

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 計算機類

學 號 1170300601

班 級 1736101

學 生 刘畅

指 導 教 師 刘宏伟

計算機科學與技術學院

2018 年 12 月

摘 要

本文对源文件 `hello.c` 在 Linux 下利用各种工具进行预处理、编译、汇编等过程，以及在 Shell 中的动态链接、存储管理、进程管理、I/O 管理等过程进行深入探索，旨在加深对课本相关知识的理解，更直观的体验 `hello` 程序从“出生”到“死亡”的全过程，获得新的收获。

关键词：操作系统；编译；汇编；链接；虚拟内存；异常控制流；

（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	- 5 -
1.3 中间结果	- 5 -
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	- 6 -
2.3 HELLO 的预处理结果解析	- 6 -
2.4 本章小结	- 10 -
第 3 章 编译	- 11 -
3.1 编译的概念与作用	- 11 -
3.2 在 UBUNTU 下编译的命令	- 11 -
3.3 HELLO 的编译结果解析	- 11 -
3.4 本章小结	- 11 -
第 4 章 汇编	- 18 -
4.1 汇编的概念与作用	- 18 -
4.2 在 UBUNTU 下汇编的命令	- 18 -
4.3 可重定位目标 ELF 格式	- 18 -
4.4 HELLO.O 的结果解析	- 21 -
4.5 本章小结	- 24 -
第 5 章 链接	- 25 -
5.1 链接的概念与作用	- 25 -
5.2 在 UBUNTU 下链接的命令	- 25 -
5.3 可执行目标文件 HELLO 的格式	- 26 -
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	- 28 -
5.7 HELLO 的动态链接分析	- 29 -
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	- 32 -
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	- 34 -
6.7 本章小结	- 40 -
第 7 章 HELLO 的存储管理.....	- 41 -
7.1 HELLO 的存储器地址空间	- 41 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理	- 41 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理	- 43 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 44 -
7.5 三级 CACHE 支持下的物理内存访问	- 45 -
7.6 HELLO 进程 FORK 时的内存映射	- 46 -
7.7 HELLO 进程 EXECVE 时的内存映射	- 47 -
7.8 缺页故障与缺页中断处理.....	- 48 -
7.9 动态存储分配管理	- 49 -
7.10 本章小结	- 50 -
第 8 章 HELLO 的 IO 管理	- 54 -
8.1 LINUX 的 IO 设备管理方法	- 54 -
8.2 简述 UNIX IO 接口及其函数	- 54 -
8.3 PRINTF 的实现分析	- 55 -
8.4 GETCHAR 的实现分析	- 57 -
8.5 本章小结	- 57 -
结论	- 58 -
附件	- 59 -
参考文献	- 60 -

第 1 章 概述

1.1 Hello 简介

P2P: From Program to Process

hello 程序的生命周期是从一个源程序（或者说源文件）开始的，即程序员通过编辑器创建并保存的文本文件，文件名是 `hello.c`。

然而，为了在系统上运行 `hello.c` 程序，实现 P2P (From Program to Process) 的过程，每条 `c` 语句都必须被其他程序转化为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放起来。

首先，`gcc` 编译器驱动程序读取程序文件 `hello.c`，然后预处理器 (`cpp`) 根据以字符 `#` 开头的命令，修改原始的 `C` 程序，得到 `hello.i`。

随后，编译器 (`cc1`) 将文本文件 `hello.i` 翻译成文本文件 `hello.s`。接下来，汇编器 (`as`) 将 `hello.s` 翻译成机器语言指令，将结果保存在目标二进制文件 `hello.o` 中。

最后，链接器 (`ld`) 合并预编译好的目标文件，得到可执行目标文件 `hello`。Linux 系统中通过内置命令行解释器 `shell` 加载运行 `hello` 程序，它会通过 `fork` 函数为 `hello` 创建一个进程，再通过 `execve` 执行 `hello`。至此，P2P 的过程就执行完毕了。

O2O: From Zero-0 to Zero-0

Linux 系统中内置命令行解释器 `shell` 加载运行 `hello` 程序，它会通过 `fork` 函数为 `hello` 创建一个进程，再通过 `execve` 执行 `hello`。对 `hello` 的进行数据处理时，需要在内存上申请空间，先删除当前虚拟地址的用户部分已存在的数据结构，为 `hello` 的代码、数据、`bss` 和栈等创建新的数据结构，利用计算机的缓存结构，传递数据，映射虚拟内存，设置程序计数器，使之指向代码区域的入口点，进入 `main` 函数，CPU 为 `hello` 分配时间片执行逻辑控制流。同时通过 Unix I/O 管理来控制输入输出。执行完成后，`hello` 的父进程 `shell` 回收 `hello`，内核从系统中删除 `hello` 的所有痕迹，释放其运行过程中占用的内存空间。至此，O2O 的过程就执行完毕了。

1.2 环境与工具

硬件

Intel (R) Core (TM) i7-6700HQ X64CPU 2.60GHz 16.0GBRAM

软件

Winsows 7, Vmware 15.0.0, Ubuntu18.04.1

开发及调试

codeblocks, gdb, objdump, gcc, edb, readelf, hexedit

1.3 中间结果

hello.i:	hello.c 预编译的结果, 用于研究预编译的作用以及进行编译器的下一步编译操作。
hello.s:	hello.i 编译后的结果, 用于研究汇编语言以及编译器的汇编操作, 可以与 hello.c 对应, 分析底层的实现。
hello.o:	hello.s 汇编后的结果, 可重定位目标程序, 没有经过链接, 用于链接器或编译器链接生成最终可执行程序。
hello.out:	hello.o 链接后生成的可执行目标文件, 可以用来反汇编或者通过 EDB、GDB 等工具分析链接过程以及程序运行过程, 包括进入 main 函数前后发生的过程。
hello:	链接之后的可执行目标文件, 可直接运行。
hello.o.s:	对可重定位目标文件反汇编得到, 可以与对可执行目标文件反汇编得到的代码对比来分析链接过程。
a.s:	对可执行目标文件反汇编得到, 可以用来分析链接过程与寻址过程。
hello.elf	hello 的 elf 格式, 用来分析各段的基本信息

(表 1.1 中间结果)

1.4 本章小结

本章简要概述了 hello 的 P2P 与 O2O 的过程, 介绍了本论文撰写的过程中, 用的软硬件环境, 以及开发与调试工具。同时列出了本论文撰写的过程中, 生成的中间结果文件的名字, 文件的作用等。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

