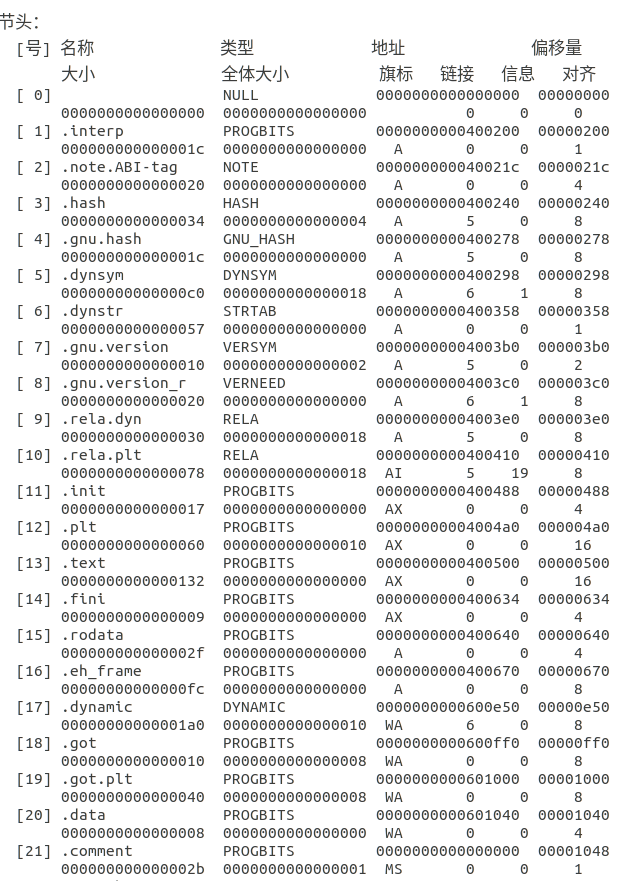


图5.4.1节头

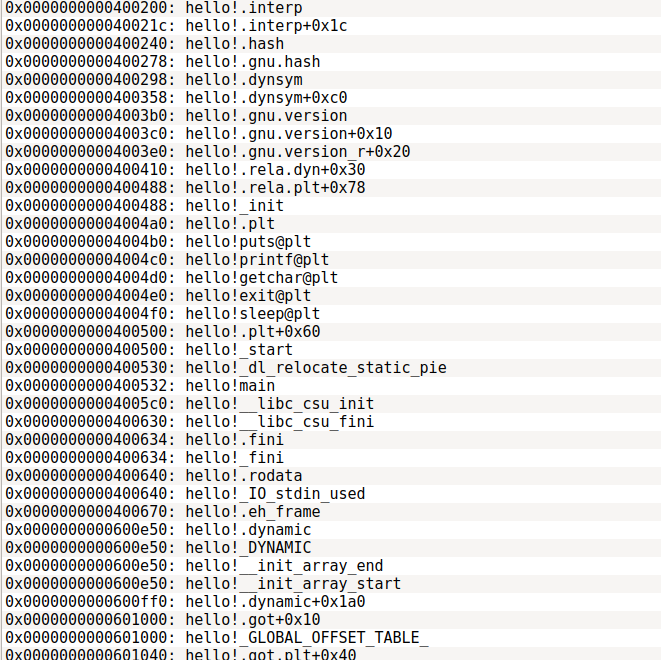


图5.4.2edb下的符号表

如图所示 ，0x400200对应的是偏移量200处。

## 5.5 链接的重定位过程分析

1.文件内容：

hello反汇编文件比hello.o反汇编文件多出很多节，可以看到hello.o反汇编文件只有一个.text节，而hello反汇编文件中包含.init .plt .got节等等。

1. 函数调用

hello调用的时候采用了确定的地址，而hello.o的地址实际上还是相对地址。

重定位过程合并输入模块，并为每个符号分配运行时地址，主要有以下两步：

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新的聚合节。

2.重定位节中的符号引用。链接器依赖于可重定位目标文件的重定位条目，修改代码节和数据节中对每个符号的引用，使得它们指向正确运行时的地址。

## 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 |
| hello!main |
| hello!puts@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 |
| hello!exit@plt |
| libc-2.27.so!exit |

## 5.7 Hello的动态链接分析

首先由节头表可以得到GOT节由0x600ff0开始

## 在运行dl\_init前，GOT表中存放的都是对应PLT条目的第二条指令

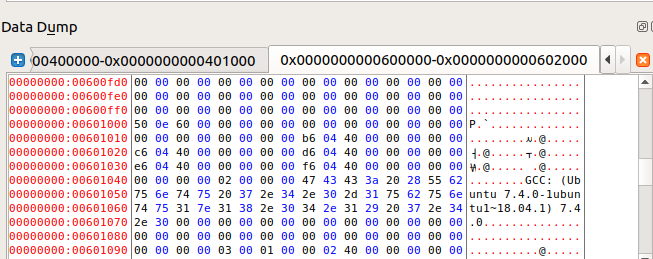
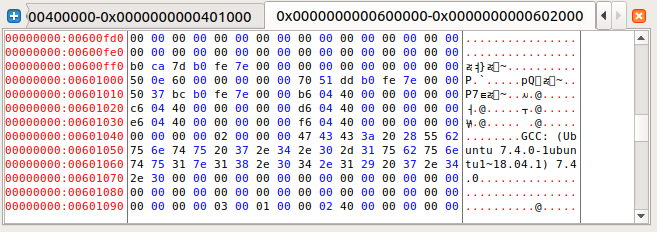


