

哈爾濱工業大學

計算機系統

大作業

題 目	程序人生-Hello's P2P
专 业	
学 号	1170400307
班 级	17 03008
学 生	刘仁彪
指 导 教 师	

计算机科学与技术学院

2018 年 12 月

摘 要

hello 程序从内容上看虽然十分简单但是实际上这个程序是十分有内涵，十分有想法的。

从形成方式上看，预处理、编译、汇编、链接等，一个不缺。

从进程上看，壳(Bash)、OS(进程管理)、fork(Process)、Hardware(CPU/RAM/IO)上等都要发挥作用。

OS(存储管理)、MMU、VA、PA、TLB、4 级页表、3 级 Cache, Pagefile、0 管理与信号处理更是全要粉墨登场。

正如一句老话说的一样，“没有菜鸡的程序，只有菜鸡的程序员。”还有另一句老话说得好，“大道至简”。当然，还有一句话说的也很有道理：“一力降十惠”，最后，我觉得这句话也有必要提一下：“天下武功，唯快不破”……

总而言之，说了这么多，中心思想只有一个：研究好 hello 程序，才是每一个武林宗师走向巅峰的第一步。

接下来，将由我这个菜鸡，来为大家介绍一下这个神奇的工具！

关键词：程序的生命周期；进程；P2P；020

目 录

第 1 章 概述	- 4 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 5 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 7 -
2.3 HELLO 的预处理结果解析	- 8 -
2.4 本章小结	- 8 -
第 3 章 编译	- 9 -
3.1 编译的概念与作用	- 9 -
3.2 在 UBUNTU 下编译的命令	- 10 -
3.3 HELLO 的编译结果解析	- 11 -
3.3.1 指令解释	- 11 -
3.3.2 数据	- 11 -
3.3.3 数据传送	- 13 -
3.3.4 数据运算	- 13 -
3.3.5 控制	- 13 -
3.3.6 函数	- 14 -
3.4 本章小结	- 15 -
第 4 章 汇编	- 16 -
4.1 汇编的概念与作用	- 16 -
4.2 在 UBUNTU 下汇编的命令	- 16 -
4.3 可重定位目标 ELF 格式	- 17 -
4.4 HELLO.o 的结果解析	- 21 -
4.5 本章小结	- 22 -
5.1 链接的概念与作用	- 23 -
5.2 在 UBUNTU 下链接的命令	- 23 -
5.3 可执行目标文件 HELLO 的格式	- 24 -
5.4 HELLO 的虚拟地址空间	- 27 -
5.5 链接的重定位过程分析	- 28 -
5.6 HELLO 的执行流程	- 30 -

5.7 HELLO 的动态链接分析	- 31 -
5.8 本章小结	- 31 -
第 6 章 HELLO 进程管理.....	- 32 -
6.1 进程的概念与作用.....	- 32 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	- 32 -
6.3 HELLO 的 FORK 进程创建过程.....	- 33 -
6.4 HELLO 的 EXECVE 过程	- 33 -
6.5 HELLO 的进程执行.....	- 34 -
6.6 HELLO 的异常与信号处理.....	- 36 -
6.7 本章小结.....	- 38 -
第 7 章 HELLO 的存储管理.....	- 39 -
7.1 HELLO 的存储器地址空间.....	- 39 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 39 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 40 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	- 42 -
7.5 三级 CACHE 支持下的物理内存访问	- 43 -
7.6 HELLO 进程 FORK 时的内存映射	- 44 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 44 -
7.8 缺页故障与缺页中断处理	- 45 -
7.9 动态存储分配管理	- 46 -
7.10 本章小结.....	- 47 -
第 8 章 HELLO 的 IO 管理	- 48 -
8.1 LINUX 的 IO 设备管理方法.....	- 48 -
8.2 简述 UNIX IO 接口及其函数.....	- 48 -
8.3 PRINTF 的实现分析.....	- 50 -
8.4 GETCHAR 的实现分析.....	- 52 -
8.5 本章小结.....	- 52 -
结论.....	- 53 -
附件.....	- 54 -
参考文献	- 55 -

第 1 章 概述

1.1 Hello 简介

Hello 虽为武林秘籍，但其创作过程并无不同。

当年，武林宗师墨云沧深入大山，历尽千辛万苦，方才选得上铁一块（hello.c 程序经过在 IDE 输入代码构建得到）。

之后为他淬火（cpp 的预处理）、冷却（ccl 的编译）、千锤（as 的汇编）、百炼（ld 的链接），方才得到这把绝世武器（可执行目标程序 hello）。

之后，大师选择下山，一统武林。每当他拔出这把剑（在 shell 中键入启动命令后），便要饮血一次（shell 为其 fork，产生子进程），剑刃也就更加锋利（从 Program 摇身一变成为 Process。）

除此以外，宗师墨云沧每次夜晚都要以手摩挲此剑（shell 为其 execve，映射虚拟内存），将内功传入其中（进入程序入口后程序开始载入物理内存），在剑中孕育剑灵（进入 main 函数执行目标代码），修养灵气（CPU 为运行的 hello 分配时间片执行逻辑控制流）。

最后，宗师临终前，将一生所修武林秘籍刻于剑上（I/O 设备管理），并将此剑藏于终南山上一隐蔽洞穴中，等待有缘人的到来……

1.2 环境与工具

（1）环境：

硬件环境：Intel Core i7-7700HQ x64CPU

软件环境：Ubuntu18.04.1 LTS

（2）工具：

开发与调试工具：vim, gcc, as, ld, edb, readelf, vs

1.3 中间结果

文件	功能
hello.i	预处理后的ASCII码文件
hello.o	汇编文件
hello.elf	可重定位目标文件
helloasm.txt	hello.o 反汇编代码
hello	最终的可执行目标文件
helloasmm.txt	hello 反汇编代码
helloelf_1k.txt	hello的部分elf内容
elfall.txt	hello的全部elf内容

1.4 本章小结

欲练神功，必先××……

欲练神兵，必先选材。只有最高大上的材料（代码），才配得上世间最顶级的剑客（编译器）。

选好了材料，第一步当然是打磨啊，先把材料处理一下啦……

第 2 章 预处理

2.1 预处理的概念与作用

(1) 概念

预处理(或称预编译)是指在进行编译的第一遍扫描(词法扫描和语法分析)之前所作的工作。预处理指令指示在程序正式编译前就由编译器进行的操作,可放在程序中任何位置。

预处理是 C 语言的一个重要功能,它由预处理程序负责完成。当对一个源文件进行编译时,系统将自动引用预处理程序对源程序中的预处理部分作处理,处理完毕自动进入对源程序的编译。

C 语言提供多种预处理功能,主要处理#开始的预编译指令,如宏定义(#define)、文件包含(#include)、条件编译(#ifdef)等。合理使用预处理功能编写的程序便于阅读、修改、移植和调试,也有利于模块化程序设计。

(2) 功能

在集成开发环境中,编译,链接是同时完成的。其实,C 语言编译器在对源代码编译之前,还需要进一步的处理:预处理。预处理的主要作用如下:

1. 将源文件中以“include”格式包含的文件复制到编译的源文件中。
2. 用实际值替换用“#define”定义的字符串。
3. 根据“#if”后面的条件决定需要编译的代码。

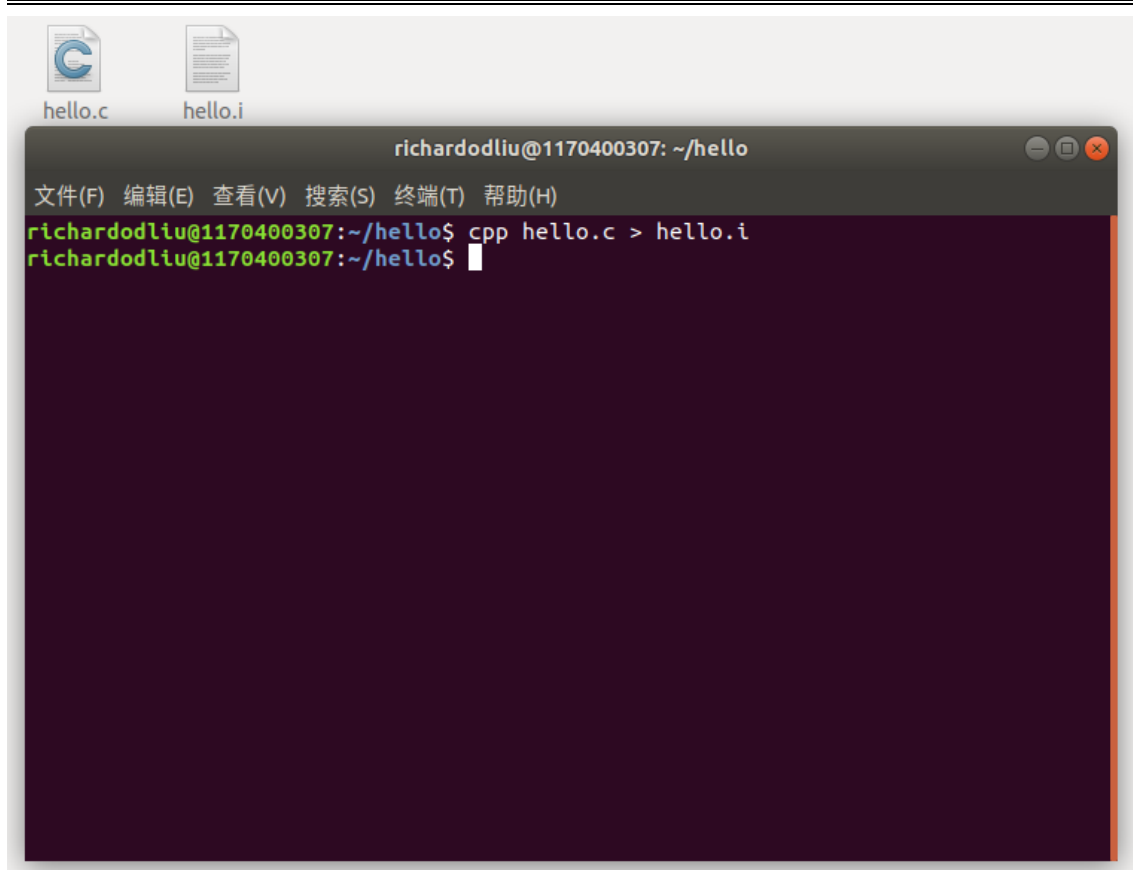
其中,作用 3 在本次作业中所给的源代码中没有体现,因此后续操作中不予考虑,以减少行文内容。

下图是常用的一些预处理命令。

预处理名称	意 义
#define	宏定义
#undef	撤销已定义过的宏名
#include	使编译程序将另一源文件嵌入到带有#include 的源文件中
#if	#if 的一般含义是如果#if 后面的常量表达式为 true, 则编译它与#endif 之间的代码, 否则跳过这些代码。命令#endif 标识一个#if 块的结束。#else 命令的功能有点象 C 语言中的 else , #else 建立另一选择 (在# if 失败的情况下)。#elif 命令意义与 else if 相同, 它形成一个 if else-if 阶梯状语句, 可进行多种编译选择。
#else	
#elif	
#endif	
#ifdef	用#ifdef 与#ifndef 命令分别表示“如果有定义”及“如果无定义”, 是条件编译的另一种方法。
#ifndef	
#line	改变当前行数和文件名称, 它们是在编译程序中预先定义的标识符 命令的基本形式如下: #line number["filename"]
#error	编译程序时, 只要遇到 #error 就会生成一个编译错误提示消息, 并停止编译
#pragma	为实现时定义的命令, 它允许向编译程序传送各种指令例如, 编译程序可能有一种选择, 它支持对程序执行的跟踪。可用#pragma 语句指定一个跟踪选择。 https://blog.csdn.net/weixin_41143631

2.2 在 Ubuntu 下预处理的命令

命令: `cpp hello.c > hello.i`



2.3 Hello 的预处理结果解析

打开 `hello.i` 之后发现，预处理后内容达到为 3188 行。

首先是 `stdio.h` 的展开。

其次是 `unistd.h` 的展开。

最后是 `stdlib.h` 的展开。

同时，修改“`#define`”以及后续内容体现在，凡是出现的宏定义内容`#define`语句均已得到执行替换操作，所以最终得到的 `hello.i` 文件的内容中是没有宏定义内容的。

`main` 函数出现在 `hello.c` 中的代码自 3099 行开始。

2.4 本章小结

真所谓：“凡铸神剑，必先精炼。”本是一块凡铁，怎么就被大师选上，成为一代神兵了呢？今时今日，我们终于了解到，第一步，必是淬火。

以凡铁之身（代码）入火炉（`cpp`）得以经受磨砺（预处理），此番经历之后，精华尽出，方才是淬火（预处理）的真谛啊！

第 3 章 编译

3.1 编译的概念与作用

(1) 概念

经过预处理得到的输出文件中，只有常量；如数字、字符串、变量的定义，以及 C 语言的关键字，如 `main`, `if`, `else`, `for`, `while`, `{`, `}`, `+`, `-`, `*`, `\` 等等。

编译程序所要作的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。

优化处理是编译系统中一项比较艰深的技术。它涉及到的问题不仅同编译技术本身有关，而且同机器的硬件环境也有很大的关系。优化一部分是对中间代码的优化。这种优化不依赖于具体的计算机。另一种优化则主要针对目标代码的生成而进行的。对于前一种优化，主要的工作是删除公共表达式、循环优化（代码外提、强度削弱、变换循环控制条件、已知量的合并等）、复写传播，以及无用赋值的删除，等等。后一种类型的优化同机器的硬件结构密切相关，最主要的是考虑是如何充分利用机器的各个硬件寄存器存放有关变量的值，以减少对于内存的访问次数。另外，如何根据机器硬件执行指令的特点（如流水线、RISC、CISC、VLIW 等）而对指令进行一些调整使目标代码比较短，执行的效率比较高，也是一个重要的研究课题。