2.2.1 概念

预处理是指预处理器（cpp）根据以字符#开头的命令，修改原始的 C 程序，比如#include <stdio.h>命令告诉预处理器读取系统头文件 stdio.h 的内容，并把它直接插入程序文本中。其他常见的预处理指令还有#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）等。预处理指令一般被用来使源代码在不同的执行环境中方便的被改或者编译。

2.1.2 作用

将头文件拷入源文件，得到另外一个 C 程序，通常是以.i 作为文件扩展名。

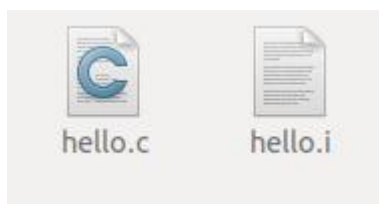
2.2 在 Ubuntu 下预处理的命令

使用 gcc -E -o hello.i hello.c 指令进行预处理过程，如下图 2.1



（图 2.1 使用 gcc -E -o hello.i hello.c 指令进行预处理过程）

生成了目标文件 hello.i，如下图 2.2:



（图 2.2 生成的目标文件 hello.i）

2.3 Hello 的预处理结果解析

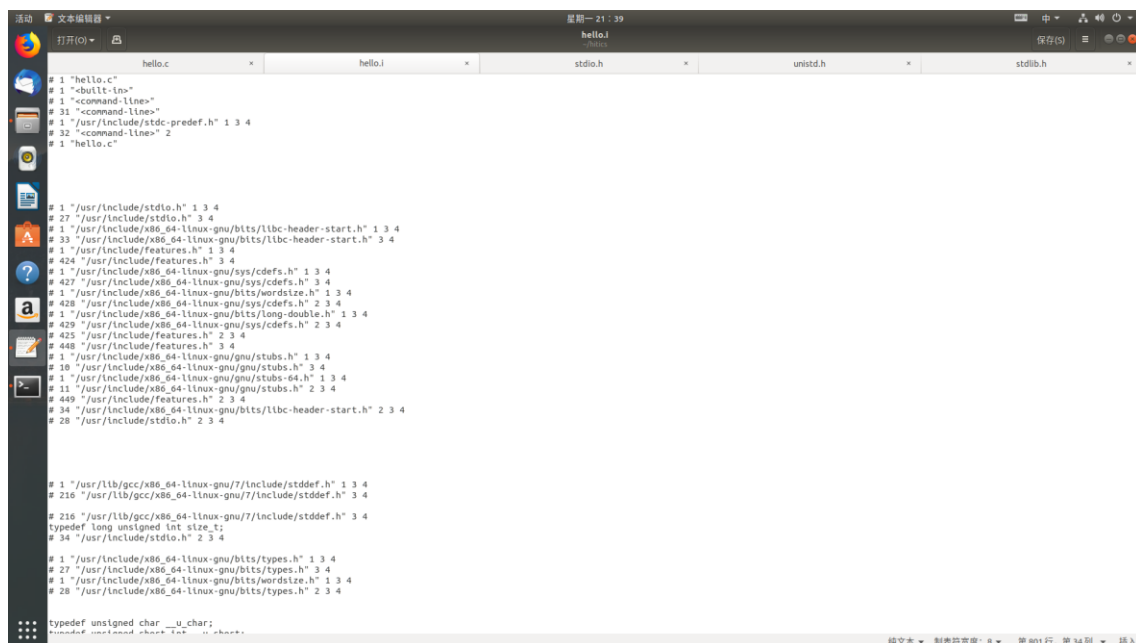
打开 hello.i，hello.c 文件，以及 hello.c 文件中包含的头文件 stdio.h,unistd.h,stdlib.h，如下图 2.3:

计算机系统课程报告



(图 2.3 查阅的头文件举例)

通过对比发现, hello.i 在 hello.c 的基础上, 把头文件的内容直接插入程序文本中, 对头文件中还包含的其他头文件, 系统会递归式的进行展开。Hello.i 文件包含声明函数、定义结构体、定义变量、定义宏等内容, 还包含原文件的函数。如图组 2.4。



(图 2.4.1 hello.i 中展开的头文件举例)

计算机系统课程报告



```
hello.c * hello.i * stdio.h * unistd.h * stdlib.h *
# 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

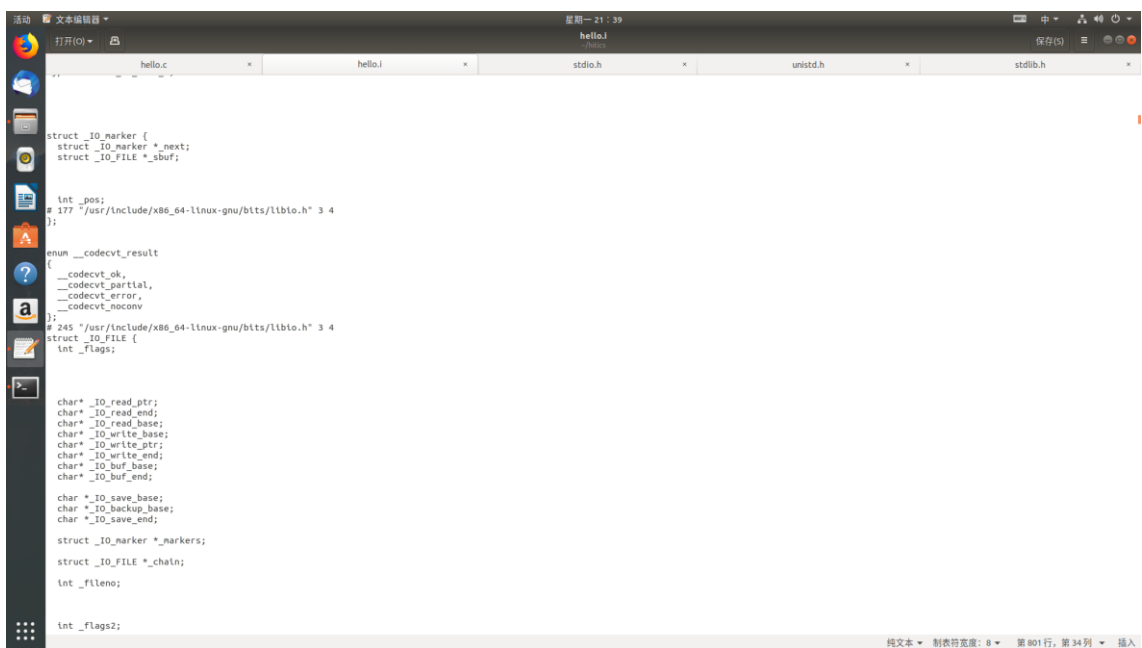
typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;

typedef long int __quad_t;
typedef unsigned long int __u_quad_t;

typedef long int __intmax_t;
typedef unsigned long int __uintmax_t;
# 188 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/typesizes.h" 1 3 4
# 131 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4

typedef unsigned long int __dev_t;
typedef unsigned int __uid_t;
typedef unsigned int __gid_t;
typedef unsigned long int __ino_t;
typedef unsigned long int __ino64_t;
typedef unsigned int __mode_t;
typedef unsigned long int __nlink_t;
typedef long int __off_t;
typedef long int __off64_t;
typedef int __pid_t;
typedef struct { int __val[2]; } __fsid_t;
typedef long int __riscv_r_t;
```

