

**计算机系统**

**大作业**

题 目 程序人生-Hello’s P2P

专 业 人工智能（2+X）

学　　 号 2021112078

班　　 级 21WL021

学 生 刘俊杉

指 导 教 师 史先俊

**计算机科学与技术学院**

**2023年4月**

**摘 要**

一个hello程序的生命周期是从一个源程序开始的，程序员通过编辑器编写并保存了一个.c文件，也就是hello.c文件。运行C预处理器（cpp）将hello.c文件进行预处理，生成一个hello.i文件。hello.i文件为一个修改了的源程序。运行C编译器（ccl）将此hello.i文件翻译为一个汇编语言的文件 hello.s。再运行汇编器( as）将hello.s文件翻译为一个可重定位目标文件 hello.o。最后运行链接器程序(1d)，该程序将hello.o和系统各种目标文件链接起来，最终创建出一个可执行月标文件 hello。

**关键词：**预处理；编译；汇编；链接；进程；存储；IO管理

**目 录**

[第1章 概述 - 4 -](#_Toc532238396)

[1.1 Hello简介 - 4 -](#_Toc532238397)

[1.2 环境与工具 - 4 -](#_Toc532238398)

[1.3 中间结果 - 4 -](#_Toc532238399)

[1.4 本章小结 - 4 -](#_Toc532238400)

[第2章 预处理 - 5 -](#_Toc532238401)

[2.1 预处理的概念与作用 - 5 -](#_Toc532238402)

[2.2在Ubuntu下预处理的命令 - 5 -](#_Toc532238403)

[2.3 Hello的预处理结果解析 - 5 -](#_Toc532238404)

[2.4 本章小结 - 5 -](#_Toc532238405)

[第3章 编译 - 6 -](#_Toc532238406)

[3.1 编译的概念与作用 - 6 -](#_Toc532238407)

[3.2 在Ubuntu下编译的命令 - 6 -](#_Toc532238408)

[3.3 Hello的编译结果解析 - 6 -](#_Toc532238409)

[3.4 本章小结 - 6 -](#_Toc532238410)

[第4章 汇编 - 7 -](#_Toc532238411)

[4.1 汇编的概念与作用 - 7 -](#_Toc532238412)

[4.2 在Ubuntu下汇编的命令 - 7 -](#_Toc532238413)

[4.3 可重定位目标elf格式 - 7 -](#_Toc532238414)

[4.4 Hello.o的结果解析 - 7 -](#_Toc532238415)

[4.5 本章小结 - 7 -](#_Toc532238416)

[第5章 链接 - 8 -](#_Toc532238417)

[5.1 链接的概念与作用 - 8 -](#_Toc532238418)

[5.2 在Ubuntu下链接的命令 - 8 -](#_Toc532238419)

[5.3 可执行目标文件hello的格式 - 8 -](#_Toc532238420)

[5.4 hello的虚拟地址空间 - 8 -](#_Toc532238421)

[5.5 链接的重定位过程分析 - 8 -](#_Toc532238422)

[5.6 hello的执行流程 - 8 -](#_Toc532238423)

[5.7 Hello的动态链接分析 - 8 -](#_Toc532238424)

[5.8 本章小结 - 9 -](#_Toc532238425)

[第6章 hello进程管理 - 10 -](#_Toc532238426)

[6.1 进程的概念与作用 - 10 -](#_Toc532238427)

[6.2 简述壳Shell-bash的作用与处理流程 - 10 -](#_Toc532238428)

[6.3 Hello的fork进程创建过程 - 10 -](#_Toc532238429)

[6.4 Hello的execve过程 - 10 -](#_Toc532238430)

[6.5 Hello的进程执行 - 10 -](#_Toc532238431)

[6.6 hello的异常与信号处理 - 10 -](#_Toc532238432)

[6.7本章小结 - 10 -](#_Toc532238433)

[第7章 hello的存储管理 - 11 -](#_Toc532238434)

[7.1 hello的存储器地址空间 - 11 -](#_Toc532238435)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 11 -](#_Toc532238436)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 11 -](#_Toc532238437)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 11 -](#_Toc532238438)

[7.5 三级Cache支持下的物理内存访问 - 11 -](#_Toc532238439)

[7.6 hello进程fork时的内存映射 - 11 -](#_Toc532238440)

[7.7 hello进程execve时的内存映射 - 11 -](#_Toc532238441)

[7.8 缺页故障与缺页中断处理 - 11 -](#_Toc532238442)

[7.9动态存储分配管理 - 11 -](#_Toc532238443)

[7.10本章小结 - 12 -](#_Toc532238444)

[第8章 hello的IO管理 - 13 -](#_Toc532238445)

[8.1 Linux的IO设备管理方法 - 13 -](#_Toc532238446)

[8.2 简述Unix IO接口及其函数 - 13 -](#_Toc532238447)

[8.3 printf的实现分析 - 13 -](#_Toc532238448)

[8.4 getchar的实现分析 - 13 -](#_Toc532238449)

[8.5本章小结 - 13 -](#_Toc532238450)

[结论 - 14 -](#_Toc532238451)

[附件 - 15 -](#_Toc532238452)

[参考文献 - 16 -](#_Toc532238453)

# 第1章 概述

## 1.1 Hello简介

P2P：程序员在Windows或Linux环境下，利用编辑器敲入C语言代码（Program），然后利用gcc编译器对C语言程序执行编译命令：hello.c文件先经过预处理器cpp，生成hello.i文件，再经过编译器ccl生成hello.s汇编程序，然后经过汇编器as生成可重定位目标程序hello.o，最后通过链接器ld链接生成可执行文件hello。在Linux终端执行./hello命令，运行该可执行文件（Process）。

O2O：可执行文件hello执行后，shell通过execve函数和fork函数创建子进程并将hello载入，通过内存映射等为其分配自己的运行空间，CPU在流水线上依次执行每一条指令，内存管理系统利用虚拟内存地址从TLB或高速缓存中取出相应数据，如果出现不命中则继续从其下一级缓存中读取数据。最后程序运行结束后，shell接受到相应的信号，启动信号处理机制，对该进程进行回收处理，释放其所占的内存并删除有关进程上下文(context)，hello程序重新回归0，即From zero to zero。

要运行hello.c首先要对hello.c做四个步骤：预处理，编译，汇编，链接，然后这些指令按照可执行目标程序的格式打包，并以二进制磁盘文件的形式保存起来，目标程序写可称为可执行目标文件。

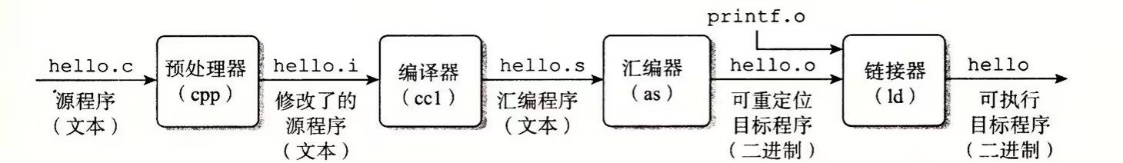


图 1编译过程

hello.c源程序已经被编译系统翻译成了hello，并被存放在磁盘上，运行可执行文件要把文件名输入到shell中。shell将加载并运行hello程序，然后等待程序终止。hello在屏幕上输出它的消息，然后终止。当在键盘上敲回车键时，shell程序就知道结束了命令的输入，然后shell执行一系列指令来加载可执行的hello文件，这些指令将hello目标文件中的代码和数据从磁盘复制到主存。一旦目标文件hello中的代码和数据被加载到主存，处理器就开始执行hello程序的main程序中的机器语言指令，将字符串中的字节从主存复制到寄存器文件，再从寄存器文件中复制到显示设备，最终显示在屏幕上。

