

哈爾濱工業大學

計算機系統

大作業

題 目 程序人生-Hello's P2P

專 業 _____

學 號 _____

班 級 _____

學 生 _____

指 導 教 師 _____

計算機科學與技術學院

2018 年 12 月

摘 要

人的一生，从十月怀胎到呱呱坠地，从牙牙学语到妙语连珠，从蹒跚学步到纵情奔跑，陪伴在我们身边的是父母。程序的一生也和人相差无多，从脑中构想到动手敲击，从单步执行到循环迭代，从调试卡机到飞速执行，驻守在旁边的是身为程序员的我们。

本文旨在讨论分析 hello 程序从简单的.c 文件一步步发展，通过预处理、编译、汇编等一系列的操作之后到最终执行的过程。通过对各个阶段的讨论和分析，有利于帮助我们更加明确的认识程序在计算机中从编写到执行的具体过程，从而达到“深入理解计算机系统”的最终目的。

关键词：hello；预处理；编译；汇编；链接；进程管理；存储管理；IO 管理；计算机系统

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

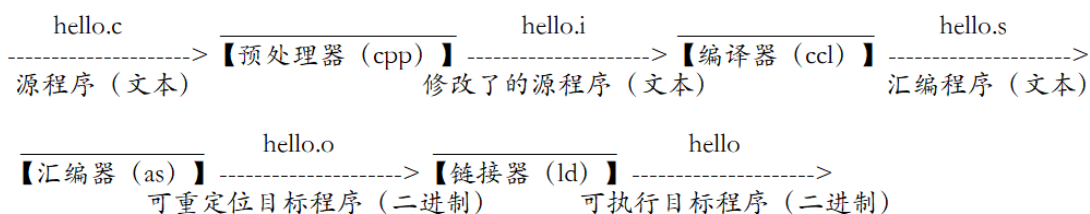
第 1 章 概述	- 4 -
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 4 -
第 2 章 预处理	- 6 -
2.1 预处理的概念与作用	- 6 -
2.2 在 UBUNTU 下预处理的命令	- 7 -
2.3 HELLO 的预处理结果解析	- 7 -
2.4 本章小结	- 8 -
第 3 章 编译	- 9 -
3.1 编译的概念与作用	- 9 -
3.2 在 UBUNTU 下编译的命令	- 9 -
3.3 HELLO 的编译结果解析	- 9 -
3.4 本章小结	- 13 -
第 4 章 汇编	- 14 -
4.1 汇编的概念与作用	- 14 -
4.2 在 UBUNTU 下汇编的命令	- 14 -
4.3 可重定位目标 ELF 格式	- 14 -
4.4 HELLO.O 的结果解析	- 16 -
4.5 本章小结	- 18 -
第 5 章 链接	- 19 -
5.1 链接的概念与作用	- 19 -
5.2 在 UBUNTU 下链接的命令	- 19 -
5.3 可执行目标文件 HELLO 的格式	- 19 -
5.4 HELLO 的虚拟地址空间	- 21 -
5.5 链接的重定位过程分析	- 21 -
5.6 HELLO 的执行流程	- 22 -
5.7 HELLO 的动态链接分析	- 23 -
5.8 本章小结	- 23 -
第 6 章 HELLO 进程管理	- 25 -
6.1 进程的概念与作用	- 25 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 25 -
6.3 HELLO 的 FORK 进程创建过程	- 25 -
6.4 HELLO 的 EXECVE 过程	- 26 -
6.5 HELLO 的进程执行.....	- 27 -
6.6 HELLO 的异常与信号处理	- 28 -
6.7 本章小结	- 30 -
第 7 章 HELLO 的存储管理.....	- 31 -
7.1 HELLO 的存储器地址空间	- 31 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 31 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 32 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 33 -
7.5 三级 CACHE 支持下的物理内存访问	- 34 -
7.6 HELLO 进程 FORK 时的内存映射	- 35 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 35 -
7.8 缺页故障与缺页中断处理.....	- 36 -
7.9 动态存储分配管理	- 37 -
7.10 本章小结	- 38 -
第 8 章 HELLO 的 IO 管理	- 39 -
8.1 LINUX 的 IO 设备管理方法	- 39 -
8.2 简述 UNIX IO 接口及其函数	- 39 -
8.3 PRINTF 的实现分析.....	- 40 -
8.4 GETCHAR 的实现分析.....	- 41 -
8.5 本章小结	- 41 -
结论	- 42 -
附件	- 43 -
参考文献.....	- 44 -

第 1 章 概述

1.1 Hello 简介

Hello 程序的生命周期是从一个高级 C 语言程序开始的。为了系统上运行 hello.c 程序，每条 C 语句都必须被其他程序转化未一系列的低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打好包，并以二进制磁盘文件的形式存在起来。目标程序也成为可执行目标文件。接下来的 p2p 过程，即从 program 变成 process 的过程，部分可以通过编译系统来完成，如下如所示，



1170300204

图 1-1 编译系统

之后在壳（bash）里面输入启动指令，进程管理就会为它 fork，创建子进程，然后通过 execve 和 mmap 等一系列操作，最终变成一个 process。

020（From Zero-0 to Zero-0）：在程序完成了相应的 p2p 过程之后，程序运行结束，处于终止状态，之后 shell 父进程回收 hello 并推出，使得 shell 又回到了运行 hello 之前的状态。即运行完 hello 之后，并不会改变 shell。或者说，hello 程序对 shell 状态的影响从 0 又变回了 0。

1.2 环境与工具

硬件环境：intel Core i7-7700HQ 64 位 CPU 8G RAM 256G SSD+1T HDD

软件环境：Win10 x64 VMware Workstation Pro 64 位 ubuntu

开发与调试工具：Code::Blocks gcc gdb edb hexedit 等

1.3 中间结果

hello.c	源文件
---------	-----

hello.i	预处理得到的文件
hello.s	编译后得到的文件
hello.o	汇编之后得到的文件
hello.txt	hello 的反汇编结果文件
hello.elf	hello.o 的 elf 格式文件
hello	链接之后得到的可执行文件

1.4 本章小结

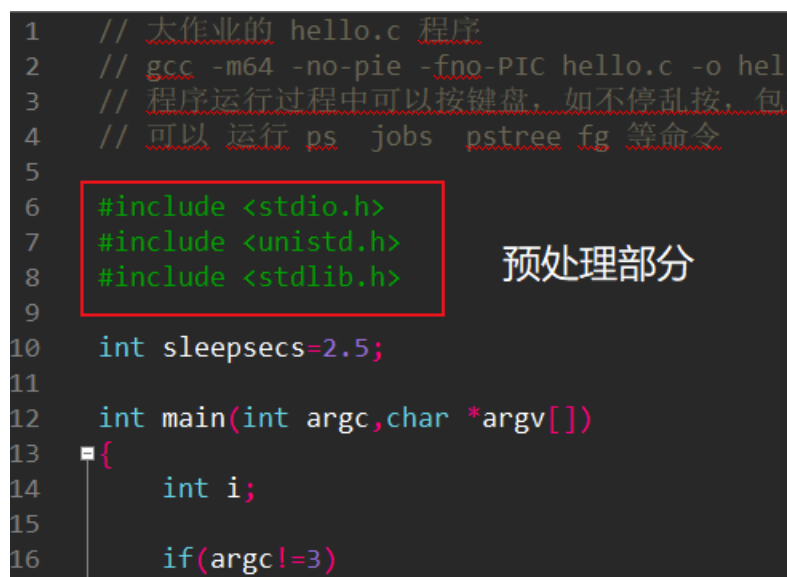
本章在宏观上浏览了 `hello` 的生命周期，并且总结了本文撰写过程中需要的软硬件环境及开发与调试工具。管中窥豹可见一斑，从简单的 `hello` 程序的分析可以推及到大部分其他程序。对此的深入体会将会对我们后续的工作又很大的帮助。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

概念：预处理一般是指在程序源码被翻译为目标代码的过程中，生成二进制代码之前的过程。典型的，由预处理器（preprocessor）对程序源代码文本进行处理，得到的结果在由编译器核心进一步编译。这个过程并不对程序的源代码进行解析，但是它把源代码分割成特定的单位----预处理记号（preprocessing token）用来支持语言特性。



```
1 // 大作业的 hello.c 程序
2 // gcc -m64 -no-pie -fno-PIC hello.c -o hell
3 // 程序运行过程中可以按键盘，如不停乱按，包
4 // 可以运行 ps jobs pstree fg 等命令
5
6 #include <stdio.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9
10 int sleepsecs=2.5;
11
12 int main(int argc,char *argv[])
13 {
14     int i;
15
16     if(argc!=3)
```

图 2-1 大作业给出 hello.c 中需要预处理的部分

功能：C 语言的预处理主要有三个方面的内容：1.宏定义 2.文件包含 3.条件编译。

宏定义是用一个标识符来表示一个字符串，这个字符串可以是常量、变量或表达式。在宏调用中将用该字符串代换宏名。宏定义可以带有参数，宏调用时是以实参代换形参。而不是“值传送”。为了避免宏代换时发生错误，宏定义中的字符串应加括号，字符串中出现的形式参数两边也应加括号。

文件包含是预处理的一个重要功能，用来将多个源文件连接成一个源文件进行编译，结果生成一个目标文件。

条件编译允许只编译源程序中满足条件的程序段，这样可以让生成的目标程序变短，从而减少了内存的开销，提升了程序的效率。

2.2 在 Ubuntu 下预处理的命令

```
Linux$ gcc -m64 -no-pie -fno-PIC hello.c -E -o hello.i
```

