

**计算机系统**

**大作业**

题　　 目 程序人生-Hello’s P2P

专　　 业 人工智能

学　　 号 2021112845

班　　 级 2103601

学 生 张智雄

指 导 老 师 郑贵滨

**计算机科学与技术学院**

**2023年5月**

**摘 要**

Hello的一生经历了许多过程，从预处理、编译、汇编、链接到进程管理、存储管理再到IO管理。其中操作系统、壳和硬件为它的表演提供了支持和保障，让它得以在计算机系统中完整运行。本文在Linux下完整探讨了hello.c从编写完成到最终执行完毕的整个生命周期，结合所学知识逐步对比解析各个过程在Linux下实现机制及原因，较为深入研究了hello.c文件的P2P和020的过程。

**关键词：**计算机系统；计算机体系结构；P2P；020

**CSDN地址：**

<https://blog.csdn.net/m0_62249876/article/details/130827769?spm=1001.2014.3001.5501>

**目 录**

[第1章 概述 - 4 -](#_Toc134091777)

[1.1 Hello简介 - 4 -](#_Toc134091778)

[1.2 环境与工具 - 5 -](#_Toc134091779)

[1.3 中间结果 - 5 -](#_Toc134091780)

[1.4 本章小结 - 5 -](#_Toc134091781)

[第2章 预处理 - 6 -](#_Toc134091782)

[2.1 预处理的概念与作用 - 6 -](#_Toc134091783)

[2.2在Ubuntu下预处理的命令 - 7 -](#_Toc134091784)

[2.3 Hello的预处理结果解析 - 7 -](#_Toc134091785)

[2.4 本章小结 - 7 -](#_Toc134091786)

[第3章 编译 - 8 -](#_Toc134091787)

[3.1 编译的概念与作用 - 8 -](#_Toc134091788)

[3.2 在Ubuntu下编译的命令 - 9 -](#_Toc134091789)

[3.3 Hello的编译结果解析 - 9 -](#_Toc134091790)

[3.4 本章小结 - 12 -](#_Toc134091791)

[第4章 汇编 - 13 -](#_Toc134091792)

[4.1 汇编的概念与作用 - 13 -](#_Toc134091793)

[4.2 在Ubuntu下汇编的命令 - 13 -](#_Toc134091794)

[4.3 可重定位目标elf格式 - 14 -](#_Toc134091795)

[4.4 Hello.o的结果解析 - 17 -](#_Toc134091796)

[4.5 本章小结 - 18 -](#_Toc134091797)

[第5章 链接 - 19 -](#_Toc134091798)

[5.1 链接的概念与作用 - 19 -](#_Toc134091799)

[5.2 在Ubuntu下链接的命令 - 19 -](#_Toc134091800)

[5.3 可执行目标文件hello的格式 - 20 -](#_Toc134091801)

[5.4 hello的虚拟地址空间 - 23 -](#_Toc134091802)

[5.5 链接的重定位过程分析 - 24 -](#_Toc134091803)

[5.6 hello的执行流程 - 24 -](#_Toc134091804)

[5.7 Hello的动态链接分析 - 25 -](#_Toc134091805)

[5.8 本章小结 - 26 -](#_Toc134091806)

[第6章 hello进程管理 - 27 -](#_Toc134091807)

[6.1 进程的概念与作用 - 27 -](#_Toc134091808)

[6.2 简述壳Shell-bash的作用与处理流程 - 27 -](#_Toc134091809)

[6.3 Hello的fork进程创建过程 - 27 -](#_Toc134091810)

[6.4 Hello的execve过程 - 28 -](#_Toc134091811)

[6.5 Hello的进程执行 - 29 -](#_Toc134091812)

[6.6 hello的异常与信号处理 - 29 -](#_Toc134091813)

[6.7本章小结 - 32 -](#_Toc134091814)

[第7章 hello的存储管理 - 33 -](#_Toc134091815)

[7.1 hello的存储器地址空间 - 33 -](#_Toc134091816)

[7.2 Intel逻辑地址到线性地址的变换-段式管理 - 33 -](#_Toc134091817)

[7.3 Hello的线性地址到物理地址的变换-页式管理 - 34 -](#_Toc134091818)

[7.4 TLB与四级页表支持下的VA到PA的变换 - 34 -](#_Toc134091819)

[7.5 三级Cache支持下的物理内存访问 - 35 -](#_Toc134091820)

[7.6 hello进程fork时的内存映射 - 36 -](#_Toc134091821)

[7.7 hello进程execve时的内存映射 - 36 -](#_Toc134091822)

[7.8 缺页故障与缺页中断处理 - 37 -](#_Toc134091823)

[7.9本章小结 - 37 -](#_Toc134091824)

[结论 - 38 -](#_Toc134091825)

[附件 - 39 -](#_Toc134091826)

[参考文献 - 40 -](#_Toc134091827)

# 第1章 概述

**（0.5分）**

## 1.1 Hello简介

Hello.c是一个简单的C语言程序，能够实现简单字符串信息的循环输入输出，其编译执行的过程可分为P2P和020两个部分：

1.1.1 P2P(From Program to Process)

此过程指的是将C语言程序文件 hello.c 转换为可执行进程的过程。在 Linux 系统中，这个过程经历了如下几个步骤：

1. **预处理：**hello.c经过C预处理器cpp(C Pre-Processor)进行预处理，进行宏展开、头文件包含等操作，生成一个被预处理过的hello.i文件。
2. **编 译：**C编译器ccl(C Compiler)对hello.i进行词法分析、语法分析和优化等操作，生成汇编代码生成hello.s。
3. **汇 编：**汇编器as(Assembler)将hello.s翻译成机器语言指令，并生成一个目标文件hello.o。
4. **链 接：**链接器ld(Linker) 将hello.o和其他依赖的库文件进行链接，生成一个可执行文件hello。
5. **运 行：**在shell内输入命令./hello，shell会通过系统调用fork()为其创建一个子进程，并在子进程执行hello。

1.1.2 020(From Zero-0 to Zero-0)

020是指将一个可执行文件hello.out载入内存并运行的过程。此过程中，0表示内存中没有hello文件的状态。具体而言，020在Linux下包含以下过程：

1. **内存载入：**当程序运行开始时，子进程调用execve函数将hello文件载入内存中。并调用mmap函数将其映射到内存中的合适位置。
2. **进程控制：**内核中进程控制器为hello进程分配时间片，使其开始执行自身的逻辑控制流。
3. **进程回收：**当程序运行结束后，父进程会回收hello进程，并在内核中删除相关的数据，使得内存恢复初始到与hello无关的状态。

## 1.2 环境与工具

1. 硬件环境：X64 CPU；2.30GHz；16G RAM；1.5THD disk
2. 软件环境：Windows11 64位；Vmware Workstation 17 Pro；Ubuntu 22.10
3. 开发与调试工具：Visual Studio 2019 64位；CodeBlocks 64位；vim+gcc； readelf； objdump；ldd；EDB等

## 1.3 中间结果

|  |  |
| --- | --- |
| 文件名称 | 功能 |
| hello.i | hello.c预处理后生成的中间文件 |
| hello.s | hello.i编译后生成的汇编文件 |
| hello.o | hello.s汇编后生成的可重定位目标文件 |
| hello | hello.o和其他库文件经链接后生成的可执行文件 |
| hello\_o.elf | 用readelf读取hello.o得到的ELF格式信息 |
| hello.elf | 由hello可执行文件生成的ELF格式文件 |
| hello\_o\_objdump.asm | 反汇编hello.o得到的.asm文件 |
| hello\_objdump.asm | 反汇编hello可执行文件得到的.asm文件 |

表1-3 中间文件名称及功能

## 1.4 本章小结

本章首先是对hello做了简单的介绍，对hello的P2P和020过程进行了简要概述，介绍了大作业过程中应用的软硬件环境和开发与调试工具，列举了任务过程中出现的中间文件及其作用，是本次任务的总纲部分，后文将依据本章做详细展开。

# 第2章 预处理

**（0.5分）**

## 2.1 预处理的概念与作用

在对Hello.c编译执行的过程中，首先需要对Hello.c文件进行预处理，得到中间文件Hello.i，为编译器进行下一步的处理做准备。

2.1.1 预处理的概念

预处理(Preprocessin)是计算机科学中编译器的一种重要处理阶段，用于在实际编译之前对源代码进行预处理和转换，主要目的是为了简化编程工作，提高代码的复用性和可维护性。此过程并不包括对源代码内容的解析，只是进行一些简单的插入、删除和替换等文本操作。

在C/C++语言中，预处理器是一种特殊的程序，负责在编译源代码之前对其进行预处理。预处理器会解析源代码中以“#”开头的预处理指令，并根据这些指令对源代码进行修改和扩展，生成新的源代码文件(\*.i)，使其更适合后续的编译、汇编和链接等过程。