经过优化得到的汇编代码必须经过汇编程序的汇编转换成相应的机器指令，方可能被机器执行。

(2) 作用

编译的过程实质上是把高级语言翻译成机器语言的过程，即对 `hello.i` 文件执行了以下几步操作：

1. 词法分析，
2. 语法分析
3. 语义分析

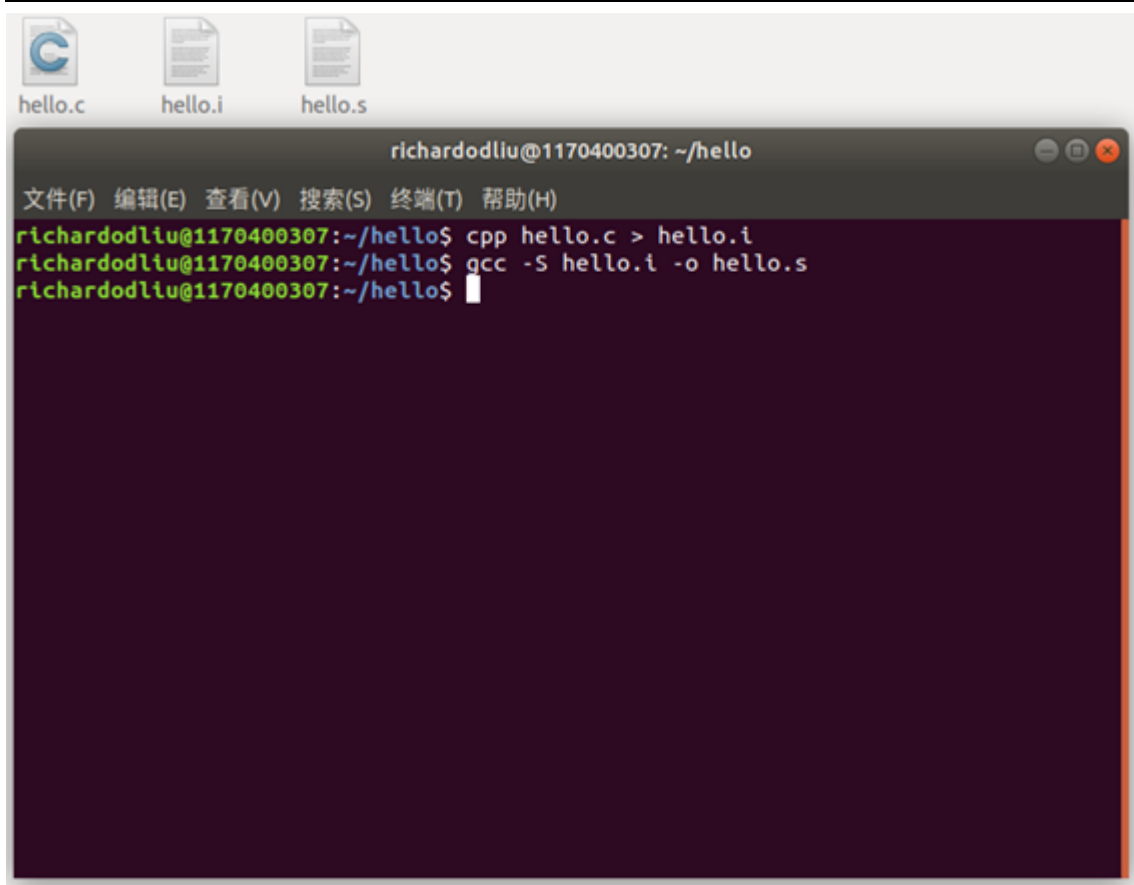
下图为常见的优化操作：

优先级问题	表达式	经常误认为的结果	实际结果
. 的优先级高于* ->操作符用于消除这个问题	*p. f	p 所指对象的字段 f (*p). f	对 p 取 f 偏移, 作为指针, 然后进行解除引用操作。*(p. f)
[] 高于*	int *ap[]	ap 是个指向 int 数组的指针 int (*ap)[]	ap 是个元素为 int 指针的数组 int *(ap[])
函数() 高于*	int *fp()	fp 是个函数指针, 所指函数返回 int。 int (*fp)()	fp 是个函数, 返回 int * int *(fp())
== 和!=高于位操作	(val & mask != 0)	(val & mask) != 0	val & (mask != 0)
== 和!=高于赋值符	c = getchar() != EOF	(c = getchar()) != EOF	c = (getchar() != EOF)
算术运算符高于位移运算符	msb << 4 + lsb	(msb << 4) + lsb	msb << (4 + lsb)
逗号运算符在所有运算符中优先级最低	i = 1, 2	i = (1, 2)	(i = 1), 2

https://blog.csdn.net/weixin_41143631

3.2 在 Ubuntu 下编译的命令

命令: gcc -Shello.i -o hello.s



3.3 Hello 的编译结果解析

3.3.1 指令解释

声明	含义
.file	可重定位目标文件
.text	已编译程序的汇编代码
.section.rodata	只读数据
.globl	全局变量
.type	函数或对象类型
.size	规模大小
.align	对指令或者数据的存放地址进行对齐的方式

3.3.2 数据

(1) 字符串

程序中共有两个字符，在 .section.rodata（只读数据节）声明：

```
.LC0:  
    .string "Usage: Hello \345\255\246\345\217\267 \345\247\223\345\220\215\357\274\201"
```

```
.LC1:
    .string "Hello %s %s\n"
```

其中汉字编码成 UTF-8 格式，一个汉字在 UTF-8 编码中占三个字节，一个\代表一个字节。

(2) 数值型数据

```
1. int sleepsecs
    .globl sleepsecs
    .data
    .align 4
    .type sleepsecs, @object
    .size sleepsecs, 4
sleepsecs:
    .long 2
```

解释如下：已赋值全局变量；编译器处理时在.data节声明该变量；设置对齐方式为4；设置类型为对象；设置大小为4字节；设置为long类型；其值为2。

2. int i

编译器将局部变量存储在寄存器或者栈空间中，在hello.s中编译器将i存储在栈上空间-4(%rbp)中，可以看出i占据了栈中的4B。

3. int argc

作为第一个参数传入。

4. 立即数

其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

(3) 数组

```
.L4:
    movq    -32(%rbp), %rax
    addq    $16, %rax
    movq    (%rax), %rdx
    movq    -32(%rbp), %rax
    addq    $8, %rax
    movq    (%rax), %rax
    movq    %rax, %rsi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    sleepsecs(%rip), %eax
    movl    %eax, %edi
    call    sleep@PLT
    addl    $1, -4(%rbp)
```

解释如下：char *argv[]为指针数组，其每个指针指向一个字符，作为参数传入main函数中；起始地址为argv；main函数中访问数组元素argv[1], argv[2]时，栈顶指针减小32，说明指针大小为8B。

3.3.3 数据传送

数据传送使用 `mov` 指令完成时，类型如下：

类型	b	w	l	q
规模	1B	2B	4B	8B

举例：

```
movl    $1, %edi
```

解释如下，将立即数 1 传入寄存器 `%edi` 中。

注意：

由于 `sleepsecs` 是 `int` 型的而 2.5 是 `float` 类型的，这就有一个隐式的类型转换，编译器将 2.5 隐式地转换成了 2 存入 `sleepsecs`。

3.3.4 数据运算

指令	效果
<code>leaq S,D</code>	$D = \&S$
<code>INC D</code>	$D += 1$
<code>DEC D</code>	$D -= 1$
<code>NEG D</code>	$D = -D$
<code>ADD S,D</code>	$D = D + S$
<code>SUB S,D</code>	$D = D - S$
<code>IMULQ S</code>	$R[\%rdx]:R[\%rax] = S * R[\%rax]$ (有符号)
<code>MULQ S</code>	$R[\%rdx]:R[\%rax] = S * R[\%rax]$ (无符号)
<code>IDIVQ S</code>	$R[\%rdx] = R[\%rdx]:R[\%rax] \bmod S$ (有符号) $R[\%rax] = R[\%rdx]:R[\%rax] \div S$
<code>DIVQ S</code>	$R[\%rdx] = R[\%rdx]:R[\%rax] \bmod S$ (无符号) $R[\%rax] = R[\%rdx]:R[\%rax] \div S$

3.3.5 控制

(1) 条件判断

jX指令	条件	描述
jmp	1	无条件
je	ZF	相等 / 结果为0
jne	~ZF	不相等 / 结果不为0
js	SF	结果为负数
jns	~SF	结果为非负数
jg	~(SF^OF)&~ZF	大于 (符号数)
jge	~(SF^OF)	大于等于 (符号数)
jl	(SF^OF)	小于 (符号数)
jle	(SF^OF) ZF	小于等于 (符号数)
ja	~CF&~ZF	大于 (无符号数)
jb	CF	小于 (无符号数)

(2) 跳转内容

1.if 语句

```
if(argc!=3)
{
    printf("Usage: Hello 学号 姓名! \n");
    exit(1);
}
```

2.for 循环

```
for(i=0;i<10;i++)
{
    printf("Hello %s %s\n",argv[1],argv[2]);
    sleep(sleepsecs);
}
```

3.3.6 函数

(1) 主函数

参数传递: 外部调用分别通过寄存器 %RDI 和 %rsi 将 argv 和 argc 传递给 main 函数

函数调用: 当新进程开始执行可执行对象文件时, 它首先调用系统启动函数。调用指令将返回地址压入堆栈, 然后调用 main 函数。

函数返回: 通常 main 函数返回 0

(2) printf

参数传递：main 函数将要打印的第一个字符串的地址传递给 %rdi，使用 put，因为只有一个字符串

函数调用：因为它是 DLL 的函数，所以它需要与位置无关的代码，因此需要 GOT 和 PLT 交互来确定运行时函数的地址。所以这里是 call put @ PLT / printf @ PLT。

函数返回：无返回值

(3) 退出

参数传递：无

函数调用：因为它是 DLL 中的一个函数，所以需要使用与位置无关的代码，因此在运行时需要 GOT 和 PLT 交互来确定函数的地址，所以这里是调用 exit @ PLT。

函数返回：无返回，直接退出

(4) 睡觉

参数传递：在调用睡眠功能之前，将睡眠秒数传递到寄存器 %EDI 中

函数调用：调用指令在调用之前将返回地址压入堆栈，因为它是动态链接库中的一个函数，需要使用与位置无关的代码，因此需要 GOT 和 PLT 交互来确定函数的地址在运行时，所以这里是睡眠 @ PLT。

函数返回：如果被信号中断，它将返回剩余的睡眠秒数，否则返回 0。

(5) getchar

参数传递：无

函数调用：调用指令在调用之前将返回地址压入堆栈，因为它是动态链接库中的一个函数，需要与位置无关的代码，因此需要 GOT 和 PLT 交互来确定运行时函数的地址时间，所以这里是调用 getchar @ PLT。

3.4 本章小结

第二步，乃是冷却（编译），冷却（编译）的目的在于将淬火（预处理）之后的凡铁（代码）更进一步升华，使其显露真容（汇编指令）。在这当中，冷水（cc1）的作用不可忽视，必然是冷却（的）主力军。

到了这里，我们的神器（目标文件）已经处理过半了，但是这个时候仍然只是一个凡人无法使用的普通器件（汇编指令），要想让真正的大侠（计算机）挥舞着它降妖除魔，我们还有很多工作需要处理。

第 4 章 汇编

4.1 汇编的概念与作用

(1) 概念

汇编过程将上一步的汇编代码转换成机器码(machine code)，这一步产生的文件叫做目标文件，是二进制格式。汇编代码转换机器码。

(2) 作用

得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。

补充内容：目标文件

目标文件由段组成。通常一个目标文件中至少有两个段：

1. 代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

2. 数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

UNIX 环境下主要有三种类型的目标文件：

1. 可重定位文件

其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。

2. 共享的目标文件

这种文件存放了适合于在两种上下文里链接的代码和数据。第一种是链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个目标文件；第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。