有两个并发的进程: shell 进程和hello进程.最开始，只有shell进程在运行，即等待命令行上的输入。当我们让它运行hello程序时，shell 通过调用一个专门的函数，即系统调用，来执行我们的请求，系统调用会将控制权传递给操作系统。操作系统保存shell进程的上下文，创建一个新的hello进程及其上下文，然后将控制权传给新的hello进程。hello 进程终止后，操作系统恢复shell进程的上下文，并将控制权传回给它，shell 进程会继续等待下一个命令行输入。

## 1.2 环境与工具

### 1.2.1 硬件环境

处理器：Intel® Core™ i7-9750H CPU @ 2.60GHz 2.59GHz

RAM：32.00GB

64位操作系统，基于x64的处理器

### 1.2.2 软件环境

### Windows10 64位；Ubuntu 16.04

### 1.2.3 开发工具

### Visual Studio 2010 64位以上；CodeBlocks 64位；vi/vim/gedit+gcc gdb

## 1.3 中间结果

表 1 生成文件

|  |  |
| --- | --- |
| **文件名** | **文件作用** |
| hello.c | 源代码文件 |
| hello.i | 预处理后生成的文件 |
| hello.s | 编译后生成的文件 |
| hello.o | 汇编后生成的可重定位目标程序 |
| hello | 链接之后生成的可执行程序 |
| hello\_o\_asm.txt | hello.o的反汇编文件 |
| hello\_o\_elf.txt | hello.o的ELF文件 |
| hello\_run\_asm | hello的反汇编文件 |
| hello\_run\_elf | hello的ELF文件 |

## 1.4 本章小结

本章进行了hello程序一生P2P，O2O的简要概述，列出了本实验的硬件环境、软件环境、开发工具和本实验中生成的中间结果文件的名字和作用。

# 第2章 预处理

## 2.1 预处理的概念与作用

在C语言的程序中可包括各种以符号#开头的编译指令，这些指令称为预处理命令。预处理命令属于C语言编译器，而不是C语言的组成部分。通过预处理命令可扩展C语言程序设计的环境。

大多数预处理器指令属于下面3种类型：

1. 宏定义：#define 指令定义一个宏，#undef指令删除一个宏定义。

2. 文件包含：#include指令导致一个指定文件的内容被包含到程序中。

3. 条件编译：#if,#ifdef,#ifndef,#elif,#else和#dendif指令可以根据编译器可以测试的条件来将一段文本包含到程序中或排除在程序之外。

预处理阶段：预处理器（cpp）根据以字符#开头的命令，修改源程序。比如第一行的#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，并把它直接插入程序文本中。结果就得到了另一个C程序，通常是以.i作为文件拓展名。

## 2.2在Ubuntu下预处理的命令



图 2 预处理命令

## 2.3 Hello的预处理结果解析

2.3.1 文件增大

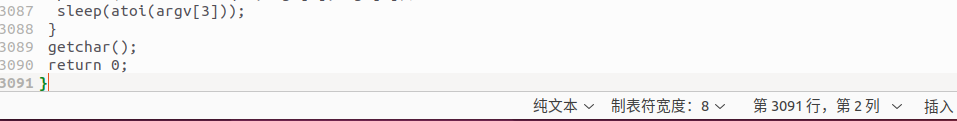


图 3 文件增大截图

整个文件变成了3091行

2.3.2源代码保留



图 4 源代码截图

hello.c程序本来的内容出现在最后，在此之前是#include <stdio.h>，

#include <unistd.h> ，#include <stdlib.h>的展开

2.3.3 模块化拼接

最开始的一段代码，是hello.c拼接的各种库文件。

图形用户界面, 文本

描述已自动生成

图 5 拼接的库文件

中间的某段代码，对很多内部的函数进行声明。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图 6 中间代码

程序的源代码在文件的末尾。

文本

描述已自动生成

图 7 源代码

## 2.4 本章小结

本章介绍了预处理的概念和作用，同时介绍了Ubuntu环境下预处理的命令，并对预处理生成的hello.i文件进行了简要分析。

# 第3章 编译

## 3.1 编译的概念与作用

编译：就是把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。编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

在hello的一生中，编译器将文本文件 hello.i 翻译成文本文件 hello.s。

C编译器ccl通过词法分析和语法分析，将合法指令翻译成等价汇编代码，经过编译过程，编译器将文本文件hello.i翻译成汇编语言文件hello.s。在hello.s中，以文本的形式描述了一条条低级机器语言指令。

## 3.2 在Ubuntu下编译的命令

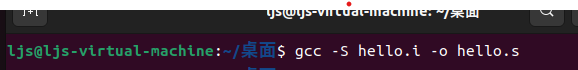


图 8 编译命令

## 3.3 Hello的编译结果解析

图形用户界面, 文本

描述已自动生成

图 9 C语言源程序

3.3.1整数和字符串常量





图 10 常量4保存位置

在语句中，常量4保存位置在.text节



文本

描述已自动生成

图 11 常量0和5保存位置

数字0和5也被存在.text节中



文本

描述已自动生成

图 12 函数保存位置

printf()、scanf()中的字符串则被存储在.rodata节中

3.3.2 变量

全局变量：

已经初始化并且初始值非零的全局变量储存在.data节，它的初始化不需要汇编语句，而是通过虚拟内存请求二进制零的页直接完成的。

局部变量：

局部变量存储在寄存器或栈中。程序中的局部变量i定义



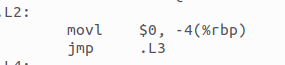


图 13 变量i保存位置

表示i被保存在%rsp的位置上

3.3.3 算数操作





图 14 算数操作

栈上存储的i变量加一

3.3.4关系操作和控制转移



文本

描述已自动生成

图 15 判断argc

将程序传入参数argc是否等于4，je用于判断cmpl产生的条件码，看是否相等



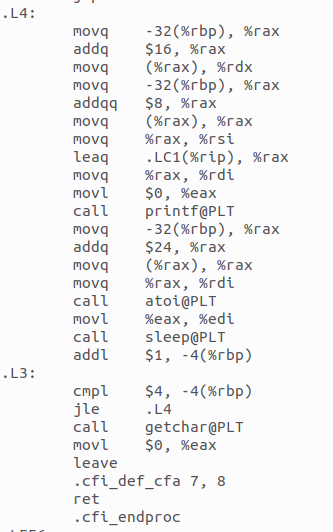


图 16 for循环中的循环条件

这里采取了初始化– 跳转– 循环判断的模式。jle用于判断cmpl产生的条件码，若后一个操作数的值小于等于前一个，则重新执行循环；

3.3.5 数组/指针/结构操作

主函数main的参数中有指针数组char \*argv[]