2.1.2 预处理的作用

预处理器(C Pre-Processor)的主要功能是在编译过程之前对源代码进行处理，将源代码中的宏定义、条件编译指令、包含其他文件的指令等预处理指令处理完毕后生成新的源代码文件，以便编译器对其进行编译。

1. **宏定义处理：**为提高代码的重用性和可读性，程序员往往通过宏定义将一些常量或表达式进行封装。而在编译时，预处理器会将宏定义展开为实际的表达式或代码，以便编译器对其进行处理。
2. **文件包含处理：**预处理器可以通过#include指令将其他文件的内容包含到当前的源代码文件中，以方便程序的开发和维护。
3. **条件编译指令处理：**预处理器可以通过#define、#ifdef、#ifndef、#endif等指令实现条件编译，以便在不同的环境下编译不同的代码，或者实现不同的功能。
4. **注释处理：**预处理器可以将源代码中所有的注释删除，以减小程序的大小和提高执行效率。

## 2.2在Ubuntu下预处理的命令

预处理的命令：gcc -E hello.c -o hello.i

图形用户界面, 文本, 应用程序

描述已自动生成

图2-2 hello.c预处理

## 2.3 Hello的预处理结果解析

使用vim打开hello.i，发现代码由23行(\*.c)拓展至了3104行(\*.i)。而原来的main函数保留在文件最后，而剩余部分则是对stdio.h、unistd.h、stdlib.h等头文件的包含展开，同时删除了所有注释，使代码更加完整而不冗余。

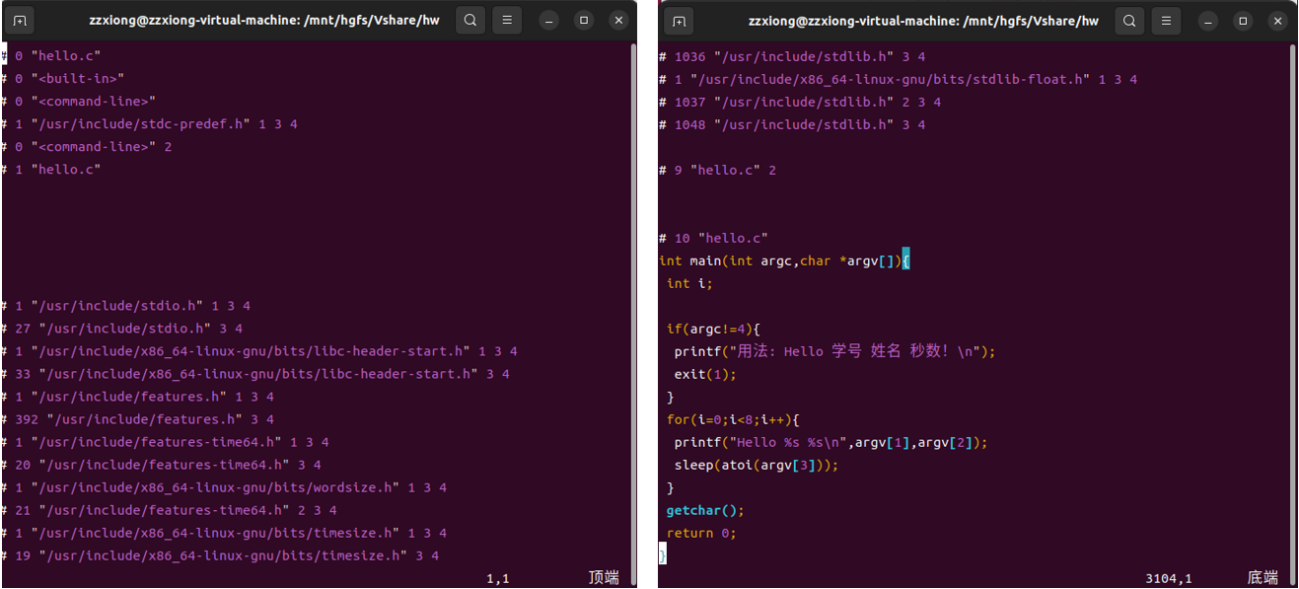


图2-3 hello.i部分代码

## 2.4 本章小结

本章主要介绍了预处理的概念及作用、gcc下的预处理指令，并对预处理器预处理hello.c生成hello.i文件的过程结合Ubuntu下实际运行结果作了简单分析。结合所学进一步了解了预处理的过程，明确了预处理在整个P2P过程中的重要性。

# 第3章 编译

**（2分）**

## 3.1 编译的概念与作用

对Hello.c预处理得到包含完整程序代码的hello.i文件过后，下一步是对hello.i进行编译，得到汇编文件hello.s。

3.1.1 编译的概念

编译(Compilation)是将高级编程语言（如C、C++等）的源代码转换成汇编语言(\*.s)的过程。在编译的过程中，源代码经过词法分析、语法分析、语义分析等步骤，转换成对应的中间表示形式，即汇编代码。编译的主要目的是提高程序的执行效率，使程序更加稳定和安全。

编译器是完成编译过程的程序，它将高级程序源代码作为输入，通过语法分析、语义分析、优化和代码生成等多个阶段，根据源代码的语法、数据类型、函数定义等信息，对源代码进行检查、转换和优化，生成对应的汇编代码文件。

3.1.2 编译的作用

编译器能够对(\*.i)中间文件进行词法分析、语法分析、语义分析、代码优化、代码生成等功能，从而将其转换为(\*.s)汇编文件。

1. **词法分析：**将源代码按照语言的语法规则进行分词，并将每个词归类为不同的语言元素，例如：关键字、标识符、运算符、分隔符等。
2. **语法分析：**将词法分析生成的语言元素按照语言的语法规则组织成一棵抽象语法树，从而描述源代码中的语法结构。
3. **语义分析：**在语法树上进行类型检查、变量和函数定义检查等操作，确保源代码符合语言的语义规则。
4. **代码优化：**对生成的中间代码进行多种等价变换，在不改变其功能的前提下优化代码，以提高程序的性能和效率。
5. **代码生成：**将优化后的中间代码翻译为汇编语言代码，并进行一些额外的处理，例如符号表的生成等。

**注**：这里的编译是指从.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

编译的命令：gcc -S hello.i -o hello.s

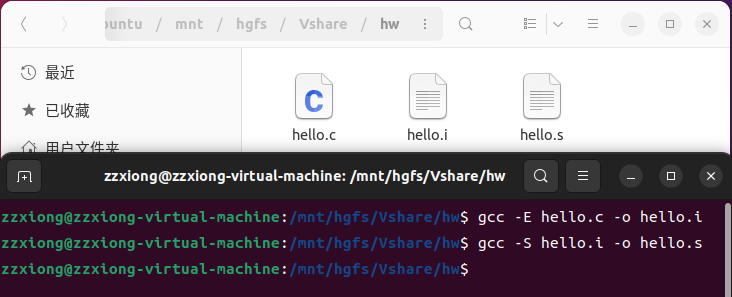


图3-2 hello.i编译

## 3.3 Hello的编译结果解析

hello.i编译得到hello.s文件，以下将对hello.s使用的伪指令、编译过程中对各个数据类型的处理以及各类操作进行分析。

3.3.1 hello.s伪指令

伪指令指导编译器和汇编器进行代码生成，首先了解各伪指令的含义有助于梳理.s文件结构，从而更好理解编译的过程。

|  |  |
| --- | --- |
| 内容 | 含义 |
| .file | 源文件声明 |
| .text | 代码段 |
| .section | 指定接下来的指令和数据所属的段 |
| .rodata | 只读代码段 |
| .align | 指令或者数据的存放地址的对齐方式 |
| .global | 声明全局符号 |
| .data | 存放已经初始化的全局和静态变量 |
| .type | 声明符号是数据/函数类型 |
| .long/.string | 声明数据类型long/string |

表3-3 hello.s伪指令及其含义

3.3.2 hello.s数据

hello.s中包含常量和变量两种类型，

1. **常量：**字符串常量(存储在.rodata节)和整数常量(立即数形式)



图3-3-2-1 hello.s常量数据

1. **变量：**循环变量-整数i（局部变量）、main()函数参数-整数argc、字符指针数组argv[ ]（外部变量，由寄存器加载至堆栈帧中）。

图表, 折线图

描述已自动生成

图3-3-2-2 hello.s变量数据

3.3.3 hello.s数据操作

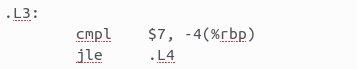
hello.s中包含赋值、比较、加法以及数组操作四种数据操作。

1. **赋值：**mov指令(对循环变量i赋初值0)



1. **比较：**cmp指令(对argc&4、i&8的数值进行比较)





1. **加法：** add指令(对循环变量进行累加i++)



1. **数组操作：**movq指令(输出时以及调用atoi函数时将argv[ ]内数据传出)

