
哈爾濱工業大學

计算机系统

大作业

题 目 程序人生-Hello's P2P

专 业 计算机科学与技术

学 号 1170300905

班 级 1736101

学 生 沈金开

指 导 教 师 刘宏伟

计算机科学与技术学院

2018 年 12 月

摘 要

通过再次回忆起计算机系统的相关知识，让 Hello 演练完它传奇的一生，通过对各项实验实践能力的训练，大大加深对课本知识的理解，从而巩固本学期所学知识点，进而达到不负所学的目的，是对计算机系统的良好总结。

关键词：CSAPP；hello；预处理；编译；反汇编；链接；进程

（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.....	6 -
2.3 HELLO 的预处理结果解析.....	6 -
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.....	17 -
第 5 章 链接	18 -
5.1 链接的概念与作用.....	18 -
5.2 在 UBUNTU 下链接的命令.....	18 -
5.3 可执行目标文件 HELLO 的格式.....	18 -
5.4 HELLO 的虚拟地址空间.....	20 -
5.5 链接的重定位过程分析.....	20 -
5.6 HELLO 的执行流程.....	22 -
5.7 HELLO 的动态链接分析.....	22 -
5.8 本章小结.....	23 -
第 6 章 HELLO 进程管理	24 -
6.1 进程的概念与作用.....	24 -
6.2 简述壳 SHELL-BASH 的作用与处理流程.....	24 -
6.3 HELLO 的 FORK 进程创建过程.....	24 -

6.4 HELLO 的 EXECVE 过程	- 25 -
6.5 HELLO 的进程执行	- 25 -
6.6 HELLO 的异常与信号处理	- 26 -
6.7 本章小结	- 28 -
第 7 章 HELLO 的存储管理.....	- 29 -
7.1 HELLO 的存储器地址空间	- 29 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 29 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 30 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换	- 31 -
7.5 三级 CACHE 支持下的物理内存访问	- 32 -
7.6 HELLO 进程 FORK 时的内存映射	- 33 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 33 -
7.8 缺页故障与缺页中断处理	- 34 -
7.9 动态存储分配管理	- 35 -
7.10 本章小结	- 36 -
第 8 章 HELLO 的 IO 管理	- 37 -
8.1 LINUX 的 IO 设备管理方法	- 37 -
8.2 简述 UNIX IO 接口及其函数	- 37 -
8.3 PRINTF 的实现分析	- 37 -
8.4 GETCHAR 的实现分析	- 38 -
8.5 本章小结	- 39 -
结论	- 39 -
附件	- 40 -
参考文献	- 41 -

第 1 章 概述

1.1 Hello 简介

Hello 在 Linux 中的进化史:

P2P: 1.cpp 的预处理,

2.gcc 的编译

3.as 的汇编

4.ld 的链接

5.shell 的运行

6.fork 子进程

020: 7.execve

8.malloc

9.回收

1.2 环境与工具

硬件: CPU Intel Core i3 3110m 500GB HDD 4G RAM

软件环境: Microsoft Windows10 专业版 64 位; VMware Workstation 14 Pro;
Ubuntu 16.04

工具: cpp, vi, gcc, as, elf, edb。

1.3 中间结果

文件名	作用
hello.c.	大作业源程序
hello.i	经过预处理后的中间文件
hello.s	经过编译后的汇编文件
hello.o	经过汇编后的可重定位目标执行文件
hello	经过链接后的可执行程序
hello1.elf	hello.o 的 ELF
hello2.elf	hello 的 ELF

1.4 本章小结

介绍了 Hello 的 P2P 与 020 过程, 还有列出了大作业的各种环境及中间各结果

(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：

程序设计领域中，预处理一般是指在程序源代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。典型地，由预处理器对程序源代码文本进行处理，得到的结果再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，但它把源代码分割或处理成为特定的单位——（用 C/C++ 的术语来说是）预处理记号用来支持语言特性（如 C/C++ 的宏调用）。

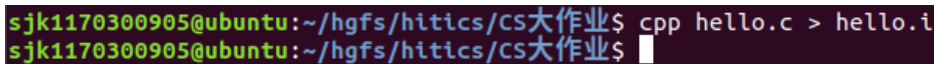
C/C++ 预处理（作用）：

最常见的预处理是 C 语言和 C++ 语言。ISO C 和 ISO C++ 都规定程序由源代码被翻译分为若干有序的阶段，通常前几个阶段由预处理器实现。预处理中会展开以 # 起始的行，试图解释为预处理指令，其中 ISO C/C++ 要求支持的包括 #if/#ifdef/#ifndef/#else/#elif/#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的 #（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

预处理器在 UNIX 传统中通常缩写为 PP，在自动构建脚本中 C 预处理器被缩写为 CPP 的宏指代。为了不造成歧义，C++(cee-plus-plus) 经常并不是缩写为 CPP，而改成 CXX。

注意预处理常被错误地当作预编译，事实上这是两个不同的概念。预处理尽管并不是 ISO C/C++ 要求的单独阶段，但“预处理”这个术语正式地出现并参与构成其它术语，如 C 的预处理翻译单元以及 C/C++ 词法规则中预处理记号这个语法分类。预编译是一些编译器支持的特性，不是 C/C++ 语言的特性或实现必须要求遵循的规则涉及到的内容，没有在 ISO C/C++ 全文中出现。

2.2 在 Ubuntu 下预处理的命令



```
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ cpp hello.c > hello.i
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$
```

图 1 预处理

2.3 Hello 的预处理结果解析

使用 vi 打开 hello.i 之后发现，整个 hello.i 程序已经拓展为 3118 行：

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello.c"

# 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
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
"hello.i" 3118L, 66102C 1,3 顶端
```

图 2hello.i

main 函数出现在 hello.c 中的代码自 3099 行开始。如下：

```
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1017 "/usr/include/stdlib.h" 2 3 4
# 1026 "/usr/include/stdlib.h" 3 4

# 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;
} 3118,1 底端
```

图 3hello.i

之前插入了大量代码，包含了相关函数的申明，如 `qsort`, `abs`, `labs`:

```
extern void qsort (void *__base, size_t __nmem, size_t __size,
    __compar_fn_t __compar) __attribute__((__nonnull__(1, 4)));
# 837 "/usr/include/stdlib.h" 3 4
extern int abs (int __x) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__));
extern long labs (long int __x) __attribute__((__nothrow__, __leaf__)) __attribute__((__const__));
```

图 4 函数

2.4 本章小结

本章主要介绍了预处理的定义与作用、解析了预处理，发现许多头文件的源代码形式。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

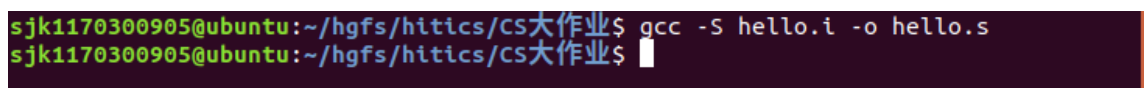
概念：

编译是将经过预处理的源代码文本文件 `hello.i` 翻译成汇编代码文本文件 `hello.s`。

作用：

将偏向于人的代码翻译成偏向于机器的代码，以便于之后机器指令的执行。

3.2 在 Ubuntu 下编译的命令



```
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ gcc -S hello.i -o hello.s
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$
```

图 5 编译

3.3 Hello 的编译结果解析

3.3.1. 编译如下：