3. 可执行文件

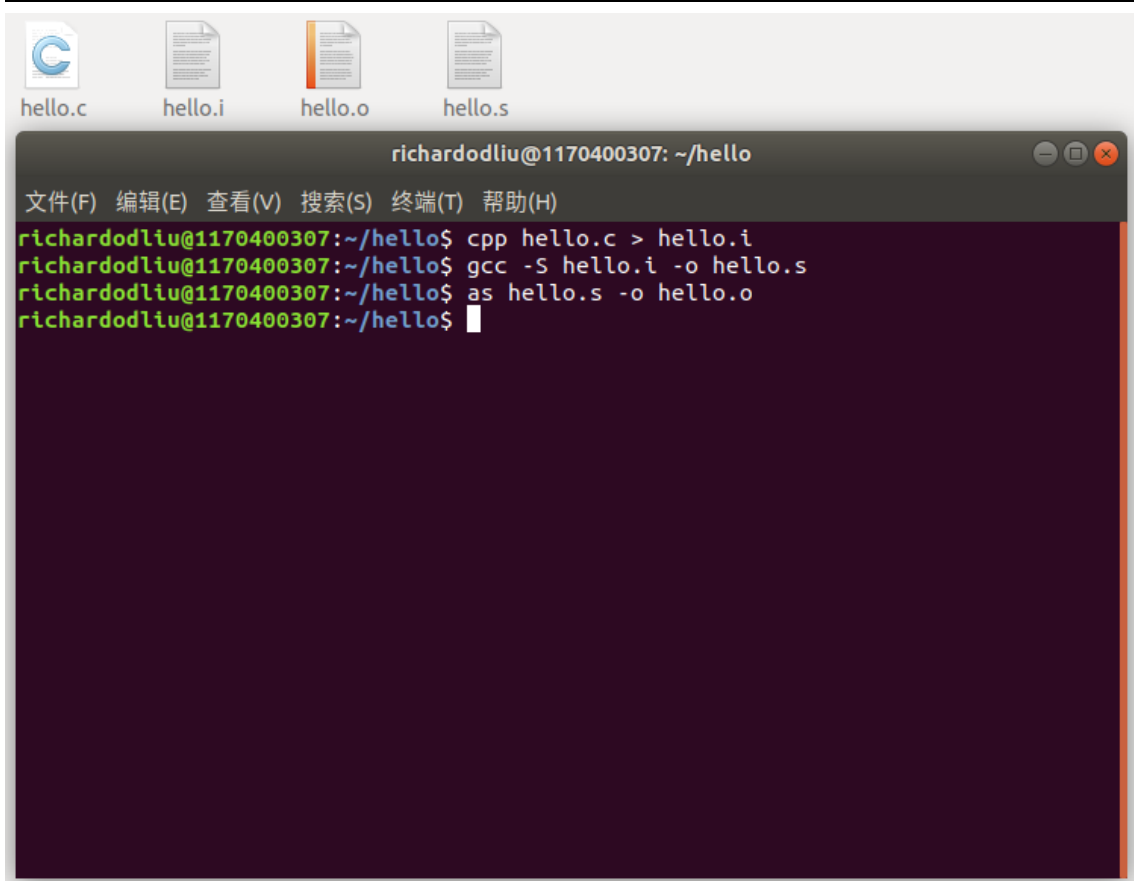
它包含了一个可以被操作系统创建一个进程来执行之的文件。

汇编程序生成的实际上是第一种类型的目标文件。

(以上内容整理自互联网)

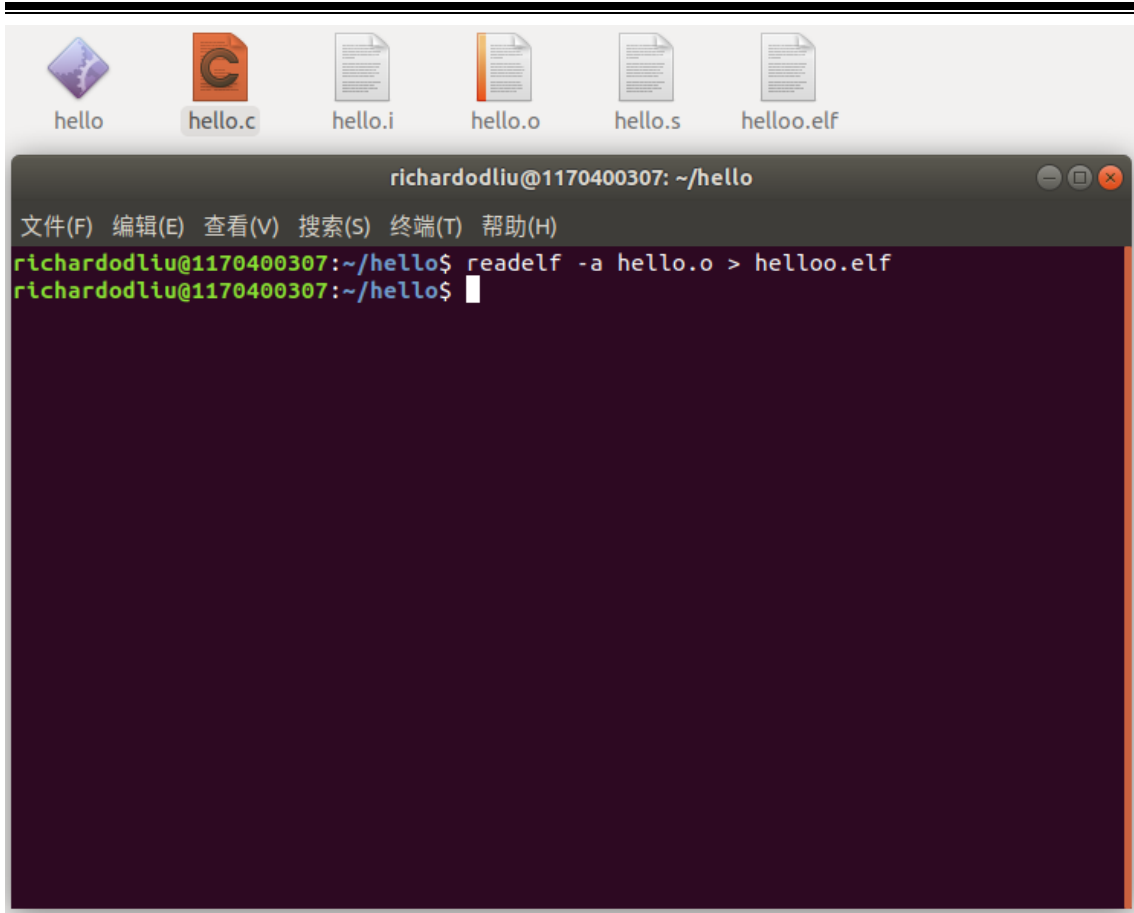
4.2 在 Ubuntu 下汇编的命令

命令：`as hello.s -o hello.o`



4.3 可重定位目标 elf 格式

指令: `readelf -a hello.o > helloo.elf`



hello.o 文件的 ELF 格式组成如下：

(1) ELF 头

Magic 序列：大小为 16B，描述了生成该文件的系统的字的大小和字节顺序。

其余内容：包含帮助链接器解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，机器类型，节头表的文件偏移量以及节标头信息，例如表中的条目大小和数量。

ELF 头:

```
Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
类别:                               ELF64
数据:                               2 补码, 小端序 (little endian)
版本:                               1 (current)
OS/ABI:                               UNIX - System V
ABI 版本:                               0
类型:                               REL (可重定位文件)
系统架构:                               Advanced Micro Devices X86-64
版本:                               0x1
入口点地址:                               0x0
程序头起点:                               0 (bytes into file)
Start of section headers:               1152 (bytes into file)
标志:                               0x0
本头的大小:                               64 (字节)
程序头大小:                               0 (字节)
Number of program headers:               0
节头大小:                               64 (字节)
节头数量:                               13
字符串表索引节头: 12
```

(2) Section Headers

节头部表, 包含了文件中出现的各个节的语义, 包括节的类型、位置和大小等信息。

节头:

[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标 链接 信息	对齐
[0]	000000000000000000	NULL	0000000000000000	00000000
[1]	.text	PROGBITS	0000000000000000	00000040
[2]	.rela.text	RELA	0000000000000000	00000340
[3]	.data	PROGBITS	0000000000000000	000000c4
[4]	.bss	NOBITS	0000000000000000	000000c8
[5]	.rodata	PROGBITS	0000000000000000	000000c8
[6]	.comment	PROGBITS	0000000000000000	000000f3
[7]	.note.GNU-stack	PROGBITS	0000000000000000	0000011e
[8]	.eh_frame	PROGBITS	0000000000000000	00000120
[9]	.rela.eh_frame	RELA	0000000000000000	00000400
[10]	.symtab	SYMTAB	0000000000000000	00000158
[11]	.strtab	STRTAB	0000000000000000	000002f0
[12]	.shstrtab	STRTAB	0000000000000000	00000418

(3) rela.text

重定位节是在.text节中表示位置的一个列表, 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。

重定位节 '.rela.text' at offset 0x340 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4
00000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4
000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4
000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 1a
00000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4
000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4
000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4

(4) rela.eh_frame

eh_frame 节的重定位信息。

重定位节 '.rela.eh_frame' at offset 0x400 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
0000000000020	0002000000002	R_X86_64_PC32	0000000000000000	.text + 0

(5) symtab

符号表，用来存放程序中定义和引用的函数和全局变量的信息。重定位需要引用的符号都在其中声明。

4.4 Hello.o 的结果解析

指令：objdump -d -r hello.o > hello.objdump

得到文件如下图所示。

hello.o: 文件格式 elf64-x86-64

Disassembly of section .text:

```
0000000000000000 <main>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 ec 20       sub     $0x20,%rsp
8: 89 7d ec          mov     %edi,-0x14(%rbp)
b: 48 89 75 e0       mov     %rsi,-0x20(%rbp)
f: 83 7d ec 03       cmpl    $0x3,-0x14(%rbp)
13: 74 16            je      2b <main+0x2b>
15: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 1c <main+0x1c>
18: R_X86_64_PC32    .rodata-0x4
1c: e8 00 00 00 00    callq  21 <main+0x21>
1d: R_X86_64_PLT32    puts-0x4
21: bf 01 00 00 00    mov     $0x1,%edi
26: e8 00 00 00 00    callq  2b <main+0x2b>
27: R_X86_64_PLT32    exit-0x4
2b: c7 45 fc 00 00 00 movl    $0x0,-0x4(%rbp)
32: eb 3b            jmp     6f <main+0x6f>
34: 48 8b 45 e0       mov     -0x20(%rbp),%rax
38: 48 83 c0 10       add     $0x10,%rax
3c: 48 8b 10          mov     (%rax),%rdx
3f: 48 8b 45 e0       mov     -0x20(%rbp),%rax
43: 48 83 c0 08       add     $0x8,%rax
47: 48 8b 00          mov     (%rax),%rax
4a: 48 89 c6          mov     %rax,%rsi
4d: 48 8d 3d 00 00 00 lea     0x0(%rip),%rdi    # 54 <main+0x54>
50: R_X86_64_PC32    .rodata+0x1a
54: b8 00 00 00 00    mov     $0x0,%eax
59: e8 00 00 00 00    callq  5e <main+0x5e>
5a: R_X86_64_PLT32    printf-0x4
5e: 8b 05 00 00 00 00 mov     0x0(%rip),%eax    # 64 <main+0x64>
60: R_X86_64_PC32    sleepsecs-0x4
```

```

64: 89 c7          mov    %eax,%edi
66: e8 00 00 00 00 callq  6b <main+0x6b>
                        67: R_X86_64_PLT32      sleep-0x4
6b: 83 45 fc 01    addl   $0x1,-0x4(%rbp)
6f: 83 7d fc 09    cmpl   $0x9,-0x4(%rbp)
73: 7e bf          jle     34 <main+0x34>
75: e8 00 00 00 00 callq  7a <main+0x7a>
                        76: R_X86_64_PLT32      getchar-0x4
7a: b8 00 00 00 00 mov     $0x0,%eax
7f: c9             leaveq  %eax
80: c3             retq

```

和编译后的文件进行比较，可以观察到如下差别：

（1）跳转指令

汇编之后跳转指令指向了明确的内存地址。

（2）函数调用

函数调用：仅编译后函数调用时指明函数名称，汇编之后调用函数的目标地址是下一条指令。因为此时尚未与共享库进行链接，无法确定函数执行的地址。同时，在`.rela.text`节中为其添加重定位条目，等待静态链接的进一步确定。

（3）全局变量

编译后文件中访问只读数据段中的字符串时使用段名称+`%rip`，汇编后改为`0+%rip`，原理同上。

4.5 本章小结

经历过冷却之后，我们来到了下一步：千锤（汇编）。千锤（汇编）的目的很简单，就是为了去掉无用杂质（无关代码），使得整个千锤后的剑形（汇编文件）相比之前更加的精简、可靠，也更便于之后的百炼（链接）。

不过，值得我们思考的是，我们一定要牢记：冷却（编译）之后一定要先千锤（汇编）才能百炼（链接），只有千锤得到的精致模具（可重定位目标文件）才有让我们进一步百炼（链接）得到绝世武器（可执行目标文件）的需要。