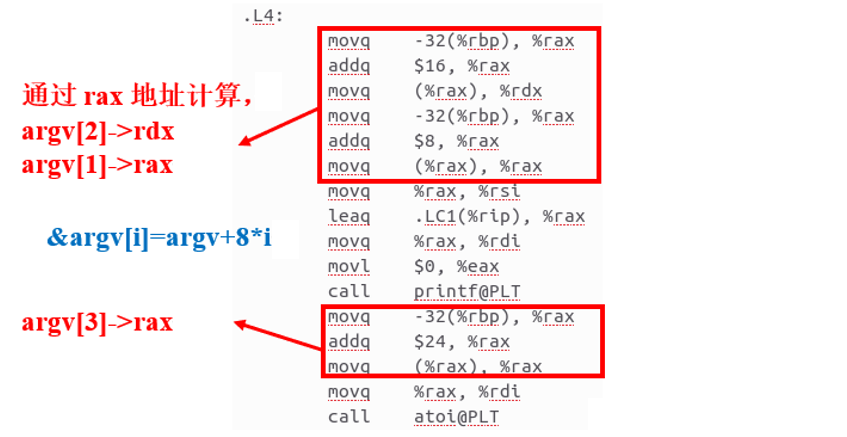


图3-3-3 hello.s数据操作

3.3.4 hello.s控制转移

hello.s中包含if条件分支引起的跳转以及for循环分支引起的跳转，通过cmpl指令设置条件码后通过je/jle指令跳转。

1. If条件跳转：cmpl判断argc是否为4，是则状态寄存器的ZF位置位，并通过je指令跳转至.L2节；否则跳过je继续顺序执行。



1. For循环跳转：cmpl判断i是否小于等于7，是则通过jle指令跳转至.L4节，继续执行循环体内的内容；否则跳过jle继续顺序执行。



图3-3-4 hello.s控制转移

3.3.5 hello.s函数操作

hello.s中包含参数传递、函数调用、函数返回等函数操作。

1. 参数传递：调用函数之前，先将相应参数传到%edi、%rdi寄存器作为函数参数(如将立即数1传入exit函数、取argv[3]的值传入atoi函数、将atoi函数的返回值传入sleep函数)。

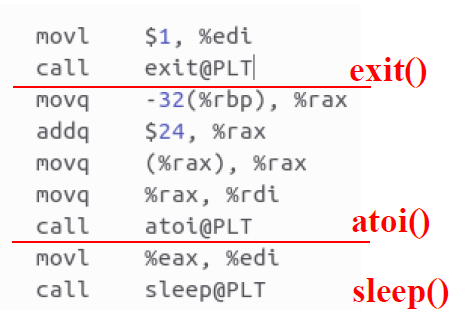


图3-3-5-1 hello.s参数传递

1. 函数调用：call指令调用puts、exit、printf、atoi、sleep、getchar函数。

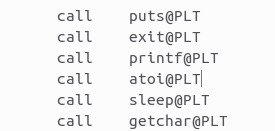


图3-3-5-2 hello.s函数调用

1. 函数返回：函数返回时，通过leave指令释放堆栈帧，并将返回值存在rax中，而后通过ret返回。

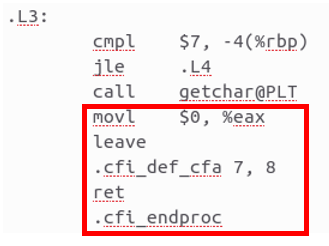


图3-3-5-3 hello.s函数返回

## 3.4 本章小结

本章详细阐述了编译的概念和作用，简单描述了Ubuntu下hello.i经过编译生成hello.s的过程，并结合hello.s的具体汇编代码解析了编译过程中编译器对各个数据类型进行的处理以及各类操作。观察发现，相较于预处理后的hello.i，hello.s的代码更为精简，几乎只包含main函数的内容，且这些汇编代码已经具备在指令级别上控制硬件资源的能力，是程序的机器级表示过程中的重要一步。

# 第4章 汇编

**（2分）**

## 4.1 汇编的概念与作用

汇编文件hello.s已经具备在指令级别上控制CPU进行数据传递等操作，但仍需要将汇编语言汇编生成机器语言hello.o来实现程序的执行。

4.1.1 汇编的概念

汇编是将编译后的汇编语言程序(\*.s)翻译成机器可识别执行的机器指令，并将这些指令打包成一种叫做可重定位目标程序，进而转换成机器语言程序(\*.o)的过程，它是编译器生成可执行文件的一个重要步骤。

汇编程序使用的是一些特殊的指令，这些指令直接对应到底层的硬件架构，因此生成的机器语言程序可以直接在计算机上运行。

4.1.2 汇编的作用

汇编能够将人类可读的汇编代码转化为机器可执行的指令，这些指令可以被计算机处理器直接识别和执行。通过汇编，程序员可以直接控制计算机底层的操作，实现高效的程序代码。

因此，汇编在操作系统、嵌入式系统、驱动程序等领域有着广泛的应用。同时，汇编也是高级语言编译器、解释器等软件工具的重要基础，因为这些工具需要将高级语言翻译成汇编代码后再进行处理。

注：这里的汇编是指从.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

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

图形用户界面, 文本, 应用程序

描述已自动生成

图4-2 hello.s汇编

## 4.3 可重定位目标elf格式

首先输入命令readelf -a hello.o > hello\_o.elf，获得hello.o的ELF格式。观察发现hello.elf文件包含ELF头(ELF Header)、节头(Section Header)、符号表、重定位节等部分。

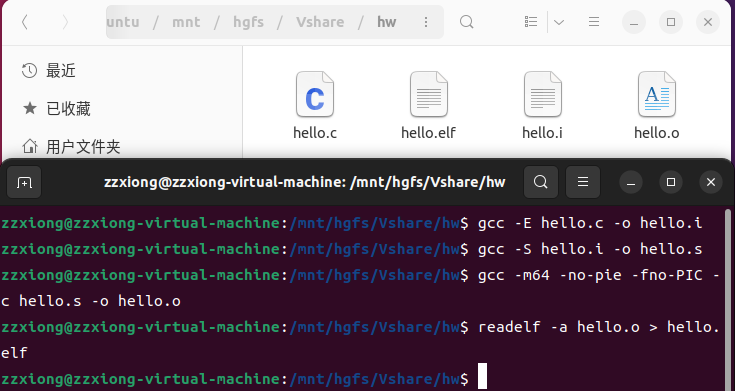


图4-3 hello.o生成hello.elf

4.3.1 ELF头(ELF Header)

ELF头从一个16字节序列magic开始，描述了生成该文件的系统的字的大小和字节顺序，开头的4字节7f 45 4c 46分别对应删除(Del) E L F的ASCII码，操作系统在加载可执行文件时会确认magic序列是否正确。

而Header剩下部分包含帮助链接器语法分析和解释目标文件的信息，包括数据类型及字节序、OS/ABI、目标文件类型(REL可重定位、EXEC可执行或共享的)、机器类型(x86-64)、节头部表(section header table)的文件偏移量、规模以及包含条目的大小和数量等相关信息。

文本

描述已自动生成

图4-3-1 ELF头信息

4.3.2 节头(Section Header)

节头列表包含文件中各节的名称、类型、地址、偏移量、大小、旗标、链接和对齐信息等内容，在hello.elf文件中共包含14个节。

文本

描述已自动生成

图4-3-2 ELF节头信息

其中部分节的名称以及内容列举如下：

|  |  |
| --- | --- |
| 名称 | 内容 |
| .text | 已编译程序的机器代码段 |
| .rela.text/.rela.eh\_frame | 被模块引用或全局变量等需要重定位的信息 |
| .data | 已初始化的全局和静态变量 |
| .bss | 未初始化/初始化0的全局和静态变量 |
| .rodata | 只读数据段 |
| .comment | 编译器版本信息 |
| .symtab | 当前目标文件中所有对符号的定义和引用. |
| .strtab | 静态链接的字符串表 |
| .shstrtab | 节区头部表名字的字符串表 |

4.3.3 符号表(Symbol Table)

符号表存放程序中定义和引用的函数和全局变量的相关信息，包括相对于目标节的起始位置偏移Value、大小Size、类型Type/Bind以及符号名称Name。由于在此还没有进行相关库函数的链接，因而Value都为0。

日历

描述已自动生成

图4-3-3 ELF符号表信息

4.3.4 重定位节(Relocation section)

此部分包含main.o中需要重定位的信息，当链接器把目标文件和其他文件组合时，需要根据具体类型修改这些位置。每条重定位条目包含偏移量、信息、类型（标识对应的地址计算算法）、符号值、符号名称、加数等信息。下图.rela.text是hello.c中调用函数的重定位条目；.rela.eh\_frame是对.text代码段的重定位条目。

日历

低可信度描述已自动生成

图4-3-4 ELF重定位节信息

## 4.4 Hello.o的结果解析

通过objdump -d -r hello.o反汇编hello.o，并与hello.s对照如下图所示。发现反汇编代码与hello.s基本相似，汇编后调用函数以及访问内存时还需链接器作用才能确定数据/函数地址，而hello.s与反汇编代码在处理这部分存在一些区别：