```

.file    "hello.c"
.text
.globl   sleepsecs
.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2
.section      .rodata
.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"
.text
.globl   main
.type    main, @function
main:
.LFB5:
.cfi_startproc
pushq    %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq     %rsp, %rbp
.cfi_def_cfa_register 6
subq     $32, %rsp
movl     %edi, -20(%rbp)

```

图 6 编译结果

3.3.2.汇编指令:

指令	含义
.file	声明源文件
.text	以下是代码段
.globl	声明一个全局变量
.data	表明在.data 节
.section .rodata	以下是.rodata 节
.align	声明对指令或者数据的存放地址进行对齐的方式
.type	用来指定是函数类型或是对象类型
.size	声明大小
.long、.string	声明一个长整数、字符类型

3.3.3.int 整型:

1.sleepsecs:

```

.file    "hello.c"
.text
.globl   sleepsecs
.data
.align 4
.type    sleepsecs, @object
.size    sleepsecs, 4
sleepsecs:
.long    2

```

图 7 整型

Sleepsecs 一开始被 .globl 声明成为全局变量，后在 .data 中设置对齐方式为 4、设置类型为对象、设置大小为 4 字节，最后声明为值为 2 的 long int 整型变量。

2.i:

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

图 8 整型

I 作为局部变量声明，在栈中分配内存，只读段和读写数据段均不做声明。

3.3.4. 数组

文件中的数组有 char *argv[] main，函数执行时输入的命令行， argv 作为存放 char 指针的数组同时是第二个参数传入，使用了寄存器 %rbp 作为数组 argc[] 的栈指针：

```
.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)
```

图 9 数组

其中，%rbp-32 是 argc[1] 的地址，之后的地址依次加 8 即可。

3.3.5. 字符串

```
.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 10 字符串

Printf 函数的命令行格式串，首先声明在 .rodata 节中，再声明类型为 string。

3.3.6. 赋值

```
int sleepsecs=2.5;

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

图 11 赋值

1.int sleepsecs=2.5 : sleepsecs 是全局变量，直接在 .data 节中将 sleepsecs 声明为值 2 的 long 类型数据。

2.i=0: 整型数据的赋值使用 mov 指令完成，在栈或寄存器中分配存储空间，根据数据的大小不同使用不同后缀。

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

图 12 赋值 i

3.3.7. 条件转移:

通过比较 `cmpl` 与跳转 `je` 来实现 `argc!=3`, 然后转移

```
    cmpl    $3, -20(%rbp)
    je      .L2
    leaq    .LC0(%rip), %rdi
```

图 13 条件转移

3.3.8. 循环

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

图 14 源代码循环

```
.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)

.L3:
    cmpl    $9, -4(%rbp)
    jle     .L4
    call    getchar@PLT
    movl    $0, %eax
    leave
```

图 15 汇编循环

先通过赋值给 `i` 初值 0, 当 `i<10` 时反复执行.L4, 当 `i=10` 则跳出, 执行后面的语句。

3.3.9. 函数:

1.main 函数

```
.LC1:
    .string "Hello %s %s\n"
    .text
    .globl main
    .type   main, @function

main:
```

图 16main 函数

Main 符号首先声明在 .text 节中, 再声明为 global 变量, 类型为函数类型。argc 和 argv[] 构造好后, 传入 `execve` 做参数, 调用加载器, 上下文运行, 所以 main 的栈

帧上方就是 `argc` 和 `argv[]`。

2. `printf` 函数:

其中寄存器 `%rsi` 存放 `argv[1]`, `%rdx` 存放 `argv[2]`。

```
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
```

图 17 `printf` 函数

3. `Sleep` 函数:

设置 `edi` 为 `sleepsecs`。

```
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep@PLT
```

图 18 `sleep` 函数

4. `exit` 函数:

```
movl    $1, %edi
call    exit@PLT
```

图 19 `exit` 函数

`exit(x)` 则将 `x` 放在 `rdi` 中，再调用 `exit` 函数。

5. `getchar` 函数:

```
call    getchar@PLT
```

图 20 `getchar` 函数

直接调用 `getchar`。

3.4 本章小结

编译器将 `.i` 的拓展程序编译为 `.s` 的汇编代码。经过编译之后，`hello` 自 C 语言变为更加低级的汇编语言。

通过一步一步的详细分析 `hello.c` 的汇编语言，深刻理解了这里的各个变量、条件控制及函数的调用过程。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

概念：

汇编程序（as）对汇编语言源程序进行汇编，生成一个扩展名为.o 的可重定位目标二进制文件。

作用：将偏向于机器编译生成的汇编语言代码翻译为以二进制为基础的机器语言代码，以便于之后链接。

4.2 在 Ubuntu 下汇编的命令

```
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ as hello.s -o hello.o
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$
```

图 21 汇编

4.3 可重定位目标 elf 格式

1. ELF 头部表：以 16 B 序列开始，包含了该文件的大小与字节顺序 其余为各种信息，如 ELF 头的大小、目标文件的类型、机器类型、字节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量等信息。

```
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ readelf -a hello.o
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:               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
```

图 22hello.o 的 ELF

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。

[号]	名称 大小	类型 全体大小	地址 旗标	链接 链接	信息	偏移量 对齐
[0]	0000000000000000	NULL	0000000000000000		0	0
[1]	.text	PROGBITS	0000000000000000		0	0
[2]	.rela.text	RELA	0000000000000000		1	8
[3]	.data	PROGBITS	0000000000000000		0	0
[4]	.bss	NOBITS	0000000000000000		0	0
[5]	.rodata	PROGBITS	0000000000000000		0	0
[6]	.comment	PROGBITS	0000000000000000		0	0
[7]	.note.GNU-stack	PROGBITS	0000000000000000		0	0
[8]	.eh_frame	PROGBITS	0000000000000000		0	0
[9]	.rela.eh_frame	RELA	0000000000000000		8	8
[10]	.symtab	SYMTAB	0000000000000000		9	8
[11]	.strtab	STRTAB	0000000000000000		0	0
[12]	.shstrtab	STRTAB	0000000000000000		0	0

图 23 节头部表

3.rela.text: .text 节相关的可重定位信息。当链接器将某个目标文件和其他目标文件组合时，.text 节中的代码被合并后，调用外部函数或者引用全局变量的指令中的地址字段需要修改。.rela.text 节中记录了各个函数名和变量名对应的偏移、值、类型、符号值等信息。

```
There is no dynamic section in this file.

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

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
000000000020  000200000002 R_X86_64_PC32  0000000000000000 .text + 0

The decoding of unwind sections for machine type Advanced Micro Devices X86-64
is not currently supported.
```

图 24 重定位表

4.4 Hello.o 的结果解析

```

sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ objdump -d -r hello.o
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 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 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 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 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
80: c3                      retq

```

图 25 反汇编

与 hello.s 有很多的相似之处，但还是有不少的差别：

1. 分支转移：

```

13: 74 16                  je      2b <main+0x2b>

```

图 26 分支转移

在转移时使用的是已经确定的地址，而不是.l1,.l2 之类的段名称操作。

2. 调用函数：

```
59:  e8 00 00 00 00      callq 5e <main+0x5e>
                          5a: R_X86_64_PLT32      printf-0x4
```

图 27 调用函数

在 callq 时是 call offset 的地址方式，而不是 hello.s 里面的 call symbol。

3. 全局变量：

没有明显的全局变量信息，采用了是 offset(%rip)的形式，而非 hello.s 使用的 symbol(%rip)