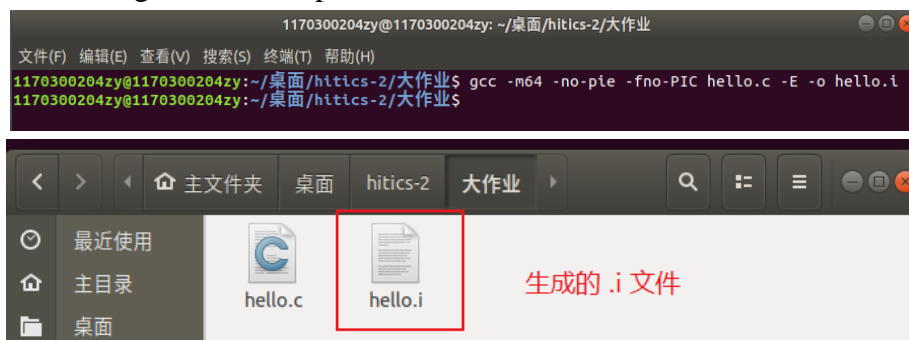


图 2-2 Linux 下生成 hello.i 文件

2.3 Hello 的预处理结果解析

使用 gedit 打开生成的 hello.i 文件之后,发现比之前的 hello.c 多出了很多东西: 例如 `#+数字+路径` 以及一些 `typedef` 定义、`extern` 等等。

```
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
```

图 2-3-1 .i 文件中多出的#内容

```
typedef __off64_t __loff_t;
typedef char *__caddr_t;

typedef long int __intptr_t;

typedef unsigned int __socklen_t;

typedef struct
{
    int __count;
    union
    {
        unsigned int __wch;
        char __wchb[4];
    } __value;
} __mbstate_t;
```

图 2-3-2 .i 文件中多出的 typedef 内容

```
extern int pclose (FILE *__stream);
```

图 2-3-3 .i 文件中多出的 extern 内容

这些中, 大部分是头文件 `stdio.h` `unistd.h` `stdlib.h` 中部分内容的插入。
在.i 文件的最后部分, 出现了原.c 文件的部分内容


```
# 9 "hello.c" 2

# 10 "hello.c"
int sleepsecs=2.5;

int main(int argc,char *argv[])
{
    int i;

    if(argc!=3)
    {
        printf("Usage: Hello 学号 姓名! \n");
        exit(1);
    }
    for(i=0;i<10;i++)
    {
        printf("Hello %s %s\n",argv[1],argv[2]);
        sleep(sleepsecs);
    }
    getchar();
    return 0;
}
```

图 2-3-4 .i 文件中包含的部分原.c 文件的内容

2.4 本章小结

本章主要展示了 `hello.c` 在编译系统中第一步预处理生成.i 文件的过程，简单说明了预处理的相关概念和作用。并通过亲自键入执行预处理命令，得到处理后的.i 文件，查看并简单分析文件中的相关内容。通过这样的过程，加深对相关知识的理解。

(第 2 章 0.5 分)

第3章 编译

3.1 编译的概念与作用

概念：编译即是将预处理之后的.i 文件转换成.s 文件，其中.s 文件中包含汇编语言程序。比较具体的概念为：

编译程序（Compiler，compiling program）也称为编译器，是指把用高级程序设计语言书写的源程序，翻译成等价的机器语言格式目标程序的翻译程序。它以高级程序设计语言书写的源程序作为输入，而以汇编语言或机器语言表示的目标程序作为输出。编译出的目标程序通常还要经历运行阶段，以便在运行程序的支持下运行，加工初始数据，算出所需的计算结果。

功能：编译的基本功能是把源程序翻译成相应的汇编语言程序，主要的工作过程有词法分析、语法分析、语义检查和中间代码生成、代码优化、目标代码生成等。主要是进行词法分析和语法分析，又称为源程序分析，分析过程中发现有语法错误则给出提示信息。

3.2 在 Ubuntu 下编译的命令

```
Linux$ gcc -m64 -no-pie -fno-PIC -S hello.i -o hello.s
```



图 3-1 Linux 下生成 hello.s 文件

3.3 Hello 的编译结果解析

3.3.1 数据

通过观察 hello.c 的源代码，发现程序中用到的数据类型有：整数、数组、字

字符串

其中字符串类型的变量出现如下图，

```

.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"

```

图 3-2 hello.s 中出现的字符串型数据

这两个字符串开头都有“.string”的前缀，表明了其数据类型。值得注意的是“Usage: Hello \345\255\246\……”里面，一些中文被翻译成了 UTF-8 编码形式。这间接表明计算机通过翻译成其他编码的形式间接地存储中文字符。

程序中的整数 `int sleepsecs` 是全局变量，且已经给出了初值。在.s 文件中，

```

.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2

```

图 3-3 hello.s 中出现的 int 型全局变量

通过这张图片，我们可以进行简单的分析：`sleepsecs` 存储在 .data 节中（.data 节中存储已初始化的全局和静态 C 变量），`.align 4` 表明对齐方式为 4，`.type sleepsecs, @object` 表明类型是对象（object），`.size sleepsecs, 4` 说明大小是 4 个字节，其后的 `.long 2` 应该是存储为 long 形式（个人猜测）。

Hello.s 中的数组型变量有 `char *argv[]` 存储的是 char 型指针

```

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

```

图 3-4 hello.s 中的 argv 数组

3.3.2 赋值

Hello 中的赋值语句有两条：

1. `int sleepsecs=2.5`; 这一条的赋值已经在之前的链接中完成（详见 2.3 节）
2. `for` 循环中的 `i=0`; 通过 `mov` 指令实现，

```
.L2:
    movl    $0, -4(%rbp)
    jmp     .L3
```

图 3-5 hello.s 中的赋值

值得注意的是，`movl` 把值传给了 `-4(%rbp)`，这是因为 `i` 是 `int` 型变量占了 4 个字节，所以要 `%rbp-4`。

3.3.3 类型转换

在全局变量定义的时候 `int sleepsecs=2.5` 里 `sleepsecs` 是 `int` 型变量却赋值为了 2.5，这里就蕴含着隐式类型转换。通过之前的分析，在内存中 `sleepsecs` 被存为了 2，说明 2.5 在转化成 `int` 型时进行了向下舍入。

3.3.4 算术操作

Hello 中涉及的算术操作有 `for` 循环中的 `i++`

```
for(i=0; i<10; i++)
```

图 3-6 hello 中的算术运算

在 `hello.s` 文件中的实现为，

```
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep
addl    $1, -4(%rbp)
```

图 3-7 hello.s 中的 `i++`

3.3.5 关系操作

Hello 中的关系操作有 `argc!=3` 和 `i<10`，实现的方式如下，

```
movl    %edi, -20(%rbp)  # argc
movq    %rsi, -32(%rbp)
cmpl    $3, -20(%rbp)    # argc ? 3
je      .L2

.L3:
    cmpl    $9, -4(%rbp)  # i ? 9
    jle     .L4
```

图 3-8 hello.s 中的 `argc!=3`图 3-9 hello.s 中的 `i<10`

有趣的是，再判断条件 `i<10` 中，与 `i` 进行 `cmpl` 的对象不是 10 而是 9，这有可能是编译器对关系操作的一种优化。

3.3.6 控制转移

一般的控制转移发生在判断条件 `cmp` 之后，所以 `hello` 中有两处控制转移，分别在两个判断语句（关系操作）`if(argv!3)` 和 `for(i=0;i<10;i++)`

控制转移的实现一般依托 `jmp` 及其扩展进行实现，

```
.L2:
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $3, -20(%rbp)
je       .L2
movl    $.LC0, %edi
call    puts
movl    $1, %edi
call    exit
.L2:
```

图 3-10 `argv!3` 的跳转(je)

```
.L2:
movl    $0, -4(%rbp)
jmp     .L3
.L4:
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi
movl    $.LC1, %edi
movl    $0, %eax
call    printf
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep
addl    $1, -4(%rbp)
.L3:
cmpl    $9, -4(%rbp)
jle     .L4
call    getchar
movl    %eax, %edi
```

图 3-11 来回跳转实现 `for` 循环

可以看到 `for` 循环语句的实现依赖于两个来回的跳转，通过逐次的比较（`i<10`）和对循环标志变量的操作（`i++`），在两个标志点之前来回跳转直到满足循环结束条件。

3.3.7 函数操作

`Hello` 中的函数操作（参数传递(地址\值)、函数调用()、函数返回 `return` 等）有，

`Printf` 函数

```
movl    $.LC0, %edi
call    puts
movl    $1, %edi
```

设置传入的参数%edi
函数调用

图 3-12 调用 `puts` 函数

```
movq    -32(%rbp), %rax
addq    $16, %rax
movq    (%rax), %rdx
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movq    %rax, %rsi
movl    $.LC1, %edi
movl    $0, %eax
call    printf
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep
```

设置参数
参数传递
函数调用

图 3-13 调用 `printf` 函数

可以看出 `printf` 函数在图 3-12 中是以 `puts` 的形式被调用，而在图 3-13 中则是以 `printf` 的形式，这应该也是编译程序的一种优化。

`Exit` 函数 `exit` 函数的传参和调用较简单，使用 `movl $1, %edi` 设置参数 使用 `call exit` 直接调用。

`Sleep` 函数同样如此，使用 `movl sleepsecs(%rip), %eax` 和 `movl %eax, %edi` 分两步设置传入的参数为 `sleepsecs`。