1. **分支转移**：在hello.s中，分支转移是由通过节头(.L2等)来标识，而在机器语言反汇编程序中，分支转移直接跳入目的地址
2. **数据访问**：在hello.s中通过.LC0(%rip)访问.rodata段数据，而反汇编语言中通过0x0(%rip) 访问。
3. **函数调用**：在hello.s中，函数调用通过call<函数名称>实现函数调用；而在反汇编程序中，call的目标地址是当前指令的下一条指令。

文本

描述已自动生成

图4-4 hello.o反汇编与hello.s部分对照

同时对比反汇编代码中左侧机器语言与右侧汇编代码，可以得出：

1. **机器语言**是由二进制代码表示的，计算机能够直接识别和执行的机器指令的集合。
2. 且汇编语言和二进制机器语言**一一映射**，即每一条汇编指令都可以用机器语言来表示，同时每一条机器指令都可以用汇编代码来表示。

## 4.5 本章小结

本章主要介绍了汇编操作，此过程将汇编语言转化为机器语言，生成可重定位目标文件，为下一步链接生成可执行文件作准备。并在Ubuntu下展示了汇编的执行方法，并利用readelf查看了hello.o的ELF格式，对其各节的基本信息做了简单分析。而后通过反汇编方式比较了hello.o的反汇编代码和hello.s的区别，比较了汇编语言与机器语言的异同，进一步加深了对汇编的概念和作用的理解。

# 第5章 链接

**（1分）**

## 5.1 链接的概念与作用

hello调用了printf、atoi等函数，它们在之前的过程中并没有被编译，而是存储在预编译的库文件中，通过链接将这些文件合并成最终的可执行文件。

5.1.1 链接的概念

在计算机科学中，链接(Linking)是指将编译后的目标文件和所需的库文件链接在一起，生成可执行文件的过程。链接又可以分为静态链接和动态链接。

**静态链接**指的是在编译链接过程中将目标文件和库文件合并成一个单独的可执行文件，即在可执行文件中嵌入了所有的目标文件和库文件的代码和数据。

**动态链接**则是在运行时将程序需要的库文件动态加载到内存中，并建立相应的映射关系，程序中只包含引用库函数的符号信息。

5.1.2 链接的作用

在编译过程中，链接的作用是将多个目标文件（包括库文件）合并成一个可执行文件或共享库，解决了在编译过程中分离编译带来的符号（函数、变量）的引用等问题，最终生成一个可执行文件。

具体而言，链接过程涉及到符号表的生成和解析、重定位等操作，其中符号表用于记录不同目标文件中的符号信息，重定位则用于将目标文件中的符号引用关联到正确的符号定义位置上。在链接过程中还可能会进行优化操作，如去除未使用的符号和函数、合并重复的函数等，以减小最终可执行文件或共享库的大小。

在实际应用中，链接提供了一种模块化的编程方式，使得程序可以被划分为能够分开编译修改的源文件集合，从而减少整个程序的复杂度和大小，提高了程序的可读性和可维护性。

**注：**这里的链接是指从 hello.o 到hello生成过程。

## 5.2 在Ubuntu下链接的命令

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

图形用户界面, 文本, 应用程序

描述已自动生成

图5-2 hello.o链接

## 5.3 可执行目标文件hello的格式

输入指令readelf -a hello > hello.elf生成hello程序的ELF格式文件，观察各段的基本信息，并与上文4.3节中由hello.o生成的hello\_o.elf进行对比。

5.3.1 ELF头(ELF Header)

ELF头与hello.elf的基本相同，不同之处在于文件类型由REG变为EXEC可执行目标文件，同时节头大小数量增加，并获得了入口地址。

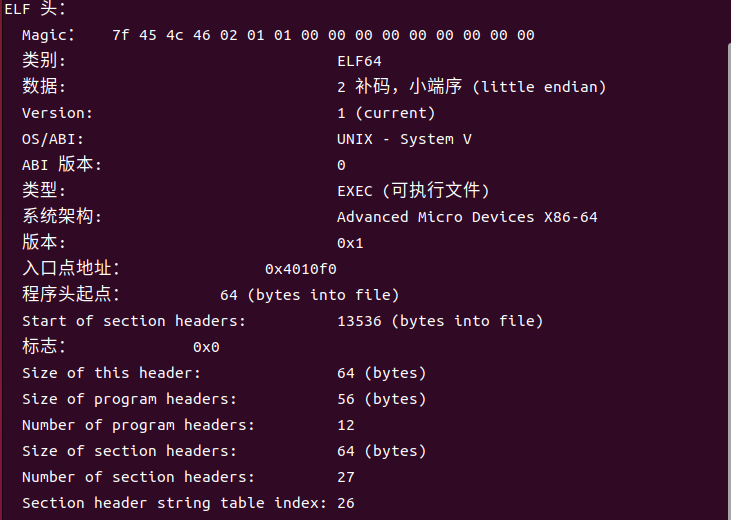


图5-3-1 ELF头信息

5.3.2 节头(Section Header)

链接后，节头数量有所增加，但各条目内包含信息种类并未发生变化。

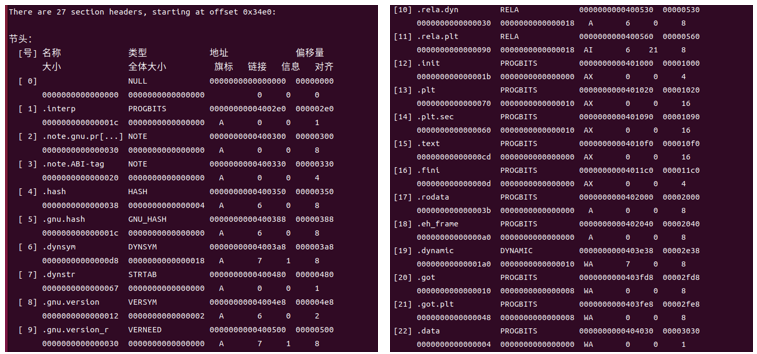


图5-3-2 ELF节头信息

5.3.3 符号表(Symbol Table)

链接后，符号表条目显著增加，包含各目标文件中符号定义及引用信息。

图形用户界面, 文本

描述已自动生成文本

描述已自动生成

图5-3-3 ELF符号表信息

5.3.4 重定位节(Relocation section)

链接后，重定位节内容变化为执行过程中需要通过动态链接调用的函数，同时类型也发生改变。

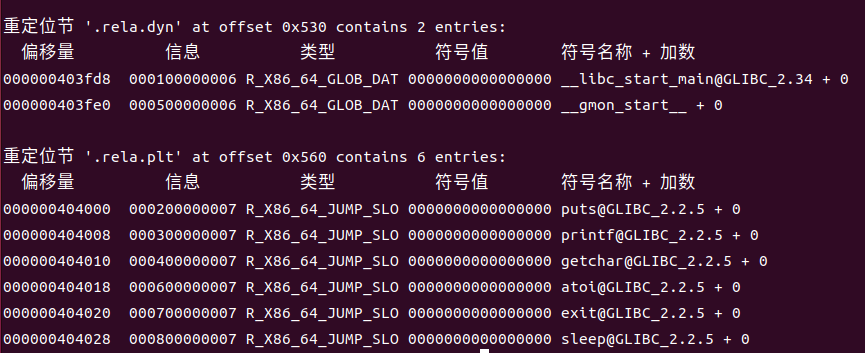


图5-3-4 ELF重定位节信息

5.3.5 其他内容

链接后，相较于hello\_o.elf，hello.elf增加了程序头和段节。二者均是在链接过程中确定，其中程序头描述了系统准备程序执行所需的段和其他信息。

表格

描述已自动生成

表格

描述已自动生成图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图5-3-5 hello.elf新增部分

## 5.4 hello的虚拟地址空间

使用edb加载hello，观察Data Dump部分，观察发现程序占有0x401000~ 0x402000的地址空间，通过5.3节中入口地址及各节的偏移量即可观察各节内容。