```
4d:  48 8d 3d 00 00 00 00      lea 0x0(%rip),%rdi      # 54 <main+0x54>
                          50: R_X86_64_PC32      .rodata+0x1a
```

图 28 全局变量

4.5 本章小结

列出 hello.o 的 ELF，了解了每个表中各种类型的相关信息，通过比较 hello.o 的反汇编与 hello.s 的异同更加深刻的理解了各个函数与全局变量的重定位。

(第 4 章 1 分)

第 5 章 链接

5.1 链接的概念与作用

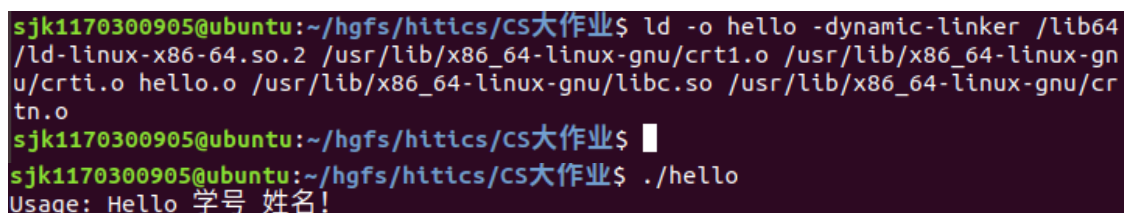
概念：

链接是将各种代码和数据片段收集并组合成一个单一可执行文件的过程。链接可以执行于编译时，即在源代码被编译成机器代码时；也可以执行于加载时，即在程序被加载器加载到内存并执行时；甚至可执行于运行时，即由应用程序来执行。

作用：

链接器使得分离编译成为可能。使得代码成为可以单独运行的可执行文件。

5.2 在 Ubuntu 下链接的命令



```
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ 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-gn
u/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/cr
tn.o
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ █
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ ./hello
Usage: Hello 学号 姓名!
```

图 29 链接

5.3 可执行目标文件 hello 的格式

```

sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ readelf -a hello
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:      EXEC (可执行文件)
  系统架构:  Advanced Micro Devices X86-64
  版本:      0x1
  入口点地址: 0x400500
  程序头起点: 64 (bytes into file)
  Start of section headers: 5928 (bytes into file)
  标志:      0x0
  本头的大小: 64 (字节)
  程序头大小: 56 (字节)
  Number of program headers: 8
  节头大小:  64 (字节)
  节头数量:  25
  字符串表索引节头: 24

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]      NULL      NULL      0000000000000000 00000000
     0000000000000000 0000000000000000      0      0      0
[ 1] .interp      PROGBITS      0000000000400200 00000200
     000000000000001c 0000000000000000      A      0      0      1
[ 2] .note.ABI-tag NOTE      000000000040021c 0000021c
     0000000000000020 0000000000000000      A      0      0      4
[ 3] .hash      HASH      0000000000400240 00000240
     0000000000000034 0000000000000004      A      5      0      8
[ 4] .gnu.hash      GNU_HASH      0000000000400278 00000278
     000000000000001c 0000000000000000      A      5      0      8
[ 5] .dynsym      DYNYSYM      0000000000400298 00000298
     00000000000000c0 0000000000000018      A      6      1      8
[ 6] .dynstr      STRTAB      0000000000400358 00000358
     0000000000000057 0000000000000000      A      0      0      1
[ 7] .gnu.version      VERSYM      00000000004003b0 000003b0
     0000000000000010 0000000000000002      A      5      0      2
[ 8] .gnu.version_r      VERNEED      00000000004003c0 000003c0
     0000000000000020 0000000000000000      A      6      1      8
[ 9] .rela.dyn      RELA      00000000004003e0 000003e0
     0000000000000030 0000000000000018      A      5      0      8
[10] .rela.plt      RELA      0000000000400410 00000410
     0000000000000078 0000000000000018      AI     5     19      8
[11] .init      PROGBITS      0000000000400488 00000488
     0000000000000017 0000000000000000      AX     0      0      4
[12] .plt      PROGBITS      00000000004004a0 000004a0
     0000000000000060 0000000000000010      AX     0      0     16
[13] .text      PROGBITS      0000000000400500 00000500
     0000000000000132 0000000000000000      AX     0      0     16
[14] .fini      PROGBITS      0000000000400634 00000634

```

	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

图 30hello 的 ELF

与 4.3 的操作和分析基本一样，不再多说。

5.4 hello 的虚拟地址空间

用 edb 打开 hello 可执行文件，可以看到在 Data Dump 中，自虚拟地址 0x400000 开始，到 0x400fff 结束，这之间每个节（.interp~.eh_frame 节）的排列（开始结束）即 Address 中声明。

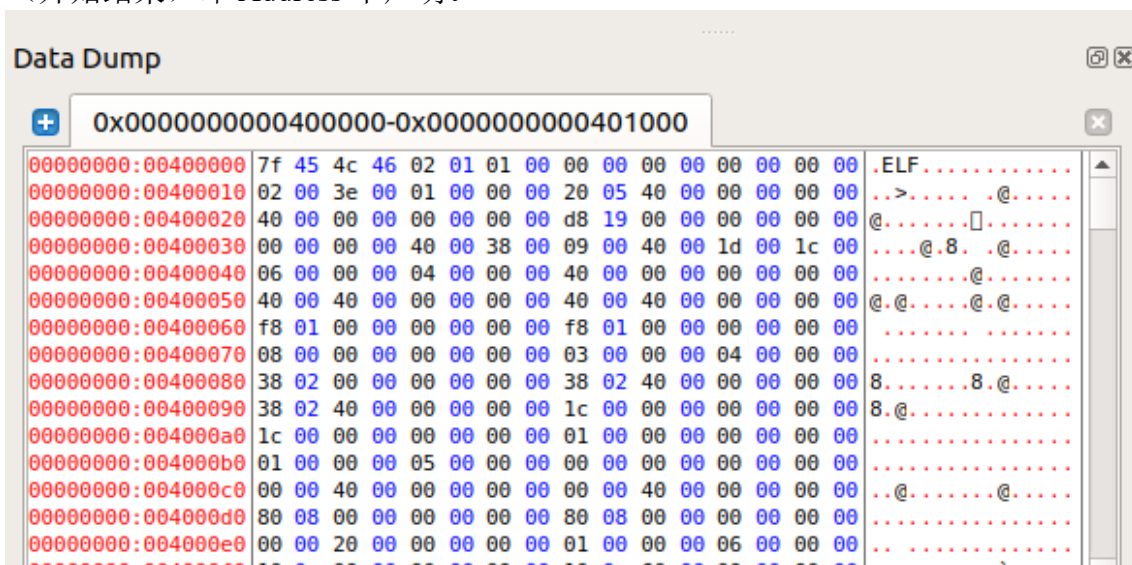


图 31hello 的 EDB

再通过 Data Dump 查看 0x400000 虚拟地址段 0x600000~0x602000，发现在 0~fff 空间中，与 0x400000~0x401000 段的存放的程序相同，在 fff 之后存放的是 .dynamic~.shstrtab 节。

5.5 链接的重定位过程分析

```

sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ objdump -d -r hello

hello:          文件格式 elf64-x86-64

Disassembly of section .init:

0000000000400488 <_init>:
 400488:    48 83 ec 08          sub    $0x8,%rsp
 40048c:    48 8b 05 65 0b 20 00  mov    0x200b65(%rip),%rax        # 600
ff8 <__gmon_start__>
 400493:    48 85 c0            test   %rax,%rax
 400496:    74 02              je     40049a <_init+0x12>
 400498:    ff d0             callq  *%rax
 40049a:    48 83 c4 08          add    $0x8,%rsp
 40049e:    c3                retq

Disassembly of section .plt:

00000000004004a0 <.plt>:
 4004a0:    ff 35 62 0b 20 00    pushq 0x200b62(%rip)        # 601008 <
_GLOBAL_OFFSET_TABLE_+0x8>
 4004a6:    ff 25 64 0b 20 00    jmpq   *0x200b64(%rip)        # 601010
<_GLOBAL_OFFSET_TABLE_+0x10>
 4004ac:    0f 1f 40 00          nopl   0x0(%rax)

```

图 32 反汇编

1. 出了许多函数，例如 .init, .plt, .fini 等：