`Getchar` 函数调用则是直接 `call getchar`。

```
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep
        movl    $1, %edi
        call    exit
call    getchar
```

图 3-14 一些简单的函数调用

3.4 本章小结

本章中，我们近距离观察和分析了汇编代码，了解了高级语言翻译成汇编语言的一些简单的方法与规则。对于一些 C 语言中的基本的数据和操作的汇编代码形式有了一定的认识。

高级语言的抽象级别比较高，大多数时候这种抽象级别的工作效率会很高。但是对于严谨的程序员来说，能够阅读和理解汇编代码是一项很重要的技能。理解汇编代码可以尝试理解编译器的优化能力，并分析下其中隐含的低效率。能够理解汇编语言与原始 C 语言之间的联系是历届计算机如何执行程序的关键一步。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

概念：汇编器（as）将.s 文件汇编程序翻译成机器语言指令把这些指令打包成可重定位目标程序的格式，并将结果保存在.o 目标文件中。也就是从编译后的文件到生成机器语言二进制程序的过程。.o 文件就是对象文件,是可重定向文件的一种,通常以 ELF 格式保存，里面包含了对各个函数的入口标记、描述。

作用：可以将汇编代码翻译成机器可以理解并执行的机器指令。

4.2 在 Ubuntu 下汇编的命令

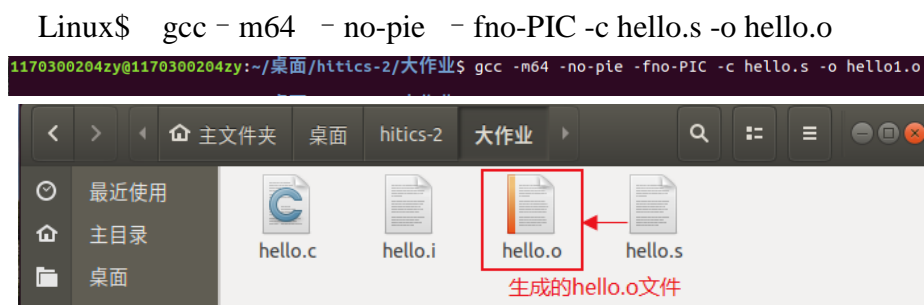


图 4-1 .o 文件的生成

4.3 可重定位目标 elf 格式

使用 **readelf -a hello.o** 指令观察.o 文件的 ELF 格式。

首先说明的是，ELF(Executable and Linking Format)是一种对象文件的格式，用于定义不同类型的对象文件(Object files)中都放了什么东西、以及都以什么样的格式去放这些东西。而 hello.o 的 ELF 格式的具体结构和内容如下，

1. ELF 头：ELF 文件头被固定地放在不同类对象文件的最前面。里面的内容大多是 ELF 文件的一些基本信息。

其中，Magic（魔数）用来指明该文件是一个 ELF 目标文件。第一个字节 7f 是个固定的数；后面的三个字节正是 E（45）、L（4c）、F（46）三个字节的 ASCII 码。

类别（CLASS）表示文件类型，这里是 64 位的 ELF 格式。

数据（Data）表示文件中的数据是按照什么格式组织（大端或小段）的，不同处理器平台数据组织格式可能就不同，如 x86 平台位小端存储格式。

之后的内容在图片里面可以简单看出，在此就不一一说明了。

```

ELF 头:
Magic:      7f 45 4c 46 02 01 01 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: 1104 (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

```

图 4-2 ELF 头的相关内容

2. 节头: 包含了各个节的名称、类型、大小、偏移量等信息。
对于各个节所存储的信息, 大致如下

.text: 已编译程序的机器代码。

.rodata: 只读数据, 比如 printf 语句中的格式串和开关 (switch) 语句的跳转表。

.data: 已初始化的全局 C 变量。局部 C 变量在运行时被保存在栈中, 既不出现在 .data 中, 也不出现在 .bss 节中。

.bss: 未初始化的全局 C 变量。在目标文件中这个节不占据实际的空间, 它仅仅是一个占位符。目标文件格式区分初始化和未初始化变量是为了空间效率在: 在目标文件中, 未初始化变量不需要占据任何实际的磁盘空间。

.symtab: 一个符号表 (symbol table), 它存放在程序中被定义和引用的函数和全局变量的信息。一些程序员错误地认为必须通过 -g 选项来编译一个程序, 得到符号表信息。实际上, 每个可重定位目标文件在 .symtab 中都有一张符号表。然而, 和编译器中的符号表不同, .symtab 符号表不包含局部变量的表目。

.rela.text: 当链接器把这个目标文件和其他文件结合时, .text 节中的许多位置都需要修改。一般而言, 任何调用外部函数或者引用全局变量的指令都需要修改。另一方面调用本地函数的指令则不需要修改。注意, 可执行目标文件中并不需要重定位信息, 因此通常省略, 除非使用者显式地指示链接器包含这些信息。

.rela.data: 被模块定义或引用的任何全局变量的信息。一般而言, 任何已初始化全局变量的初始值是全局变量或外部定义函数的地址都需要被修改。

.debug: 一个调试符号表, 其有些表目是程序中定义的局部变量和类型定义, 有些表目是程序中定义和引用的全局变量, 有些是原始的 C 源文件。只有以 -g 选项调用编译驱动程序时, 才会得到这张表。

.line: 原始 C 源程序中的行号和 .text 节中机器指令之间的映射。只有以 -g 选项调用编译驱动程序时, 才会得到这张表。

.strtab: 一个字符串表, 其内容包括 .symtab 和 .debug 节中的符号表, 以及节头部中的节名字。字符串表就是以 null 结尾的字符串序列。

节头:

[号]	名称	类型	地址	偏移量	对齐
	大小	全体大小	旗标 链接 信息		
[0]		NULL	0000000000000000	00000000	
	0000000000000000	0000000000000000	0	0	0
[1]	.text	PROGBITS	0000000000000000	00000040	
	0000000000000081	0000000000000000	AX	0	1
[2]	.rela.text	RELA	0000000000000000	00000340	
	00000000000000c0	0000000000000018	I	10	1 8
[3]	.data	PROGBITS	0000000000000000	000000c4	
	0000000000000004	0000000000000000	WA	0	0 4
[4]	.bss	NOBITS	0000000000000000	000000c8	
	0000000000000000	0000000000000000	WA	0	0 1
[5]	.rodata	PROGBITS	0000000000000000	000000c8	
	000000000000002b	0000000000000000	A	0	0 1
[6]	.comment	PROGBITS	0000000000000000	000000f3	
	000000000000002b	0000000000000001	MS	0	0 1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	0000011e	
	0000000000000000	0000000000000000	0	0	1
[8]	.eh_frame	PROGBITS	0000000000000000	00000120	
	0000000000000038	0000000000000000	A	0	0 8
[9]	.rela.eh_frame	RELA	0000000000000000	00000400	
	0000000000000018	0000000000000018	I	10	8 8
[10]	.symtab	SYMTAB	0000000000000000	00000158	
	0000000000000198	0000000000000018	11	9	8
[11]	.strtab	STRTAB	0000000000000000	000002f0	
	000000000000004d	0000000000000000	0	0	1
[12]	.shstrtab	STRTAB	0000000000000000	00000418	
	0000000000000061	0000000000000000	0	0	1

图 4-3 节头

3. 重定位节:当链接器把这个目标文件和其他文件结合时, .text 节中的许多位置都需要修改。一般而言, 任何调用外部函数或者引用全局变量的指令都需要修改。另一方面调用本地函数的指令则不需要修改。值得注意的是, 可执行目标文件中并不需要重定位信息, 因此通常省略, 除非使用者显式地指示链接器包含这些信息。

重定位节 '.rela.text' at offset 0x310 contains 8 entries:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000016	00050000000a	R_X86_64_32	0000000000000000	.rodata + 0
00000000001b	000b00000002	R_X86_64_PC32	0000000000000000	puts - 4
000000000025	000c00000002	R_X86_64_PC32	0000000000000000	exit - 4
00000000004c	00050000000a	R_X86_64_32	0000000000000000	.rodata + 1e
000000000056	000d00000002	R_X86_64_PC32	0000000000000000	printf - 4
00000000005c	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4
000000000063	000e00000002	R_X86_64_PC32	0000000000000000	sleep - 4
000000000072	000f00000002	R_X86_64_PC32	0000000000000000	getchar - 4

重定位节 '.rela.eh_frame' at offset 0x3d0 contains 1 entry:

偏移量	信息	类型	符号值	符号名称 + 加数
000000000020	000200000002	R_X86_64_PC32	0000000000000000	.text + 0

图 4-4 .rela.txt 节

由图可见, hello 中的重定位信息包括 puts、exit、printf、sleepsecs、sleep、getchar 的重定位声明。另外的, .rodata+0 对应的是 .L0 即第一个 printf 中的字符串, 而.rodata+1e 对应的是 .L1 即第二个 printf 中的字符串。.rela.eh_frame 中存放的是 eh_frame 节的重定位信息。

4. 符号表 (symbol table): 存放在程序中被定义和引用的函数和全局变量的信息。

4.4 Hello.o 的结果解析

键入语句 **objdump -d -r hello.o > hello.txt** 并打开生成的 hello.txt 文件, 与之前的 hello.s 文件的内容进行比较分析。