文本, 表格

描述已自动生成

图5-4 edb虚拟内存窗口

## 5.5 链接的重定位过程分析

输入指令objdump -d -r hello，观察其中的的main函数，并与4.4节中hello.o的反汇编代码进行对比。

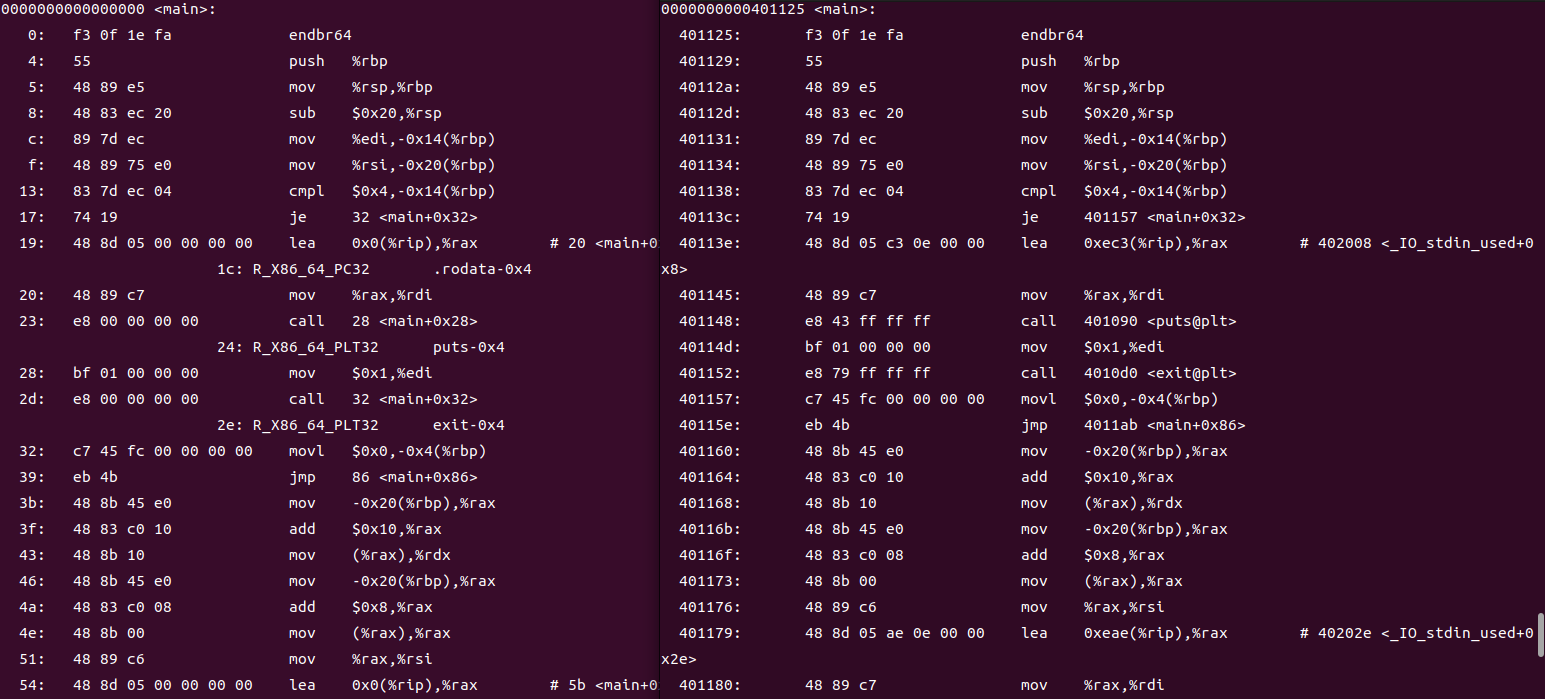


图5-5 反汇编对比（左为hello.o反汇编，右为hello反汇编）

对比两段反汇编代码可以发现：

1. hello比hello.o多出了许多文件节以及外部链接的函数，如printf函数、.init节和.plt节等；
2. hello反汇编代码中jle/je跳转地址是确定的地址，而hello.o反汇编代码中的是相对偏移地址。
3. hello中函数调用时使用call直接跟函数代码所在的目标地址，指向对应的代码段，而hello.o反汇编代码中同样是相对偏移地址。

由此可以分析得出，编译时，重定位的作用是将编译后的目标文件中的地址和符号关联起来，使得它们可以正确地在内存中加载和执行。

具体来讲，链接器在完成符号解析以后，链接器根据重定位节和符号定义将相同类型的节合并，生成ELF节。同时对程序分配虚拟内存空间，使得程序仅有唯一的运行地址。而后对引用符号进行重定位，根据.rel\_data和.rel\_text节中保存的重定位信息修改.text节和.data节中对每个符号的引用。

## 5.6 hello的执行流程

使用EDB执行hello，从加载hello到\_start，到call main，以及程序终止的所有过程如下表。

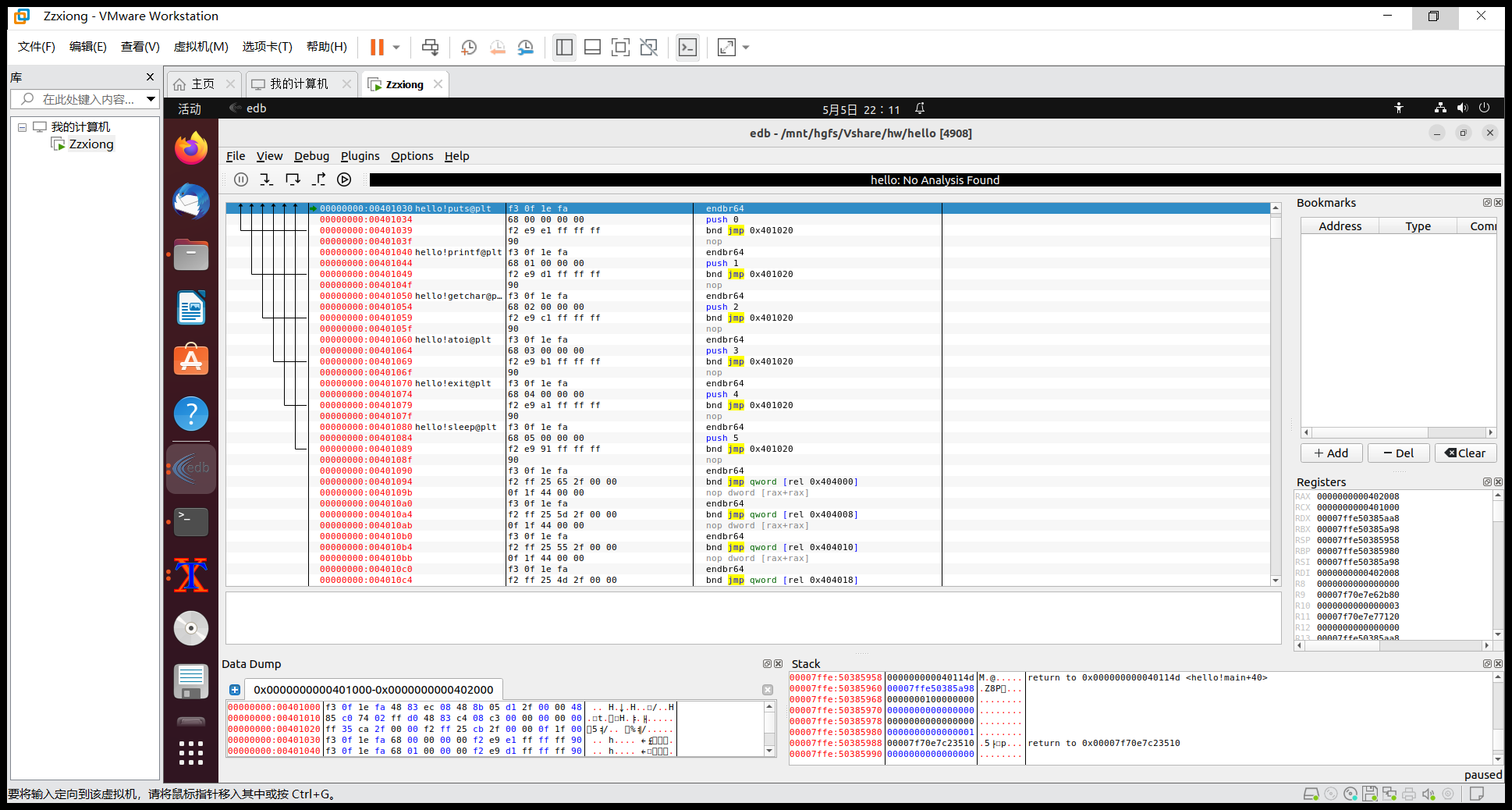


图5-6 EDB执行hello

|  |  |
| --- | --- |
| 程序名称 | 程序地址 |
| ld-2.27.so!\_dl\_start | 0x7ffe68c3a148 |
| ld-2.27.so!\_dl\_init | 0x7fce8cc47630 |
| hello!\_start | 0x401090 |
| hello!\_init | 0x401000 |
| ld-2.27.so!.plt | 0x401020 |
| hello!main | 0x4010c1 |
| hello!puts@plt | 0x401030 |
| hello!printf@plt | 0x401040 |
| hello!getchar@plt | 0x401050 |
| hello!atoi@plt | 0x401060 |
| hello!exit@plt | 0x401070 |
| hello!sleep@plt | 0x401080 |
| hello!\_dl\_relocate\_static\_pie | 0x4010c0 |
| -libc-2.27.so!\_\_libc\_csu\_init | 0x4005c0 |
| libc-2.27.so!exit | 0x7fce8c889128 |