第5章 链接

5.1 链接的概念与作用

(1) 概念

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。

例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

(2) 功能

根据开发人员指定的同库函数的链接方式的不同，链接处理可分为两种：

1. 静态链接

在这种链接方式下，函数的代码将从其所在的静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。

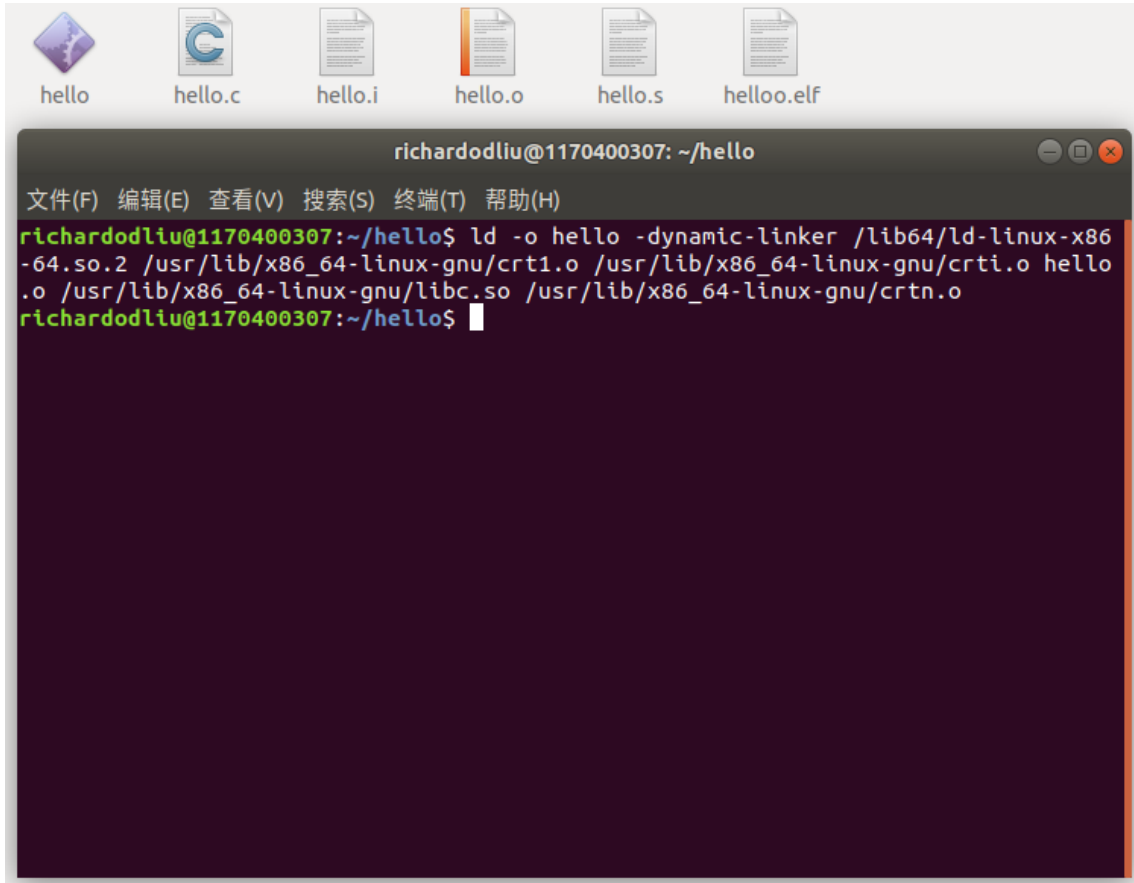
2. 动态链接

在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害。

5.2 在 Ubuntu 下链接的命令

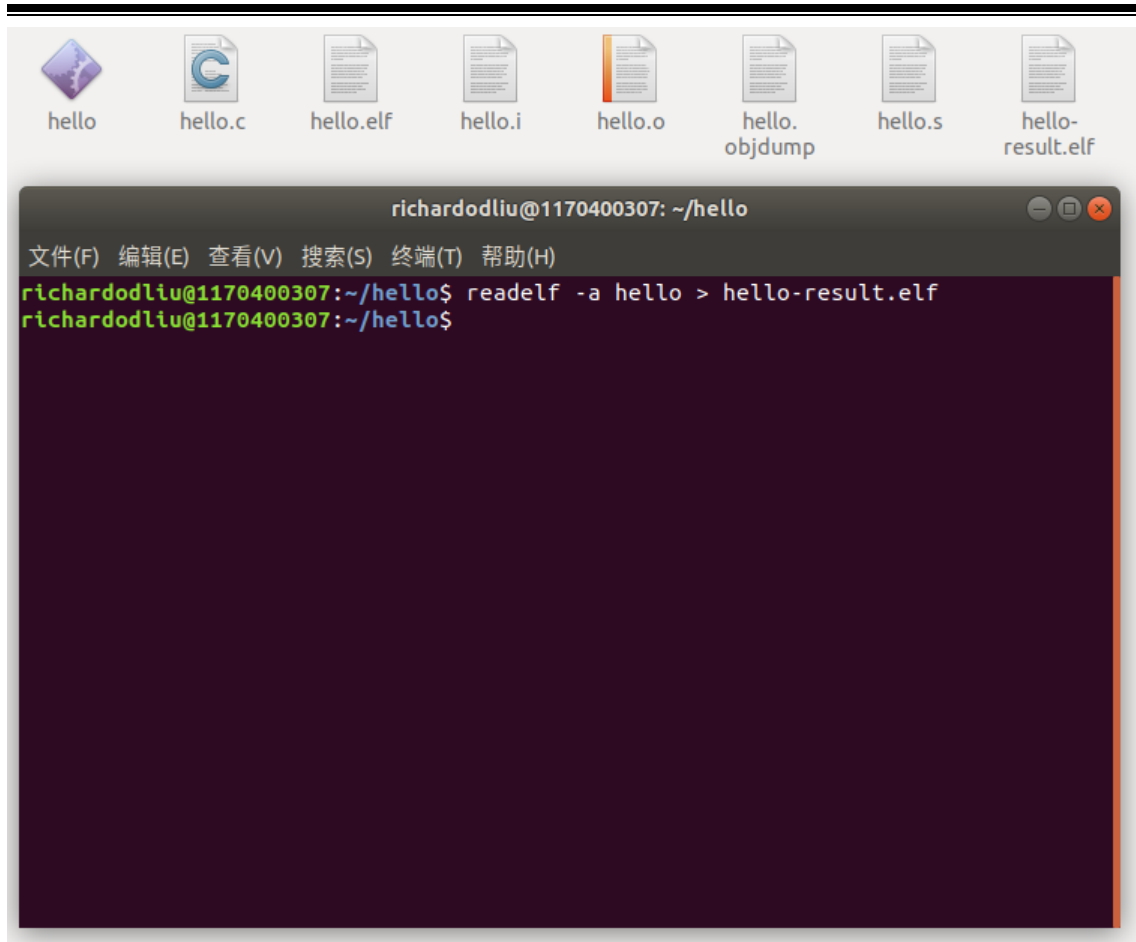
```
指 令   :   ld   -o   hello   -dynamic-linker   /lib64/ld-linux-x86-64.so.2  
/usr/lib/x86_64-linux-gnu/crt1.o   /usr/lib/x86_64-linux-gnu/crti.o   hello.o
```


/usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crti.o



5.3 可执行目标文件 hello 的格式

指令: readelf -a hello > hello-result.elf



在 ELF 格式文件中，Section Headers 声明 `hello` 中的所有 section 信息，包括程序中的 `size` 和 `offset`。因此根据 Section Headers 中的信息，我们可以使用 HexEdit 来定位每个的间隔（起始位置和大小）。地址是程序加载到虚拟地址的起始地址。

节头:

[号]	名称	类型	地址	偏移量
	大小	全体大小	旗标 链接 信息	对齐
[0]	000000000000000000	NULL	0000000000000000	00000000
	000000000000000000	0000000000000000	0 0 0	0
[1]	.interp	PROGBITS	0000000000400200	00000200
	00000000000000001c	0000000000000000	A 0 0	1
[2]	.note.ABI-tag	NOTE	000000000040021c	0000021c
	000000000000000020	0000000000000000	A 0 0	4
[3]	.hash	HASH	0000000000400240	00000240
	000000000000000034	00000000000000004	A 5 0	8
[4]	.gnu.hash	GNU_HASH	0000000000400278	00000278
	00000000000000001c	0000000000000000	A 5 0	8
[5]	.dynsym	DYNSYM	0000000000400298	00000298
	0000000000000000c0	00000000000000018	A 6 1	8
[6]	.dynstr	STRTAB	0000000000400358	00000358
	000000000000000057	0000000000000000	A 0 0	1
[7]	.gnu.version	VERSYM	00000000004003b0	000003b0
	000000000000000010	00000000000000002	A 5 0	2
[8]	.gnu.version_r	VERNEED	00000000004003c0	000003c0
	000000000000000020	0000000000000000	A 6 1	8
[9]	.rela.dyn	RELA	00000000004003e0	000003e0
	000000000000000030	00000000000000018	A 5 0	8
[10]	.rela.plt	RELA	0000000000400410	00000410
	000000000000000078	00000000000000018	AI 5 19	8
[11]	.init	PROGBITS	0000000000400488	00000488
	000000000000000017	0000000000000000	AX 0 0	4
[12]	.plt	PROGBITS	00000000004004a0	000004a0
	000000000000000060	00000000000000010	AX 0 0	16
[13]	.text	PROGBITS	0000000000400500	00000500
	000000000000000132	0000000000000000	AX 0 0	16

[14]	.fini	PROGBITS	0000000000400634	00000634
	0000000000000009	0000000000000000	AX 0 0	4
[15]	.rodata	PROGBITS	0000000000400640	00000640
	000000000000002f	0000000000000000	A 0 0	4
[16]	.eh_frame	PROGBITS	0000000000400670	00000670
	00000000000000fc	0000000000000000	A 0 0	8
[17]	.dynamic	DYNAMIC	0000000000600e50	00000e50
	00000000000001a0	0000000000000010	WA 6 0	8
[18]	.got	PROGBITS	0000000000600ff0	00000ff0
	0000000000000010	0000000000000008	WA 0 0	8
[19]	.got.plt	PROGBITS	0000000000601000	00001000
	0000000000000040	0000000000000008	WA 0 0	8
[20]	.data	PROGBITS	0000000000601040	00001040
	0000000000000008	0000000000000000	WA 0 0	4
[21]	.comment	PROGBITS	0000000000000000	00001048
	000000000000002a	0000000000000001	MS 0 0	1
[22]	.symtab	SYMTAB	0000000000000000	00001078
	00000000000000498	0000000000000018	23 28	8
[23]	.strtab	STRTAB	0000000000000000	00001510
	00000000000000150	0000000000000000	0 0	1
[24]	.shstrtab	STRTAB	0000000000000000	00001660
	00000000000000c5	0000000000000000	0 0	1

5.4 hello 的虚拟地址空间

调试工具：EDB

程序映射在内存中地址：0x400000-0x401000

通过 ELF 格式文件中的程序头表可以看到动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。在下面可以看出，程序包含 7 个段。

(1) PHDR: 保存程序头表。

(2) 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（如动态链接器）。

(3) 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（如字符串）、程序的目标代码等，并且分为读写两个部分。

(4) DYNAMIC 保存了由动态链接器使用的信息。

(5) NOTE 保存辅助信息。

(6) GNU_STACK: 权限标志，标志栈是否是可执行的。

(7) GNU_RELRO: 指定在重定位结束之后那些内存区域是需要设置只读。

(以上内容由我根据室友李大鑫的报告整理得到，故作转载声明)

程序头:

Type	Offset	VirtAddr	PhysAddr
FileSiz	MemSiz	Flags	Align
PHDR	0x0000000000000040	0x000000000000400040	0x000000000000400040
	0x000000000000001c0	0x000000000000001c0	R 0x8
INTERP	0x0000000000000200	0x000000000000400200	0x000000000000400200
	0x000000000000001c	0x000000000000001c	R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x000000000000400000	0x000000000000400000
	0x0000000000000076c	0x0000000000000076c	R E 0x200000
LOAD	0x0000000000000e50	0x000000000000600e50	0x000000000000600e50
	0x000000000000001f8	0x000000000000001f8	RW 0x200000
DYNAMIC	0x0000000000000e50	0x000000000000600e50	0x000000000000600e50
	0x000000000000001a0	0x000000000000001a0	RW 0x8
NOTE	0x000000000000021c	0x00000000000040021c	0x00000000000040021c
	0x00000000000000020	0x00000000000000020	R 0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x0000000000000e50	0x000000000000600e50	0x000000000000600e50
	0x000000000000001b0	0x000000000000001b0	R 0x1

具体信息如下:

这里对应5.3中的地址

这里对应5.3中的名称

这里是详细的数据段的信息

5.5 链接的重定位过程分析

(1) 节内容增加:

节名称	描述
.interp	保存ld.so的路径
.note.ABI-tag	Linux下特有的section
.hash	符号的哈希表
.gnu.hash	GNU拓展的符号的哈希表
.dynsym	运行时/动态符号表
.dynstr	存放.dynsym节中的符号名称
.gnu.version	符号版本
.gnu.version_r	符号引用版本
.rela.dyn	运行时/动态重定位表
.rela.plt	.plt节的重定位条目
.init	程序初始化需要执行的代码
.plt	动态链接-过程链接表
.fini	当程序正常终止时需要执行的代码
.eh_frame	contains exception unwinding and source language information.
.dynamic	存放被ld.so使用的动态链接信息
.got	动态链接-全局偏移量表-存放变量
.got.plt	动态链接-全局偏移量表-存放函数
.data	初始化了的数据
.comment	一串包含编译器的NULL-terminated字符串

(2) 函数增加：在使用 `ld` 命令链接的时候，指定了动态链接器为 64 的 `/lib64/ld-linux-x86-64.so.2`，`crt1.o`、`crti.o`、`crtm.o` 中主要定义了程序入口 `_start`、初始化函数 `_init`，`_start` 程序调用 `hello.c` 中的 `main` 函数，`libc.so` 是动态链接共享库，其中定义了 `hello.c` 中用到的 `printf`、`sleep`、`getchar`、`exit` 函数和 `_start` 中调用的 `__libc_csu_init`，`__libc_csu_fini`，`__libc_start_main`。链接器将上述函数加入。

(3) 函数调用：链接器在完成符号解析以后，就把代码中的每个符号引用和正好一个符号定义（即它的一个输入目标模块中的一个符号表条目）关联起来。此时，链接器就知道它的输入目标模块中的代码节和数据节的确切大小。然后就可以开始重定位步骤了，在这个步骤中，将合并输入模块，并为每个符号分配运行时的地址。在 `hello` 到 `hello.o` 中，首先是重定位节和符号定义，链接器将所有输入到 `hello` 中相同类型的节合并为同一类型的新的聚合节。例如，来自所有的输入模块的 `.data` 节被全部合并成一个节，这个节成为 `hello` 的 `.data` 节。然后，链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每一个符号。当这一步完成时，程序中的每条指令和全局变量都有唯一的运行时内存地址了。然后是重定位节中的符号引用，链接器会修改 `hello` 中的代码节和数据节中对每一个符号的引用，使得他们指向正确的运行地址。重定位算

法如下：

```

1  foreach section s {
2      foreach relocation entry r {
3          refptr = s + r.offset; /* ptr to reference to be relocated */
4
5          /* Relocate a PC-relative reference */
6          if (r.type == R_X86_64_PC32) {
7              refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8              *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9          }
10
11         /* Relocate an absolute reference */
12         if (r.type == R_X86_64_32)
13             *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14     }
15 }