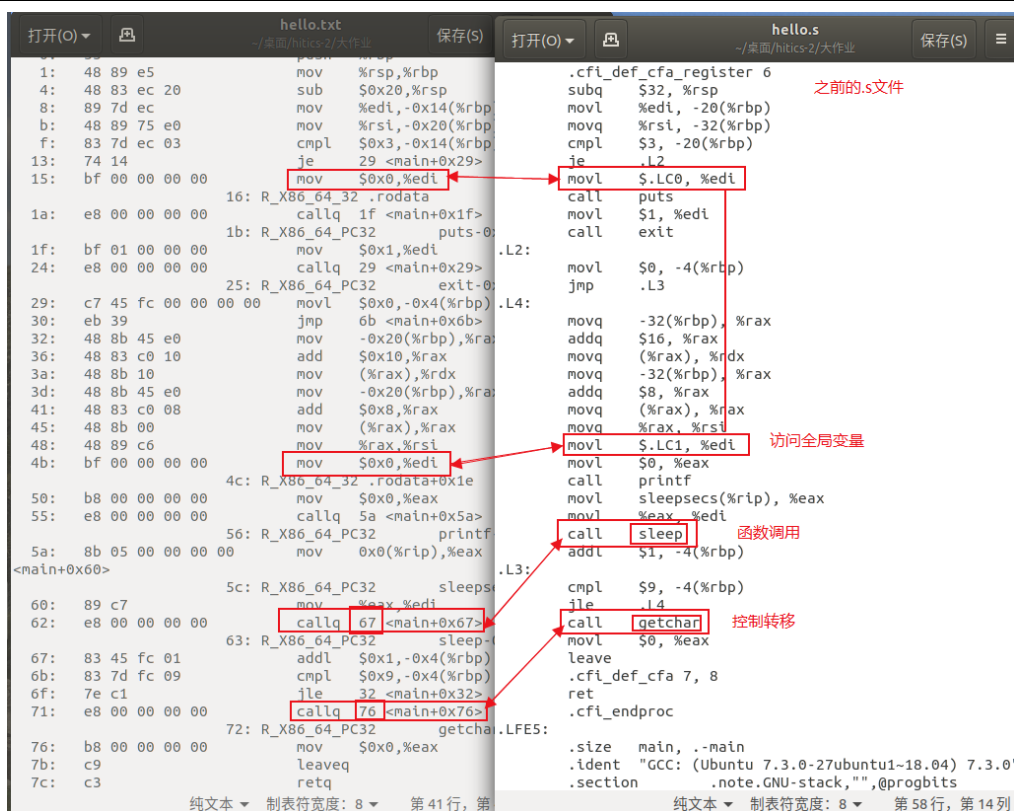


图 4-5 反汇编文件与.s 文件的比较

通过比较，我们发现了一些很有意思的区别。

对于全局变量访问来说，hello.s 中采用的是 \$+段名称的形式，而反汇编文件中是 \$+数，

函数调用方面，hello.s 中 call 之后直接跟着函数的名称，而反汇编文件中则是 call 的目标地址。这是因为，在编译阶段，gcc -c 使得函数调用被一个名称占位符写入，也就是 call sleep、call exit 的形式，而真正的 sleep、exit 的地址要在链接的阶段被真正的地址取代。在链接阶段因为 sleep、exit 的真正地址已经知道，所以用真正的地址去取代字符型函数名，从而变成一个数字。

在控制转移方面，跳转指令后面跟着的是段名称，而反汇编文件中为一个数字。这是因为，段名称仅仅是汇编语言中的助记符，机器是无法识别的，在汇编成机器语言之后显然需要经过修改为相应的地址。

机器指令与汇编语言在一定程度上有着一一对应的关系。汇编程序中存有汇编指令（助记符）和机器指令之前对应关系的对照表。通过扫遍查找对照表可以将汇编指令序列快速的翻译为机器指令序列，这样就可以将汇编文件翻译为目标程序，进而通过链接后生成可执行的机器码文件（如 exe 文件）。

4.5 本章小结

机器指令是计算机执行的命令，而汇编语言是具有一定意义的文字命令，与机器语言一一对应。汇编语言可以通过会变得到机器语言，机器语言可以通过反汇编得到汇编语言。汇编过程还包括变量内存管理，即经过汇编之后所有的变量和函数都变成了地址，而常量也变成了对应的值。但是汇编语言还是不够直观，一个简单的动作需要大量的语句来描述，因此才有了高级语言。这就是高级语言、汇编语言和机器语言之间的大致简单关系。

本章中，简要分析了 `hello.s` 生成 `hello.o` 的大致过程，并观察了 `.o` 文件的 ELF 格式，又使用了 `objdump` 进行了反汇编，体会了反汇编文件与 `.s` 文件之间的差别所在。这对于我们理解程序的发展又是很重要且关键的一步。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

概念：链接（linking）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，链接可以执行与编译时，也就是在源代码被演绎成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。

作用：理解链接器将帮助我们构造大型程序；理解链接器将帮助我们避免一些危险的编程错误；理解链接将帮助我们理解语言的作用域规则是如何实现的；理解链接将帮助我们理解其他重要的系统概念；理解链接将是我们能够利用共享库。

5.2 在 Ubuntu 下链接的命令

```
Linux$ ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o
/usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o
```



图 5-1 链接形成可执行文件

5.3 可执行目标文件 hello 的格式

使用 readelf 观察 helle：readelf -a hello > hello.elf
打开生成的 hello.elf 并作如下观察分析，

1. ELF 头：和之前分析的 ELF 头大致相似，包含了 magic（魔数）、类别、数据、

版本等一系列的信息。在此不做过多分析。

2. 节头:

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]	0000000000000000	NULL	0000000000000000	0	0	0
[1]	.interp 000000000000001c	PROGBITS	0000000000400200	A	0	1
[2]	.note.ABI-tag 0000000000000020	NOTE	000000000040021c	A	0	4
[3]	.hash 0000000000000034	HASH	0000000000400240	A	5	8
[4]	.gnu.hash 000000000000001c	GNU_HASH	0000000000400278	A	5	8
[5]	.dynsym 00000000000000c0	DYNSYM	0000000000400298	A	6	1
[6]	.dynstr 0000000000000057	STRTAB	0000000000400358	A	0	1
[7]	.gnu.version 0000000000000010	VERSYM	00000000004003b0	A	5	2
[8]	.gnu.version_r 0000000000000020	VERNEED	00000000004003c0	A	6	1
[9]	.rela.dyn 0000000000000030	RELA	00000000004003e0	A	5	8
[10]	.rela.plt 0000000000000078	RELA	0000000000400410	AI	5	19
[11]	.init 0000000000000017	PROGBITS	0000000000400488	AX	0	4
[12]	.plt 0000000000000060	PROGBITS	00000000004004a0	AV	0	16

图 5-2 节头信息

观察节头，可以发现里面标注了各节的基本信息，如大小、类型、地址等，根据偏移量等信息，我们就可以通过 hexedit 等工具访问各个节的位置。

3. 程序头 (program headers):

程序头:					
Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags	Align	
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040	R	0x8
INTERP	0x0000000000000200	0x0000000000400200	0x0000000000400200	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000	R E	0x200000
LOAD	0x0000000000000e50	0x0000000000600e50	0x0000000000600e50	RW	0x200000
DYNAMIC	0x0000000000000e50	0x0000000000600e50	0x0000000000600e50	RW	0x8
NOTE	0x000000000000021c	0x000000000040021c	0x000000000040021c	R	0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000	RW	0x10
GNU_RELRO	0x0000000000000e50	0x0000000000600e50	0x0000000000600e50	R	0x1

图 5-3 程序头信息

在程序头里有许多栏数据，其代表的分别是：PHDR：程序头表；INTERP：程序执行前需要调用的解释器；LOAD：程序目标代码和常量信息；DYNAMIC：动态链接器所使用的信息；NOTE：辅助信息；GNU_EH_FRAME：保存异常信息；GNU_STACK：使用系统栈所需要的权限信息；GNU_RELRO：保存在重定位之后只读信息的位置。

4. 之后还有一些重定位头等信息，在此不做分析。

5.4 hello 的虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息。

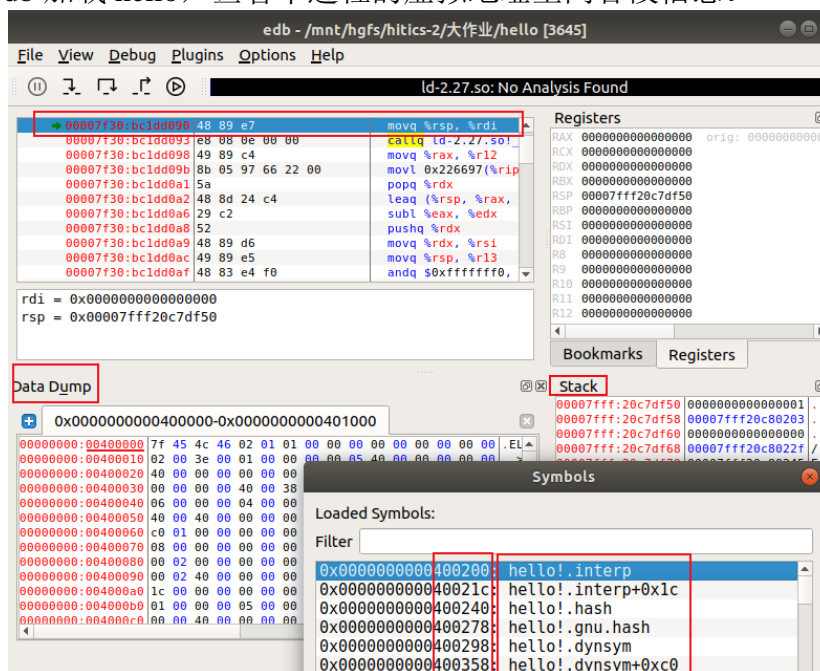


图 5-4 edb 中观察 hello

可以看到，在 Data Dump 中存放着详细的数据段信息，在 symbols 和汇编代码区域有着对应于图 5-3 中的地址

5.5 链接的重定位过程分析

使用语句 `objdump -d -r hello` 分析 hello 与 hello.o 的不同。



图 5-5 objdump 反汇编 hello

观察得到的文件，发现较之前多了许多节，大致信息如下：

.init 节：存储着程序初始化需要执行的代码 .plt 节：动态链接-过程链接表

.text 节：程序代码段 .fini 节：程序正常终止需要执行的代码

在之前的反汇编代码中，并未出现.init 节、.plt 节等节。而且有：hello.o 中的相对地址偏移在 hello 中为虚拟内存地址，hello 中多出了很多附加函数的 push 和 jmp 信息，而且 hello.o 中的控制转移和函数调用的地址在 hello 变成了虚拟内存地址。