图 17 main函数

char \*占8个字节，argc表示传入的长度

把argc存在%edi中

把argv存在%rsi中

文本

描述已自动生成

图 18 汇编代码

M[%rbp-32+16]和M[%rbp-32+8]，分别得到argv[1]和argv[2]两个字符串的首地址

3.3.6函数操作

X86-64中,过程调用传递参数规则：第1~6个参数一次储存在%rdi、%rsi、%rdx、%rcx、%r8、%r9这六个寄存器中，剩下的参数保存在栈当中。



文本

描述已自动生成

图 19 函数操作代码

把argc存在%edi中

把argv存在%rsi中





图 20 print函数相关代码

传入了字符串参数的首地址，条件满足后调用



文本, 信件

描述已自动生成

图 21 for函数代码



****

图 22 exit函数代码

参数传入1





图 23 sleep函数代码

随着for循环被调用





图 24 getchar函数代码

## 3.4 本章小结

本章介绍了编译的概念以及过程。通过hello函数分析了c语言如何转换成为汇编代码。介绍了汇编代码如何实现变量、常量、传递参数以及分支和循环。

# 第4章 汇编

## 4.1 汇编的概念与作用

概念：汇编器（as）将.s 文件翻译成机器语言指令，把这些指令打包成一种叫可重定位目标程序的格式，并将结果保存在目标文件（后缀为.o）中。  
作用：将汇编代码转换成真正机器可以读懂的二进制代码。

## 4.2 在Ubuntu下汇编的命令



图 25 汇编命令

## 4.3 可重定位目标elf格式

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。

4.3.1 readelf命令：



图 26 readelf命令

4.3.2 ELF头：

包含了系统信息，编码方式，ELF头大小，节的大小和数量等等一系列信息。ELF头内容如下：

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剩下的部分包含帮助连接器语法分析和解释目标文件的信息。包括ELF头的大小、目标文件的类型等。

一个典型的ELF可重定位目标文件包含下面几个节：

.txt：已编译程序的机器代码。

.rodata：只读数据。

.data：已初始化的全局和静态局部C变量。

.bss：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。

.symtab：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。

.rel.text：一个.text节中位置的列表。

.rel.data：被模块引用或定义的所有全局变量的重定位信息。

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

文本

中度可信度描述已自动生成

图 27 ELF头截图

4.3.3节头表

描述了.o文件中出现的各个节的类型、位置、所占空间大小等信息。

图形用户界面, 文本, 应用程序, 表格

描述已自动生成

图 28 节头表截图

4.3.4 重定位节：

各个段引用的外部符号等在链接时需要通过重定位对这些位置的地址进行修改。链接器会通过重定位节的重定位条目计算出正确的地址。

hello.o需重定位：.rodata中的模式串，puts，exit，printf，slepsecs，sleep，getchar等符号。

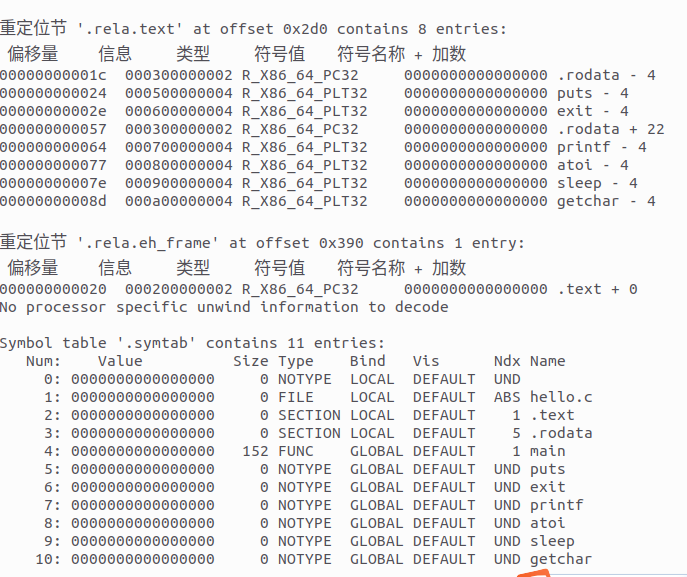


图 29 重定位节

4.3.5符号表：

.symtab存放在程序中定义和引用的函数和全局变量的信息。

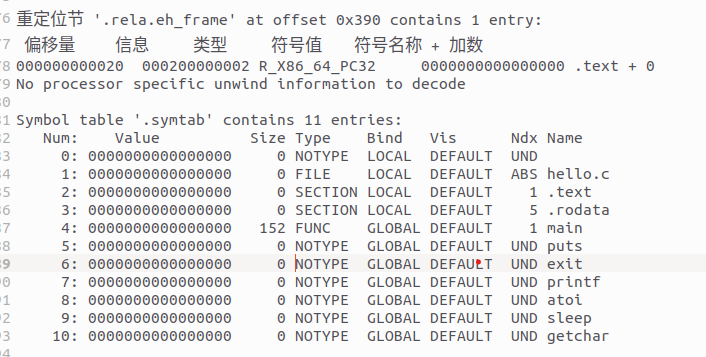


图 30 符号表

## 

## 4.4 Hello.o的结果解析



图 31 反汇编hello.o

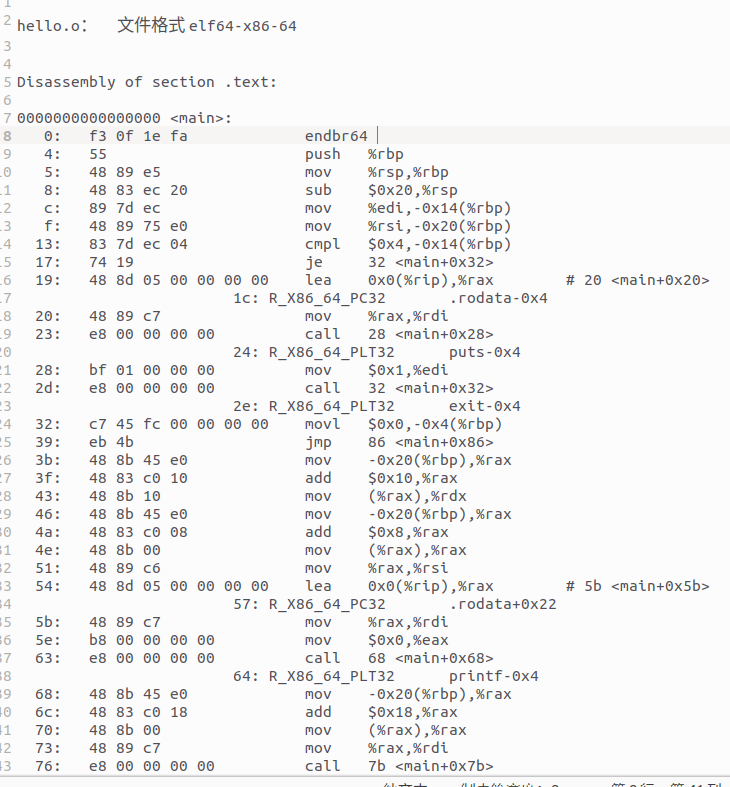


图 32 反汇编文件

与第三章的hello.s对比，首先在指令前增加了其十六进制表示，即机器语言。其次在操作数上，hello.s中操作数为十进制，而hello.o的反汇编中操作数为十六进制。在条件跳转语句上，hello.o的反汇编文件用的是相对偏移量，而hello.s中是函数名。