```

整体示意图如下：



5.6 hello 的执行流程

函数执行流程图如下：

程序名称	程序地址
ld-2.27.so!_dl_start	0x7fce 8cc38ea0
ld-2.27.so!_dl_init	0x7fce 8cc47630
hello!_start	0x400500
libc-2.27.so!_libc_start_main	0x7fce 8c867ab0
-libc-2.27.so!__cxa_atexit	0x7fce 8c889430
-libc-2.27.so!_libc_csu_init	0x4005c0
hello!_init	0x400488
libc-2.27.so!_setjmp	0x7fce 8c884c10
-libc-2.27.so!_sigsetjmp	0x7fce 8c884b70
--libc-2.27.so!__sigjmp_save	0x7fce 8c884bd0
hello!main	0x400532
hello!puts@plt	0x4004b0
hello!exit@plt	0x4004e0
*hello!printf@plt	--
*hello!sleep@plt	--
*hello!getchar@plt	--
ld-2.27.so!_dl_runtime_resolve_xsave	0x7fce 8cc4e680
-ld-2.27.so!_dl_fixup	0x7fce 8cc46df0
--ld-2.27.so!_dl_lookup_symbol_x	0x7fce 8cc420b0
libc-2.27.so!exit	0x7fce 8c889128

5.7 Hello 的动态链接分析

定位到.interp 段的地址 0x400238，可以看到 hello 程序所用到的动态库。即 /lib64/ld-linux-x86-64.so.2。

在 edb 调试之后我们发现原先 0x00600a10 开始的 global_offset 表是全 0 的状态，在执行过 _dl_init 之后被赋上了相应的偏移量的值。这说明 dl_init 操作是给程序赋上当前执行的内存地址偏移量，这是初始化 hello 程序的一步。

（这里写的不好，可以改改）

5.8 本章小结

经过了最后一步百炼（链接），我们终于得到了这把绝世神器（可执行目标文件），接下来我们就要来领略它的风采。

但是正如伟大词人毛泽东主席所言一般：“雄关漫道真如铁，而今迈步从头越。”纵然造就了绝世的武器（可执行目标文件），如果使用不当的话（异常进程），不仅不能伤人，反而会伤了自己。因此接下来，我们就需要练就一身无敌的功法（shell 运行环境），来学习使用这把兵器。

第 6 章 hello 进程管理

6.1 进程的概念与作用

(1) 概念

进程是执行中的程序的实例。系统中的每个程序都在某个进程的上下文中运行。该过程为应用程序提供以下两个关键抽象：

1. 一个独立的逻辑控制流程，它提供了我们的程序专用处理器的错觉。
2. 一个私有地址空间，提供我们的程序独占使用内存系统的错觉。

(2) 作用

1. 判断服务器的健康状态

运维工程师最主要的工作就是保证服务器安全、稳定地运行。理想的状态是，在服务器出现问题，但是还没有造成服务器宕机或停止服务时，就人为干预解决了问题。进程管理最主要的工作就是判断服务器当前运行是否健康，是否需要人为干预。首先判断这个进程是否是正常进程，如果是正常进程，则说明你的服务器已经不能满足应用需求，你需要更好的硬件或搭建集群了；如果是非法进程占用了系统资源，则更不能直接中止进程，而要判断非法进程的来源、作用和所在位置，从而把它彻底清除。当然，如果服务器数量很少，我们完全可以人为通过进程管理命令来进行监控与干预；但如果服务器数量较多，那么人为手工监控就变得非常困难了，这时我们就需要相应的监控服务，如 cacti 或 nagios。总之，进程管理工作中最重要的工作就是判断服务器的健康状态，最理想的状态是服务器宕机之前就解决问题，从而避免服务器的宕机。

2. 查看系统中所有的进程

我们需要查看系统中所有正在运行的进程，通过这些进程可以判断系统中运行了哪些服务、是否有非法服务在运行。

3. 杀死进程

这是进程管理中最不常用的手段。当需要停止服务时，会通过正确关闭命令来停止服务（如 apache 服务可以通过 `service httpd stop` 命令来关闭）。只有在正确终止进程的手段失效的情况下，才会考虑使用 `kill` 命令杀死进程。

6.2 简述 Shell-bash 的作用与处理流程

(1) 作用

shell 是一个交互式应用程序级程序，代表用户运行其他程序。shell 执行一系列读取/评估步骤，然后终止。读取步骤从用户读取命令行。evaluate 步骤解析命令行并代表用户运行程序。

(2) 流程

1. 读取输入命令。
2. 阅读命令并获取所有参数
3. 如果是内置命令，立即执行；否则调用相应的程序来分配子进程并运行。
4. 接受键盘输入信号并处理。

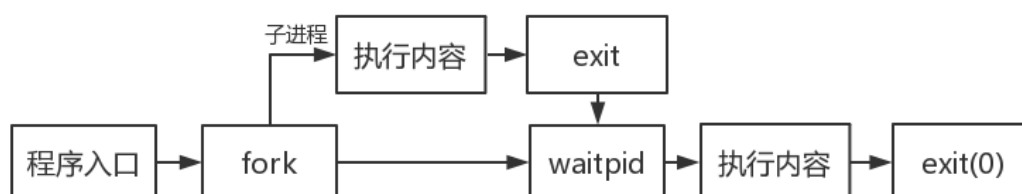
6.3 Hello 的 fork 进程创建过程

指令：./hello 1170400307 刘仁彪

系统中的每个程序都在某个进程的上下文中运行。该过程包括程序需要正确转换的状态。此状态包括程序的代码和存储在内存中的数据，其堆栈，通用寄存器的内容，程序计数器，环境变量以及打开的文件描述符集。

子级获取父级用户级虚拟地址空间的相同（但单独）副本，包括代码和数据段，堆，共享库和用户堆栈。子级还获取任何父级打开文件描述符的相同副本，这意味着子进程在调用 fork 时可以读取和写入父进程中打开的任何文件。父项和新创建的子项之间最显著的区别是它们具有不同的 PID

示意图如下：



6.4 Hello 的 execve 过程

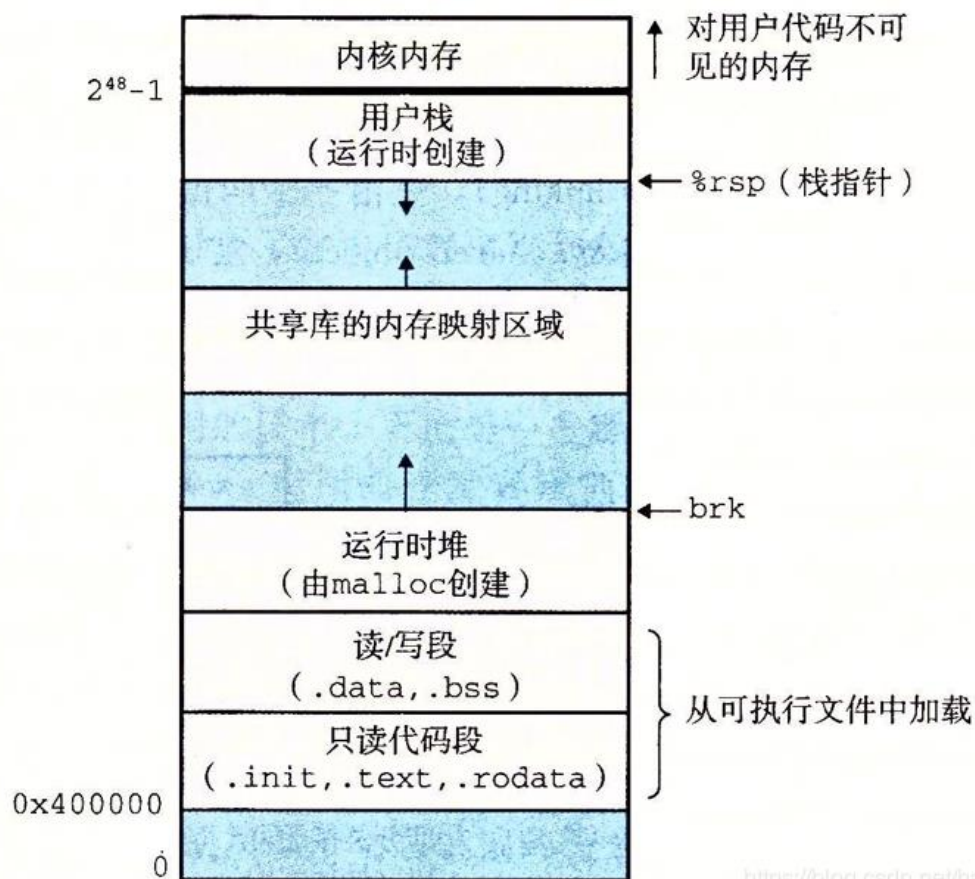
execve 函数在当前进程的上下文中加载并运行新程序。execve 函数使用参数列表 argv 和环境变量列表，加载并运行可执行目标文件名。

执行加载后，它会调用启动代码。启动代码设置堆栈并将控制权传递给新程序的主程序，该程序具有表格的原型或等效。

主要步骤包括：

- (1) 删除现有用户区域
- (2) 映射私人区域
- (3) 映射共享区域
- (4) 设置程序计数器

加载器创建的内存映像如下：



6.5 Hello 的进程执行

逻辑控制流：一系列程序计数器 PC 的值的序列叫做逻辑控制流，进程是轮流使用处理器的，在同一个处理器核心中，每个进程执行它的流的一部分后被抢占（暂时挂起），然后轮到其他进程。

时间片：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。

用户模式和内核模式：处理器通常使用一个寄存器提供两种模式的区分，该寄存器描述了进程当前享有的特权，当没有设置模式位时，进程就处于用户模式中，用户模式的进程不允许执行特权指令，也不允许直接引用地址空间中内核区内的代码和数据；设置模式位时，进程处

于内核模式，该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。

上下文信息: 上下文就是内核重新启动一个被抢占的进程所需要的状态, 它由通用寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构等对象的值构成。

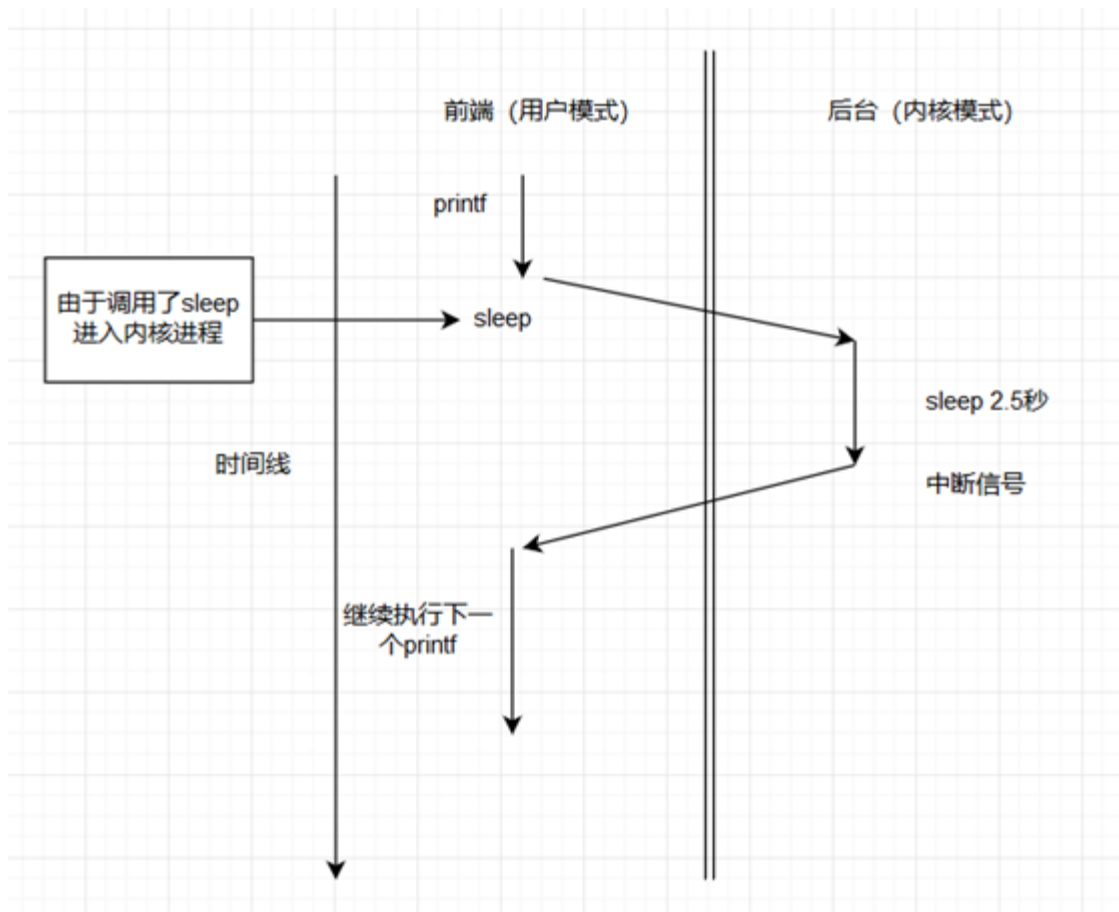
(以上内容由我室友李大鑫根据 CSAPP 整理得到, 故作转载声明)

并发执行多个流的一般现象称为并发。流程与其他流程轮流的概念也称为多任务处理。进程执行其一部分流的每个时间段称为时间片。因此, 多任务也被称为时间切片。

在执行进程的某些时刻, 内核可以决定抢占当前进程并重新启动先前抢占的进程。这个决定称为调度, 由内核中的代码处理, 称为调度程序。当内核选择要运行的新进程时, 我们说内核已经安排了进程。在内核计划运行新进程之后, 它会抢占当前进程, 并使用称为上下文切换的机制将控制权转移到新进程。