重定位：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。

5.6 hello 的执行流程

使用 edb 执行 hello，并将调用和跳转的各个子程序名列出如下：

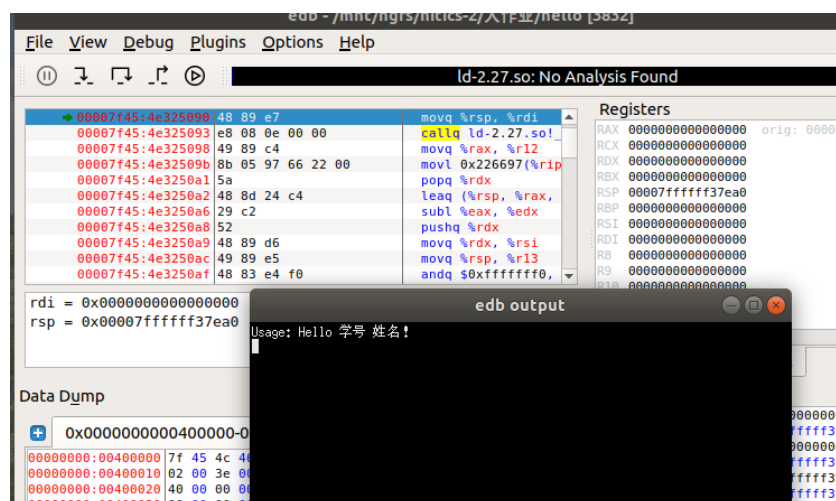


图 5-6 在 edb 中运行 hello

名称
ld-2.27.so!_dl_start

5.7 Hello 的动态链接分析

图 5-7 执行 dl init 前后的变化

5.8 本章小结

- 23 -

接以及到运行时的共享库的动态链接。我们使用了 `hello` 的实际试例来描述基本的机制。并且观察了可执行文件的 `ELF` 格式并与之前几章的内容进行了比较。运用了 `edb` 调试工具分析了 `hello` 的虚拟地址空间、重定位过程、运行顺序和动态链接的过程，有助于我们对于链接基本概念的理解。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。进程的概念主要有两点：第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为进程。

简单来说，shell 是一个交互性应用及程序，代表用户运行其他程序。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 执行一系列的读/求值步骤。按步骤读取用户的命令行，求值步骤解析命令。代表用户运行。Shell 命令行的执行过程分为 15 步，分别为：

1、读取命令行 2、命令历史替换 3、别名替换 4、花括号扩展 5、波浪号替换 6、I/O 重定向 7、变量替换 8、命令替换 9、单词解析 10、文件名生成 11、引用字符处理 12、进程替换 13、环境处理 14、执行命令 15、跟踪执行过程

6.3 Hello 的 fork 进程创建过程

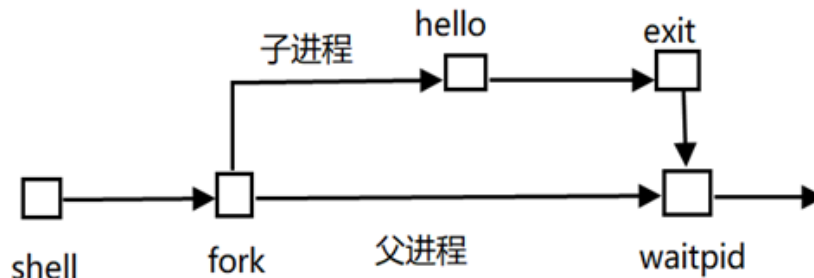


图 6-1 hello 的 fork 进程创建过程

6.4 Hello 的 execve 过程

当 fork 之后，子进程调用 `execve` 函数在当前进程的上下文中加载并运行一个新程序 `hello`，`execve` 调用驻留在内存中的操作系统代码执行 `hello` 进程，加载器删除子进程现有的虚拟内存段并创建一组新的代码数据、堆和栈。创建的新内容被初始化为零，通过将虚拟地址空间中的页映射到可执行文件的页大小的片，新的代码和数据段被初始化为可执行文件中的内容。最后加载器设置 PC 指向 `_start` 地址，`_start` 最终调用 `hello` 中的 `main` 函数。除了一些头部信息，在加载过程中没有任何从磁盘到内存的数据复制。直到 CPU 引用一个被映射的虚拟页时才会进行复制，这时，操作系统利用他的页面调度机制自动将页面从磁盘传送到内存。

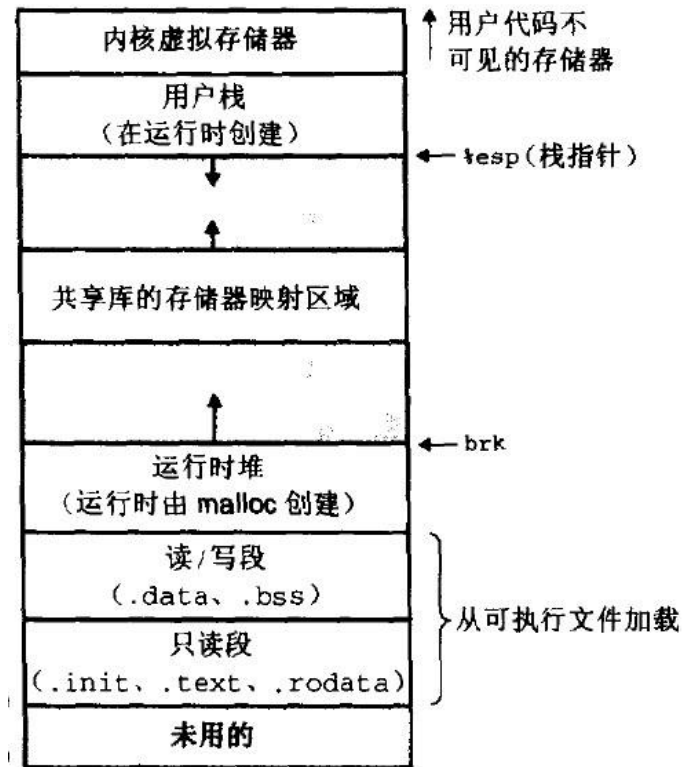
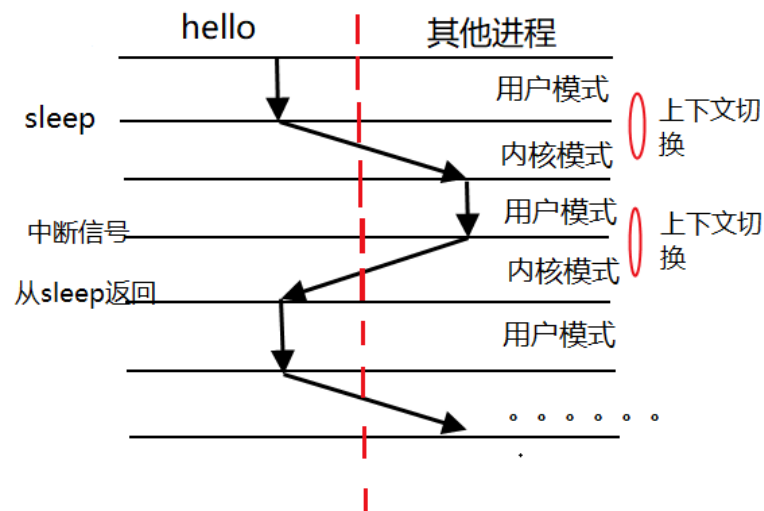


图 6-2 进程地址空间

6.5 Hello 的进程执行

图 6-3 `hello` 进程下辖区切换的剖析

操作系统内核使用一种称为上下文切换（context switch）的较高层形式的异常

控制流来实现多任务。内核为每个进程为耻以上下文（context）。上下文就是内核重新启动一个被强占的进程所需的状态。在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被强占了的进程。这种决策叫做调度（scheduling），是由内核中的被称为调度器的代码处理的。在内核调度了一个新的进程运行后，他就抢占当前的进程，并使用一种称为上下文切换的机制来讲控制转移到新的进程，上下文切换 1）保存当前进程的上下文，2）恢复某个先前被强占的进程被保存的上下文，3）将控制传递给这个新恢复的进程。