表5-6 hello执行流程

## 5.7 Hello的动态链接分析

对于动态共享链接库中PIC函数，编译器没有办法预测函数的运行时地址，所以需要为其添加重定位记录，并等待动态链接器处理。为避免运行时修改调用模块的代码段，链接器采用延迟绑定的策略。动态链接器使用过程链接表PLT和全局偏移量表GOT实现函数的动态链接。其中GOT 中存放函数目标地址，PLT使用 GOT中地址跳转到目标函数。查询hello得.got起始位置为0x403fd8，而.got.plt起始位置为0x404000。

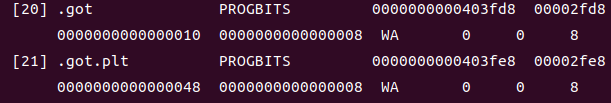


图5-7-1 查询起始位置

查看edb的Data Dump，在调用dl\_init前，其内容如下：

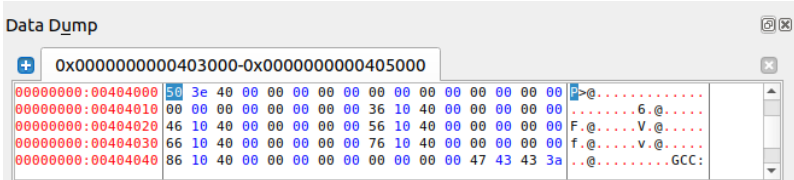


图5-7-2 调用前.got

调用后，.got中的条目已经改变，说明动态链接完成。

图片包含 游戏机

描述已自动生成

图5-7-3 调用后.got

## 5.8 本章小结

本章主要介绍了链接的概念和作用，以及在Ubuntu下使用ld指令进行链接的方法。通过对hello和hello.o的反汇编代码以及ELF格式文件的比较分析，深入理解了重定位的概念和过程，介绍了hello的虚拟地址、重定位过程、执行流程和动态链接过程，并使用edb对其执行流程、动态链接过程进行了较为细致的分析。

通过本章的学习，能够更好地理解程序的编译链接过程和目标文件的生成过程，加深对计算机系统底层原理的认识。

# 第6章 hello进程管理

**（1分）**

## 6.1 进程的概念与作用

hello生成可执行程序后，具体的执行还需要在操作系统的进程管理之下。

6.1.1 进程的概念

进程是一个执行程序中的实例，系统中的每个程序都运行在某个进程的上下文(context)中。进程是操作系统中最基本的执行单元，它负责完成特定的任务，可以是一个独立的应用程序，也可以是系统本身的一部分。

进程具有独立的程序计数器（PC）、寄存器、堆栈、文件描述符等，并与其他进程相互隔离，使得多个进程可以在同一时间内并行运行，进程还可以通过进程间通信（IPC）等机制实现不同进程之间的数据共享和协作。

6.1.2 进程的作用

进程是操作系统中的重要概念，它可以提高系统的性能和灵活性，实现复杂的计算和任务处理。其作用主要包括以下几个方面：

1. **实现任务并发执行：**多个进程可以在同一时间内并发执行不同的任务，从而提高系统的资源利用率和效率。
2. **实现资源的共享：**进程可以共享某些资源，如打印机、网络等，从而避免资源的浪费和冲突。
3. **实现数据的保护：**每个进程都有独立的地址空间，不同进程之间不能直接访问彼此的内存空间，从而保护了进程的数据安全。
4. **实现进程间的通信：**进程可以通过进程间通信机制（如管道、消息队列、共享内存等）进行数据交换和协调工作，从而实现更加复杂的任务。

## 6.2 简述壳Shell-bash的作用与处理流程

壳(Shell)是一种命令解释器，是用户与操作系统之间的接口。Bash(Bourne Again Shell)是Linux和macOS系统中默认的Shell。它能够接收用户输入的命令或脚本，并将其转换为操作系统内核能够理解和执行的指令，从而控制和管理系统的行为和资源。

Bash的处理流程大致可以分为以下几个步骤：

1. 读取用户输入的命令或脚本文件。
2. 将输入字符串切分，分析输入内容，解析命令和参数, 将命令行的参数改造为系统调用execve()内部处理所要求的形式
3. 判断命令是否为内置命令，是则立即执行，否则调用fork()来创建子进程，自身调用wait()来等待子进程完成，同时在程序执行期间始终接受键盘输入信号，并对输入信号做相应处理。
4. 当子进程运行时，调用execve()函数，同时根据命令的名字指定的文件到目录中查找可行性文件，调入内存并执行这个命令。
5. 当子进程完成处理后，向父进程shell报告，此时终端进程被唤醒，做完必要的判别工作后，再发提示符，让用户输入新命令。

## 6.3 Hello的fork进程创建过程

在命令行输入./hello命令运行hello程序，此命令不是内置命令，因此Bash通过调用fork函数创建一个新的运行的子进程。fork函数的执行过程中，操作系统会为新的子进程分配一个新的标识符(PID)，然后在内核中分配一个进程控制块(PCB)，将其挂在PCB表上。接着，操作系统会将父进程的环境复制到子进程中，包括大部分PCB的内容，并为其分配资源，包括程序、数据、栈等。

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及虚拟地址空间相同的（但是独立的）一份副本，包括代码段、数据段、堆、共享库以及用户栈。子进程还将获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用fork时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大的区别在于它们有不同的PID。

## 6.4 Hello的execve过程

在执行execve系统调用时，首先会将当前子进程的用户空间栈清空，接着会将要执行的程序的命令行参数argv和环境变量envp压入栈中，然后会将控制权转移到hello的入口点，即main函数。在这个过程中，execve还会负责将hello所需的库文件加载到内存中，并将程序需要的内存空间初始化。

如果execve执行成功，则原有进程的代码和数据会被完全替换，而新程序会成为新的进程映像，接着会开始运行。如果在执行过程中出现错误，例如找不到指定的程序，则execve会返回一个负值，表明执行失败，而原有进程将继续运行。

## 6.5 Hello的进程执行

hello进程在操作系统中的执行是由进程调度器对进程进行时间片调度，并通过上下文切换实现进程的执行。在执行过程中，操作系统合理调度，根据需要在用户态和核心态之间进行切换，并在进程结束后清除其资源。

6.5.1 进程调度的概念

进程调度是操作系统管理进程并分配处理器资源的过程。在进程执行期间，处理器按照一定的时间片轮流使用各个进程。

在进程执行过程中，内核可以随时决定抢占当前进程，并开始另一个进程。这种决策通常基于一些因素，例如进程的优先级、等待时间、资源使用情况等。进程被抢占后，内核会保存该进程的上下文信息，包括寄存器、程序计数器、用户栈和内核栈等，以便重新启动该进程时恢复其原始状态。

同时，内核会使用上下文切换机制，将原始进程的上下文信息保存到内核中，然后载入下一个进程的上下文信息，并将控制权转移到新进程的主函数中。

6.5.2 用户态与核心态

操作系统中存在两种特权级别：用户模式和内核模式。用户模式下，进程只能访问自己的地址空间，不允许直接访问内核区的代码和数据；而内核模式下，进程可以执行指令集中的任何命令，并访问系统中的任何内存位置。这样的划分保证了系统的安全性，防止用户程序直接访问内核数据结构或系统硬件资源

当操作系统决定运行hello进程时，将在进程调度器中保存当前执行进程的上下文信息，并将控制权转移到hello进程的上下文。此时，CPU进入用户态。

当hello进程需要执行需要特权级别的操作（如I/O操作），会导致CPU进入核心态，此时操作系统会保存当前进程的上下文，并执行需要的特权操作。完成后，操作系统将控制权返回给hello进程，CPU重新进入用户态，并将保存的上下文信息恢复到CPU中。