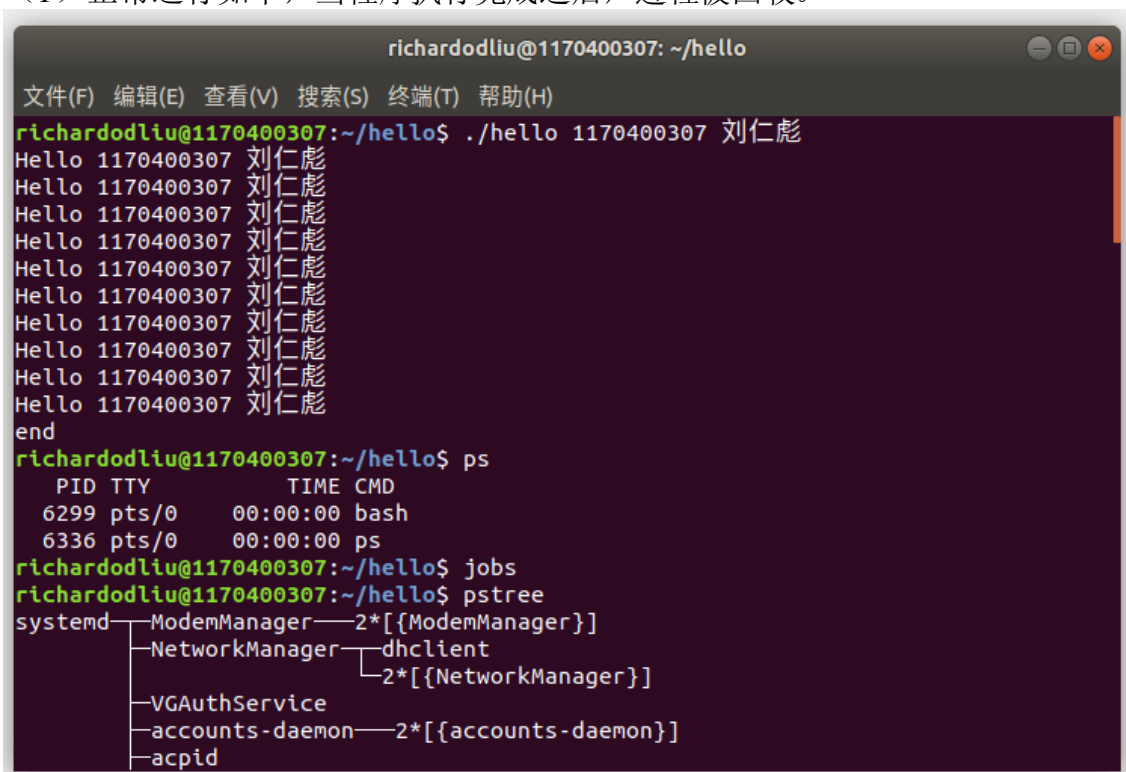
当内核代表用户执行系统调用时, 可能会发生上下文切换。如果系统调用由于等待某个事件发生而阻塞, 那么内核可以将当前进程置于休眠状态并切换到另一个进程。

另一个例子是睡眠系统调用, 这是一个将调用进程置于休眠状态的显式请求。



6.6 hello 的异常与信号处理

(1) 正常运行如下，当程序执行完成之后，进程被回收。



```
richardodliu@1170400307: ~/hello
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
richardodliu@1170400307:~/hello$ ./hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
end
richardodliu@1170400307:~/hello$ ps
  PID TTY          TIME CMD
  6299 pts/0        00:00:00 bash
  6336 pts/0        00:00:00 ps
richardodliu@1170400307:~/hello$ jobs
richardodliu@1170400307:~/hello$ pstree
systemd--ModemManager--2*[{ModemManager}]
         --NetworkManager--dhclient
         --                2*[{NetworkManager}]
         --VGAAuthService
         --accounts-daemon--2*[{accounts-daemon}]
         --acpid
```

(2) 是在程序输出 2 条 info 之后按下 ctrl-z 的结果，当按下 ctrl-z 之后，shell 父进程收到 SIGSTP 信号，信号处理函数的逻辑是打印屏幕回显、将 hello 进程挂起，通过 ps 命令我们可以看出 hello 进程没有被回收，此时他的后台 job 号是 1，调用 fg 1 将其调到前台，此时 shell 程序首先打印 hello 的命令行命令，hello 继续运行打印剩下的 8 条 info，之后输入字串，程序结束，同时进程被回收。

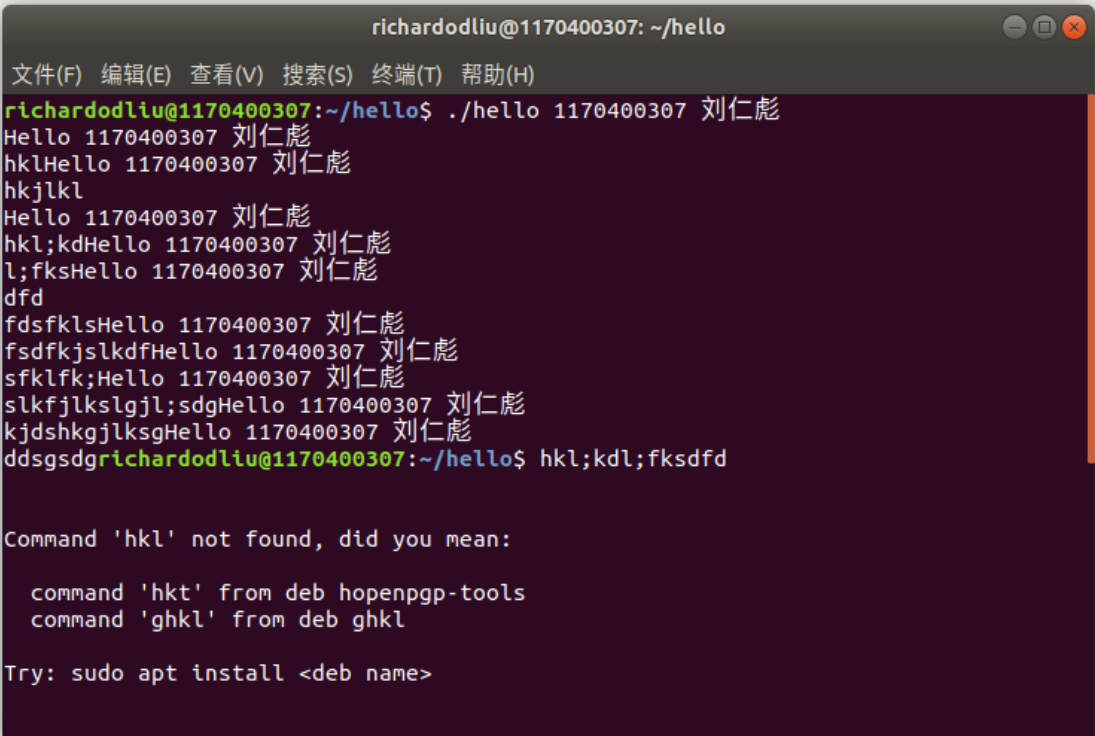
```
richardodliu@1170400307: ~/hello
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
richardodliu@1170400307:~/hello$ ./hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
^Z
[1]+ 已停止                  ./hello 1170400307 刘仁彪
richardodliu@1170400307:~/hello$ ps
  PID TTY          TIME CMD
  6357 pts/0        00:00:00 bash
  6383 pts/0        00:00:00 hello
  6411 pts/0        00:00:00 ps
richardodliu@1170400307:~/hello$ fg 1
./hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
end
richardodliu@1170400307:~/hello$ jobs
richardodliu@1170400307:~/hello$ pstree
systemd└─ModemManager─2*[{ModemManager}]
```

(3) 是在程序输出 3 条 info 之后按下 ctrl-c 的结果，当按下 ctrl-c 之后，shell 父进程收到 SIGINT 信号，信号处理函数的逻辑是结束 hello，并回收 hello 进程。

```
richardodliu@1170400307: ~/hello
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
richardodliu@1170400307:~/hello$ ./hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
^C
richardodliu@1170400307:~/hello$ ps
  PID TTY          TIME CMD
  6423 pts/0        00:00:00 bash
  6477 pts/0        00:00:00 ps
richardodliu@1170400307:~/hello$ jobs
richardodliu@1170400307:~/hello$ pstree
systemd└─ModemManager─2*[{ModemManager}]
        └─NetworkManager─dhclient
                        2*[{NetworkManager}]
        └─VGAAuthService
        └─accounts-daemon─2*[{accounts-daemon}]
        └─acpid
        └─avahi-daemon─avahi-daemon
        └─bluetoothd
        └─boltd─2*[{boltd}]
        └─colord─2*[{colord}]
        └─cron
        └─cups-browsed─2*[{cups-browsed}]
        └─cupsd
        └─2*[{dbus-daemon}]
```

(4) 是在程序运行中途乱按的结果，可以发现，乱按只是将屏幕的输入缓存到 stdin，当 getchar 的时候读出一个 '\n' 结尾的字串（作为一次输入），其他字串

会当做 shell 命令行输入。



```
richardodliu@1170400307: ~/hello
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
richardodliu@1170400307:~/hello$ ./hello 1170400307 刘仁彪
Hello 1170400307 刘仁彪
hklHello 1170400307 刘仁彪
hkjkl
Hello 1170400307 刘仁彪
hkl;kdHello 1170400307 刘仁彪
l;fksHello 1170400307 刘仁彪
dfd
fdsfklHello 1170400307 刘仁彪
fsdfkjslkdfHello 1170400307 刘仁彪
sfklfk;Hello 1170400307 刘仁彪
slkfjklsgjl;sdgHello 1170400307 刘仁彪
kjdshkgjlksgHello 1170400307 刘仁彪
ddsgsdgrichardodliu@1170400307:~/hello$ hkl;kdl;fkdfd

Command 'hkl' not found, did you mean:

  command 'hkt' from deb hopenpgp-tools
  command 'ghkl' from deb ghkl

Try: sudo apt install <deb name>
```

6.7 本章小结

欲要使用绝世的兵器（可执行目标文件），必先练就绝世的剑法（shell 运行的环境）。

一代宗师的绝世剑法，必然是锋利无比的，既能伤人，又能伤及。为了防止剑气伤及自身（异常指令），必须学会如何控制神剑（异常环境）。

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：是指由程序产生的与段相关的偏移地址部分，即 `hello.o` 里面的相对偏移地址。

线性地址：地址空间(address space) 是一个非负整数地址的有序集合，如果地址空间中的整数是连续的，那么我们说它是一个线性地址空间，即 `hello` 里面的虚拟内存地址。

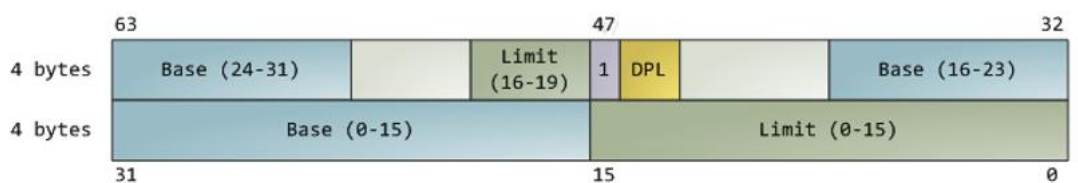
虚拟地址：CPU 通过生成一个虚拟地址，即 `hello` 里面的虚拟内存地址。

物理地址：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。计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有一个唯一的物理地址，即 `hello` 在运行时虚拟内存地址对应的物理地址。

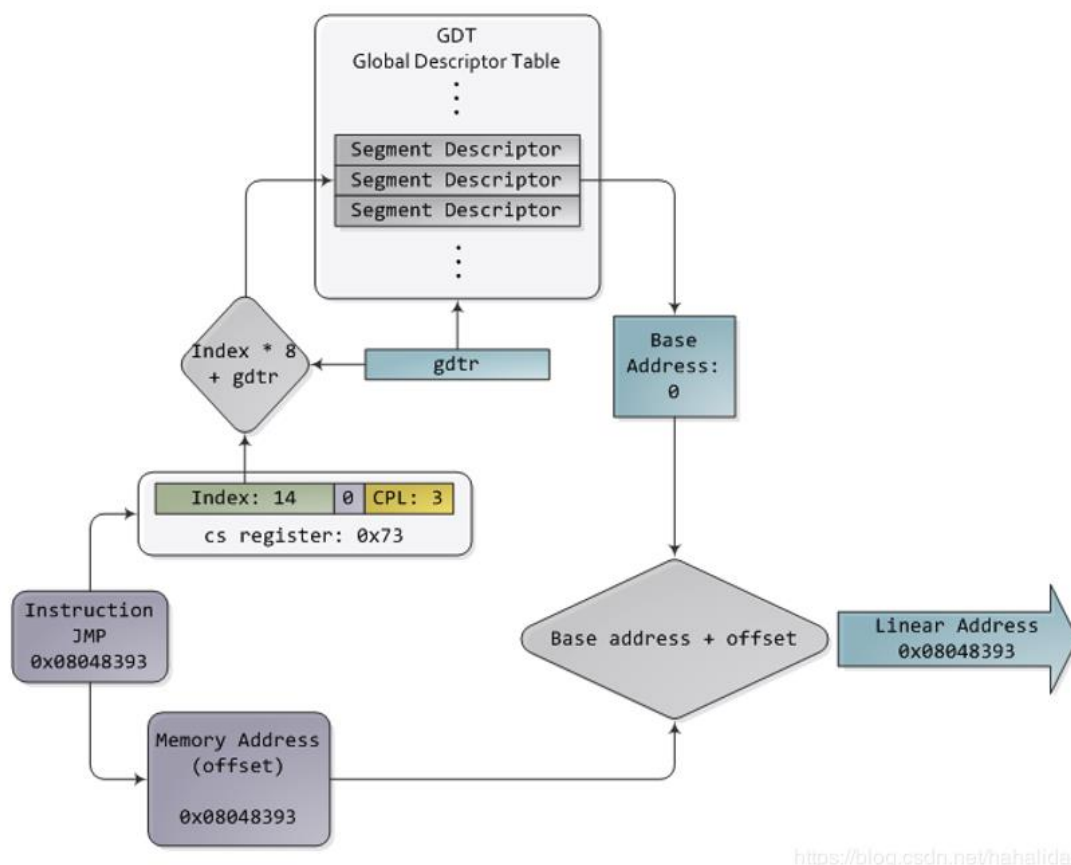
7.2 Intel 逻辑地址到线性地址的变换-段式管理

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：

索引号是“段描述符(segment descriptor)”的索引，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，每一个段描述符由 8 个字节组成，如图：