(图 2.4.2 hello.i 中声明举例)



```
hello.c * hello.i * stdio.h * unistd.h * stdlib.h *
struct _IO_marker {
    struct _IO_marker *next;
    struct _IO_FILE *_sbuf;
};

int __pos;
# 177 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
};

enum __codecvt_result
{
    __codecvt_ok,
    __codecvt_partial,
    __codecvt_error,
    __codecvt_noconv
};
# 245 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
struct _IO_FILE {
    int _flags;

    char* __IO_read_ptr;
    char* __IO_read_end;
    char* __IO_read_base;
    char* __IO_write_base;
    char* __IO_write_ptr;
    char* __IO_write_end;
    char* __IO_buf_base;
    char* __IO_buf_end;

    char* __IO_save_base;
    char* __IO_backup_base;
    char* __IO_save_end;

    struct _IO_marker *_markers;

    struct _IO_FILE *_chain;

    int _fileno;

    int _flags2;
};
```

(图 2.4.3 hello.i 中结构体举例)

计算机系统课程报告



```
extern int renameat (int __oldfd, const char *__old, int __newfd,
                    const char *__new) __attribute__((__nothrow__, __leaf__));

extern FILE *tmpfile (void) ;
# 173 "usr/include/stdio.h" 3 4
extern char *tmpnam (char *__s) __attribute__((__nothrow__, __leaf__));

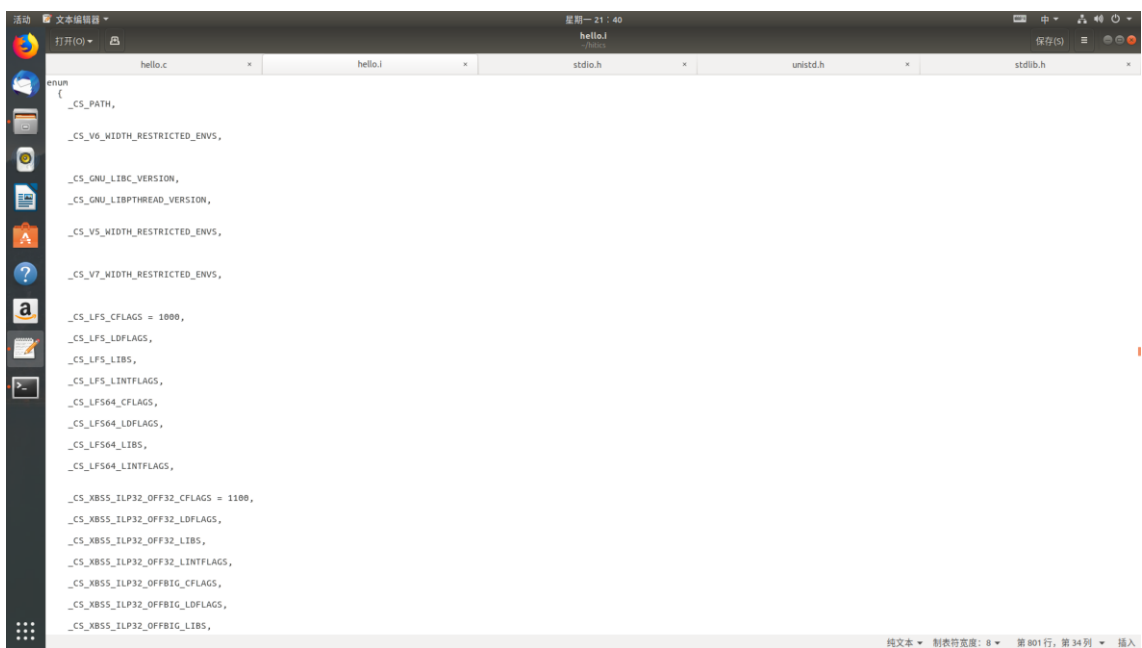
extern char *tmpnam_r (char *__s) __attribute__((__nothrow__, __leaf__));
# 190 "usr/include/stdio.h" 3 4
extern char *tempnam (const char *__dir, const char *__pfx)
    __attribute__((__nothrow__, __leaf__)) __attribute__((__malloc__));

extern int fclose (FILE *__stream);

extern int fflush (FILE *__stream);
# 213 "usr/include/stdio.h" 3 4
extern int fflush_unlocked (FILE *__stream);
# 232 "usr/include/stdio.h" 3 4
extern FILE *fopen (const char *__restrict __filename,
                    const char *__restrict __modes);

extern FILE *freopen (const char *__restrict __filename,
                     const char *__restrict __modes,
                     FILE *__restrict __stream);
# 265 "usr/include/stdio.h" 3 4
extern FILE *fdopen (int __fd, const char *__modes) __attribute__((__nothrow__, __leaf__));
# 278 "usr/include/stdio.h" 3 4
extern FILE *fmemopen (void *__s, size_t __len, const char *__modes)
    __attribute__((__nothrow__, __leaf__)) __attribute__((__malloc__));
```