在数的表示上，hello.s中的操作数表现为十进制，而hello.o反汇编代码中的操作数为十六进制。说明机器语言的构成，与汇编语言的映射关系。特别是机器语言中的操作数与汇编语言不一致，特别是分支转移函数调用等。

在控制转移上，hello.s使用.L2和.LC1等段名称进行跳转，而反汇编代码使用目标代码的虚拟地址跳转。不过目前留下了重定位条目，跳转地址为零。它们将在链接之后被填写正确的位置。

在函数调用上，hello.s直接call函数名称，而反汇编代码中call的是目标的虚拟地址。但和上一条的情况类似，只有在链接之后才能确定运行执行的地址，目前目的地址是全0，并留下了重定位条目。

## 4.5 本章小结

本章介绍了汇编。经过汇编器，汇编语言转化为机器语言，hello.s文件转化为hello.o可重定位目标文件。我们研究了可重定位目标文件elf格式，接触了了readelf命令、elf头、节头部表、重定位节、符号表。我们对比hello.s和hello.o，分析了汇编语言到机器语言的变化。

# 

# 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：链接可以使得分离编译成为可能。不用将一个大型的应用程序组织为一个巨大的源文件，而是可以将它分解成更小、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，而不必重新编译其他文件。

## 5.2 在Ubuntu下链接的命令

文本

描述已自动生成

图 33 链接命令

## 5.3 可执行目标文件hello的格式



图 34 hello的ELF格式

5.3.1 ELF文件头：

表格

描述已自动生成

图 35 ELF头

5.3.2 节头：

描述了各个节的大小、偏移量和其他属性。链接器链接时，会将各个文件的相同段合并成一个大段，并且根据这个大段的大小以及偏移量重新设置各个符号的地址。

图形用户界面, 应用程序, 表格, Excel

描述已自动生成

图 36 节头

## 5.4 hello的虚拟地址空间

使用edb加载hello， data dump窗口可以查看加载到虚拟地址中的hello程序。program headers告诉链接器运行时加载的内容并提供动态链接需要的信息。

可以看出虚拟地址空间起始地址为0x400000

表格

中度可信度描述已自动生成

图 37 elf文件

由elf文件可看出.inerp偏移量为0x2e0，在edb对应位置找到：

表格

描述已自动生成

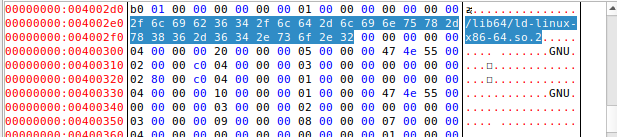
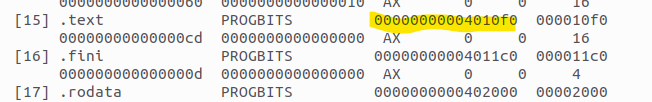


图 38 .inerp偏移量

同样可以在edb中找到.text段和.rodata 段等：



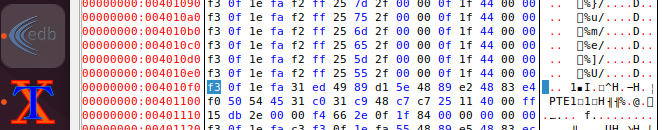


图 39 .text段

图形用户界面, 文本, 应用程序

描述已自动生成



图 40 .rodata 段

程序包含PHDR，INTERP，LOAD，DYNAMIC，NOTE，GNU\_STACK，GNU\_RELRO几个部分，其中PHDR 保存程序头表。INTERP 指定在程序已经从可执行文件映射到内存之后，必须调用的解释器。LOAD 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量数据、程序的目标代码等。DYNAMIC 保存了由动态链接器使用的信息。NOTE 保存辅助信息。GNU\_STACK：权限标志，用于标志栈是否是可执行。GNU\_RELRO：指定在重定位结束之后哪些内存区域是需要设置只读。