图5.7.1运行之前GOT表

在运行dl\_init后，GOT表中存放的就是对应的函数的地址



5.7.2运行之后GOT表

## 5.8 本章小结

本章完成了对hello.o的链接工作。通过使用ld链接命令可以生成hello.out 文件，同时分析了elf头文件的信息，以及利用了edb来具体的了解hello执行过程以及动态链接过程中的细节，更加深入的理解了链接和重定位的作用。通过完成本阶段的链接的工作，我们可以进一步分洗hello在执行过程中的进程管理。

**（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：进程的经典定义就是一个执行中的程序的实例。系统中的每个程序都运行在某个进程的上下文中。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、以及打开文件描述符的集合。

## 作用：进程给应用程序提供的关键抽象有两种：

## a) 一个独立的逻辑控制流，提供一个假象，程序独占地使用处理器。

## b) 一个私有的地址空间，提供一个假象，程序在独占地使用系统内存。

## 6.2 简述壳Shell-bash的作用与处理流程

## 作用：Shell俗称壳，是指“为使用者提供操作界面”的软件（命令解析器），它接收用户命令，然后调用相应的应用程序。

## 处理流程：

## 1.打印提示信息

## 2.等待用户输入

## 3.接受命令

## 4.解释命令

## 5.找到该命令，执行命令，如果命令含有参数，输入的命令解释它

## 6.执行完成，返回第一步

## 6.3 Hello的fork进程创建过程

Shell通过分析命令行中的./hello首先在内置命令中查找是否存在hello命令，因为不存在便通过调用fork产生一个子进程。

## 6.4 Hello的execve过程

Fork产生子进程之后，使用execve函数在当前进程中载入新的hello进程。

## 6.5 Hello的进程执行

Hello执行之前内核正在调他进程，hello进程被加载后内核挂起其他进程，开始执行hello。Hello有独立的逻辑控制流，与其他进程的逻辑控制流将会有部分的重叠，被称为并发。Hello执行它的控制流的每一时间段叫做时间片。当hello执行到sleep函数时会被挂起secs秒，内核调度其他进程。时间结束后hello进程重新进入待执行队列中等待内核调度。内核调度之后执行到getchar函数的时候调用read函数，陷入内核。内核中的陷阱处理程序请求来自磁盘处理器的DMA传输，传输结束之后磁盘发出中断信号，表示数据传送完成。内核判断其他进程执行时间足够之后，将会进行上下文切换，再次切换到hello进程，传递到read之后的指令，hello进程继续执行。