6.6 hello 的异常与信号处理

```
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$ ./hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$
```

图 6-4-1 正常运行

```
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$ ./hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
^Z
[1]+ 已停止 ./hello 1170300204 赵跃
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$
```

图 6-4-2 Ctrl-Z

```
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$ ./hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
^C
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$
```

图 6-4-3 Ctrl-C

```
1170300204zy@1170300204zy:~/桌面/hitics-2/大作业$ ps
  PID TTY          TIME CMD
 2390 pts/0        00:00:00 bash
 2444 pts/0        00:00:00 hello
 2447 pts/0        00:00:00 ps
```

图 6-4-4 ps

```

1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$ fg 1
./hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃

1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$ jobs
1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$

```

图 6-4-5 fg 和 jobs

```

1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$ ./hello 1170300204 赵跃
Hello 1170300204 赵跃
safsadjkHello 1170300204 赵跃
adsjakcHello 1170300204 赵跃
ads
Hello 1170300204 赵跃
davs

Hello 1170300204 赵跃
eHello 1170300204 赵跃
efef
Hello 1170300204 赵跃
Hello 1170300204 赵跃
Hello 1170300204 赵跃
EFewf
Hello 1170300204 赵跃
1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$ davs

Command 'davs' not found, did you mean:

  command 'dave' from deb libhttp-dav-perl
  command 'dav' from deb dav-text

Try: sudo apt install <deb name>

1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$
1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$
1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$ eefef
eefef: 未找到命令
1170300204zy@1170300204zy:~/桌面/hitcs-2/大作业$ EFewf
EFewf: 未找到命令

```

图 6-4-6 运行过程中乱按

图 6-4-7 pstree

- 1) 中断 SIGSTP 挂起程序
2) 终止 SIGINT 终止程序

现代系统能够对系统状态的变化作出反应。一般而言，我们把这些突变称为异常控制流（Exceptional Control Flow, ECF）。异常控制流发生在计算机系统的各个层次。作为程序员理解 ECF 很重要：理解 ECF 将帮助我们理解重要的系统概念；理解 ECF 将帮助我们理解应用程序是如何与操作系统交互的；理解 ECF 将帮助我们编写有趣的新应用程序；理解 ECF 将帮助我们理解并发；理解 ECF 将帮助我们理解软件异常是如何工作的。

(第 6 章 1 分)

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

物理地址 (physical address)：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。

虚拟内存(virtual memory)：这是对整个内存的抽象描述。它是相对于物理内存来讲的，可以直接理解成“不直实的”，“假的”内存，例如，一个 0x08000000 内存地址，它并不对应物理地址上那个大数组中 0x08000000 - 1 那个地址元素；之所以是这样，是因为现代操作系统都提供了一种内存管理的抽象，即虚拟内存。进程使用虚拟内存中的地址，由操作系统协助相关硬件，把它转换成真正的物理地址。

逻辑地址 (logical address)：逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址。

线性地址(linear address)：也叫虚拟地址(virtual address)，跟逻辑地址类似，它也是一个不真实的地址，如果逻辑地址是对应的硬件平台段式管理转换前地址的话，那么线性地址则对应了硬件页式内存的转换前地址。

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分组成，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。

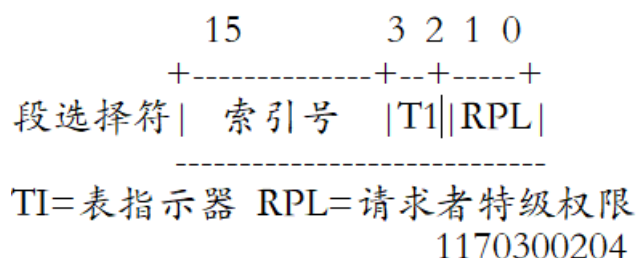


图 7-1 段选择符

其中 T1 有两种取值：=0 时，说明选择全局描述符表 (GDT)；=1 时，说明选择局部描述符表 (LDT)

RPL 有四个取值：00、01、10、11，分别表示第 1~4 级。

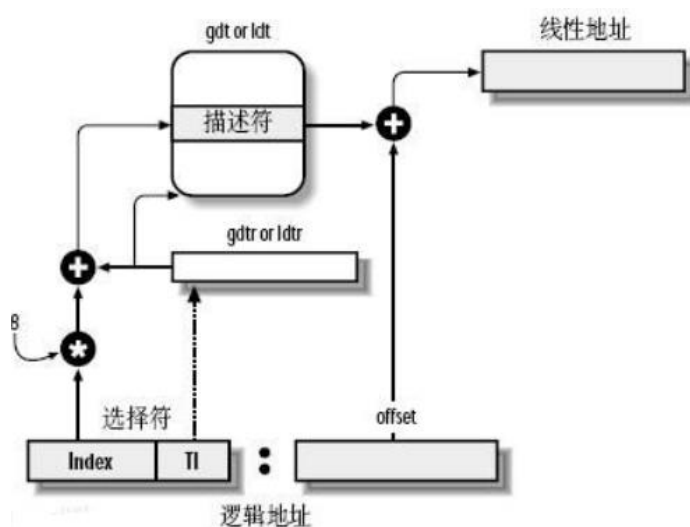


图 7-2 关系图

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，

1、看段选择符的 TI=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。

2、拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，即得到了基地址。

3、把 Base + offset，就是要转换的线性地址了。

7.3 Hello 的线性地址到物理地址的变换-页式管理

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

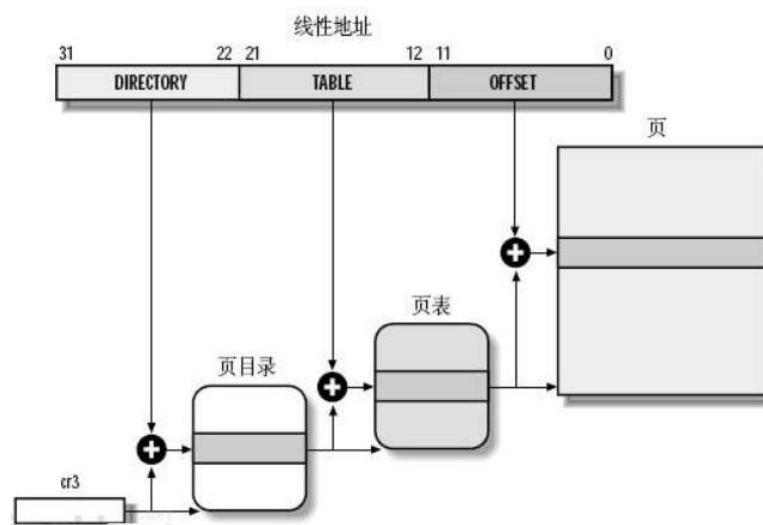


图 7-3 页式管理

分页单元中，页目录是唯一的，它的地址放在 CPU 的 cr3 寄存器中，是进行地址转换的开始点。每一个活动的进程，因为都有其独立的对应的虚拟内存（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，将别的保存下来。

每一个 32 位的线性地址被划分为三部份，面目录索引(10 位)：页表索引(10 位)：偏移(12 位) 并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。（参考了博客园中的内容 <https://www.cnblogs.com/zengkefu/p/5452792.html>）

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

给出了 Core i7 MMU 如何使用四级的页表来将虚拟地址翻译成物理地址。36 位 VPN 被划分成四个 9 位的片，每个片被用作到一个页表的偏移量。CR3 寄存器包含 L1 页表的物理地址。VPN 1 提供到一个 L1 PTE 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN 2 提供到一个 L2 PTE 的偏移量，以此类推。（图片和内容均来自 CSAPP 课本）

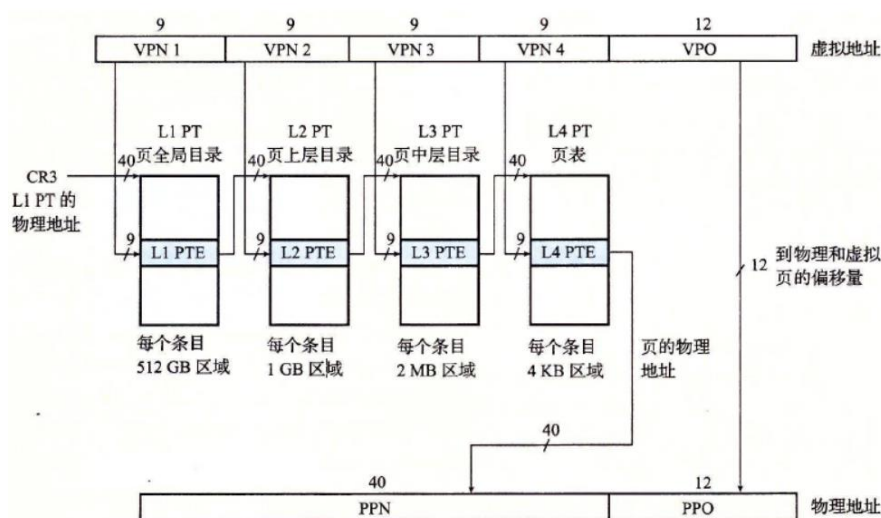


图 7-4 Core i7 页表翻译 (PT: 页表, PTE: 页表条目, VPN: 虚拟页号, VPO: 虚拟页偏移, PPN 物理页号, PPO: 物理页偏移量。图中还给出了这四级页表的 Linux 名字)