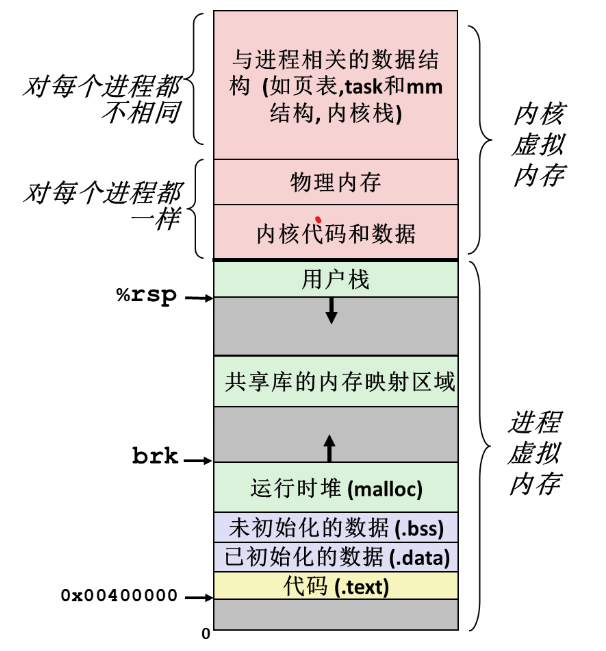


图 41 虚拟内存分配示意图

## 5.5 链接的重定位过程分析



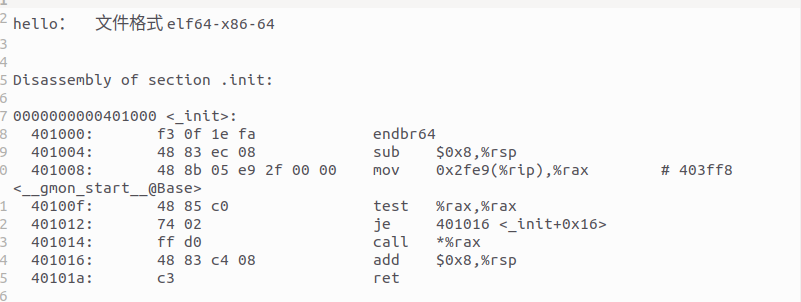


图 42 重定位文件

5.5.1 新增函数：

链接加入了在hello.c中用到的库函数，如exit、printf、sleep、getchar等函数。

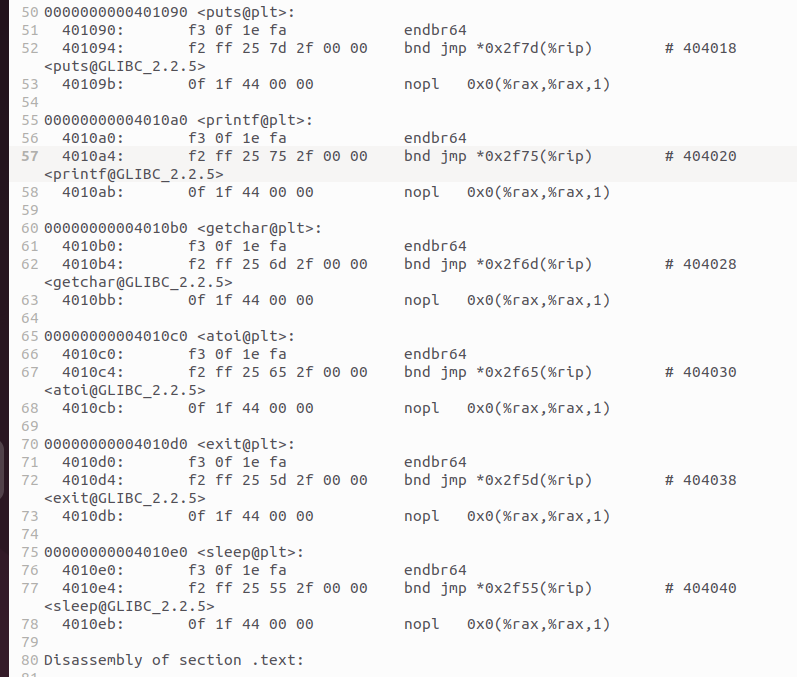


图 43 库函数

5.5.2 新增节：

hello中增加了.init和.plt节，和一些节中定义的函数。

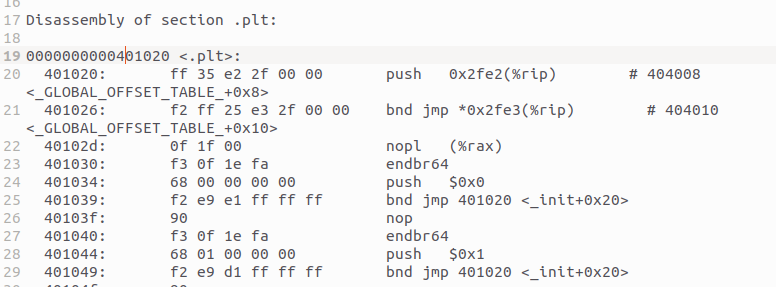


图 44 新增节

5.5.3 函数调用地址：

hello实现了调用函数时的重定位，因此在调用函数时调用的地址已经是函数确切的虚拟地址。

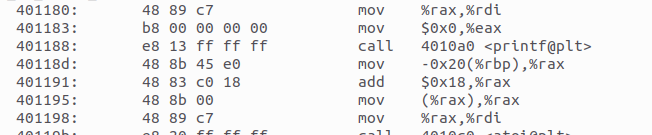


图 45 调用的虚拟地址

5.5.4控制流跳转地址：

hello实现了调用函数时的重定位，因此在跳转时调用的地址已经是函数确切的虚拟地址。



图 46 虚拟地址

5.5.5 链接的过程：

链接就是链接器（ld）将各个目标文件（各种.o文件）组装在一起，文件中的各个函数段按照一定规则累积在一起。从.o提供的重定位条目将函数调用和控制流跳转的地址填写为最终的地址。

## 5.6 hello的执行流程

从加载hello到\_start，到call main,以及程序终止的所有过程如下：  
\_dl\_start 地址：0x7f894f9badf0  
\_dl\_init 地址： 0x7f894f9cac10  
\_start 地址：0x4010f0  
\_libc\_start\_main 地址：0x7fce59403ab0  
\_cxa\_atexit 地址：0x7f38b81b9430  
\_libc\_csu\_init 地址：0x4011c0  
\_setjmp 地址：0x7f38b81b4c10  
\_sigsetjmp 地址：0x7efd8eb79b70  
\_sigjmp\_save 地址：0x7efd8eb79bd0  
main 地址：0x401125   
若argc != 4  
puts 地址：0x401090  
exit 地址：0x4010d0  
此时输出窗口打印出“用法: Hello 学号 姓名 秒数！”，程序结束  
若argc == 4  
printf 地址：0x4010a0  
atoi  地址：0x4010c0  
sleep 地址：0x4010e0 (以上三个函数在循环体中执行5次)  
此时窗口打印5行“Hello 2021112078 ljs”

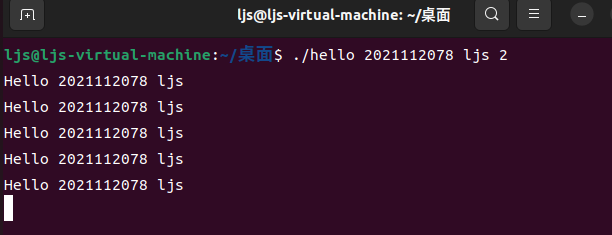


图 47 程序输入示例

getchar 地址：0x4010b0  
等待用户输入回车，输入回车后：  
\_dl\_runtime\_resolve\_xsave 地址：0x7f5852241680  
\_dl\_fixup 地址：0x7f5852239df0  
\_uflow 地址：0x7f593a9a10d0  
exit 地址：0x7f889f672120  
程序终止。

## 5.7 Hello的动态链接分析

在elf文件中可以找到：

表格

中度可信度描述已自动生成

图 48 elf文件

进入edb查看：

表格

描述已自动生成

表格

描述已自动生成

图 49 edf查看文件

## 动态链接的基本思想是把程序按照模块拆分成各个相对独立的部分，在程序运行时才将它们链接在一起形成一个完整的程序，在调用函数时，编译器没有办法预测这个函数运行时的地址，正常的方法是为该引用生成一条重定位记录，然后动态链接器在程序加载的时候再解析它。

延时绑定是通过GOT和PLT实现的，根据hello.elf可知，GOT起始表位置为0x404000

## 5.8 本章小结

本章主要介绍了链接的过程，利用edb、gdb、objdump等工具对链接的ELF文件、虚拟地址空间、重定位过程等进行详细分析，同时对hello程序的执行流程以及动态链接也做了简要介绍。至此，hello程序已经从程序代码转换成了一个可执行文件。

# 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：进程为程序提供了一种假象，程序好像是独占的使用处理器和内存。处理器好像是无间断地一条接一条地执行我们程序中的指令。

## 6.2 简述壳Shell-bash的作用与处理流程

shell是一个交互型的应用级程序，也被称为命令解析器，它为用户提供一个操作界面，接受用户的输入命令，并调度相应的应用程序。

首先从终端读入输入的命令,对输入的命令进行解析,如果该命令为内置命令，则立即执行命令，否则调用fork创建一个新的子进程，在该子进程的上下文中执行指定的程序。判断该程序为前台程序还是后台程序，如果为前台程序则等待程序执行结束，若为后台程序则将其放回后台并返回。在过程中 shell可以接受从键盘输入的信号并对其进行处理。