## 6.6 hello的异常与信号处理

可能会出现的异常：

中断：信号SIGTSTP，默认行为是停止直到下一个SIGCONT

终止：信号SIGINT，默认行为是终止

1.hello程序正常运行，结束后按回车

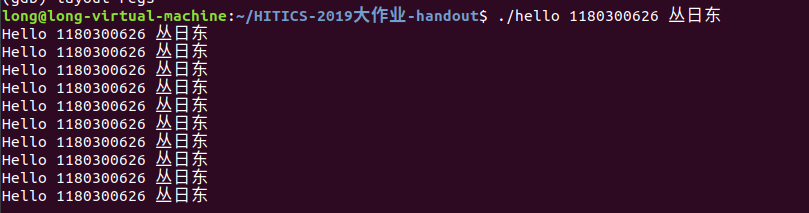


图6.6.1正常运行

2.hello运行过程中按下Ctrl+Z。父进程收到SIGSTP信号，hello进程被挂起



图6.6.2按下Ctrl+Z

3.hello运行过程中按下Ctrl+C。父进程收到SIGINT信号，hello进程被终止并回收

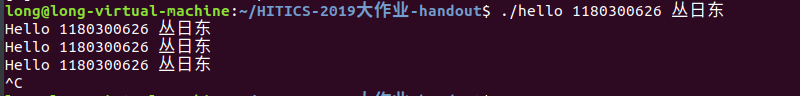


图6.6.3按下Ctrl+C

4.hello运行过程中按下多次回车，可以看到回车键会被读入。并且在hello执行结束之后其他回车被当作命令输入。

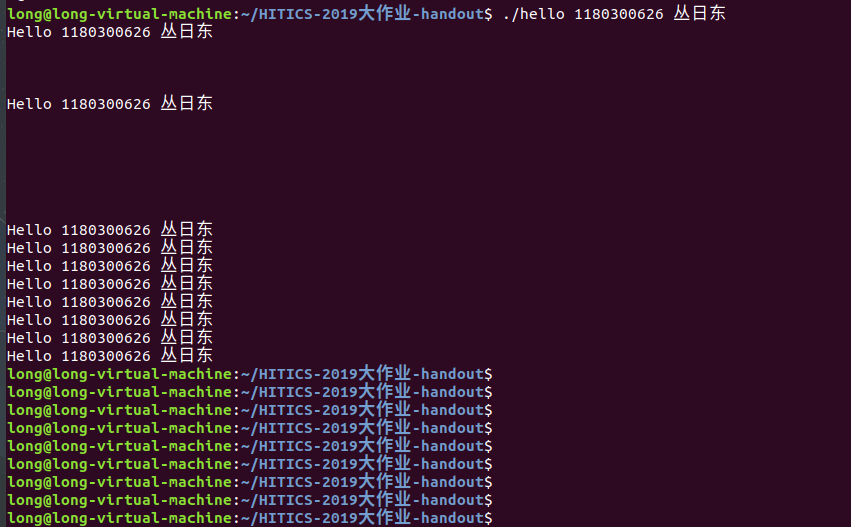


图6.6.4按下多次回车

5.运行时随意输入字符。随意输入的字符会被读入。并且在hello结束之后被当作命令输入。



图6.6.5运行时随意输入字符

6. Ctrl+Z运行之后运行ps命令。Hello进程被挂起，ps命令列出当前所有的进程。

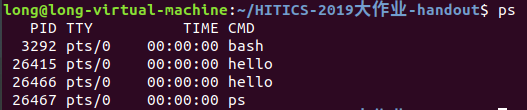


图6.6.6运行ps

7. Ctrl+Z运行之后执行jobs命令。Jobs列出所有已经启动的任务当前的状态。



图6.6.7运行jobs