(图 2.4.4 hello.i 中 extern 举例)



```
enum
{
    _CS_PATH,

    _CS_V6_WIDTH_RESTRICTED_ENVS,

    _CS_GNU_LIBC_VERSION,
    _CS_GNU_LIBPTHREAD_VERSION,

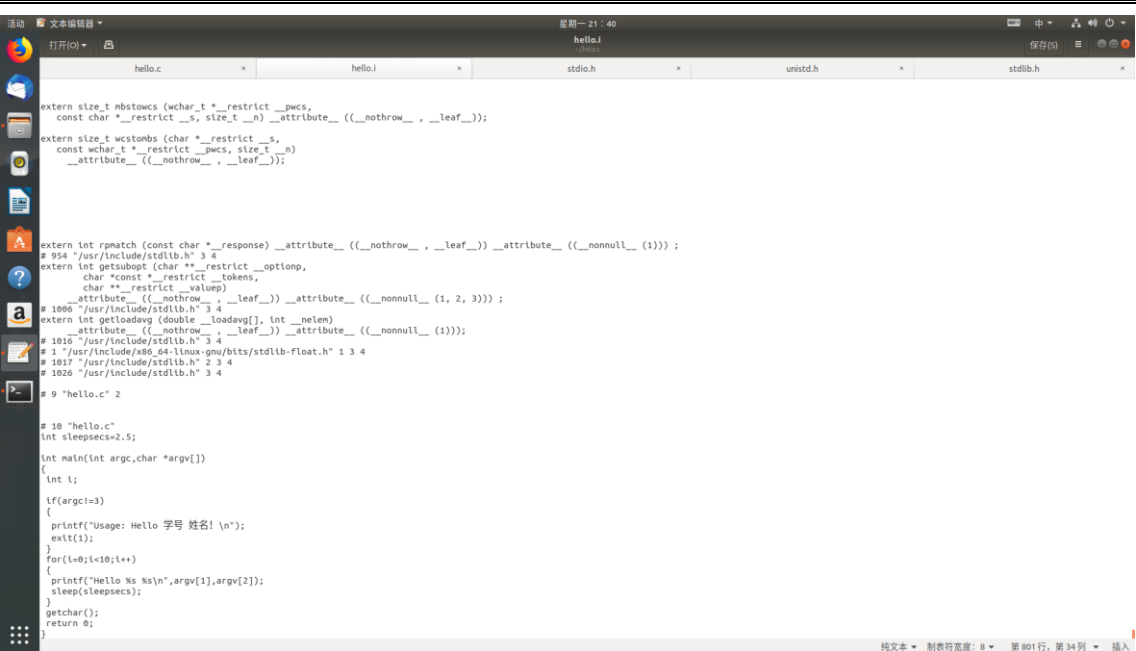
    _CS_V5_WIDTH_RESTRICTED_ENVS,

    _CS_V7_WIDTH_RESTRICTED_ENVS,

    _CS_LFS_CFLAGS = 1000,
    _CS_LFS_LDFLAGS,
    _CS_LFS_LIBS,
    _CS_LFS_LINTFLAGS,
    _CS_LFS64_CFLAGS,
    _CS_LFS64_LDFLAGS,
    _CS_LFS64_LIBS,
    _CS_LFS64_LINTFLAGS,

    _CS_XBSS_ILP32_OFF32_CFLAGS = 1100,
    _CS_XBSS_ILP32_OFF32_LDFLAGS,
    _CS_XBSS_ILP32_OFF32_LIBS,
    _CS_XBSS_ILP32_OFF32_LINTFLAGS,
    _CS_XBSS_ILP32_OFFBIG_CFLAGS,
    _CS_XBSS_ILP32_OFFBIG_LDFLAGS,
    _CS_XBSS_ILP32_OFFBIG_LIBS,
```

(图 2.4.5 hello.i 中枚举举例)



```
extern size_t mbstowcs (wchar_t *__restrict __pwcs,
const char *__restrict __s, size_t __n) __attribute__((__nothrow__, __leaf__));

extern size_t wcstombs (char *__restrict __s,
const wchar_t *__restrict __pwcs, size_t __n)
__attribute__((__nothrow__, __leaf__));

extern int rpnatch (const char *__response) __attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));
# 954 "/usr/include/stdlib.h" 3 4
extern int getsubopt (char **__restrict __optionp,
char *const *__restrict __tokens,
char **__restrict __valpp)
__attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1, 2, 3)));
# 1006 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
__attribute__((__nothrow__, __leaf__)) __attribute__((__nonnull__(1)));
# 1016 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/stdio.h" 1
# 1017 "/usr/include/stdio.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 字号 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 2.4.6 hello.i 中 main 函数位于末尾)

由此可知，预处理的过程将头文件拷入源文件，得到另外一个 C 程序，通常是以 .i 作为文件扩展名。

2.4 本章小结

本章介绍了预处理的概念及作用，以及 Ubuntu 下预处理对应的指令，同时解析了预处理得到的文件的内容，直观的展现了预处理的结果。预处理这一过程，为 hello 的下一步处理做好了准备。

(第 2 章 0.5 分)

第 3 章 编译

3.1 编译的概念与作用

3.1.1 概念

编译器 (cc1) 将文本文件 `hello.i` 经过词法分析、语法分析、语义检查和中间代码生成、代码优化、目标代码生成的过程翻译成文本文件 `hello.s`，它包含了一个汇编语言程序。

3.1.2 作用

将 `hello.i` 翻译成 `hello.s`。

注意：这儿的编译是指从 `.i` 到 `.s` 即预处理后的文件到生成汇编语言程序

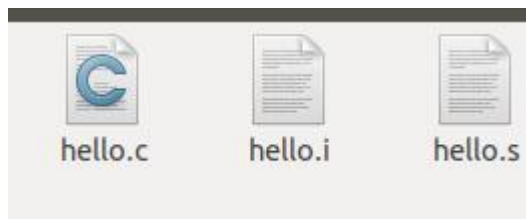
3.2 在 Ubuntu 下编译的命令

使用 `gcc -S hello.i -o hello.s` 指令，进行编译过程，如下图 3.1：



(图 3.1 使用 `gcc -S hello.i -o hello.s` 指令进行编译过程)

生成了目标文件 `hello.s`，如下图 3.2：



(3.2 生成的 `hello.s` 文件)

3.3 Hello 的编译结果解析

3.3.1 汇编指令

指令	对应内容
.file	声明源文件
.text	声明代码段
.data	声明数据段
.align	声明指令及数据存放地址的对齐方式
.type	指定类型
.size	声明大小
.section .rodata	声明 rodata 段
.globl	声明全局变量
.long、.string	声明一个 long、string 类型

(表 3.1 汇编指令)

3.3.2 变量

```

.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

```

(图 3.3 hello 的汇编文件声明部分)

如图 3.3，首先声明源文件为 hello.c，然后将 sleepsecs 变量声明为全局变量放入 4 字节对齐的 .data 节。同时声明 sleepsecs 的大小为 4 字节的，编译器设置其为 long 型变量，值为 2。然后声明只读数据节，包含两个字符串。“Usage: Hello 学号 姓名! \n”，“Hello %s %s\n”，可以发现字符串被编码成 UTF-8 格式，一个汉字在 utf-8 编码中占三个字节，以 ‘\345’ 开头，一个\代表一个字节。随后声明 main 为函数类型。