其中 **Base** 字段，它描述了一个段的开始位置的线性地址，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，由段选择符中的 **T1** 字段表示选择使用哪个， $=0$ ，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。如下图：



为了进行段管理，除了为每个程序设置段图像表外，操作系统还必须为整个主存储系统建立一个真实的主存储器管理表，包括占用区域表和可用区域表。占用区域表的每个项目（行）用于指示主存储器的哪些区域已被占用，程序的哪个段已被占用，以及主存储器中段的起始点和长度。另外，可以设置诸如在进入主存储器之后是否已经重写段的字段，使得当从主存储器释放段时，可以决定是否将其写回到其在辅助存储器中的原始位置以减少辅助操作。可用区域表的每个项目（行）指示每个未占用的基址和区域大小。当段从辅助存储器加载到主存储器中时，操作系统将一个条目添加到占用区域表并修改可用区域表。当段从主存储器退出时，它将其占用区域表中的项（行）移动到可用区域表中，并处理它是否可以与其他可用区域合并以修改可用区域表。当程序完成或由更高优先级的程序替换时，应该将程序的所有段的项目从占用区域表移动到可用区域表并相应地进行处理。

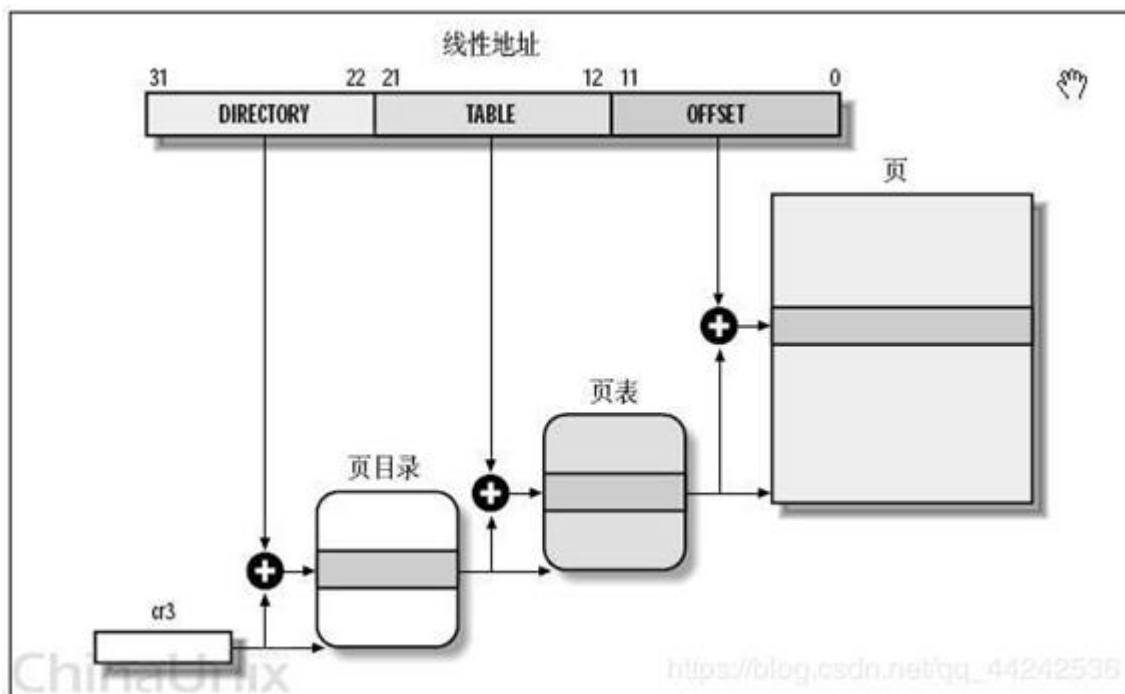
7.3 Hello 的线性地址到物理地址的变换-页式管理

从概念上讲，虚拟内存被组织为存储在磁盘上的 N 个连续字节大小单元的数组。每个字节都有一个唯一的虚拟地址，用作数组的索引。磁盘上阵列的内容

缓存在主内存中。与内存层次结构中的任何其他缓存一样，磁盘上的数据被分区为块，这些块用作磁盘和主内存之间的传输单元。VM 系统通过将虚拟内存划分为称为虚拟页面的固定大小块来处理此问题。每个虚拟页面的大小为 $P = 2^p$ 个字节。类似地，物理内存被划分为物理页面，也是 P 字节大小。

线性地址被分成固定长度的组，称为页面，例如 32 位机器。线性地址最高可达 4G，可分为 4 KB 页面。此页面分为大量的 `total_pages [2^20]`，共 20 页。这个大型数组就是我们所说的页面目录。目录中的每个目录项都是一个地址 - 相应页面的地址。另一种“页面”，我们称之为物理页面，或页面框架，页面 `zheng`。寻呼单元将所有物理内存划分为固定长度管理单元。它的长度通常与内存页面一一对一。请注意，`total_page` 数组有 2^{20} 个成员，每个成员都是一个地址（32 位，一个地址是 4 个字节），所以它占用 4 MB 的内存来代表这样一个数组。为了节省空间，引入了两级管理模型来组织寻呼单元。

如图所示：



由上图可得：

(1) 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。

(2) 每一个活动的进程，因为都有其独立的对应的虚拟内存（页目录也是唯一的），那么它也就对应了一个独立的页目录地址。——运行一个进程，需要将它页目录地址放到 `cr3` 寄存器中。

(3) 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, 得到最终我们想要的物理地址。

通过虚拟内存的页式管理, 可以做到如下的事情:

(1) 方便链接: 每个进程正是因为虚拟内存的页式管理可以拥有相同的虚拟内存映像而不用去管具体在物理内存中存在哪个地方的事情。这些都交给 MMU 进行 *address translation*

(2) 方便加载: 在一个新的子进程加载进一个可执行文件时, 将这个文件的所有东西都复制进自己的虚拟地址空间, 而是通过将虚拟页映射到某一块物理内存中。第一次执行存放在某一块区域中的命令来触发缺页中断, 这时虚拟页会自动滑入。

(3) 方便共享: 当两个进程想要共享代码段、数据段的时候, 比如共享库, 就没有必要在每个进程中都复制一份, 这是非常浪费内存空间的, 可以将本进程的一个虚拟页映射到物理内存, 这样持有一份副本就可以为多个进程所用。

(4) 方便内存分配: 每个进程都有自己独有的代码、数据、堆、栈, 如果是通过物理地址进行分配的话, 那么必须就把它们分配到连续的物理内存当中, 这是具有一定难度的事情, 但是通过虚拟页式管理可以将连续的虚拟页映射到不连续的物理页, 这样就降低了分配的难度。

(以上内容由刘帅根据 CSAPP 整理得到, 故作转载声明)

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

首先, 虚拟地址由 VPN 和 VPO 组成。VPN 可以用作 TLB 中的索引。如上图所示, TLB 可以看作是 PTE 的缓存, 它将常用的 PTE 缓存到 TLB 中以加速虚拟地址的转换。TLB 高度连接, 应设计为一次存储更多 PTE。如果在 TLB 中可以找到对应于 VPN 的 PTE, 即 TLB 命中, 则 TLB 直接给出 PPN, 然后 PPO 是 VPO, 从而构成物理地址。

如果你不能做 TLB 命中, 你必须到四级页面表找到地址。在 i7 中, VPN 有 36 位, 分为四个部分。从左到右的前三个九位对应于第一个三级页表中的偏移量, 页面表中的相应页表项指向下一个页表, 而后九个 VPN 对应于偏移量在页面表中。页表的最后一级中的页表条目存储在 PPN 中

例如，VPN1 对应于第一级页表中的页表条目，其指向下一级页表中的页表，然后依赖于 VPN2 在页表中查找其对应的页表条目。同样，这也指第三级页表中的页表，然后依赖于 VPN3 在页表中查找相应的页表项。到第四级页表中的页表，然后依靠 VPN4 查找相应的页表项，这个页表项存放在 PPN 中，在第四级页表中最多可存储 512G 的内存，显然一般没那么多。

最后，VPO 可以用作 PPO 来查找存储在相应物理页面上的内容。

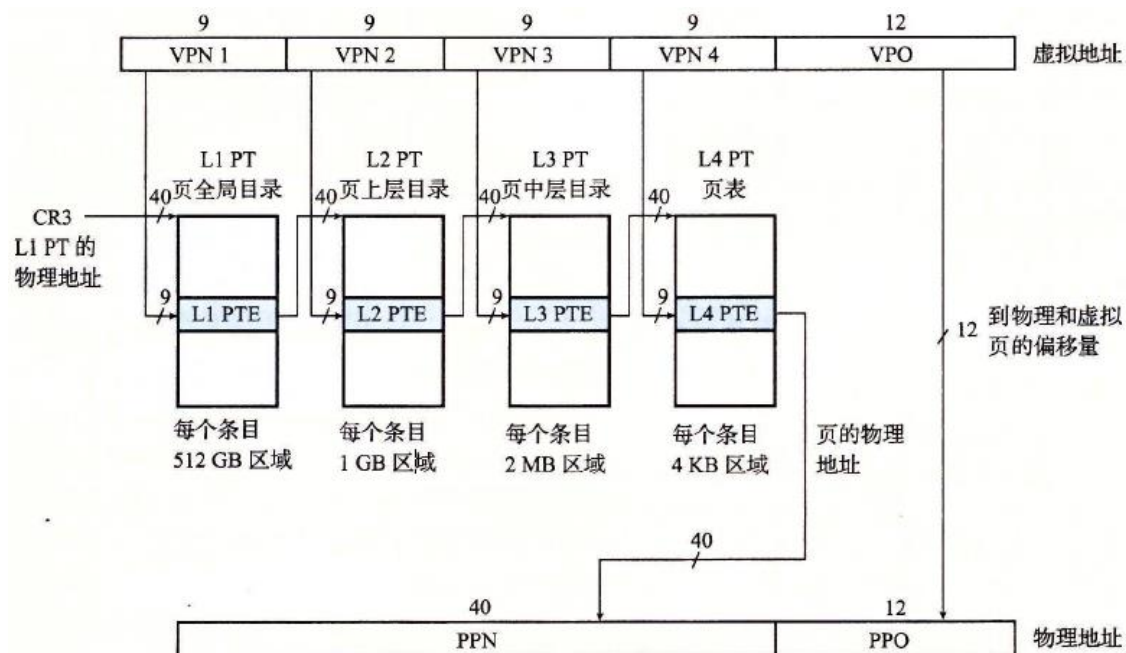


图 9-25 Core i7 页表翻译(PT: 页表, PTE: 页表条目, VPN: 虚拟页号, VPO: 虚拟页偏移, PPN: 物理页号, PPO: 物理页偏移量。图中还给出了这四级页表的 Linux 名字)

7.5 三级 Cache 支持下的物理内存访问

得到物理地址之后，先将物理地址拆分成 CT（标记）+CI（索引）+CO（偏移量），然后在一级 cache 内部找，如果未能寻找到标记位为有效的字节（miss）的话就去二级和三级 cache 中寻找对应的字节，找到之后返回结果。