8. Ctrl+Z运行之后执行pstree命令。 pstree列出进程之间的树状图。

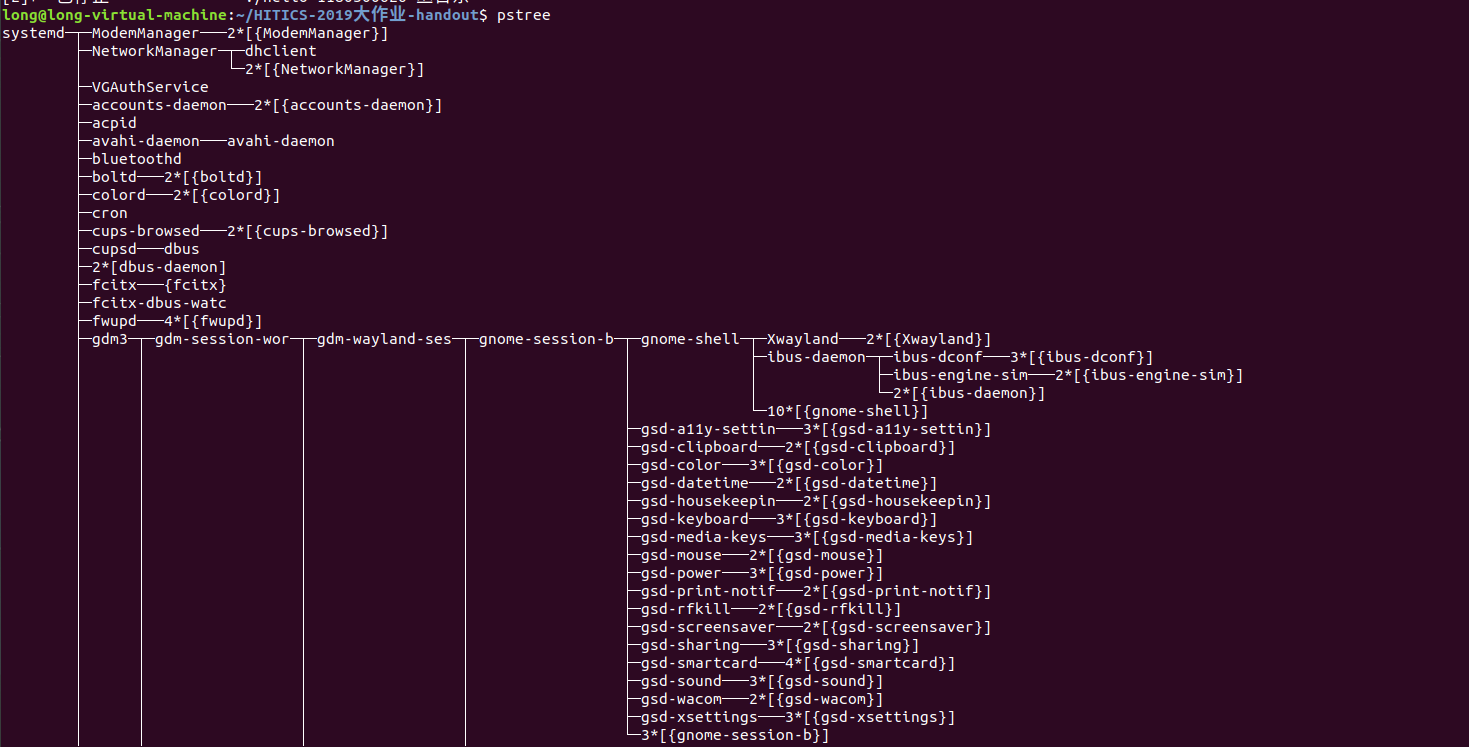


图6.6.8运行pstree命令

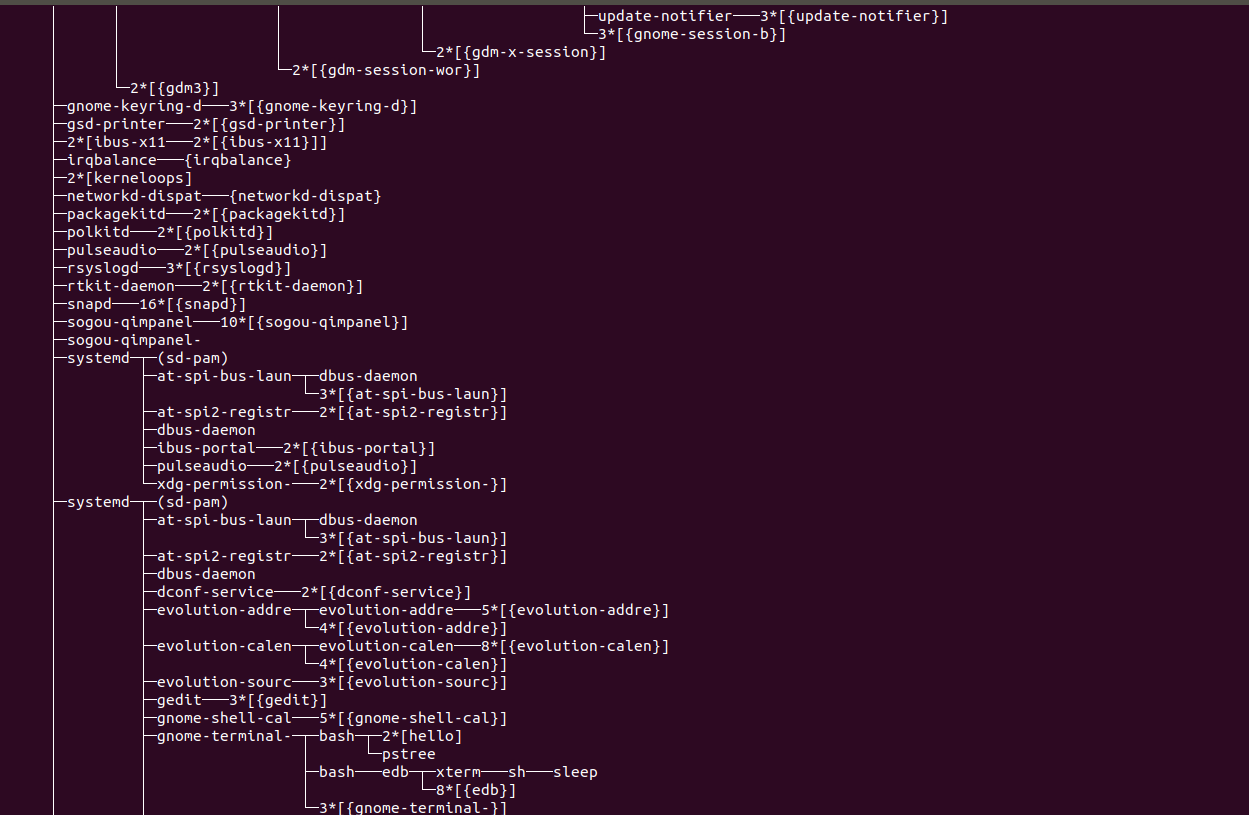


图6.6.9运行pstree命令

## 6.7本章小结

本章介绍了fork进程和execve进程的具体过程，并且对其中的虚拟内存映像等结构做出说明。同时阐述了逻辑控制流中内核的调度及上下文切换等机制。最后介绍了执行过程中可能的异常和信号处理。

**（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（Logical Address） 是指由程序产生的与段相关的偏移地址部分。例如，你在进行C语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于你当前进程数据段的地址，不和绝对物理地址相干。只有在Intel实模式下，逻辑地址才和物理地址相等（因为实模式没有分段或分页机制,Cpu不进行自动地址转换）；逻辑也就是在Intel 保护模式下程序执行代码段限长内的偏移地址（假定代码段、数据段如果完全一样）。应用程序员仅需与逻辑地址打交道，而分段和分页机制对您来说是完全透明的，仅由系统编程人员涉及。应用程序员虽然自己可以直接操作内存，那也只能在操作系统给你分配的内存段操作。在hello程序里面就是hello.o里的相对偏移地址。