```

0000000000400488 <_init>:
 400488:    48 83 ec 08          sub    $0x8,%rsp
 40048c:    48 8b 05 65 0b 20 00  mov    0x200b65(%rip),%rax        # 600
ff8 <__gmon_start__>
00000000004004a0 <.plt>:
 4004a0:    ff 35 62 0b 20 00    pushq 0x200b62(%rip)        # 601008 <
_GLOBAL_OFFSET_TABLE_+0x8>
 4004a6:    ff 25 64 0b 20 00    jmpq   *0x200b64(%rip)        # 601010
<_GLOBAL_OFFSET_TABLE_+0x10>
 4004ac:    0f 1f 40 00          nopl   0x0(%rax)

0000000000400634 <_fini>:
 400634:    48 83 ec 08          sub    $0x8,%rsp
 400638:    48 83 c4 08          add    $0x8,%rsp
 40063c:    c3                retq

```

图 33 函数

2. 函数调用：直接跳转到下一条指令的地址，例如 getchar 函数：

```

00000000004004d0 <getchar@plt>:
 4004d0:    ff 25 52 0b 20 00    jmpq   *0x200b52(%rip)        # 601028
<getchar@GLIBC_2.2.5>
 4005a7:    e8 24 ff ff ff      callq   4004d0 <getchar@plt>

```

图 34 函数调用

链接器解析重定条目时会发现对外部函数调用的类型为 R_X86_64_PLT32 的重定位，动态链接库中的函数加入了 PLT 中，.text 与 .plt 节相对距离也已经确定了，此时链接器要计算相对距离，会对动态链接库中的函数的调用值改为 PLT 中相应函数与下条指令的地址，即对应函数。对于此类重定位链接器为其构造 .plt

与.got.plt。

4. rodata 引用:

重定位记录分为 PC 相对地址和绝对地址的引用, 根据.rela.text 和.rela.data 中的重定位记录, 在.symtab 中寻找到需要修改记录的符号, 并根据符号与位置信息对目标进行修改。若是本地符号, 计算偏移量并修改目标位置; 否则创建.got 表。

5.6 hello 的执行流程

使用 edb 执行 hello, 说明从加载 hello 到_start, 到 call main, 以及程序终止的过程。列出其调用与跳转的各个子程序名称或程序地址。

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
hello!main	0x400532
hello!puts@plt	0x4004b0
hello!exit@plt	0x4004e0
libc-2.27.so!exit	0x7fce 8c889128

5.6 Hello 的动态链接分析

动态链接过程分成两步: 首先, 进行静态链接以生成部分链接的可执行目标文件 hello, 该文件中仅包含共享库中的符号表和重定位表信息, 而共享库中的代码和数据并没有被合并到 hello 中, 这时引进链接表 PLT+全局偏移量表 GOT 实现函数的动态链接, 每个被这个目标模块引用的全局数据目标都有一个 8 字节条目, 编译器还为 GOT 中每个条目生成一个重定位记录; 然后, 在加载 hello 时, 由加载器将控制权转移到指定的动态链接器, 由动态链接器代码和数据进行重定位并加载共享库, 以生成最终的存储空间中完全链接的可执行目标, 在完成重定位和加载共享库后, 动态链接器把控制权转移到程序 hello。

在 dl_init 调用之前, PIC 函数调用的目标地址都实际指向 PLT 中的代码逻辑, PLT 中函数调用指令的下一条指令地址存放在 GOT。

```

00000000:00600ff8 00 00 00 00 00 00 00 00 28 0e 60 00 00 00 00 00 .....(.....
00000000:00601008 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00601018 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 .....@.....@.....
00000000:00601028 06 05 40 00 00 00 00 00 16 05 40 00 00 00 00 00 .....@.....@.....
00000000:00601038 26 05 40 00 00 00 00 00 36 05 40 00 00 00 00 00 .....6.....

```

图 35 指令地址

之后跳转到这个地址，发现正式动态链接库的入口地址。