7.5 三级 Cache 支持下的物理内存访问

CPU 会首先发出一个虚拟地址给 TLB 进行搜索。如果地址命中则直接发送到 L1 中, 若未命中, 则会首先在页表里面寻找, 搜索到之后再发给 L1。进入 L1 之后, 对于物理地址的寻找仍需分析是否命中, 此时会通过一种称为 CPU 高速缓存机制把地址翻译阶段的性能提到较高的水平。

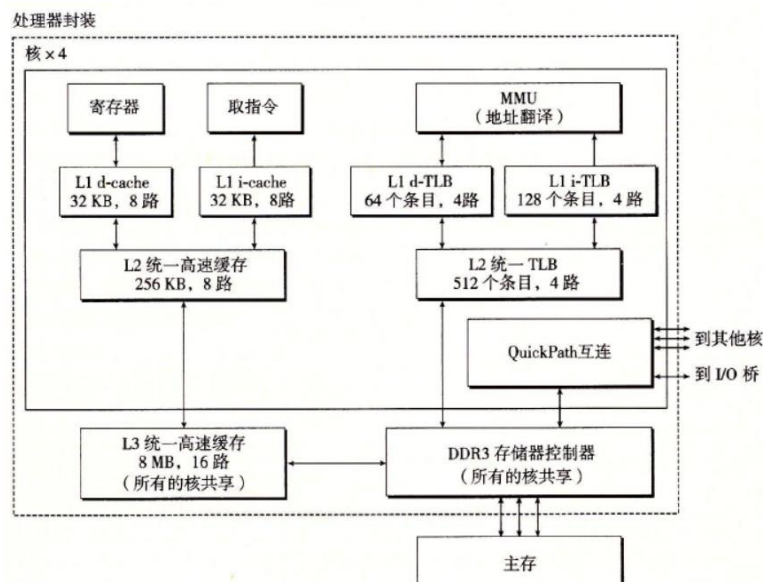


图 7-5 Core i7 的内存系统 (摘自 CSAPP 课本图 9-21)

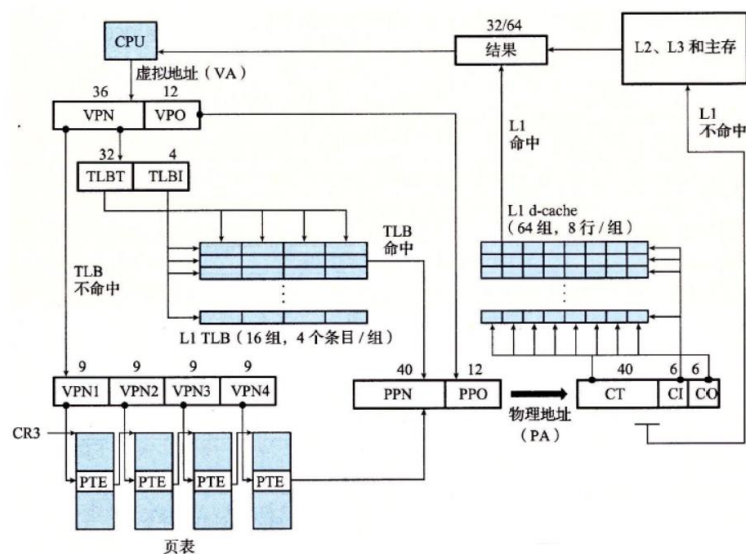


图 7-6 Core i7 地址翻译的概况。为了简化，没有显示 i-cache、i-TLB 和 L2 统一 TLB
(摘自 CSAPP 课本图 9-22)

7.6 hello 进程 fork 时的内存映射

当 fork 函数被 shell 调用时，内核为 hello 进程创建各种数据结构，并分配给它一个唯一的 PID。为了给 hello 进程创建虚拟内存，它创建了 hello 进程的 mm_struct、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

7.7 hello 进程 execve 时的内存映射

Execve 函数再当前进程的上下文中加载并运行一个新程序。

Execve 函数加载并运行可执行目标文件 filename，且带参数列表 argv 和环境变量 envp。之后又当出现错误时，例如找不到 filename，execve 才会返回到调用程序。所以，与 fork 一次调用两次返回不同，execve 调用依次从不返回。

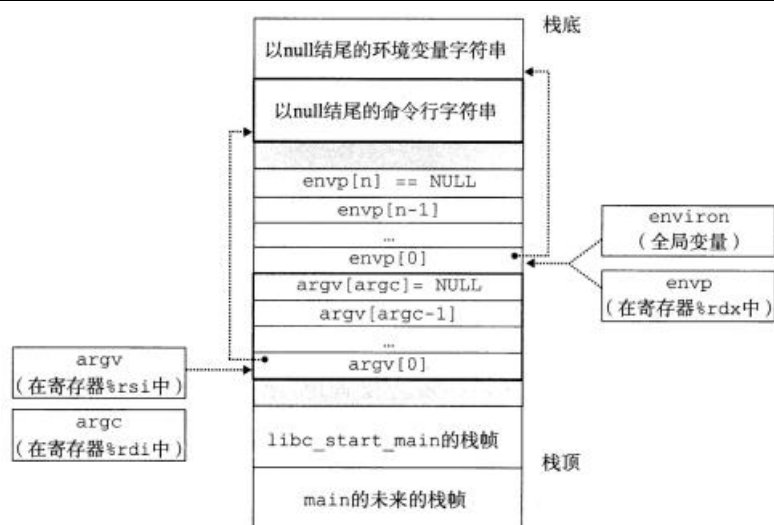


图 7-7 一个新程序开始时，用户栈的典型组织结构

7.8 缺页故障与缺页中断处理

缺页故障：进程线性地址空间里的页面不必常驻内存，在执行一条指令时，如果发现他要访问的页没有在内存中（即存在位为 0），那么停止该指令的执行，并产生一个页不存在的异常，对应的故障处理程序可通过从外存加载该页的方法来排除故障，之后，原先引起的异常的指令就可以继续执行，而不再产生异常。

产生缺页中断的几种情况：

1、当内存管理单元（MMU）中确实没有创建虚拟物理页映射关系，并且在该虚拟地址之后再没有当前进程的线性区（vma）的时候，可以肯定这是一个编码错误，这将杀掉该进程；

2、当 MMU 中确实没有创建虚拟页物理页映射关系，并且在该虚拟地址之后存在当前进程的线性区 vma 的时候，这很可能是缺页中断，并且可能是栈溢出导致的缺页中断；

3、当使用 malloc/mmap 等希望访问物理空间的库函数/系统调用后，由于 linux 并未真正给新创建的 vma 映射物理页，此时若先进行写操作，将和 2 产生缺页中断的情况一样；若先进行读操作虽然也会产生缺页异常，将被映射给默认的零页，等再进行写操作时，仍会产生缺页中断，这次必须分配 1 物理页了，进入写时复制的流程；

4、当使用 fork 等系统调用创建子进程时，子进程不论有无自己的 vma，它的 vma 都有对于物理页的映射，但它们共同映射的这些物理页属性为只读，即 linux 并未给予进程真正分配物理页，当父子进程任何一方要写相应物理页时，导致缺页中断的写时复制。

缺页中断处理的大致步骤：1) 保护 CPU 现场 2) 分析中断原因 3) 转入缺页处理中断函数 4) 恢复 CPU 现场，继续执行

7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆(heap)(见图 9-33)。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长(向更高的地址)。对于每个进程，内核维护着一个变量 brk(读做“break”)，它指向堆的顶部。

分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片(chunk)，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

图 7-8 教材中对于动态内存分配器的描述

内存分配算法：

首次适配算法(first fit)：存储管理器沿着内存段链表搜索直到找到一个足够大的空洞

下次适配(next fit)：每次找到合适的空洞时都记住当时的位置，在下次寻找空洞时从上次结束的地方开始搜索，而不是每次都从头开始

最佳适配算法(best fit)：试图找出最接近实际需要的大小的空洞，而不是把一个以后可能会用到的大空洞先使用，实际性能其实很差，因为会造成很多空洞。每次被调用时都要搜索整个链表，因此会比首次适配算法慢

最坏匹配算法(worst fit)：在每次分配时，总是将最大的那个空闲区切去一部分，分配给请求者；进程链表和空闲链表分离，这样可以加快链表的查找速度，但使得内存的回收变得更加复杂，速度更慢

快速匹配算法(quick fit)：为一些经常被用到长度的空洞设立单独的链表。快速适配算法寻找一个指定大小的空洞是十分迅速的，但在一个进程结束或被换出时寻找它的邻接块以查看是否可以合并是非常费时间的

隐式空闲链表：

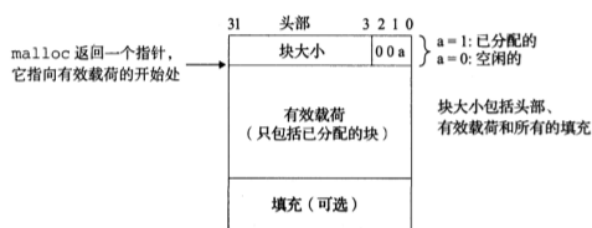


图 9-35 一个简单的堆块的格式

我们称这种结构为隐式空闲链表，是因为空闲块是通过头部中的大小字段隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。注意，我们需要某种特殊标记的结束块，在这个示例中，就是一个设置了已分配位而大小为零的终止头部(terminating header)。(就像我们将在 9.9.12 节中看到的，设置已分配位简化了空闲块的合并。)