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

虚拟地址：一个带虚拟内存的系统中，CPU从一个有N=2^n个地址空间中生成虚拟地址。虚拟地址其实就是线性地址。

物理地址（Physical Address） 是指出现在CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。在hello程序中MMU负责将虚拟地址转换为物理地址，并且通过物理地址在缓存中寻找缓冲块。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

一个逻辑地址由两部份组成，段标识符、段内偏移量。段标识符是由一个16位长的字段组成，称为段选择符。其中前13位是一个索引号。后面3位包含一些硬件细节，如图7-1：



图7.2.1 段选择符

索引号，是“段描述符(segment descriptor)”，段描述符具体地址描述了一个段。这样，很多个段描述符，就组了一个数组，叫“段描述符表”，这样，可以通过段标识符的前13位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段，由8个字节组成，如图7-2：

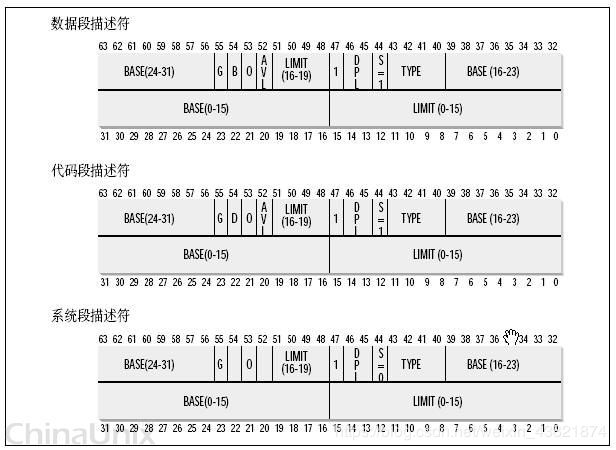


图7.2.2 段描述符

图示比较复杂，可以利用一个数据结构来定义它，不过，在此只关心一样，就是Base字段，它描述了一个段的开始位置的线性地址。 Intel设计的本意是，一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。那究竟什么时候该用GDT，什么时候该用LDT呢？这是由段选择符中的T1字段表示的，=0，表示用GDT，=1表示用LDT。 GDT在内存中的地址和大小存放在CPU的gdtr控制寄存器中，而LDT则在ldtr寄存器中。再看图7-3比起来要直观些：

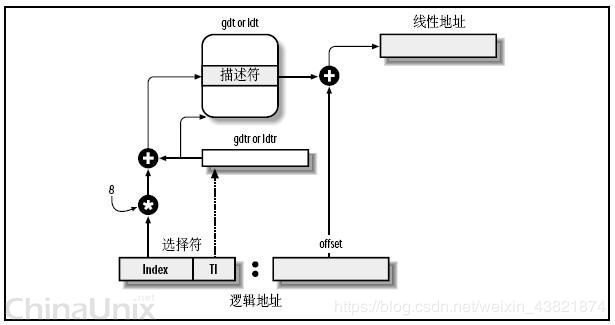


图7.2.3- 概念间的关系说明

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，

1、看段选择符的T1=0还是1，知道当前要转换是GDT中的段，还是LDT中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

2、拿出段选择符中前13位，可以在这个数组中，查找到对应的段描述符，这样，它了Base，即基地址就知道了。

3、把Base + offset，就是要转换的线性地址了。 还是挺简单的，对于软件来讲，原则上就需要把硬件转换所需的信息准备好，就可以让硬件来完成这个转换了。

## 7.3 Hello的线性地址到物理地址的变换-页式管理

CPU的页式内存管理单元，负责把一个线性地址，转换为物理地址。从管理和效率的角度出发，线性地址被分为以固定长度为单位的组，称为页(page)，例如一个32位的机器，线性地址最大可为4G，可以用4KB为一个页来划分，这页，整个线性地址就被划分为一个tatol\_page[2^20]的大数组，共有2的20个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。 另一类“页”，我们称之为物理页，或者是页框(frame)、页桢的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。 这里注意到，这个total\_page数组有2^20个成员，每个成员是一个地址（32位机，一个地址也就是4字节），那么要单单要表示这么一个数组，就要占去4MB的内存空间。为了节省空间，引入了一个二级管理模式的机器来组织分页单元。如图7-4 。

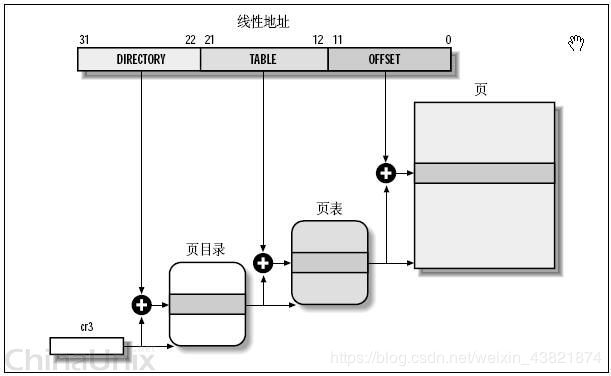


图7.3.1 二级管理模式图

分页单元中，页目录是唯一的，它的地址放在CPU的cr3寄存器中，是进行地址转换的开始点。

每一个活动的进程，因为都有其独立的对应的虚似内存（页目录也是唯一的），那么它也对应了一个独立的页目录地址。——运行一个进程，需要将它的页目录地址放到cr3寄存器中，将别个的保存下来。