## 6.3 Hello的fork进程创建过程

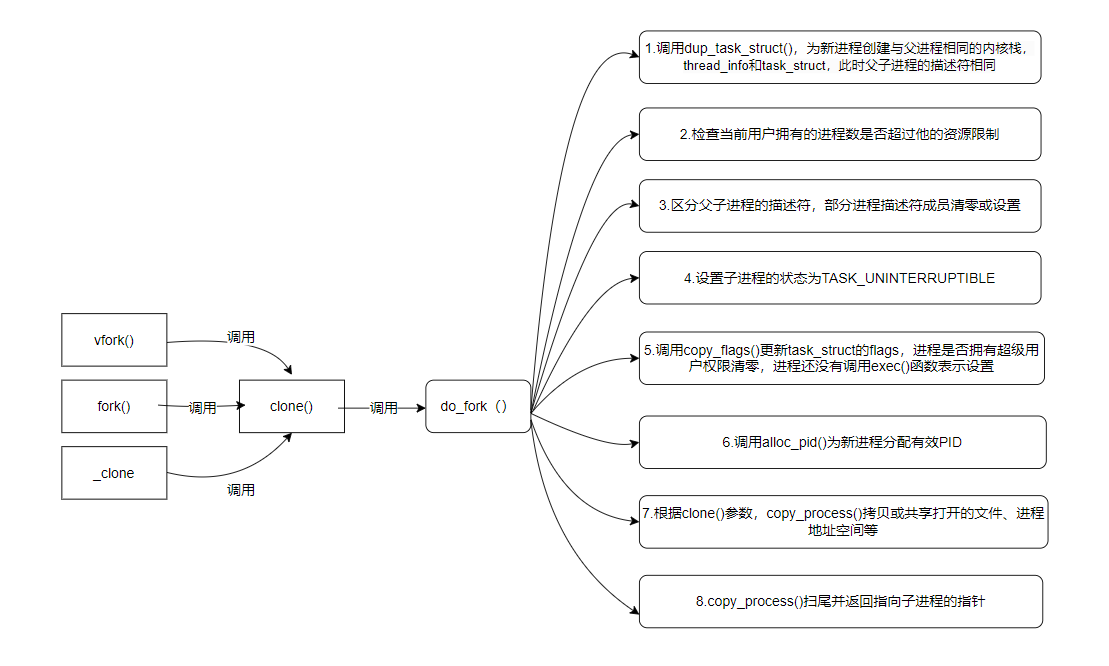


图 50 fork进程调用过程

## 6.4 Hello的execve过程

Execve的参数包括需要执行的程序(通常是argv[0])、参数argv、环境变量envp。 1. 删除已存在的用户区域（自父进程独立）。

2. 映射私有区：为Hello的代码、数据、.bss和栈区域创建新的区域结构，所有这些区域都是私有的、写时才复制的。

3. 映射共享区：比如Hello程序与标准C库libc.so链接，这些对象都是动态链接到Hello的，然后再用户虚拟地址空间中的共享区域内。

4. 设置PC：exceve做的最后一件事就是设置当前进程的上下文中的程序计数器，使之指向代码区域的入口点。

5. execve在调用成功的情况下不会返回，只有当出现错误时，例如找不到需要执行的程序时，execve才会返回到调用程序。

图示

描述已自动生成

图 51 execve过程

## 6.5 Hello的进程执行

6.5.1 逻辑控制流和时间片：

进程的运行本质上是CPU不断从程序计数器 PC 指示的地址处取出指令并执行，值的序列叫做逻辑控制流。操作系统会对进程的运行进行调度，执行进程A->上下文切换->执行进程B->上下文切换->执行进程A->… 如此循环往复。 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，这种决策就叫做调度，是由内核中称为调度器的代码处理的。当内核选择一个新的进程运行，我们说内核调度了这个进程。在内核调度了一个新的进程运行了之后，它就抢占了当前进程，并使用上下文切换机制来将控制转移到新的进程。在一个程序被调运行开始到被另一个进程打断，中间的时间就是运行的时间片。

6.5.2 用户模式和内核模式：

用户模式的进程不允许执行特殊指令，不允许直接引用地址空间中内核区的代码和数据。

内核模式进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

6.5.3 上下文：

上下文就是内核重新启动一个被抢占的进程所需要恢复的原来的状态，由寄存器、程序计数器、用户栈、内核栈和内核数据结构等对象的值构成。

6.5.4 调度的过程：

在对进程进行调度的过程，操作系统主要做了两件事：加载保存的寄存器，切换虚拟地址空间。

6.5.5 用户态与核心态转换：

为了能让处理器安全运行，需要限制应用程序可执行指令所能访问的地址范围。因此划分了用户态与核心态。

核心态可以说是拥有最高的访问权限，处理器以一个寄存器当做模式位来描述当前进程的特权。进程只有故障、中断或陷入系统调用时才会得到内核访问权限，其他情况下始终处于用户权限之中，保证了系统的安全性。

## 6.6 hello的异常与信号处理

6.6.1 异常类型与处理方式

表2：异常类型

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 原因 | 同步/异步 | 返回行为 |
| 中断 | 来自I/O设备的信号 | 异步 | 总是返回下一条指令 |
| 陷阱 | 有意的异常 | 同步 | 总是返回下一条指令 |
| 故障 | 潜在可恢复的错误 | 同步 | 可能返回当前指令 |
| 终止 | 不可恢复的错误 | 同步 | 不会返回 |

6.6.2 发送信号

按下Ctrl+Z：进程收到 SIGSTP 信号， hello 进程挂起。用ps查看其进程PID，可以发现hello的PID是3377；再用jobs查看此时hello的后台 job号是1，调用 fg 将其调回前台。

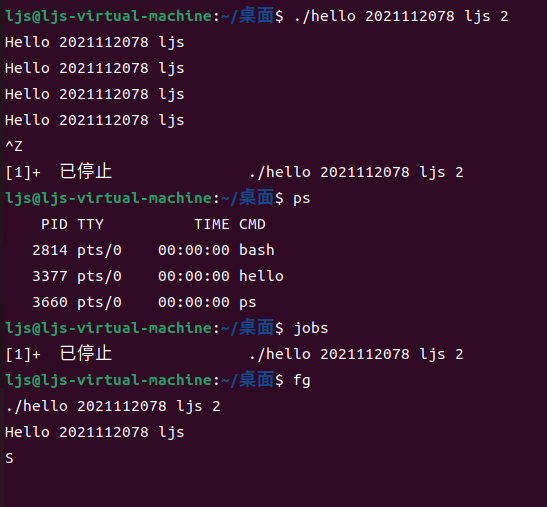


图 52 输入ctrl+z

Ctrl+C：进程收到 SIGINT 信号，结束 hello。在ps中查询不到其PID，在job中也没有显示，可以看出hello已经被彻底结束。

文本

描述已自动生成

图 53 输入ctrl+c

中途乱按：只是将屏幕的输入缓存到缓冲区。乱码被认为是命令。

文本

描述已自动生成

图 54 中途乱按

Kill命令：挂起的进程被终止，在ps中无法查到到其PID。

文本

描述已自动生成

图 55 输入Kill命令

## 6.7本章小结

本章介绍了有关进程管理的多个概念。介绍了shell的作用和处理流程，以及利用fork创建子进程、利用execve加载进程的方法。展示hello程序执行的具体过程，以及异常信号的处理机制。