```

subq    $32, %rsp
movl     %edi, -20(%rbp)
movq     %rsi, -32(%rbp)

```

(图 3.4 hello 中主函数的参数)

如图 3.4，主函数的 int argc, char *argv[] 两个参数，分别将存放在栈中 rbp

寄存器指向地址-20 和-32 处，其中%edi 代表 argc，%rsi 代表 argv[]。

```

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 3.5 hello 中局部变量 i)

整型局部变量 i 位于栈空间-0x4(%rbp)位置，占 4 个字节大小。

立即数，如判断 argc!=3、i<10 等直接作为立即数存放于指令中。

3.3.3 算数、逻辑操作

指令	效果	描述
INC D	D = D+1	加 1
DEC D	D = D-1	减 1
NEG D	D = -D	取负
ADD S,D	D = D+S	加
SUB S,D	D = D-S	减
IMUL S,D	D = D*S	乘
IMULQ S	R[%rdx]:R[%rax]=S*R[%rax]	有符号全乘法
MULQ S	R[%rdx]:R[%rax]=S*R[%rax]	无符号全乘法
IDIVQ S	R[%rdx]=R[%rdx]:R[%rax] mod S R[%rax]=R[%rdx]:R[%rax] div S	有符号除法
DIVQ S	R[%rdx]=R[%rdx]:R[%rax] mod S R[%rax]=R[%rdx]:R[%rax] div S	无符号除法
leaq S,D	D = &S	加载有效地址

(表 3.2 算数、逻辑操作)

程序中涉及的算数操作有：

```

|         addl    $1, -4(%rbp)
|.L3:

```

(图 3.6 i 的自增)

i++, 对计数器 i 自增, 使用程序指令 addl, 后缀 l 代表操作数是一个 4B 大小的数据。

```
leaq .LC1(%rip), %rdi
```

(图 3.7 LC1 段地址的加载)

汇编中使用 leaq .LC1(%rip), %rdi, 使用了加载有效地址指令 leaq 计算 LC1 的段地址 %rip+.LC1 并传递给 %rdi。

3.3.4 数组操作

```
movq -32(%rbp), %rax
addq $16, %rax
movq (%rax), %rdx
movq -32(%rbp), %rax
addq $8, %rax
movq (%rax), %rax
movq %rax, %rsi
```

(图 3.8 数组操作)

根据 argv 首地址获得 argv[1] 和 argv[2], 获得一个地址, 然后通过 add 指令加某个数值进行索引。可以发现, argv 数组作为一个 char* 类型的数组, char* 的大小是 8 个字节, 所以 argv[1] 加 0x8 得到 argv[2]。

3.3.5 关系操作、控制转移

指令	效果	描述
CMP S1,S2	S2-S1	比较-设置条件码
TEST S1,S2	S1&S2	测试-设置条件码
SET** D	D=**	按照**将条件码设置D
J**	——	根据**与条件码进行跳转

(图 3.9 关系操作、控制转移)

指令	同义名	跳转条件	描述
<code>jmp Label</code>		1	直接跳转
<code>jmp *Operand</code>		1	间接跳转
<code>je Label</code>	<code>jz</code>	ZF	相等/零
<code>jne Label</code>	<code>jnz</code>	-ZF	不相等/非零
<code>js Label</code>		SF	负数
<code>jns Label</code>		-SF	非负数
<code>jg Label</code>	<code>jnle</code>	$\neg(SF \wedge OF) \wedge \neg ZF$	大于 (有符号>)
<code>jge Label</code>	<code>jnl</code>	$\neg(SF \wedge OF)$	大于或等于 (有符号 \geq)
<code>jl Label</code>	<code>jnge</code>	$SF \wedge OF$	小于 (有符号<)
<code>jle Label</code>	<code>jng</code>	$(SF \wedge OF) \vee ZF$	小于或等于 (有符号 \leq)
<code>ja Label</code>	<code>jnbe</code>	$\neg CF \wedge \neg ZF$	超过 (无符号>)
<code>jae Label</code>	<code>jnb</code>	$\neg CF$	超过或相等 (无符号 \geq)
<code>jb Label</code>	<code>jnae</code>	CF	低于 (无符号<)
<code>jbe Label</code>	<code>jna</code>	$CF \vee ZF$	低于或相等 (无符号 \leq)

(图 3.10 跳转指令)

```

movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
cmpl    $3, -20(%rbp)
je       .L2
leaq    .LC0(%rip), %rdi
call    puts@PLT
movl    $1, %edi
call    exit@PLT

```

(图 3.11 参数个数比较)

编译器编译的执行逻辑首先和 3 进行比较, 如果相等则不满足 $argc != 3$, 那么通过 `je` 跳过下面的操作。如果不相等, 则执行大括号内部的操作。

```

.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
    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   .cfi_def_cfa 7, 8
    ret
.cfi endproc

```

(图 3.12 循环条件判断)

编译器实现 for 循环同样是通过 JMP 跳转。判断 i 是否小于 10。计算 i-9，设置条件码，为下一步 jle 利用条件码进行跳转做准备。如果小于等于 9 则跳转继续执行循环，否则不跳转，顺序执行循环外的指令。

3.3.6 函数操作

main 函数

传递控制

main 函数被系统启动函数 `__libc_start_main` 调用，call 指令将下一条指令的地址 dest 压栈，然后跳转到 main 函数。

传递数据

外部调用过程向 main 函数传递参数 argc 和 argv，使用 %rdi 和 %rsi 存储，函数出口为 return 0，将 %eax 设置 0 返回。

分配和释放内存

使用 %rbp 记录栈底，结束时调用 leave 指令，恢复栈空间为调用之前的状态，然后返回，将下一条要执行指令的地址设置为 dest。

printf 函数

传递数据

第一次 printf 将 %rdi 设置为 “Usage: Hello 学号 姓名! \n” 字符串的首地址。第二次 printf 设置 %rdi 为 “Hello %s %s\n” 的首地址，将 %rsi 设置为 argv[1]，%rdx 设置为 argv[2]。

控制传递

第一次 printf 使用 call puts@PLT；第二次 printf 使用 call printf@PLT。

exit 函数

传递数据

将%edi 设置为 1。

控制传递

call exit@PLT。

sleep 函数

传递数据

将%edi 设置为 sleepsecs。

控制传递

call sleep@PLT。

getchar 函数

控制传递

call gethcar@PLT

3.4 本章小结

本章完成了对 hello.i 的编译工作。编译器实现将 C 语言代码转换成汇编代码，从而最终转换为机器代码。生成汇编代码的这个过程需要对数据、操作都进行对应转换，数据包括常量、变量（全局/局部/静态）、表达式、宏等，操作包括算术、逻辑、位、关系、函数等操作。使用 Ubuntu 下的编译指令可以将其转换为.s 汇编语言文件。完成了对汇编代码的解析工作，为下一阶段的汇编过程做好准备。