处理器封装

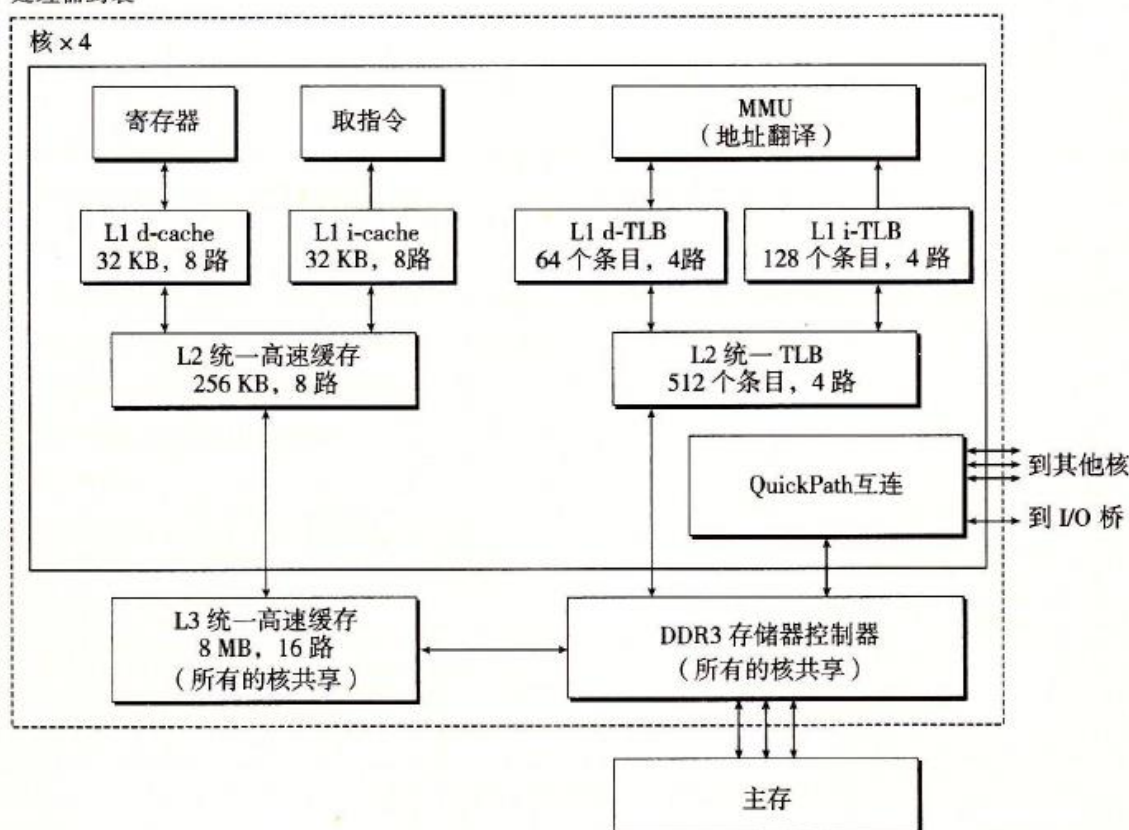


图 9-21 Core i7 的内存系统

7.6 hello 进程 fork 时的内存映射

当前进程调用 `fork` 函数时，内核会为新进程创建各种数据结构，并为其分配唯一的 PID。要为新进程创建虚拟内存，它会创建当前进程的 `mm_struct`，区域结构和页表的精确副本。它将两个进程中的每个页面标记为只读，并将两个进程中的每个区域结构标记为私有写入时。

当 `fork` 在新进程中返回时，新进程现在具有与调用 `fork` 时存在的虚拟内存的精确副本。当任一进程执行任何后续写入时，写时复制机制会创建新页面，从而为每个进程保留私有地址空间的抽象。

7.7 hello 进程 `execve` 时的内存映射

`execve` 函数在 `shell` 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：

删除已存在的用户区域。删除 `shell` 虚拟地址的用户部分中的已存在的区域结

构。

映射私有区域。为 `hello` 的代码、数据、`bss` 和栈区域创建新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 `hello` 文件中的 `.text` 和 `.data` 区。`bss` 区域是请求二进制零的，映射到匿名文件，其大小包含在 `hello` 中。栈和堆区域也是请求二进制零的，初始长度为零。图 7.7 概括了私有区域的不同映射。

映射共享区域。如果 `hello` 程序与共享对象（或目标）链接，比如标准 C 库 `libc.so`，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

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

7.8 缺页故障与缺页中断处理

情况 1：段错误：首先，先判断这个缺页的虚拟地址是否合法，那么遍历所有的合法区域结构，如果这个虚拟地址对所有的区域结构都无法匹配，那么就返回一个段错误

情况 2：非法访问：接着查看这个地址的权限，判断一下进程是否有读写改这个地址的权限。

情况 3：如果不是上面两种情况那就是正常缺页，那就选择一个页面牺牲然后换入新的页面并更新到页表。

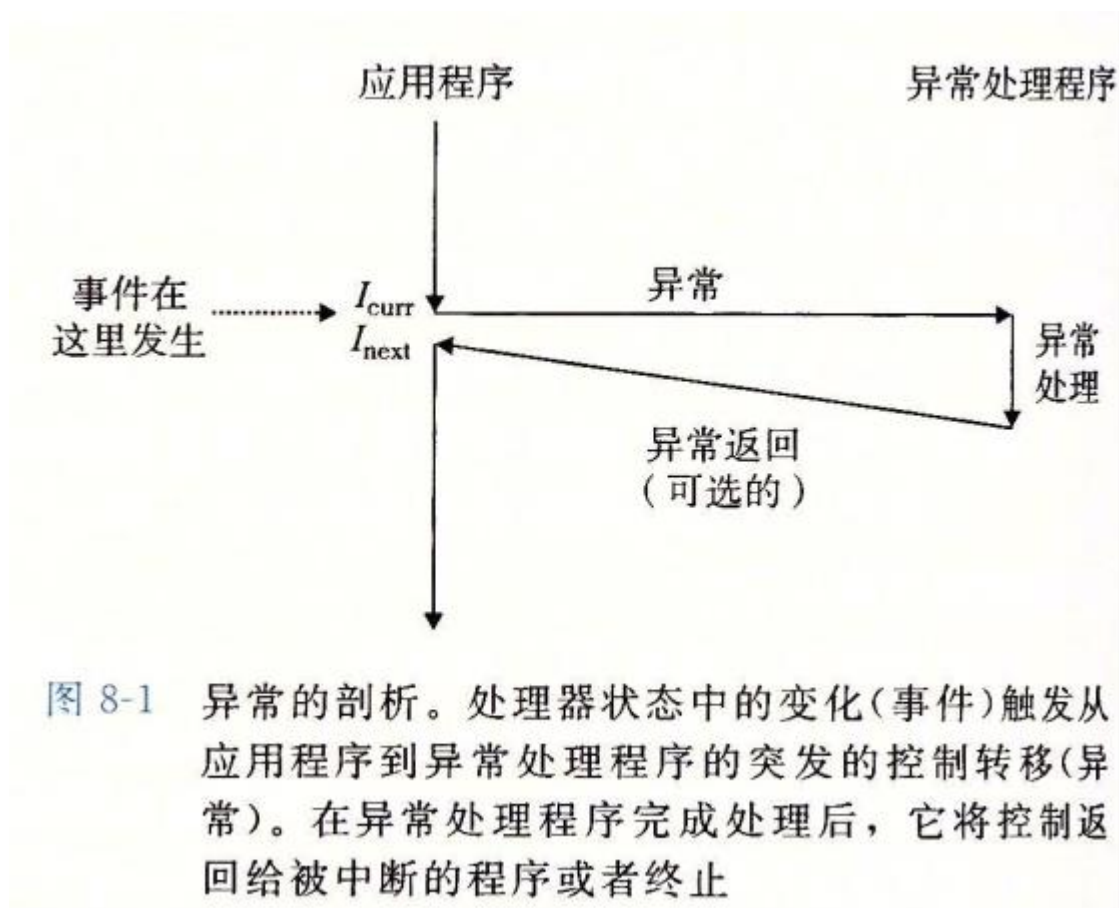


图 8-1 异常的剖析。处理器状态中的变化(事件)触发从应用程序到异常处理程序的突发的控制转移(异常)。在异常处理程序完成处理后，它将控制返回给被中断的程序或者终止

7.9 动态存储分配管理

使用动态内存分配的最主要原因是有些数据的大小事先未知，需要根据程序运行状况动态调整。分配器将堆视为一组不同大小的块，他们可以是空闲的，也可以是已经分配的。通过 `malloc` 和 `free` 函数，可以实现这两种状态之间的转换。

动态内存分配器维护一个进程的虚拟内存区域，称为堆 (heap)。它紧接着未初始化的数据 (.bss 段) 开始，并向上生长。对每个进程，内核维护一个指针 `brk`，指向堆的顶端。

2.1 动态内存分配器的基本原理 (5 分)

动态内存分配器维护一个进程的虚拟内存区域，称为堆 (heap)。它紧接着未初始化的数据 (.bss 段) 开始，并向上生长。对每个进程，内核维护一个指针 `brk`，指向堆的顶端。

分配器将堆视为一组不同大小的块 (block) 的集合来维护。每个块要么是已分配的，要么是空闲的。空闲块保持空闲，直到它显式地被应用程序所分配。已分配的块保持已分配状态，直到它被释放，这种释放要么是程序显式执行的，要

么是内存分配器自身隐式执行的（Garbage Collection）。

2.2 带边界标签的隐式空闲链表分配器原理（5分）

为每个块添加一个脚部，脚部就是头部的一个副本。如果每个块包括一个脚部，那么分配器可以通过检查它的脚部，判断前面一个块的起始位置和状态。这个脚部总是在距当前块开始位置一个字的距离。

2.3 显式空闲链表的基本原理（5分）

因为程序不需要访问一个空闲块的主体，因此我们可以在空闲块的主体里面显性地加入上一个和下一个空闲块的地址，形成一个（双向的）显式空闲链表。

维护空闲链表的方法有两种：其一是使用后进先出（LIFO）的顺序，将新释放的块放置在链表的开始处，这样分配器会首先检查最近使用过的块。

另一种是按照地址顺序来维护链表，其中每个块的地址都小于它后继的地址。此时，释放一个块需要的时间是线性的，但这样的内存利用率可以得到提升。

7.10 本章小结

有了神剑、有了剑法，能阻止他成为一代武林宗师的便只有内功心法了！

而天下内功（内存管理），纷纷扰扰，无非是对自身潜力（内存）的调用。而在这当中，排第一的便是九阳神功（动态内存管理），讲究的乃是无中生有、造化万象，实乃无上心法。

第 8 章 hello 的 I/O 管理

8.1 Linux 的 I/O 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，这使得所有的输入和输出都能以一种统一且一致的方式来执行，这就是 Unix I/O 接口。

8.2 简述 Unix I/O 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。应用程序只需记住这个描述符。

2. Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）。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一个称为 `end-of-file(EOF)` 的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的“EOF 符号”。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。

接口函数：

(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。

```
int fd;    /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

<https://blog.csdn.net/dxyinme>

(2) `int close(fd)`，`fd` 是需要关闭的文件的描述符，`close` 返回操作结果。

```
int fd;    /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

<https://blog.csdn.net/dxyinme>

(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。

```
char buf[512];
int fd;    /* file descriptor */
int nbytes; /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

<https://blog.csdn.net/dxyinme>

(4) `ssize_t write(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

```
char buf[512];
int fd;          /* file descriptor */
int nbytes;      /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

<https://blog.csdn.net/dxyinma>

8.3 printf 的实现分析

参数中明显采用了可变参数的定义，可以看到 `*fmt` 是一个 `char` 类型的指针，指向字符串的起始位置。

`printf` 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;
}
```

`vsprintf` 函数的作用是将所有的参数内容格式化之后存入 `buf`，然后返回格式化数组的长度。

`vsprintf` 代码：

```

int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

    for (p = buf; *fmt; fmt++)
    {
        if (*fmt != '%') //忽略无关字符
        {
            *p++ = *fmt;
            continue;
        }

        fmt++;

        switch (*fmt)
        {
            case 'x': //只处理%x一种情况
                itoa(tmp, *((int*)p_next_arg)); //将输入参数值转化为
                字符串保存在tmp
                strcpy(p, tmp); //将tmp字符串复制到p处
                p_next_arg += 4; //下一个参数值地址
                p += strlen(tmp); //放下一个参数值的地址
                break;
            case 's':
                break;
            default:
                break;
        }
    }

    return (p - buf); //返回最后生成的字符串的长度
}

```

write 函数是将 buf 中的 i 个元素写到终端的函数。

write 代码:

```

write:
    mov eax, NR_write
    mov ebx, [esp + 4]
    mov ecx, [esp + 8]
    int INT_VECTOR_SYS_CALL

```

8.4 getchar 的实现分析

异步异常-键盘中断的处理：键盘中断处理子程序，调用 read 系统函数，通过系统调用接受按键扫描码转成 ASCII 码，保存到系统的键盘缓冲区。

```

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;
}

```

<https://blog.csdn.net/dxyinme>

8.5 本章小结

多年以来，一代宗师在这把剑上花费了无数的心血，以内功孕育之（INPUT），终于心地至诚，将自己的无上功法铭刻于绝世武器之上。

临死之前，他在终南山下寻了一个洞穴，准备静静等待有缘人赶来，将武林秘籍传授于他（OUTPUT）

结论

现在，让我们静静地回味一代武林宗师墨云沧的一生：

- (1) 编写：借助 IDE 键入代码。
- (2) 预处理：将 `hello.c` 及外部的库展开合并。
- (3) 编译：将 `hello.i` 编译成为汇编文件。
- (4) 汇编：将 `hello.s` 会变成成为可重定位目标文件。
- (5) 链接：链接相关文件生成可执行目标文件。
- (5) 运行：在 `shell` 中输入开始运行指令。
- (6) 创建子进程：调用 `fork` 为其创建子进程。
- (7) 运行程序：调用 `execve` 启动加载器，加映射虚拟内存，进入程序入口后程序开始载入物理内存，然后进入主函数。
- (8) 执行指令：CPU 为其分配时间片，在一个时间片中，`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从而进行内存访问。
- (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) 处理信号：调用 `shell` 的信号处理函数。
- (12) 结束：回收子进程，内核删除相关数据。

附件

文件	功能
hello.i	预处理后的ASCII码文件
hello.o	汇编文件
hello.elf	可重定位目标文件
helloasm.txt	hello.o 反汇编代码
hello	最终的可执行目标文件
helloasmm.txt	hello 反汇编代码
helloelf_1k.txt	hello的部分elf内容
elfall.txt	hello的全部elf内容

参考文献

- [1] <https://www.cnblogs.com/pianist/p/3315801.html>
- [2] <http://www.cnblogs.com/linear/p/6773378.html>
- [3] <https://blog.csdn.net/nkguohao/article/details/8950744>.
- [4] https://blog.csdn.net/weixin_41143631/article/details/81221777
- [5] https://blog.csdn.net/sinat_27421407/article/details/78829508
- [6] <https://blog.csdn.net/liuchunjie11/article/details/80252811>