# 第7章 hello的存储管理

## 7.1 hello的存储器地址空间

7.1.1 逻辑地址

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。在这里指的是hello.o中的内容。

7.1.2 线性地址

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hello的代码会产生段中的偏移地址，加上相应段的基地址就生成了一个线性地址。

7.1.3 虚拟地址

CPU启动保护模式后，程序hello运行在虚拟地址空间中。注意，并不是所有的“程序”都是运行在虚拟地址中。CPU在启动的时候是运行在实模式的，Bootloader以及内核在初始化页表之前并不使用虚拟地址，而是直接使用物理地址的。

7.1.4 物理地址

放在寻址总线上的地址。放在寻址总线上，如果是读，电路根据这个地址每位的值就将相应地址的物理内存中的数据放到数据总线中传输。如果是写，电路根据这个地址每位的值就在相应地址的物理内存中放入数据总线上的内容。物理内存是以字节(8位)为单位编址的。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

在 Intel 平台下，逻辑地址(logical address)是 selector:offset 这种形式，selector 是 CS 寄存器的值，offset 是 EIP 寄存器的值。如果用 selector 去 GDT( 全局描述符表 ) 里拿到 segment base address(段基址) 然后加上 offset(段内偏移)，这就得到了 linear address。我们把这个过程称作段式内存管理。

一个逻辑地址由段标识符和段内偏移量组成。段标识符是一个16位长的字段（段选择符）。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。

全局的段描述符，放在“全局段描述符表(GDT)”中，一些局部的段描述符，放在“局部段描述符表(LDT)”中。

给定一个完整的逻辑地址段选择符+段内偏移地址，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。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，就得到了其基地址。Base + offset = 线性地址。



图 56 地址计算过程

## 7.3 Hello的线性地址到物理地址的变换-页式管理

页式管理是一种内存空间存储管理的技术，页式管理分为静态页式管理和动态页式管理。将各进程的虚拟空间划分成若干个长度相等的页(page)，页式管理把内存空间按页的大小划分成片或者页面（page frame），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技术实现了内外存存储器的统一管理。

图示

描述已自动生成

图 57 虚拟地址存储原理

## 7.4 TLB与四级页表支持下的VA到PA的变换

7.4.1翻译后备缓冲器

每次CPU产生一个虚拟地址，MMU（内存管理单元）就必须查阅一个PTE（页表条目），以便将虚拟地址翻译为物理地址。在最糟糕的情况下，这会从内存多取一次数据，代价是几十到几百个周期。如果PTE碰巧缓存在L1中，那么开销就会下降1或2个周期。然而，许多系统都试图消除即使是这样的开销，它们在MMU中包括了一个关于PTE的小的缓存，称为翻译后备缓存器（TLB）。

7.4.2 多级页表：

将虚拟地址的VPN划分为相等大小的不同的部分，每个部分用于寻找由上一级确定的页表基址对应的页表条目。

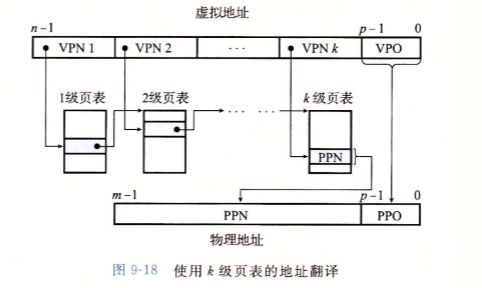


图 58 使用k级页表的地址翻译

7.4.3 VA到PA的变换

处理器生成一个虚拟地址，并将其传送给MMU。MMU用VPN向TLB请求对应的PTE，如果命中，则跳过之后的几步。MMU生成PTE地址(PTEA).，并从高速缓存/主存请求得到PTE。如果请求不成功，MMU向主存请求PTE，高速缓存/主存向MMU返回PTE。PTE的有效位为零, 因此 MMU触发缺页异常，缺页处理程序确定物理内存中的牺牲页 (若页面被修改，则换出到磁盘——写回策略)。缺页处理程序调入新的页面，并更新内存中的PTE。缺页处理程序返回到原来进程，再次执行导致缺页的指令。

在多级页表的情况下，无非就是不断通过索引 – 地址 – 索引 - 地址重复四次进行寻找。

## 7.5 三级Cache支持下的物理内存访问

CPU发送一条虚拟地址，随后MMU按照7.4所述的操作获得了物理地址PA。根据cache大小组数的要求，将PA分为CT（标记位）CI(组索引)，CO（块偏移）。根据CI寻找到正确的组，依次与每一行的数据比较，有效位有效且标记位一致则命中。如果命中，直接返回想要的数据。如果不命中，就依次去L2,L3,主存判断是否命中，命中时将数据传给CPU同时更新各级cache的储存。

图示, 示意图

描述已自动生成

图 59 物理内存访问

## 7.6 hello进程fork时的内存映射

当fork函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的PID，同时为这个新进程创建虚拟内存，创建当前进程的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位只读，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在新进程中返回时，新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。

图示

描述已自动生成

图 60 内存映射

## 7.7 hello进程execve时的内存映射

exeeve 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。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l.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亭区域内。

(4)设置程序计数器。cxccve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。如图所示:

图示

描述已自动生成

图 61 execve时的内存映射

## 

## 7.8 缺页故障与缺页中断处理

如果程序执行过程中发生了缺页故障，则内核调用缺页处理程序。处理程序执行如下步骤:

(1)检查虚拟地址是否合法，如界不合法则触发一个段错误，终止这个进程。

(2)检查进程是否有读、写或执行该区域页面的权限，如果不只有则触发保护异常，程序终止。

(3)两步检查都无误后，内核选择一个牺牲页面，如果该页面被修改过则将其交换出去，换入新的页面并更新页表。然后将控制转移给hello进程，再次执行触发缺页故障的指令。

图示

描述已自动生成

图 62 缺页处理

## 

## 7.9动态存储分配管理

动态储存分配管理使用动态内存分配器(如malloc)来进行。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。每个块就是一个连续的虚拟内存页，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的状态，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。动态内存分配主要有两种基本方法与策略：

7.9.1 带边界标签的隐式空闲链表分配器管理

带边界标记的隐式空闲链表的每个块是由一个字的头部、有效载荷、额外填充以及一个字的尾部组成。

隐式空闲链表：空闲块通过头部的大小字段隐含地连接着。分配器遍历堆中所有的块，间接地遍历整个空闲块的集合。

当一个应用请求一个k字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。分配器有三种放置策略：首次适配、下一次适配和最佳适配。分配器在面对释放一个已分配块时，可以合并相邻的空闲块，其中一种简单的方式，是利用隐式空闲链表的边界标记来进行合并。

7.9.2 显式空间链表管理

显式空闲链表是将堆的空闲块组织成一个双向链表，在每个空闲块中，都包含一个前驱与一个后继指针。进行内存管理。在显式空闲链表中。可以采用后进先出的顺序维护链表，将最新释放的块放置在链表的开始处，也可以采用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的后继地址，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。

## 7.10本章小结

本章介绍了存储器地址空间、段式管理、页式管理，VA 到 PA 的变换、物理内存访问， hello 进程fork时和execve 时的内存映射、缺页故障与缺页中断处理、包括隐式空闲链表和显式空闲链表的动态存储分配管理。