(第3章2分)

第 4 章 汇编

4.1 汇编的概念与作用

4.1.1 概念

汇编器 (as) 将 .s 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, .o 文件是一个二进制文件, 它包含程序的指令编码。

4.1.2 作用

将 hello.s 翻译成 hello.o。

注意: 这儿的汇编是指从 .s 到 .o 即编译后的文件到生成机器语言二进制程序的过程。

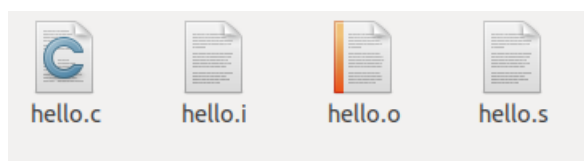
4.2 在 Ubuntu 下汇编的命令

使用 `gcc -c hello.s -o hello.o` 指令, 进行汇编过程, 如下:



(图 4.1 使用 `gcc -c hello.s -o hello.o` 指令进行汇编过程)

生成了目标文件 hello.o



(图 4.2 生成的目标文件)

4.3 可重定位目标 elf 格式

ELF 头	描述生成该文件的系统字的大小和字节顺序
.text	已编译程序的机器代码
.rodata	只读数据
.data	已初始化的全局和静态 C 变量

.bss	未初始化的全局和静态 C 遍历
.symtab	存放程序中定义和引用的函数和全局变量信息
.rel.text	一个.text 节中位置的列表，链接时修改
.rel.data	被模块引用或定义的所有全局变量的重定位信息
.debug	条目是局部变量、类型定义、全局变量及 C 源文件
.line	C 源程序中行号和.text 节机器指令的映射
.strtab	.symtab 和.debug 中符号表及节头部中节的名字
节头部表	描述目标文件的节

(表 4.1 elf 格式)

ELF 头以一个 16 字节的 magic 序列开始，这个序列描述了生成该文件的系统的字的大小和字节顺序。ELF 头剩下的部分包含帮助链接器语法分析和解释目标文件的信息。其中包括 ELF 头的大小、目标文件的类型、机器类型、节头部表的文件偏移，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的头目。hello.elf 的 ELF 头如下：

```

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: 1144 (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 4.3 elf 头)

节头部表记录了各节名称、类型、地址、偏移量、大小、全体大小、旗标、连接、信息、对齐信息。hello.elf 节头部表如下：

节头:

[号]	名称 大小	类型 全体大小	地址 旗标	链接	信息	偏移量 对齐
[0]		NULL	0000000000000000			00000000
	0000000000000000	0000000000000000		0	0	0
[1]	.text	PROGBITS	0000000000000000			00000040
	0000000000000081	0000000000000000	AX	0	0	1
[2]	.rela.text	RELA	0000000000000000			00000338
	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
	0000000000000025	0000000000000001	MS	0	0	1
[7]	.note.GNU-stack	PROGBITS	0000000000000000			00000118
	0000000000000000	0000000000000000		0	0	1
[8]	.eh_frame	PROGBITS	0000000000000000			00000118
	0000000000000038	0000000000000000	A	0	0	8
[9]	.rela.eh_frame	RELA	0000000000000000			000003f8
	0000000000000018	0000000000000018	I	10	8	8
[10]	.symtab	SYMTAB	0000000000000000			00000150
	00000000000000198	0000000000000018		11	9	8
[11]	.strtab	STRTAB	0000000000000000			000002e8
	000000000000004d	0000000000000000		0	0	1
[12]	.shstrtab	STRTAB	0000000000000000			00000410
	0000000000000061	0000000000000000		0	0	1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

There are no section groups in this file.

(图 4.4 节头部表)

.rela.text 记录了一个.text 节中位置的列表,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。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。包含了main 函数调用的.rodata 节、puts、exit、printf、sleep、getchar 函数以及全局变量 sleepsecs 的偏移量、信息、类型、符号值、符号名称及加数。链接器会依据重定向节的信息对可重定向的目标文件进行链接得到可执行文件。

rela.eh_frame 记录了.text 的信息。hello.elf 的重定位节如下:

重定位节 '.rela.text' at offset 0x338 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

(图 4.5 重定位节一)

```

重定位节 '.rela.eh_frame' at offset 0x3f8 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.

Symbol table '.symtab' contains 17 entries:
Num:      Value      Size Type      Bind      Vis      Ndx Name
0: 0000000000000000  0 NOTYPE   LOCAL   DEFAULT  UND
1: 0000000000000000  0 FILE    LOCAL   DEFAULT  ABS hello.c
2: 0000000000000000  0 SECTION LOCAL   DEFAULT  1
3: 0000000000000000  0 SECTION LOCAL   DEFAULT  3
4: 0000000000000000  0 SECTION LOCAL   DEFAULT  4
5: 0000000000000000  0 SECTION LOCAL   DEFAULT  5
6: 0000000000000000  0 SECTION LOCAL   DEFAULT  7
7: 0000000000000000  0 SECTION LOCAL   DEFAULT  8
8: 0000000000000000  0 SECTION LOCAL   DEFAULT  6
9: 0000000000000000  4 OBJECT  GLOBAL  DEFAULT  3 sleepsecs
10: 0000000000000000 129 FUNC    GLOBAL  DEFAULT  1 main
11: 0000000000000000  0 NOTYPE  GLOBAL  DEFAULT  UND _GLOBAL_OFFSET_TABLE_
12: 0000000000000000  0 NOTYPE  GLOBAL  DEFAULT  UND puts
13: 0000000000000000  0 NOTYPE  GLOBAL  DEFAULT  UND exit
14: 0000000000000000  0 NOTYPE  GLOBAL  DEFAULT  UND printf
15: 0000000000000000  0 NOTYPE  GLOBAL  DEFAULT  UND sleep
16: 0000000000000000  0 NOTYPE  GLOBAL  DEFAULT  UND getchar

No version information found in this file.

```

(图 4.6 重定位节二、符号表)

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

利用 `objdump -d -r hello.o` 得到 `hello.o` 的反汇编，