隐式空闲链表的优点是简单。显著的缺点是任何操作的开销，例如放置分配的块，要求对空闲链表进行搜索，该搜索所需时间与堆中已分配块和空闲块的总数呈线性关系。

图 7-9 隐式空闲链表（摘自 CSAPP 课本）

显示空闲链表：

一种更好的方法是将会空闲块组织为某种形式的显式数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred(前驱)和 succ(后继)指针，如图 9-48 所示。

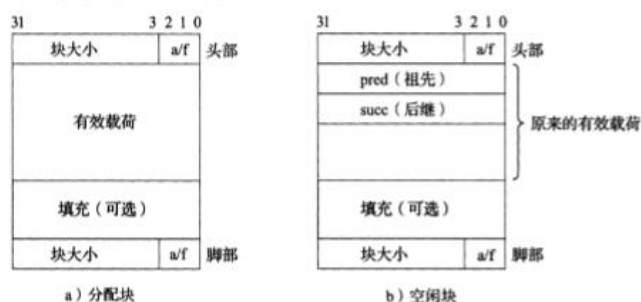


图 9-48 使用双向空闲链表的堆块的格式

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于我们所选择的空闲链表中块的排序策略。

图 7-10 显示空闲链表（摘自 CSAPP 课本）

7.10 本章小结

存储区是计算机结构中必不可少的一部分，每个用户程序都需要向操作系统申请存储资源。因此，程序的存储管理就显得尤为重要。本章中，我们分别了解分析了 intel 的段式管理和页式管理、三级 Cache 支持下的物理内存访问的相关内容。之后又观察了 hello 在 fork 和 execve 的内存映射。最后探讨了缺页故障和缺页异常处理及动态内存分配管理的相关内容。对于达成“深入理解计算机系统”的最终目标又近了一步。

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

设备的模型化：文件

设备管理：unix io 接口

一个 Linux 文件就是一个 m 个字节的序列： $B_1, B_2, B_3, \dots, B_{m-1}$

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。

8.2 简述 Unix IO 接口及其函数

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：

1) 打开文件 2) Linux shell 创建的每一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）3) 改变当前的文件位置 4) 读写文件 5) 关闭文件。

函数：

1.进程是通过调用 `open` 函数来打开一个已存在的文件或者创建一个新文件的：

```
int open(char *filename, int flags, mode_t mode);
```

返回：若成功则为新文件描述符，若出错为-1。

`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.进程通过调用 `close` 函数关闭一个打开的文件。

```
int close(int fd);
```

返回：若成功则为 0，若出错则为-1。

值得注意的是，关闭一个已经关闭的描述符会出错。

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


```
ssize_t read(int fd, void *buf, size_t n);
```

返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1。

read 函数从描述符为 fd 的当前文件位置复制最多 n 个字节到内存位置 buf。返回值-1 表示一个错误，而返回值 0 表示 EOF。否则，返回值表示的是实际传送的字节数量。

```
ssize_t write(int fd, const void *buf, size_t n);
```

返回：若成功则为写的字节数，若出错则为-1。

write 函数从内存位置 buf 复制至多 n 个字节到描述符 fd 的当前文件位置。

图 10-3 展示了一个程序使用 read 和 write 调用一次一个字节地从标准输入复制到标准输出。

8.3 printf 的实现分析

要想分析 printf 的实现，首先来观察一下 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;
}
```

图 8-1 printf 函数的函数体

可以看到

Printf 使用了可变形参，很显然，我们需要一种方法来让函数体可以知道具体调用时参数的个数。

观察这一行代码：va_list arg = (va_list)((char*)&fmt + 4);

va_list 的定义是 typedef char *va_list，这说明他是一个字符型指针。其中 (char*)&fmt + 4 标识的是第一个参数。之后一行的 i = vsprintf(buf, fmt, arg); 可以看到 vsprintf 返回的是打印出来的字符串的长度。vsprintf 的作用是格式化，它接受确定输出格式的格式字符串 fmt，用格式字符串对个数变化的参数进行格式化，产生格式化输出。

再之后的 `write` 函数实现的功能则是将 `buf` 中的 `i` 个元素写到终端中去。

由此我们可以大致得到 `printf` 的执行过程：从 `vsprintf` 生成显示信息，到 `write` 系统函数，到陷阱-系统调用 `int 0x80` 或 `syscall`。字符显示驱动子程序：从 ASCII 到字模库到显示 `vram`（存储每一个点的 RGB 颜色信息）。显示芯片按照刷新频率逐行读取 `vram`，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

参考资料地址：<https://www.cnblogs.com/pianist/p/3315801.html>

8.4 getchar 的实现分析

`getchar()` 函数的作用就是从终端输入一个字符，没有参数但是有一个 `int` 类型的返回值。一般的使用形式是 `char a = getchar();`

网络上百科中对 `getchar()` 的介绍是：`getchar` 由宏实现：`#define getchar()getc(stdin)`。`getchar` 有一个 `int` 型的返回值。当程序调用 `getchar` 时，程序就等着用户按键。用户输入的字符被存放在键盘缓冲区中。直到用户按回车为止（回车字符也放在缓冲区中）。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：键盘中断处理子程序。接受按键扫描码转成 `ascii` 码，保存到系统的键盘缓冲区。

`getchar` 等调用 `read` 系统函数，通过系统调用读取按键 `ascii` 码，直到接受到回车键才返回。

8.5 本章小结

如果从编程的角度去理解 IO，那么，IO 的主体就是其应用程序的运行态，即进程，特别强调的是我们的应用程序其实并不存在实质的 IO 过程，真正的 IO 过程是操作系统的事情。我们把应用程序的 IO 操作分为了两种动作：IO 调用和 IO 执行。IO 调用是由进程发起的，IO 执行时擦从左系统的工作。因此，更准确些来说，此时所说的 IO 时应用程序对操作系统 IO 功能的一次触发，即 IO 调用。

本章我们简单了解了 Linux 的 IO 设备管理方法和 Unix IO 接口及函数，之后又大致分析了 `printf` 和 `getchar` 的实现过程。

(第8章 1分)

结论

Hello 的一生是短暂的，但是短暂并不意味着渺小：

1. 编写：在键盘敲击声中诞生的.c 文件
2. 预处理：预处理器对 hello 进行预处理变成.i 文件
3. 编译：.i 文件被编译生成.s 文件
4. 汇编：.s 文件汇编形成可重定位目标文件.o 文件
5. 链接：链接器将.o 文件和相关的库等链接称为可执行文件 hello
6. 运行：在 shell 中键入运行指令，开始运行
7. 创建子进程：shell 壳使用 fork 为 hello 创建子进程
8. 运行程序：shell 壳使用 execve 加载并运行程序
9. 进程执行：CPU 给 hello 分配时间片，完成上下文切换等内容
10. 处理异常：对程序中的异常和信号进行处理
11. 访问内存和动态内存申请：hello 访问内存，并完成动态内存分配
12. 执行结束：hello 终止并且被 shell 回收

终于写完了大作业！！

回顾自己一个学期以来计算机系统课程的学习，从刚开始的初出茅庐的新鲜感，到之后头晕目眩的冗长期，到做实验加班加点忙到通宵，到写大作业呕心沥血，最后真的是感触颇多。

这门课程是我认为到目前以来所接受知识量最大的一门课。课程的学习不光需要看书记忆，更需要自己亲身的动手实践。亲身下河知深浅，亲口尝梨知酸甜，可以说如果不经过自己的思考和动手，那么学习这门课程的意义，或者说作用，就要大打折扣。计算机系统是一门庞杂繁复的学科，其中凝结了无数前人的智慧，当我们认真去发现、去钻研的时候就会频频惊叹于人类智慧的伟大。

Hello 虽然是一个非常简单的程序，但是纵观它的一生，从预处理到最后结束的整个生命周期，在整个系统的协同配合下又显得是那么的精彩。

虽然计算机系统的课程结束了，但是计算机系统的知识，仍然值得我们去反复学习，回味。

(结论 0 分，缺失 -1 分，根据内容酌情加分)

附件

hello.c	源文件
hello.i	预处理得到的文件
hello.s	编译后得到的文件
hello.o	汇编之后得到的文件
hello.txt	hello 的反汇编结果文件
hello.elf	hello.o 的 elf 格式文件
hello	链接之后得到的可执行文件

(附件 0 分，缺失 -1 分)

参考文献

为完成本次大作业你翻阅的书籍与网站等

- [1] 可执行文件（ELF）格式的理解
<https://www.cnblogs.com/LiuYanYGZ/p/5574602.html>
- [2] ELF 格式文件符号表全解析及 readelf 命令使用方法
<https://blog.csdn.net/edonlii/article/details/8779075>
- [3] 进程控制(2): 进程操作
<https://www.cnblogs.com/xiaomanon/p/4201006.html>
- [4] 线程和进程的基本概念 . 百度文库
<https://wenku.baidu.com/view/c8aaf49449649b6648d74776.html>
- [5] LINUX 逻辑地址、线性地址、物理地址和虚拟地址
<http://bbs.chinaunix.net/thread-2083672-1-1.html>
- [6] 缺页中断-----缺页中断处理（内核、用户）
https://blog.csdn.net/m0_37962600/article/details/81448553
- [7] printf 函数实现的深入剖析
<https://www.cnblogs.com/pianist/p/3315801.html>
- [8] 百度百科
- [9] 《深入理解计算机系统》

（参考文献 0 分，缺失 -1 分）