每一个32位的线性地址被划分为三部份，面目录索引(10位)：页表索引(10位)：偏移(12位) 依据以下步骤进行转换：

(1)从cr3中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

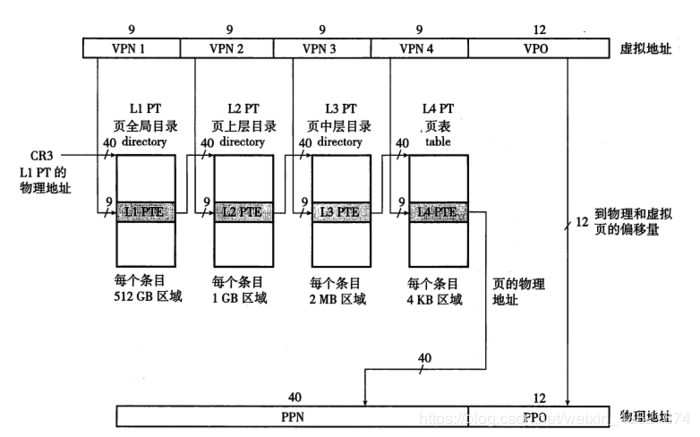
(2)根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到页表中去了。

(3)根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；

(4)将页的起始地址与线性地址中最后12位相加，得到最终的物理地址。

## 7.4 TLB与四级页表支持下的VA到PA的变换

## 该图给出了Core i7 MMU如何使用四级页表来将虚拟地址翻译成物理地址。36位的虚拟地址被分割成4个9位的片。CR3寄存器包含L1页表的物理地址。VPN1有一个到L1 PTE的偏移量，找到这个PTE以后又会包含到L2页表的基础地址；VPN2包含一个到L2PTE的偏移量，找到这个PTE以后又会包含到L3页表的基础地址；VPN3包含一个到L3PTE的偏移量，找到这个PTE以后又会包含到L4页表的基础地址；VPN4包含一个到L4PTE的偏移量，找到这个PTE以后就是相应的PPN（物理页号）。



## 图7.4.1VA到PA的变换

## 7.5 三级Cache支持下的物理内存访问

如图为Core i7的内存系统：

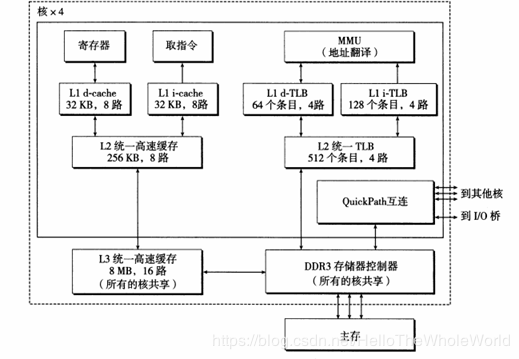


图7.5.1Core i7的内存系统

## 针对物理内存访问，主要对各类高速缓存存储器的读写策略做出说明：

## 当CPU执行一条读内存字w的指令，它向L1高速缓存请求这个字。如果L1高速缓存由w的一个缓存的副本，那么就得到L1的高速缓存命中，高速缓存会很快抽取出w并返回给CPU。否则就是缓存不命中，当L1高速缓存向主存请求包含w的块的一个副本时，CPU必须等待。当被请求的块最终从内存到达时，L1高速缓存将这个快存放在他的一个高速缓存行里，从被缓存的块中抽取字w，然后返回给CPU。总体来看，高速缓存确定一个请求是否命中，然后抽取出被请求的字的过程，分为三步，（1）组选择、（2）行匹配、（3）字抽取。

## 直接映射高速缓存读策略：

## 直接映射高速缓存E=1，即每组只有一行。组选择是通过组索引位标识组。高速缓存从w的地址中间抽取出s个组索引位，这些位被解释为一个对应于一个组号的无符号整数，来进行组索引。行匹配中，确定了某个组i，接下来需要确定是否有字w的一个副本存储在组i包含的一个高速缓存行里，因为直接映射高速缓存只有一行，如果有效位为1且标志位相同则缓存命中，根据块偏移位即可查找到对应字的地址并取出；若有效位为1但标志位不同则冲突不命中，有效位为0则为冷不命中，此时都需要从存储器层次结构下一层取出被请求的块，然后将新的块存储在组索引位指示的组中的一个高速缓存行中。

## 组相联高速缓存读策略：

## 组相联高速缓存每个组都会保存多余一个的高速缓存行，组选择与直接映射高速缓存的组选择一样，通过组索引位标识组。行匹配时需要找遍组中所有行，找到标记位有效位均相同的一行则缓存命中；如果CPU请求的字不在组的任何一行中，则缓存不命中，选择替换时如果存在空行选择空行，如果不存在空行则通过替换策略替换其中一行。