```

00000000:00601010 00 00 00 00 00 00 00 00 e6 04 40 00 00 00 00 00 .....H.@..
00000000:00601020 f6 04 40 00 00 00 00 00 06 05 40 00 00 00 00 00 .....@.@..
00000000:00601030 16 05 40 00 00 00 00 00 26 05 40 00 00 00 00 00 .....&.@..
00000000:00601040 36 05 40 00 00 00 00 00 00 00 00 00 00 00 00 00 6.@.....
00000000:00601050 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 .....

```

图 36 指令地址

最后函数调用则跳转到 PLT，执行 GOT 地址的下一条指令，函数和重定位表地址反复压栈，接着访问动态链接器，得到函数运行时地址，重写 GOT，再将控制传递给目标函数。

5.7 本章小结

根据 hello.elf 研究了 hello 的链接全过程，深刻分析了虚拟地址，重定位，执行流程与动态链接的详细过程。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：

进程的概念主要有两点：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、数据区域和堆栈。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。

作用：

操作系统引入进程的概念的原因：

从理论角度看，是对正在运行的程序过程的抽象；

从实现角度看，是一种数据结构，目的在于清晰地刻画动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。

6.2 简述 Shell-bash 的作用与处理流程

作用：

在计算机科学中，Shell 俗称壳（用来区别于核），是指“为使用者提供操作界面”的软件（命令解析器）。它类似于 DOS 下的 `command.com` 和后来的 `cmd.exe`。它接收用户命令，然后调用相应的应用程序。同时它又是一种程序设计语言。作为命令语言，它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

处理流程：

- (1) 从终端读入输入的命令。
- (2) 将输入字符串切分获得所有的参数
- (3) 如果是内置命令则立即执行
- (4) 否则调用相应的程序执行
- (5) shell 应该接受键盘输入信号，并对这些信号进行相应处理。

6.3 Hello 的 fork 进程创建过程

父进程中在调用 `fork()` 派生新进程，实际上相当于创建了一个进程的拷贝；即在 `fork()` 之前的进程拥有的资源会被复制到新的进程中去。网络服务器在处理并发请求时，也可以采取这种派生新进程的方式：父进程调用 `accept()` 后调用 `fork()` 来处理每一个连接。那么，所接受的已连接的套接口随后就在父子进程中共享。通常来说，子进程会在这连接套接口中读和写操作，父进程则关闭这个已连的套接口。

`fork()`有两个典型用法：(1)一个进程进行自身的复制，这样每个副本可以独立的完成具体的操作，在多核处理器中可以并行处理数据。这也是网络服务器的其中一个典型用途，多进程处理多连接请求。(2)一个进程想执行另一个程序。比如一个软件包含了两个程序，主程序想调起另一个程序的话，它就可以先调用 `fork` 来创建一个自身的拷贝，然后通过 `exeve` 函数来替换成将要运行的新程序。

6.4 Hello 的 `execve` 过程

前面提到了, `fork`,等复制出来的进程是父进程的一个副本,我们想加载新的程序,可以通过 `execve` 来加载和启动新的程序。

内核中实际执行 `execv()`或 `execve()`系统调用的程序是 `do_execve()`, 这个函数先打开目标映像文件, 并从目标文件的头部读入若干字节, 然后调用另一个函数 `search_binary_handler()`, 在此函数里面, 它会搜索 Linux 支持的可执行文件类型队列, 让各种可执行程序的处理程序前来认领和处理。如果类型匹配, 则调用 `load_binary` 函数指针所指向的处理函数来处理目标映像文件。

通过参数传递了寄存集合和可执行文件的名称(filename), 而且还传递了指向了程序的参数 `argv` 和环境变量 `envp` 的指针

6.5 Hello 的进程执行

进程上下文信息:

用户空间的应用程序, 通过系统调用, 进入内核空间。这个时候用户空间的进程要传递很多变量、参数的值给内核, 内核态运行的时候也要保存用户进程的一些寄存器值、变量等。所谓的“进程上下文”, 可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。当发生进程调度时, 进行进程切换就是上下文切换。操作系统必须对上面提到的全部信息进行切换, 新调度的进程才能运行。而系统调用进行的模式切换。模式切换与进程切换比较起来, 容易很多, 而且节省时间, 因为模式切换最主要的任务只是切换进程寄存器上下文的切换。

进程时间片:

连续执行同一个进程的时间段称为时间片。

用户态与核心态转换:

(1) 当一个任务(进程)执行系统调用而陷入内核代码中执行时, 称进程处于内核运行态(内核态)。此时处理器处于特权级最高的(0级)内核代码中执行。当进程处于内核态时, 执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

(2) 当进程在执行用户自己的代码时, 则称其处于用户运行态(用户态)。此时处理器在特权级最低的(3级)用户代码中运行。当正在执行用户程序而突然

被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。

进程调度：

用户进程数进程调度一般都多于处理机数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用处理机。

无论是在批处理系统还是分时系统中，用户进程数一般都多于处理机数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用处理机。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程，以使之执行。

引起进程调度的原因有以下几类，

- (1)正在执行的进程执行完毕。这时，如果不选择新的就绪进程执行，将浪费处理机资源。
- (2)执行中进程自己调用阻塞原语将自己阻塞起来进入睡眠等状态。
- (3)执行中进程调用了 P 原语操作，从而因资源不足而被阻塞；或调用了 v 原语操作激活了等待资源的进程队列。
- (4)执行中进程提出 I/O 请求后被阻塞。
- (5)在分时系统中时间片已经用完。
- (6)在执行完系统调用等系统程序后返回用户进程时，这时可看作系统进程执行完毕，从而可调度选择一新的用户进程执行。

以上都是在可剥夺方式下的引起进程调度的原因。在 CPU 执行方式是可剥夺时，还有

- (7)就绪队列中的某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

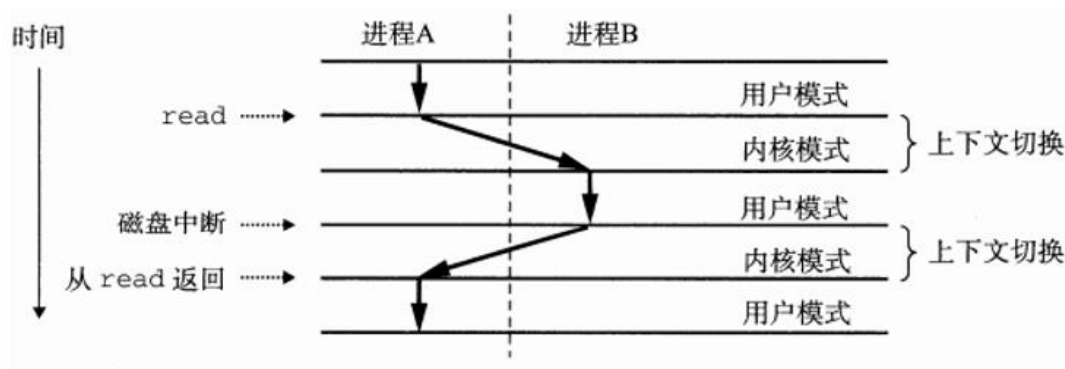


图 37 上下文处理

6.6 hello 的异常与信号处理

- 1.故障：缺页异常，hello 进程的页表被映射到 hello 文件，然而实际代码拷贝至内存 仍未完成，在执行到相应地址的代码时会引发缺页异常。
- 2.终止：不可恢复错误发生。
- 3.中断：接受到键盘键入的信号，如 ctrl Z ,ctrl C 等

```
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ ./hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai

Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
^Z
[1]+  已停止                  ./hello 1170300905 shenjinkai
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ ./hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
Hello 1170300905 shenjinkai
^C
sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ ps
  PID TTY          TIME CMD
  4271 pts/0        00:00:00 bash
  4986 pts/0        00:00:00 hello
  4988 pts/0        00:00:00 hello
  4989 pts/0        00:00:00 ps