```

11170300601@ubuntu:~/hitics$ 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     $0x0,%eax
7f:  c9                      leaveq  %rax
80:  c3                      retq
11170300601@ubuntu:~/hitics$

```

(图 4.7 hello.o 的反汇编)

```

.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)
    movq    %rsi, -32(%rbp)
    cmpl    $3, -20(%rbp)
    je      .L2
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $1, %edi
    call    exit@PLT
.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
    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
    .cfi_def_cfa 7, 8
    ret

```

(图 4.8 hello.s)

通过二者的对比，我们可以发现：

反汇编得到的文件中只有对文件最简单的描述，记录了文件格式和 .text 代码段；而 hello.s 中有对文件的描述，全局变量的完整描述（包括 .type .size .align

大小及数据类型)以及.rodata只读数据段。

两者均包含main函数的汇编代码,但是区别在于hello.s的汇编代码由一段段的语句构成,同时声明了程序起始位置及其基本信息等;而反汇编得到的文件则是由一整块的代码构成,除需要链接后才能确定的地址(此时用空白占位符占位),包含有完整的跳转逻辑和函数调用。

分支转移:反汇编得到的代码跳转指令的操作数使用的不是助记符表示的段名称,而是已确定的实际指令地址。

函数调用:在hello.s中,函数调用是call之后直接跟着函数名称,而在反汇编程序中,call的目标地址是当前下一条指令。反汇编得到的代码中,call地址后为占位符(4个字节的0),指向的是下一条地址的位置。这是因为hello.c中调用的库函数调用需要通过动态链接器才能确定函数的运行时执行地址。

变量访问:在hello.s中,访问方式为段名称+%rip,对.rodata中printf的格式串的访问需要通过助记符.LC0、.LC1等。

在反汇编得到的代码中,访问方式为0+%rip,对.rodata中printf的格式串的访问需要通过链接时重定位的绝对引用确定地址,因此在汇编代码相应位置仍用占位符表示。

两者访问参数的方式相同即通过栈帧结构及%rbp相对寻址访问。

4.5 本章小结

本章完成了对hello.s的汇编工作。使用Ubuntu下的汇编指令将其转换为.o可重定位目标文件。对汇编后产生的hello.o的可重定位的ELF文件进行举例分析,对反汇编文件与hello.s进行对比,对二者之间的差别进行比较。为下一阶段的链接做好准备。

(第4章1分)

第 5 章 链接

5.1 链接的概念与作用

5.1.1 概念

人们把每个源代码模块独立的进行编译，然后按照需要将它们组装起来，这个组装的过程就是链接（Linking）。

5.1.2 作用

- 1.合并各个.obj 文件的 section，合并符号表，进行符号解析；
- 2.符号地址重定位；
- 3.生成可执行文件

注意：这儿的链接是指从 hello.o 到 hello 生成过程。

5.2 在 Ubuntu 下链接的命令

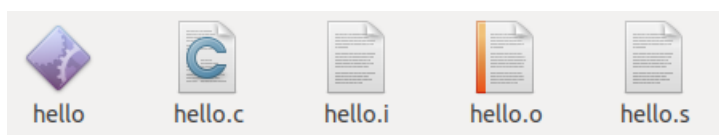
使用 `ld -o hello -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o /usr/lib/x86_64-linux-gnu/libc.so /usr/lib/x86_64-linux-gnu/crtn.o` 指令，进行链接过程，如下：

A terminal window titled 'l1170300601@ubuntu: ~/hitics' showing the execution of the linker command. The command is: `ld -o hello.out -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`. The output is not visible in the screenshot.

```
l1170300601@ubuntu: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
l1170300601@ubuntu:~/hitics$ ld -o hello.out -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 使用连接命令进行链接过程）

生成了目标文件 hello



（图 5.2 生成的目标文件）

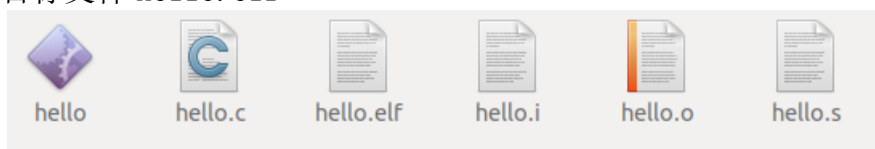
5.3 可执行目标文件 hello 的格式

通过 `readelf -a hello > hello.elf` 命令生成 hello 程序的 ELF 格式文件如下：



(图 5.3 使用命令生成 elf)

得到目标文件 hello.elf



(图 5.4 生成的目标文件)

节头部表中包含了各段的基本信息，包括名称、类型、地址、偏移量、大小、全体大小、旗标、链接、信息、对齐等信息：

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]	0000000000000000	NULL	0000000000000000	0	0	0
[1]	.interp	PROGBITS	0000000000400200	0	0	00000200
[2]	.note.ABI-tag	NOTE	000000000040021c	0	0	0000021c
[3]	.hash	HASH	0000000000400240	0	0	00000240
[4]	.gnu.hash	GNU_HASH	0000000000400278	0	0	00000278
[5]	.dynsym	DYNSYM	0000000000400298	0	0	00000298
[6]	.dynstr	STRTAB	0000000000400358	0	0	00000358
[7]	.gnu.version	VERSYM	00000000004003b0	0	0	000003b0
[8]	.gnu.version_r	VERNEED	00000000004003c0	0	0	000003c0
[9]	.rela.dyn	RELA	00000000004003e0	0	0	000003e0
[10]	.rela.plt	RELA	0000000000400410	0	0	00000410
[11]	.init	PROGBITS	0000000000400488	0	0	00000488
[12]	.plt	PROGBITS	00000000004004a0	0	0	000004a0
[13]	.text	PROGBITS	0000000000400500	0	0	00000500
[14]	.fini	PROGBITS	0000000000400634	0	0	00000634
[15]	.rodata	PROGBITS	0000000000400640	0	0	00000640
[16]	.eh_frame	PROGBITS	0000000000400670	0	0	00000670
[17]	.dynamic	DYNAMIC	0000000000600e50	0	0	00000e50
[18]	.got	PROGBITS	0000000000600ff0	0	0	00000ff0
[19]	.got.plt	PROGBITS	0000000000601000	0	0	00001000
[20]	.data	PROGBITS	0000000000601040	0	0	00001040
[21]	.comment	PROGBITS	0000000000000000	0	0	00001048
[22]	.syntab	SYMTAB	0000000000000000	0	0	00001070
[23]	.strtab	STRTAB	0000000000000000	0	0	00001508
	0000000000000150			0	0	1