## 全相联高速缓存读策略：

## 全相联高速缓存只包含一个组，其行匹配和字选择与组相联高速缓存中一样，区别主要是规模大小的问题。

## 写策略：分为两种，直写和写回。

## 直写是立即将w的高速缓存块写回到紧接着的低一层中。虽然简单，但是只写的缺点是每次写都会引起总线流量。

## 写回尽可能的推迟更新，只有当替换算法要驱逐这个更新过的块时，才把它写回到紧接着的第一次层中，由于局部性，写回能显著减少总线流量，但增加了复杂性。处理写不命中有两种方法一种为写分配，加载相应的的低一层的块到高速缓存中，然后更新这个高速缓存块。另一种方法为非写分配，避开高速缓存，直接把这个字写到低一层中，直写高速缓存通常是非写分配的，写回高速缓存通常是写分配的。

## 7.6 hello进程fork时的内存映射

## 首先了解共享对象在虚拟内存中的应用：

## 一个对象可以被映射到虚拟内存的一个区域，要么作为共享对象，要么作为私有对象。如果一个进程将一个共享对象映射到它的虚拟地址空间的一个区域内，那么这个进程对这个区域的任何写操作，对于那些也把这个共享对象映射到它们虚拟内存的其他进程而言，也是可见的。而且，这些变化也会反映在磁盘上的原始对象中。

## 另一方面，对于一个映射到私有对象的区域做的改变，对于其他进程来说是不可见的，并且进程对这个区域所做的任何写操作都不会反映在磁盘上的对象中。一个映射到共享对象的虚拟内存区域叫做共享区域。类似地，也有私有区域。

## 其次，关注写时复制这一概念：

## 私有对象使用一种叫写时复制来映射至虚拟内存中，多个进程可将一个私有对象映射到其内存不同区域，共享该对象同一物理副本对于每个映射私有对象的进程，相应私有区域的页表条目都被标记为只读，并且区域结构被标记为私有的写时复制。只要没有进程试图写它自己的私有区域，它们就可以继续共享物理内存中对象的一个单独副本。然而，只要有一个进程试图写私有区域内的某个页面，那么这个写操作就会触发一个保护故障。

## 当故障处理程序注意到保护异常是由于进程试图写私有的写时复制区域中的一个页面而引起的，它就会在物理内存中创建这个页面的一个新副本，更新页表条目指向这个新的副本，然后恢复这个页面的可写权限。

Fork被shell调用，内核会为hello创建子进程，同时会创建各种数据结构并分配给hello唯一的PID。为了给hello创建虚拟内存，内核创建了当前进程的mm\_struct、区域结构和样表的原样副本，并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写时复制。

## 7.7 hello进程execve时的内存映射

主要步骤如下：

1. 删除已存在的用户区域
2. 映射hello私有区域
3. 映射共享区域
4. 设置程序计数器PC

加载器映射用户地址空间区域如下：

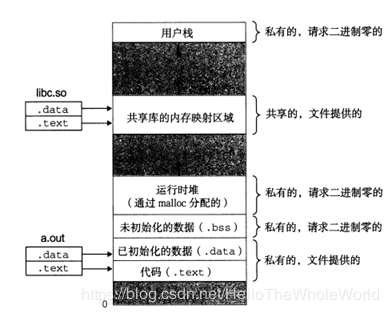


图7.7.1

## 7.8 缺页故障与缺页中断处理

## 在虚拟内存的习惯说法中，DRAM缓存不命中称为缺页(page fault) 。CPU引用了VP3中的一个字，VP3并未缓存在DRAM中。地址翻译硬件从内存中读取PTE3,从有效位推断出VP3未被缓存，并且触发一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，在此例中就是存放在PP3中的VP4。如果VP4已经被修改了，那么内核就会将它复制回磁盘。无论哪种情况，内核都会修改VP4的页表条目，反映出VP4不再缓存在主存中这一事实。

## 缺页处理程序从磁盘上用VP3的副本取代VP4，在缺页处理程序重新启动导致缺页的指令之后，该指令将从内存中正常地读取字，而不会再产生异常。

## 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。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。

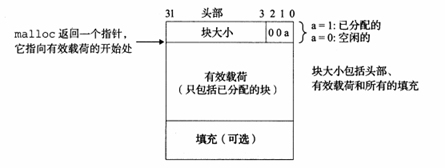
## 分配器有两种基本风格，都要求应用显式地分配块，它们的不同之处在于由哪个实体来负责释放已分配的块。

## 显式分配器：要求应用显式地释放任何已分配的块。

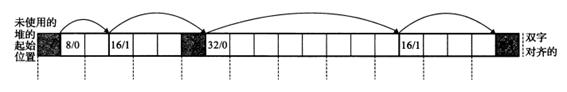
## 隐式分配器：要求分配器检测一个已分配块何时不再使用，那么就释放这个块，自动释放未使用的已经分配的块的过程叫做垃圾收集。而自动释放未使用的已分配的块的过程叫做垃圾收集。