sjk1170300905@ubuntu:~/hgfs/hitcs/CS大作业$ jobs
[1]-  已停止                  ./hello 1170300905 shenjinkai
[2]+  已停止                  ./hello 1170300905 shenjinkai
```

6.7 本章小结

(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

结合 hello 说明逻辑地址、线性地址、虚拟地址、物理地址的概念。

逻辑地址：逻辑地址是指由程序产生的与段相关的偏移地址部分。

线性地址：线性地址是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。

物理地址：放在寻址总线上的地址。放在寻址总线上，如果是读，电路根据这个地址每位的值就将相应地址的物理内存中的数据放到数据总线中传输。如果是写，电路根据这个地址每位的值就将相应地址的物理内存中放入数据总线上的内容。

虚拟地址：CPU 启动保护模式后，程序运行在虚拟地址空间中。注意，并不是所有的“程序”都是运行在虚拟地址中。CPU 在启动的时候是运行在实模式的，内核在初始化页表之前并不使用虚拟地址，而是直接使用物理地址的。。物理内存是以字节(8 位)为单位编址的。

7.2 Intel 逻辑地址到线性地址的变换-段式管理

逻辑地址转换为线性地址的过程可以描述如下：

- 1、CPU 根据进程所处的状态将上述的选择描述符装入到那四个 16 位段寄存器中的一个；
- 2、然后将逻辑地址的偏移量装入到 8 个 32 位的通用寄存器中的某一个中；
- 3、然后 MMU 中的分段部件开始对逻辑地址进行处理；
- 4、首先根据选择描述符中的前 13 位得到段描述符的索引，根据这个索引可以得到对应的段描述符；
- 5、找到段描述符之后，这几乎等于是找到了所有相应段的信息，当然我们可以提取出段的基地址，有了这个基地址，再加上逻辑地址的偏移量就得到了新的地址，这就是线性地址。

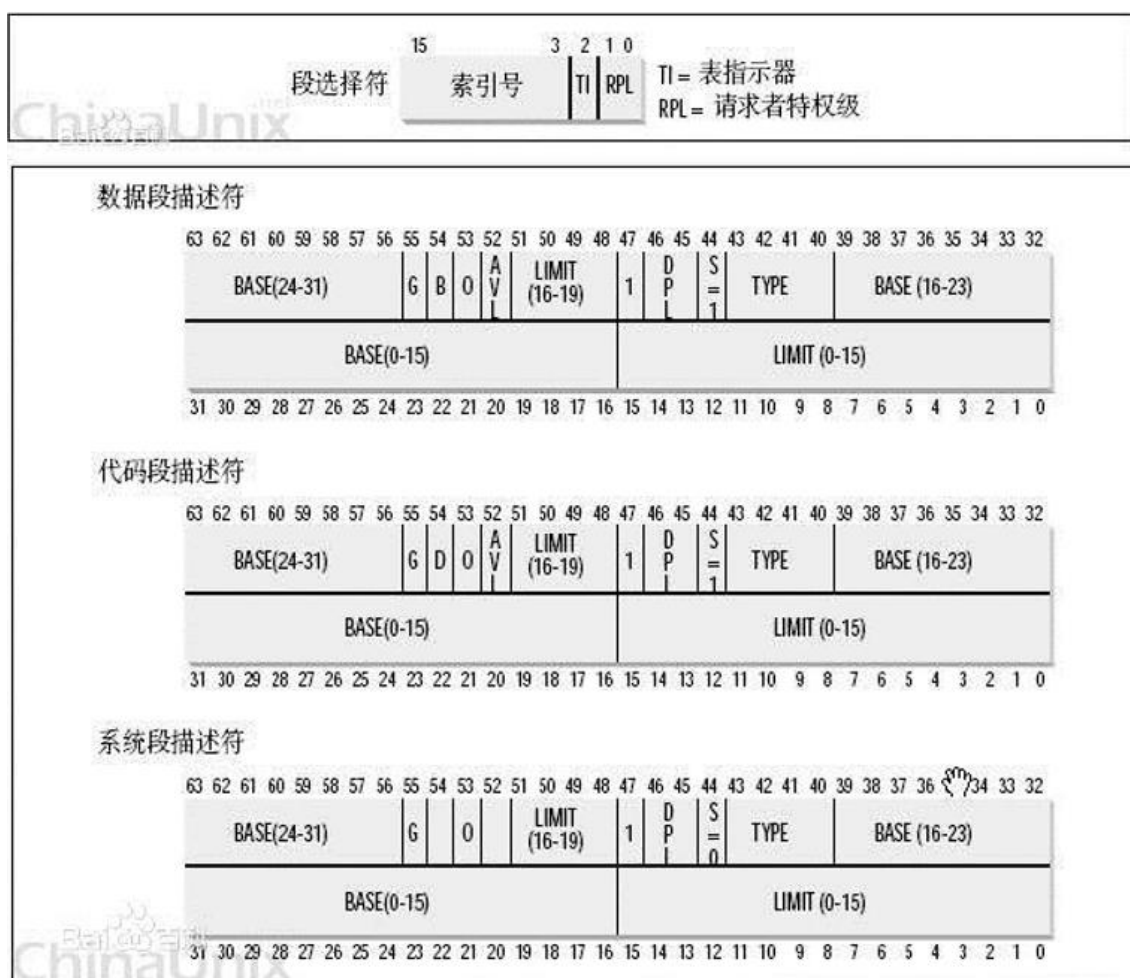


图 39 段描述

7.3 Hello 的线性地址到物理地址的变换-页式管理

分页的基本原理是把内存划分成大小固定的若干单元，每个单元称为一页，每页包含 4k 字节的地址空间。这样每一页的起始地址都是 4k 字节对齐的。为了能转换成物理地址，我们需要给 CPU 提供当前任务的线性地址转物理地址的查找表，即页表。注意，为了实现每个任务的平坦的虚拟内存，每个任务都有自己的页目录表和页表。

每个活动的任务，必须要先分配给它一个页目录表，并把页目录表的物理地址存入 cr3 寄存器。页表可以提前分配好，也可以在用到的时候再分配。

Linux 中逻辑地址等于线性地址，我们要转换的是线性地址。转换的过程是由 CPU 自动完成的，Linux 所要做的就是准备好转换所需的页目录表和页表。

当然进程在这个过程中是被蒙蔽的，它自己的感觉还是正常访问到了物理内存。

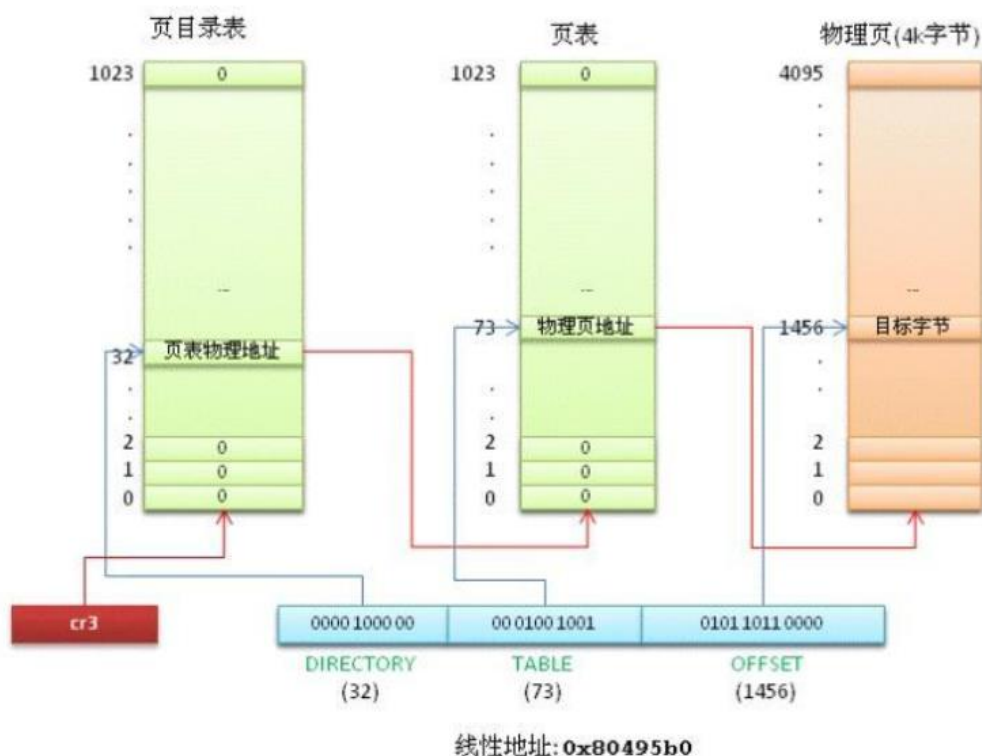


图 40 线性地址转物理地址

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

VA: virtual address 称为虚拟地址, PA: physical address 称为物理地址。

VA 到 PA 的映射过程:

首先将 CPU 内核发送过来的 32 位 VA[31:0]分成三段, 前两段 VA[31:20]和 VA[19:12]作为两次查表的索引, 第三段 VA[11:0]作为页内的偏移, 查表的步骤如下:

(1)从协处理器 CP15 的寄存器 2(TTB 寄存器, translation table base register)中取出保存在其中的第一级页表(translation table)的基地址, 这个基地址指的是 PA, 也就是说页表是直接按照这个地址保存在物理内存中的。

(2)以 TTB 中的内容为基地址, 以 VA[31:20]为索引值在一级页表中查找出一项($2^{12}=4096$ 项), 这个页表项(也称为一个描述符, descriptor)保存着第二级页表(coarse page table)的基地址, 这同样是物理地址, 也就是说第二级页表也是直接按这个地址存储在物理内存中的。

(3)以 VA[19:12]为索引值在第二级页表中查出一项($2^8=256$), 这个表项中就保存着物理页面的基地址, 我们知道虚拟内存管理是以页为单位的, 一

个虚拟内存的页映射到一个物理内存的页框，从这里就可以得到印证，因为查表是以页为单位来查的。

(4)有了物理页面的基地址之后，加上 $VA[11:0]$ 这个偏移量($2^{12}=4KB$)就可以取出相应地址上的数据了。

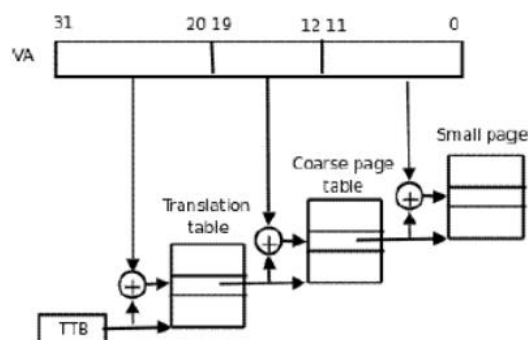


图 9. Translation Table Walk

图 41TLB

7.5 三级 Cache 支持下的物理内存访问

1. 寻找地址并发送物理地址给一级 cache
2. 分别解析出 C0, CT 等信息，取出 C0 的数据，并根据 CT 判断是否命中
3. 若匹配成功即命中，则读取数据否则提交给二级三级 cache 重复上述步骤
4. 若均为命中进入主存中查找

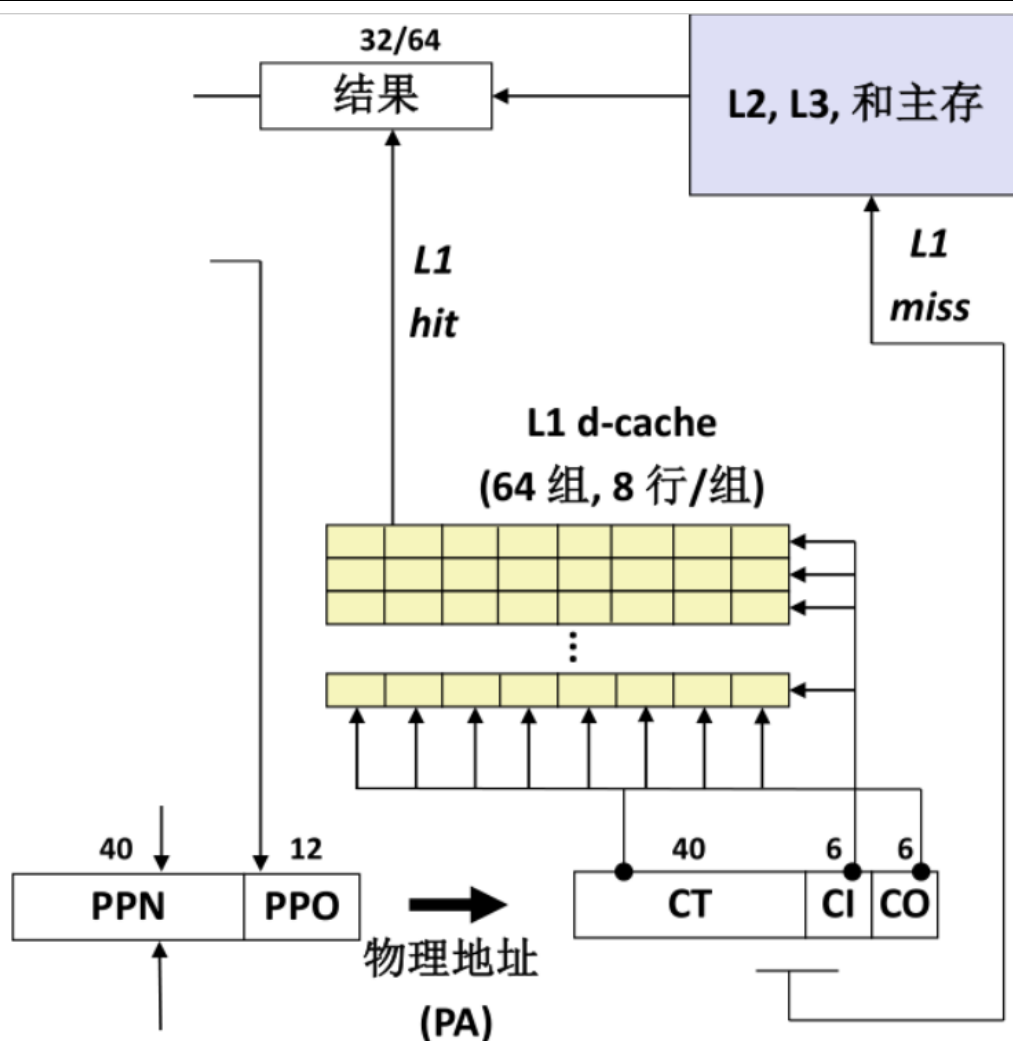


图 42cache

7.6 hello 进程 fork 时的内存映射

shell 进程调用 fork 函数，会建各种数据结构，并分配给它一个唯一的 PID，为了给这个新进程创建虚拟内存，创建了当前进程的 mm_struct、区域结构和页表的原样副本。它将这两个进程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

1. 删除已有的区域，除已经存在的区域结构
2. 映射私有区域即为新程序的代码、数据、bss 和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。例如 hello 中.data 区以及.text 区。
3. 映射共享区域，例如 hello 程序与共享对象 libc.so 链接，libc.so 是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

7.8 缺页故障与缺页中断处理

什么是缺页中断？

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

缺页中断处理一般流程：

- 1.硬件陷入内核，在堆栈中保存程序计数器，大多数当前指令的各种状态信息保存在特殊的 `cpu` 寄存器中。
- 2.启动一个汇编例程保存通用寄存器和其他易丢失信息，以免被操作系统破坏。
- 3.当操作系统发现缺页中断时，尝试发现需要哪个虚拟页面。通常一个硬件寄存器包含了这些信息，如果没有的话操作系统必须检索程序计数器，取出当前指令，分析当前指令正在做什么。
- 4.一旦知道了发生缺页中断的虚拟地址，操作系统会检查地址是否有效，并检查读写是否与保护权限一致，不过不一致，则向进程发一个信号或者杀死该进程。如果是有效地址并且没有保护错误发生则系统检查是否有空闲页框。如果没有，则执行页面置换算法淘汰页面。