# 第8章 hello的IO管理

## 8.1 Linux的IO设备管理方法

Linux将文件所有的I/O设备都模型化为文件，甚至内核也被映射为文件。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。Linux就是基于Unix I/O实现对设备的管理。

设备的模型化:所有IO设备都被模型化为文件，所有的输入和输出都能被当做相应文件的读和写来执行。

设备管理:Linux 内核有一个简单、低级的接口，成为Unix IO，是的所有的输入和输出都能以一种统一且一致的方式来执行。

## 8.2 简述Unix IO接口及其函数

8.2.1 Unix lO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，来宣告它想要访间一个I/O设备。内核返回一个小的非负整致，叫做描述符，他在后续对此文件的所有操作中标识这个文件。

(2)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。

(3)读写文件。一个读操作就是从文件复制n个字节到内存，从当前文f位置k 开始，然后增加k到k+n。给定一个大小为m字节的文件，k>=m时执行型读操作会触发一个EOF的条件。

(4)关闭文件。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地址中。

8.2.2 Unix IO函数

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

进程通过调用open函数来打开一个存在的文件或是创建一个新文件的。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(fd)

进程通过调用close函数关闭一个打开的文件, fd是需要关闭的文件的描述符。(3 )ssize\_t read(int fd,void \*buf,sizc \_t n)

rcad函数从描述符为fd的当前文件位置赋值最多n个字节到内存位置 buf。返回值-1表示一个错误，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。

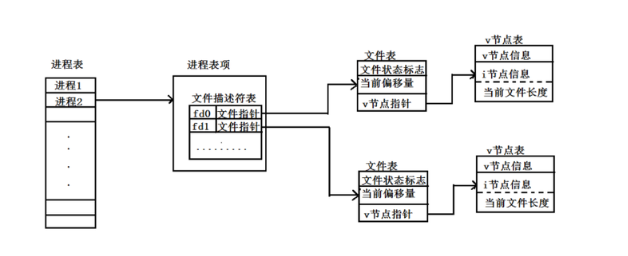


图 63 文件处理进程表

## 

## 8.3 printf的实现分析

printf函数的函数体  
int printf(const char \*fmt, ...)  
{  
int i;  
char buf[256];  
      va\_list arg = (va\_list)((char\*)(&fmt) + 4);  
     i = vsprintf(buf, fmt, arg);  
     write(buf, i);  
        return i;  
}

va\_list是字符指针， (char\*)(&fmt＋4)长示fmt后的第一个参数的地址。vsprintl函数返回值是要打印出来的字符中的长度，其作用是格式化，产生格式化的输出并保存在 buf中。最后的write函数即为写操作，把 buf中的i个元素的值写到终端。、

在write函数中，追踪之后的结果如下:

write:

mov eax,\_NR\_write

mov ebx,[esp + 4]

mov ecx,[esp + 8]

int INT\_VECTOR\_SYS\_CALL

一个int INT\_VECTOR\_SYS\_CALL表示要通过系统来调用sys\_call函数。在write函数中可以理解为其功能为显示格式化了的字符串。

字符显示驱动了程序: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

当程序调用getchar时，程序等待用户按键，用户输入的字符被存放在键盘缓冲区中直到用户按回车(回车也在缓冲区中)。

当用户键入回车之后，getchar才开始从stdio流中每次读入一个字符。getchar函数的返回值是用户输入的第一个字符的ascii码,如出错返回-1,且将用户输入的字符回显到屏幕。如用户在按回车之前输入了不止一个字符,其他字符会保留在键盘缓存区中,等待后续getchar调用读取。也就是说，后续的getchar调用不会等待用户按键,而直接读取缓冲区中的字符,直到缓冲区中的字符读完为后,才等待用户按键。

异步异常-键盘中断的处理：当用户按键时触发键盘终端，操作系统将控制转移到键盘中断处理子程序，中断处理程序执行，接受按键扫描码转成ascii码，保存到系统的键盘缓冲区，显示在用户输入的终端内。当中断处理程序执行完毕后，返回到下一条指令运行。

## 

## 8.5本章小结

本章介绍了 Linux 的 I/O 设备的基本概念和管理方法，以及Unix I/O 接口及其函数。最后分析了printf 函数和 getchar 函数的工作过程。

# 

# 结论

首先由程序员将hel1o 代码从键盘输入，依次要经过一下步骤:

1、预处理(cpp)。将 hello.c进行预处理，将文件调用的所有外部库文件合并展开，生成一个经过修改的hello.i文件。

2、编译(ccl)。将 hello.i文件翻译成为一个包含汇编语言的文件 hello.s 。

3、汇编(as)。将hello.s翻译成为一个可重定位目标文件hello.o。

4、链接(ld)。将 hello.o文件和可重定位目标文件和动态链接库链接起来，生成一个可执行目标文件 hello。

5、运行。在shell 中输入./hello 2021112078 ljs 2。

6、创建进程。终端判断输入的指令不是shell内置指令，于是调用fork函数创建一个新的子进程。

7、加载程序。shell 调用execve函数，启动加载器，映射虚拟内存，进入程序入口后程序开始载入物理内存，然后进入main函数。

8、执行指令:CPU为进程分配时间片，在一个时问片中，hello享有CPU资源，顺序执行自己的控制逻辑流。

9、访问内存: MMU将程序中使用的虚拟内存地址:通过页表映射成物理地址。10、动态申请内存:printf会调用malloc向动态内存分配器申请堆中的内存。11、信号管理:当程序在运行的时候我们输入Ctrl+c，内核会发送SIGINT信号给进程并终止前台作业。当输入Ctr1+z时，内核会发送SIGTSTP信号给进程，并将前台作业停止挂起。

12、终止:当了进程执行完成时，内核安排父进程回收了进程，将了进程的退出状态传递给父进程。内核删除为这个进程创建的所有数据结构。

通过本次实验，我深切感受到计算机系统的精细和强大，每一个简弟的任务都需要计算机的各种复杂的操作来完成，这背后体现出了严谨的逻辑和现代工艺的精巧。

# 

# 附件

# 表3 ：附件文件

|  |  |
| --- | --- |
| **文件名** | **文件作用** |
| hello.c | 源代码文件 |
| hello.i | 预处理后生成的文件 |
| hello.s | 编译后生成的文件 |
| hello.o | 汇编后生成的可重定位目标程序 |
| hello | 链接之后生成的可执行程序 |
| hello\_o\_asm.txt | hello.o的反汇编文件 |
| hello\_o\_elf.txt | hello.o的ELF文件 |
| hello\_run\_asm | hello的反汇编文件 |
| hello\_run\_elf | hello的ELF文件 |

# 参考文献

[1] https://blog.csdn.net/qq\_47582681/article/details/118109899

[2] <https://blog.csdn.net/qq_45499104/article/details/118224864>

[3] https://zhidao.baidu.com/question/646759751623911205.html

[4] https://blog.csdn.net/qq\_52943402/article/details/124833002

[5] <https://blog.csdn.net/dong_daxia/article/details/95328479>

[6] <https://blog.csdn.net/dingdingdodo/article/details/107747509>