文本, 信件

描述已自动生成

图6-5 进程上下文切换

## 6.6 hello的异常与信号处理

hello程序执行过程中出现的异常可能有中断、陷阱、故障、终止等

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

hello具体运行过程中，可能产生SIGINT、SIGKILL、SIGSEGV、SIALARM、SIGCHLD等信号，具体处理如下：

1. Ctrl+Z：进程收到SIGSTP信号，hello停止，此时进程并未回收，而是后台运行，通过ps指令可以对其进行查看，还可以通过fg指令将其调回前台。

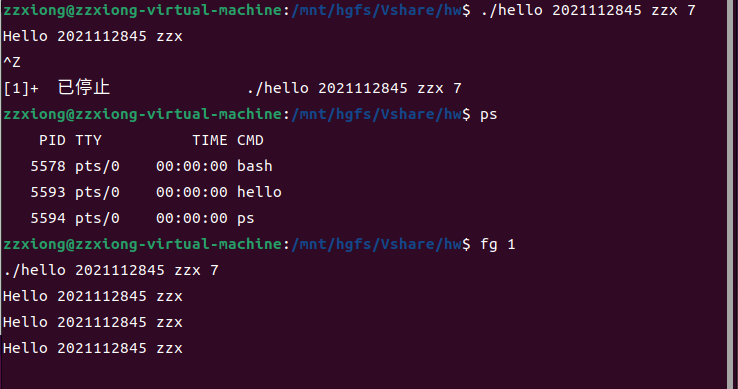


图6-6-1 SIGSTP信号处理

1. Ctrl+C：进程收到SIGINT信号，hello终止。在ps中查询不到此进程及其PID，在jobs中也没有显示。

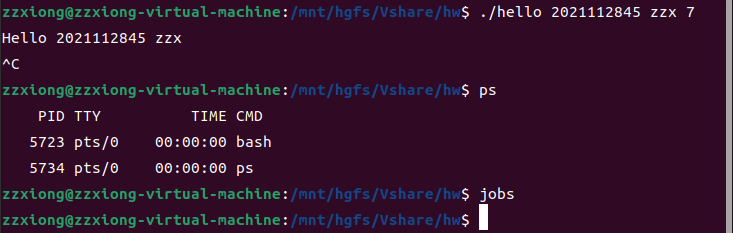


图6-6-2 SIGINT信号处理

1. 中途乱按：将屏幕的输入均缓存到缓冲区，乱码被认为是命令。hello结束后,缓冲区中其他字符会当作Shell的命令读入。

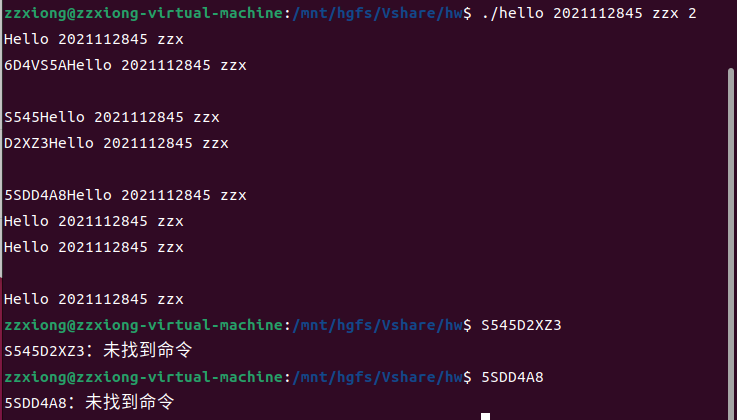


图6-6-3 键盘乱按处理

1. kill命令：挂起的进程被终止，在ps中无法查到到其PID。

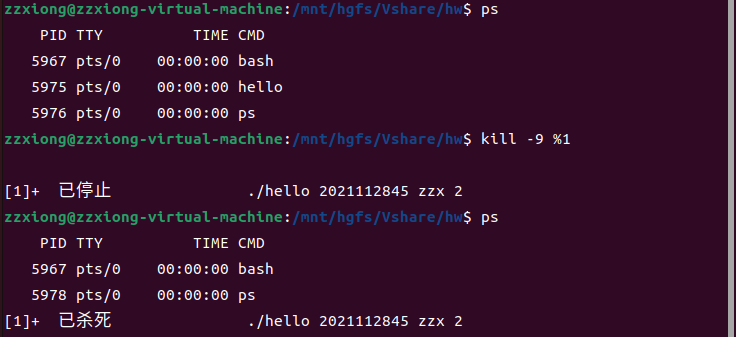


图6-6-4 kill指令

1. pstree命令：用树状图显示所有进程结构。

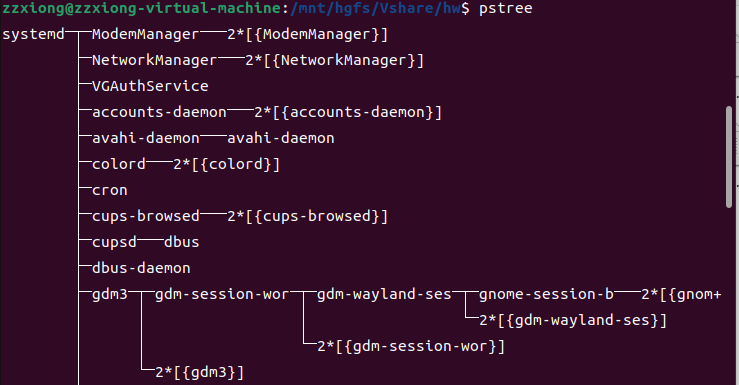


图6-6-5 pstree指令

## 6.7本章小结

本章简述了进程及Bash-Shell的基本概念与作用。并以hello为例分析了hello程序使用fork创建子进程的过程以及使用execve加载并运行用户程序的过程，最后对hello对于异常以及信号的处理结合实际操作进行了解析。

# 第7章 hello的存储管理

**（2分）**

## 7.1 hello的存储器地址空间

在计算机系统中，程序在执行时需要访问内存中的数据和指令，因此需要用到地址的概念，不同的地址代表了不同的抽象层次。

**逻辑地址**是指进程代码访问内存的地址空间，是由程序生成的地址，由段基址和偏移量组成，与实际物理内存无关。

**线性地址**是指逻辑地址经过分段、分页机制转换后得到的地址，由描述符和偏移量组成，地址空间中整数连续，它是虚拟地址空间的地址。

**虚拟地址**是指应用程序对内存的地址请求，即程序所看到的地址，也就是逻辑地址和线性地址的总和。

**物理地址**是指实际存在于内存中的地址，通过页表机制由线性地址转换而来。用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。

执行hello程序时，程序的指令和数据首先被加载到逻辑地址空间，经过分段、分页转换后得到线性地址，最终通过页表机制转换为物理地址，才能被实际执行。

## 7.2 Intel逻辑地址到线性地址的变换-段式管理

Intel处理器中，段式存储管理中以段为单位分配内存，每段分配一个连续的内存区，但各段之间不要求连续，且长度不一。其优点是易于编译、管理、修改和维护，但会产生内存的浪费。

完整的逻辑地址包含段选择符和段内偏移地址两部分。段选择符选择对应的段，在x86保护模式下，段描述符（段基线性地址、长度、权限等）无法直接存放在段寄存器中。Intel处理器将段描述符集中存放在GDT或LDT中，而段寄存器存放的是段描述符在GDT或LDT内的索引值。

将段首地址作为基地址加上偏移地址即可将逻辑地址映射到线性地址空间。

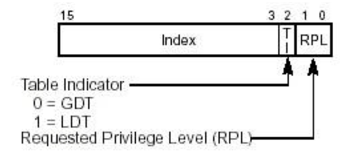


图7-2 段选择符

## 7.3 Hello的线性地址到物理地址的变换-页式管理

将程序的逻辑地址空间和物理内存划分为等长的页面，这种分配方式便于维护，且不容易产生碎块。

在页式存储管理方式中地址结构由两部构成，前一部分是虚拟页号(VPN)，后一部分为虚拟页偏移量(VPO)。如图所示，页号用于在页表中查找对应的页表项，页表项中包含了该虚拟页所映射的物理页号以及访问权限等信息，通过将物理页号和页内偏移量组合得到最终的物理地址。

图示

描述已自动生成

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(Translation Lookaside Buffer)是一种高速缓存，用于存储最近被使用的虚拟地址（VA）到物理地址（PA）的映射。当CPU访问一个虚拟地址时，首先会查找TLB，如果命中则可以直接得到对应的物理地址；如果未命中，则需要使用页表进行地址翻译。

在四级页表的支持下，当CPU访问一个虚拟地址时，先将该虚拟地址中的页目录项索引、页表项索引和页内偏移提取出来，并通过页目录表和页表来查找物理地址。查找过程中，四级页表中的每一级都有相应的页目录表和页表，可以将虚拟地址转换为物理地址。同时，如果TLB中未命中，会触发一次页表缺失异常，内核会进行相应的处理，向TLB中添加此条目。

TLB和四级页表的结合使用，能够提高地址翻译的效率，减少对内存访问的次数，从而提升系统性能。

箱线图

描述已自动生成

图7-4 Inter Core i7地址翻译

## 7.5 三级Cache支持下的物理内存访问

三级缓存是一种采用多级缓存的存储体系，可以提高计算机内存访问速度和效率。在三级缓存的架构中，缓存分为L1、L2和L3三级，每一级缓存都有不同的容量和访问速度。

当CPU需要访问内存时，首先在L1缓存中查找数据，先找组索引位，然后与标志位对比。如果L1缓存中未命中，则需要从存储层次结构中的下一层（即L2缓存）查找。如若仍未命中，则会继续在L3缓存中查找。如果在L3缓存中也未命中，则会从主存中获取数据。