- 5.如果选择的页框脏了，则将该页写回磁盘，并发生一次上下文切换，挂起产生缺页中断的进程让其他进程运行直到写入磁盘结束。且回写的页框必须标记为忙，以免其他原因被其他进程占用。
- 6.一旦页框干净后，操作系统查找所需页面在磁盘上的地址，通过磁盘操作将其装入，当页面被装入后，产生缺页中断的进程仍然被挂起，并且如果有其他可运行的用户进程，则选择另一用户进程运行。
- 7.当磁盘中断发生时，表明该页已经被装入，页表已经更新可以反映他的位置，页框也标记位正常状态。
- 8.恢复发生缺页中断指令以前的状态，程序计数器重新指向这条指令。
- 9.调度引发缺页中断的进程，操作系统返回调用他的汇编例程
- 10.该例程恢复寄存器和其他状态信息，返回到用户空间继续执行，就好像缺页中断没有发生过。

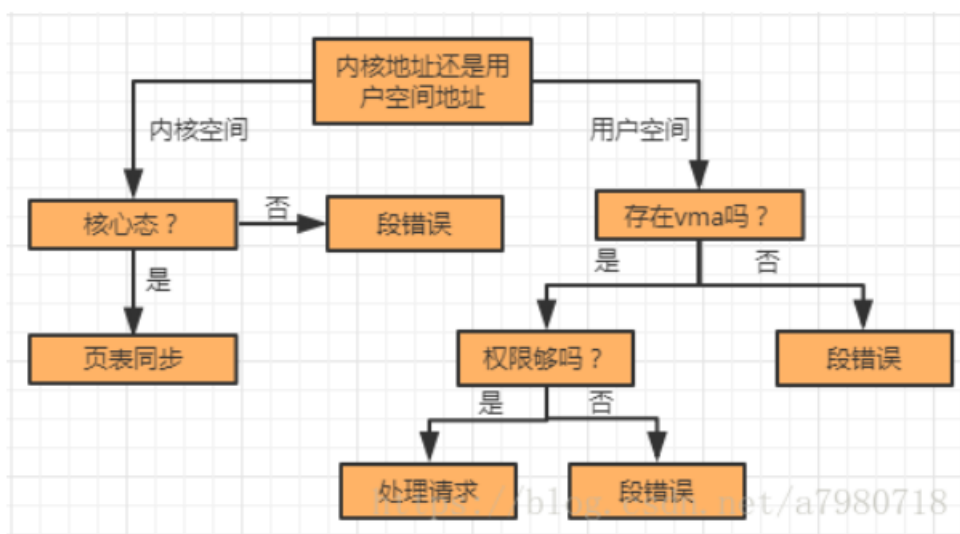


图 43 缺页中断

7.9 动态存储分配管理

Printf 会调用 malloc，请简述动态内存管理的基本方法与策略。

动态存储分配管理由动态内存分配器完成。动态内存分配器维护着一个进程的虚拟内存区域，称为堆。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后开始，并向上生长（向更高的地址）。分配器将堆视为一组不同大小的块的集合来维护。

每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可以用来分配。空闲块保持空闲，直到它显式地被应用程序所分配。

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。

动态内存分配器从堆中获得空间，将对应的块标记为已分配，回收时将堆标记为未分配。而分配和回收的过程中，往往涉及到分割、合并等操作。

动态内存分配器的目标是在对齐块的基础上，尽可能地提高吞吐率及空间占用率，即减少因为内存分配造成的碎片。其实现常见的数据结构有隐式空闲链表、显式空闲链表、分离空闲链表，常见的放置策略有首次适配、下一次适配和最佳适配。

为了更好的介绍动态存储分配的实现思想，以隐式空闲分配器的实现原理为例进行介绍。

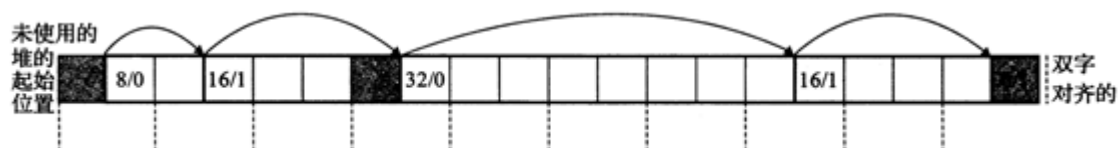


图 44 堆

隐式空闲链表分配器的实现涉及到特殊的数据结构。其所使用的堆块是由一个子的头部、有效载荷，以及可能的一些额外的填充组成的。头部含有块的大小以及是否分配的信息。有效载荷用来存储数据，而填充块则是用来对付外部碎片以及对齐要求。基于这样的基本单元，便可以组成隐式空闲链表。

通过头部记录的堆块大小，可以得到下一个堆块的大小，从而使堆块隐含地连接着，从而分配器可以遍历整个空闲块的集合。在链表的尾部有一个设置了分配位但大小为零的终止头部，用来标记结束块。

当请求一个 k 字节的块时，分配器搜索空闲链表，查找足够大的空闲块，其搜索策略主要有首次适配、下一次适配、最佳适配三种。

一旦找到空闲块，如果大小匹配的不是太好，分配器通常会将空闲块分割，剩下的部分形成一个新的空闲块。如果无法搜索到足够空间的空闲块，分配器则会通过调用 `sbrk` 函数向内核请求额外的堆内存。

当分配器释放已分配块后，会将释放的堆块自动与周围的空闲块合并，从而提高空间利用率。为了实现合并并保证吞吐率，往往需要在堆块中加入头部进行带边界标记的合并。

7.10 本章小结

具体研究了存储器地址空间、逻辑地址到线性地址的变换-段式管理、线性地址到物理地址的变换-页式管理、TLB 与四级页表支持下的 VA 到 PA 的变换三级 Cache 支持下的物理内存访问、hello 进程 `fork` 时的内存映射、`execve` 时的内存映射、页故障与缺页中断处理、态存储分配管理等内容

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

设备的模型化：文件

设备管理：unix io 接口

所有的 IO 设备都被模型化为文件，而所有的输入和输出都被 当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

8.2 简述 Unix IO 接口及其函数

首先是常用的几个函数 `open` , `read`,`write`,`lseek`, `close`

`open` 函数：函数原型 `int open(char * path,int oflag,...)`

返回值是一个文件描述符 `path` 顾名思义就是文件名 `oflag` 文件是打开方式 第三个形参应用于创建文件时使用 `/*创建文件其实还有一个 create 函数使用 以及 openat 由于还未使用过这个函数和 open 的差异 所以不在此处累赘*/` `open` 函数 使用 `if` 判断的时候 注意小细节

`read` 函数：函数原型 `ssize_t read(int fd , void* buf , size_t nbytes)` 返回值是文件读取字节数 在好几种情况下会出现返回值不等于文件读取字节数 也就是第三个参数 `nbytes` 的情况 第二个形参 `buf` 读取到 `buf` 的内存 文件偏移量(`current file offset`)受改变

`write` 函数：函数原型 `ssize_t write(int fd , const void* buf, size_t nbytes)` 返回值是文件写入字节数 `fd` 是文件描述符 将 `buf` 内容写入 `nbytes` 个字节到文件 但这里需要注意默认情况是需要系统在队列中等待写入（打开方式不同也会不同） //以上三个出错都返回-1

`lseek` 函数 `off_t lseek(int fd, off_t offset , int whence)` 返回值成功函数返回新的文件偏移量 失败-1 `fd` 文件描述符 `off_t` 是有符号的整数 `whence` 其实是和 `off_t` 配套使用的 `SEEK_SET` 文件开始处 `SEEK_CUR` 当前值的相对位置 `SEEK_END` 文件长度+-

`close` 函数原型：`int close(int fd)` 返回值 成功返回 0 失败-1 关闭文件描述符。

8.3 printf 的实现分析

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 != '%')
        {
            *p++ = *fmt;
            continue;
        }
        fmt++;
        switch (*fmt)
        {
            case 'x':
                itoa(tmp, *((int*) strcpy(p, tmp));
                p_next_arg += 4;
                p += strlen(tmp);
                break;
            case 's':
                break;
            default:
                break;
        }
    }
    return (p - buf);
}