## 隐式空闲链表的堆块格式及其组织格式如下图：

## ​ 图7.9.1 隐式链表堆块格式

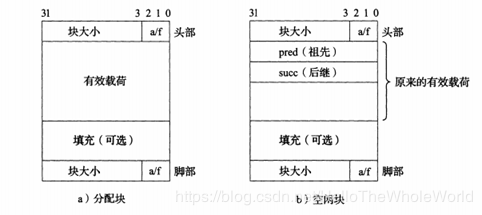


## ​ 图7.9.2 隐式链表组织格式



## 隐式空闲链表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。

## 显式空闲链表的堆块格式如下图：



## ​ 图7.9.3 显示链表组织格式

## 显式空闲链表有两种方式来维护一种是先进后出，另一种是地址顺序。此处不详细展开。

## 放置空闲块的策略有三种，分别是首次适配、下一次适配、最佳适配。

## 首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配和首次适配很相似，只不过不是从链表的起始处开始每次搜索，而是从上一.次查询结束的地方开始。最佳适配检查每个空闲块，选择适合所需请求大小的最小空闲块。7.10本章小结

本章具体完成了对Linux虚拟地址空间的分析。Intel的段式管理和页式管理机制，以及TLB与多级页表支持下的VA到PA的转换，同时对cache支持下的物理内存访问做了说明。最后介绍了 hello的内存映射，以及动态内存的分配。

**（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

## 设备管理：unix io接口

## 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。

## 8.2 简述Unix IO接口及其函数

## Unix I/O接口：

## 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。应用程序只需要记住这个描述符。

## 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，它们可以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，显式地将改变当前文件位置为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时执行读写操作会触发一个称为end-of-file（EOF）的条件，应用程序能检测到这个条件。类似地，写操作就是从内存中复制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，它就会通知内核关闭这个文件。作为相应，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去。无论一个进程因何种原因终止时，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：

## 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：

## int open(char \*filename, int flags, mode\_t mode);

## 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。

## 进程通过调用close 函数关闭一个打开的文件。

## int close(int fd);

## 返回0，成功；返回-1，失败。

## 3.应用程序是通过分别调用read 和write 函数来执行输入和输出的。

## ssize\_t read(int fd, void \*buf, size\_t n);

## ssize\_t write(int fd, const 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，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。

## write 函数从内存位置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

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;

}

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

Getchar实现方式如下：

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;

}

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

本章是hello程序实现的最后一步，具体介绍了Linux系统下的I/O设备管理方法，并且通过条列出I/O接口以及函数，分析printf和getchar函数的具体实现过程。通过概念和实例的结合，使得我们对Linux的I/O更加透彻，为hello程序的分析画上圆满的句号。

**（第8章1分）**

# 结论

Hello经过的历程：

1.hello 被I/O设备输入到磁盘当中。

2.hello.c被预处理为hello.i，将外部库，宏定义等内容展开。

3.hello.i被编译器编译成汇编文件hello.s

4.hello.s被汇编器汇编成可重定位目标文件hello.o

5.链接器将hello.o文件和外部文件链接成可执行文件hello

6.在shell中输入命令，通过fork生成子进程，通过execve加载并运行hello进程

7.运行过程中，hello的代码和数据被从磁盘复制到内存，并生成虚拟内存映像

8.MMU将虚拟地址翻译成物理地址完成程序中所需要的访存功能。

9.内核通过调度完成hello和其他进程的上下文切换

10.hello运行结束，shell负责回收。

深切感悟：

通过本次大实验具体指导和要求，我们经历了hello程序短暂的一生，虽然仅仅八个章节，但却是整个计算机系统中重要的基石，也正是通过这些概念的延伸拓展才有了现如今的多姿多彩的计算机系统。Hello程序走过的每一步，预处理，编译，汇编，链接……虽然简单平凡，甚至略显枯燥，但却是计算机系统中必不可少的成分。计算机高效有序的运行，离不开千千万万这样的程序的运转。Cache进进出出，信号走走停停，内存整整齐齐……这样的有序的动作无时无刻不在计算机内部中完成着，作为程序员，我们有理由去了解更多有关这些操作中的细节，让计算机系统尽善尽美。遍历hello程序的一生，正是我们为此努力而迈出的第一步。

**（结论0分，缺失 -1分，根据内容酌情加分）**

# 附件

hello.c ：hello源代码

hello.i ：预处理后的文本文件

hello.s ：hello.i编译后的汇编文件

hello.o ：hello.s汇编后的可重定位目标文件

hello.elf : hello.o文件的ELF格式

hello\_out.elf : hello文件的ELF格式

hello\_objdump ：hello的反汇编代码

hello.0\_objdump ：hello.o的反汇编代码

hello ：链接后的可执行文件。

**（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] https://www.cnblogs.com/pianist/p/3315801.html

[8] 深入理解计算机系统（第三版）