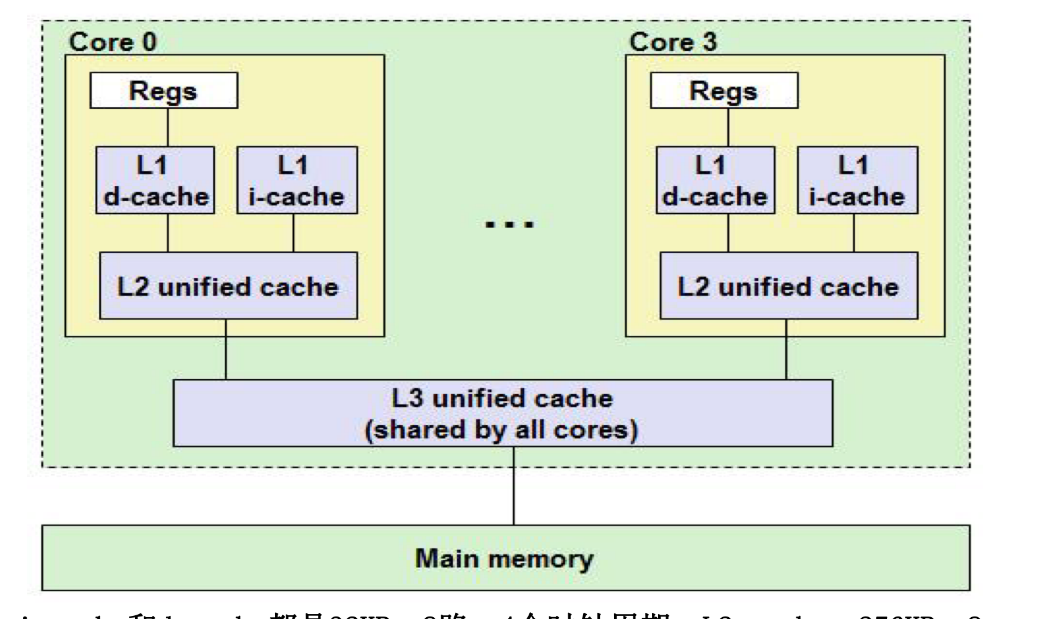


图7-4 Inter Core i7的Cache结构

如果在三级缓存中找到了需要的数据，则可以直接访问缓存中的数据，从而提高访问速度。如果在三级缓存中没有找到，则需要从主存中获取数据，并将数据存入三级缓存中，以便下次访问时可以更快地获取数据。

## 7.6 hello进程fork时的内存映射

当fork函数被调用时，内核为新进程创建虚拟内存，创建当前进程的mm\_struct、vm\_area\_struct链表和页表的原样副本。同时将两个进程中的每个页面都标记为只读，每个区域结构(vm\_area\_struct)都标记为私有的写时复制(COW)。

在新进程中返回时，新进程拥有与调用fork的父进程相同的虚拟内存。随后的写操作会通过写时复制机制创建新页面，实现内存空间的分离，也就为每个进程提供了私有的虚拟地址空间。

## 7.7 hello进程execve时的内存映射

execve函数在当前进程中加载并运行新程序hello时，需要以下几个步骤：

1. 删除已有页表和结构体vm\_area\_struct链表，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. 创建新的页表和结构体vm\_area\_struct链表，包括目标文件提供的代码和初始化的数据映射到.text和.data段，.bss和栈映射到匿名文件。所有这些新的区域都是私有的写时复制的。
3. 将需要动态链接的libc.so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)，指向代码区域的入口点，Linux根据需要换入代码和数据页面。

图示

描述已自动生成

图7-7 execve时的内存映射

## 7.8 缺页故障与缺页中断处理

**缺页故障**指访问一个虚拟地址的数据时，对应的物理地址不在内存中，从而引发的异常情况。页面命中完全是由硬件完成的，而处理缺页异常是由硬件和操作系统内核协作完成的。

当发生缺页故障时，处理器会向操作系统发出缺页中断信号，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），而后调入新的页面，并更新内存中的PTE。最后缺页处理程序返回到原来进程，再次执行导致缺页的指令，从而完成内存访问。

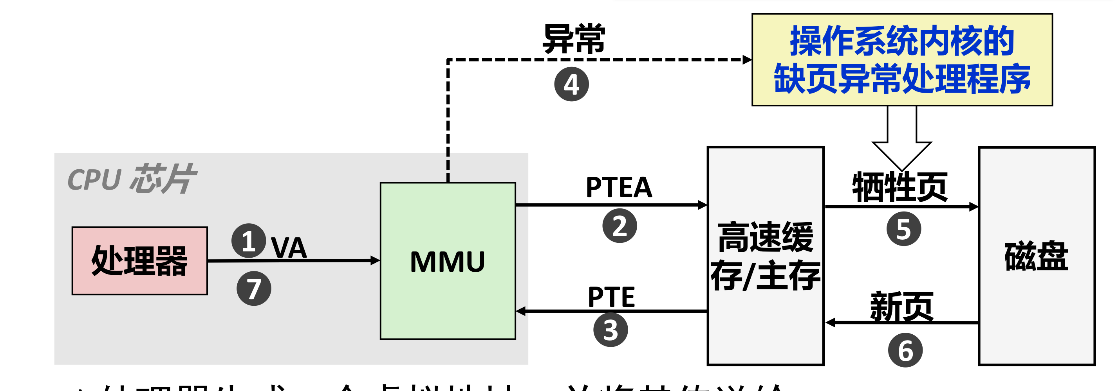


图7-8 缺页中断处理

## 7.9本章小结

本章重点介绍了存储器地址空间中各种类型地址的含义，分析了通过段式管理实现逻辑地址到线性地址的变换、页式管理实现线性地址到物理地址变换的过程。还介绍了TLB与四级页表及三级Cache下的地址翻译及访问过程，进程调用fork、execve时的内存映射、缺页故障和处理等相关内容。进一步加深了对程序的存储管理的理解。

# 结论

hello 程序终于完成了它“艰辛”的一生。hello 的一生包含如下阶段：

1. **预处理：**预处理器cpp将头文件内容插入程序文本中，完成字符串替换和删除多余空白字符，生成包含完整程序代码的预处理文件hello.i。
2. **编译：**通过词法分析和语法分析等，编译器ccl将hello.i翻译成具备在指令级别上控制硬件资源能力的汇编语言文件hello.s。
3. **汇编：**汇编器as将汇编程序翻译成机器语言指令，而后打包成可重定位目标程序hello.o。
4. **链接：**链接器ld将hello.o与动态链接库链接整合为单一文件，生成完全链接的可执行目标文件hello。
5. **进程载入：**通过Bash键入命令./hello 2021112845 zzx 3，操作系统为程序fork新进程并通过execve加载代码和数据到为其提供的私有虚拟内存空间，程序开始执行。
6. **进程控制：**由进程调度器对进程进行时间片调度，并通过上下文切换实现hello的执行，程序计数器(PC)更新，CPU按顺序取指，执行程序控制逻辑。
7. **内存访问：**内存管理单元MMU将逻辑地址逐步转换成物理地址，通过三级Cache访问物理内存/磁盘中的数据。
8. **信号处理：**进程接收信号，调用相应的信号处理函数对信号进行终止、停止、前/后台运行等处理。
9. **进程回收：**Shell等待并回收子进程，内核删除为进程创建的所有资源。

我的感想：计算机系统的设计与实现涉及多个领域，包括硬件、操作系统、编译器、网络等。即使是一个简单的 hello.c也需要操作系统综合其他部分进行许多复杂的操作，并且每一步都经过了设计者的深思熟虑，在有限的硬件资源下尽可能地提高了程序的时间和空间性能。

通过学习和实践，我深刻认识到计算机系统的复杂性和重要性，它们是现代科技和生活的基石。深入理解计算机系统的设计原理和实现细节，可以提高代码编写和性能优化的能力，更好的提升作为计算机专业学生的专业素养。

# 附件

|  |  |
| --- | --- |
| 文件名称 | 功能 |
| hello.c | hello的C语言源代码 |
| hello.i | hello.c预处理后生成的中间文件 |
| hello.s | hello.i编译后生成的汇编文件 |
| hello.o | hello.s汇编后生成的可重定位目标文件 |
| hello | hello.o和其他库文件经链接后生成的可执行文件 |
| hello\_o.elf | 用readelf读取hello.o得到的ELF格式信息 |
| hello.elf | 由hello可执行文件生成的ELF格式文件 |
| hello\_o\_objdump.asm | 反汇编hello.o得到的.asm文件 |
| hello\_objdump.asm | 反汇编hello可执行文件得到的.asm文件 |

# 参考文献

[1] RANDALE.BRYANT, DAVIDR.O’HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.

[2]进程的创建过程-fork函数https://blog.csdn.net/lyl194458/article/details/79695110

[3] argc argv的概念https://baike.baidu.com/item/argc%20argv/10826112?fr=aladdin

[4]段页式访存-逻辑地址到线性地址转换https://www.jianshu.com/p/fd2611cc808e

[5]预处理器https://zh.m.wikipedia.org/zh-sg/C%E9%A2%84%E5%A4%84%E7%90% 86%E5%99%A8