```

```

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;
}

```

分析：printf 首先确定 arg 这一格式化参数，然后调用 vsprintf 函数判断%标志符号（如%d），其中无关符号自动略过，返回需要输出的结果串及长度，最后通过系统函数 write 输出长度 i 的串 buf 到屏幕上

8.4 getchar 的实现分析

```

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;
}
```

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

8.5 本章小结

介绍了 UNIX IO 的管理方法、接口及函数,主要分析了 `printf` 和 `getchar` 两个函数的实现。

(第 8 章 1 分)

结论

对 `Hello.c` 这一源程序依次进行预处理,编译,汇编,链接的操作之后,生成可执行程序 `Hello`。在 `shell` 中使其运行, `fork` 产生子进程,则 `Hello` 从 `program` 成为 `Process`,这个过程即为 `P2P` (`Program to process`)。紧接着进行 `execve`,通过依次进行对虚拟内存的映射,物理内存的载入,进入主函数执行代码。`hello` 调用 `write` 等系统函数在屏幕打印信息,之后 `shell` 父进程 `bash` 回收 `Hello` 的进程,一切相关的数据结构被删除。以上即为 `020` (`From Zero to Zero`) 的全部过程

(结论 0 分, 缺失 -1 分, 根据内容酌情加分)

附件

hello.c. 大作业源程序
hello.i 经过预处理后的中间文件
hello.s 经过编译后的汇编文件
hello.o 经过汇编后的可重定位目标执行文件
hello 经过链接后的可执行程序
hello1.elf hello.o 的 ELF
hello2.elf hello 的 ELF

(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] www.jianshu.com 简书
- [2] blog.csdn.net CSDN
- [3] baike.baidu.com 百度百科
- [4] www.pianshen.com 程序员大本营
- [5] www.cnblogs.com 博客园
- [6] 兰德尔 E.布莱恩特 大卫 R.奥哈拉伦. 深入理解计算机系统（第 3 版）. 机械工业出版社.

（参考文献 0 分，缺失 -1 分）