(图 5.5 节头部表)

```
[24] .shstrtab          STRTAB          0000000000000000 00001658
      00000000000000c5 0000000000000000          0      0      1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

There are no section groups in this file.
```

(图 5.5 续 节头部表)

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，

.interp 节

00000000:00400238 hello!.interp	2f	db 0x2f
00000000:00400239	6c	insb %dx, (%rdi)
00000000:0040023a	69 62 36 34 2f 6c 64	imull \$0x646c2f34, 0x36(...
00000000:00400241	2d 6c 69 6e 75	subl \$0x756e696c, %eax
00000000:00400246	78 2d	js 0x400275

(图 5.6 .interp 节)

.init 节

00000000:004004a8 hello!_init	48 83 ec 08	subq \$8, %rsp
00000000:004004ac	48 8b 05 45 0b 20 00	movq 0x200b45(%rip), %rax
00000000:004004b3	48 85 c0	testq %rax, %rax
00000000:004004b6	74 05	je 0x4004bd
00000000:004004b8	e8 83 00 00 00	callq hello!.plt+0x70
00000000:004004bd	48 83 c4 08	addq \$8, %rsp
00000000:004004c1	c3	retq

(图 5.7 .init 节)

.plt 节

00000000:004004d0 hello!.plt	ff 35 32 0b 20 00	pushq 0x200b32(%rip)
00000000:004004d6	ff 25 34 0b 20 00	jmpq *0x200b34(%rip)
00000000:004004dc	0f 1f 40 00	nopl (%rax)

(图 5.8 .plt 节)

.text 节

00000000:00400550	hello!_start	31 ed	xorl %ebp, %ebp
00000000:00400552		49 89 dl	movq %rdx, %r9
00000000:00400555		5e	popq %rsi
00000000:00400556		48 89 e2	movq %rsp, %rdx
00000000:00400559		48 83 e4 f0	andq \$0xffffffff0, %rsp
00000000:0040055d		50	pushq %rax
00000000:0040055e		54	pushq %rsp
00000000:0040055f		49 c7 c0 40 07 40 00	movq \$0x400740, %r8
00000000:00400566		48 c7 c1 d0 06 40 00	movq \$0x4006d0, %rcx
00000000:0040056d		48 c7 c7 46 06 40 00	movq \$0x400646, %rdi
00000000:00400574		e8 87 ff ff ff	callq hello!__libc_start...
00000000:00400579		f4	hlt
00000000:0040057a		66 0f 1f 44 00 00	nopw (%rax, %rax)

(图 5.9 .text 节)

.data 节

00000000:00601048	hello!data_start	00 00	addb %al, (%rax)
00000000:0060104a		00 00	addb %al, (%rax)
00000000:0060104c		00 00	addb %al, (%rax)
00000000:0060104e		00 00	addb %al, (%rax)

(图 5.10 .data 节)

.bss 节

00000000:0060105c	hello!completed.7596	00 00	addb %al, (%rax)
00000000:0060105e		00 00	addb %al, (%rax)

(图 5.11 .bss 节)

查看虚拟地址段 0x600000~0x602000, 在 0~fff 段, 该段与 0x400000~0x401000 段的程序相同, fff 段后的是.dynamic~.shstrtab 节。

5.5 链接的重定位过程分析

文件内容

hello 和 hello.o 除了在反汇编生成的汇编代码有所不同, hello 的反汇编文件还在开头比 hello.o 多了 .init、.fini、.plt 和 .plt.got 节, 其中 .init 节是程序初始化需要执行的代码, .fini 是程序正常终止时需要执行的代码, .plt 和 .plt.got 节分别是动态链接中的过程链接表和全局偏移量表。而 hello.o 反汇编文件只有 Disassembly of section .text。链接器加入 _start、_init, __libc_csu_init, __libc_csu_fini, __libc_start_main 等函数。

函数调用

hello.o 反汇编文件中, call 地址后为占位符 (4 个字节的 0); 而 hello 在

生成过程中使用了动态链接共享库。`.text` 与 `.plt` 节相对距离已经确定，链接器计算相对距离，将对动态链接库中函数的调用值改为 PLT 中相应函数与下条指令的相对地址，指向对应函数。

数据访问

hello.o 反汇编文件中，对 `.rodata` 中 printf 的格式串的访问需要通过链接时重定位的绝对引用确定地址，因此在汇编代码相应位置仍为占位符表示，对 `.data` 中已初始化的全局变量 sleepsecs 为 0+%rip 的方式访问；而 hello 反汇编文件中对应全局变量已通过重定位绝对引用被替换为固定地址。`.rodata` 与 `.text` 节之间的相对距离确定，链接器直接修改 call 之后的值为目标地址与下一条指令的地址之差，指向相应的字符串。

hello.o 中 main 的地址从 0 开始，并且不存在调用的如 printf 这样函数的代码，很多地方都有重定位标记。而链接后的 hello 程序中 main 的地址不是 0。即链接后，增加了函数的个数，添加了头文件中的函数；确定了相对寻址，将动态库函数指向了 PLT；确定了函数的起始地址。

5.6 hello 的执行流程

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

(表 5.1 hello 的执行流程)

5.7 Hello 的动态链接分析

在调用共享库函数时，编译器没有办法预测这个函数的地址，所以为该引用生成一条重定位记录，然后动态链接器在程序加载的时候解析它。GNU 编译系统使用延迟绑定，将过程地址的绑定推迟到第一次调用该过程时，通过 GOT 和 PLT 实现。

PLT: PLT 是一个数组，其中每个条目是 16 字节代码。PLT[0] 指向动态链接器。每个被可执行程序调用的库函数都有 PLT 条目。每个条目都负责调用一个具体的函数。

GOT: GOT 也是一个数组，其中每个条目是 8 字节地址。和 PLT 联合使用时，GOT[0] 和 GOT[1] 包含动态链接器在解析函数地址时的有效信息。GOT[2] 是动态链接器在 ld-linux.so 模块中的入口点。其余的每个条目对应于一个被调用的函数，其地址在第一次调用时被解析，结束后将其指向正确的函数运行时地址。每个条目都有一个相匹配的 PLT 条目。

根据 hello ELF 文件可知，GOT 起始表位置为 0x601000。通过 data dump 观察，在调用函数 dl_init 之前 0x601008 后的 16 个字节均为 0：

00000000:00601000	28 0e 60 00 00 00 00 00	00 00 00 00 00 00 00 00	(.
00000000:00601010	00 00 00 00 00 00 00 00	a6 04 40 00 00 00 00 00@....

(图 5.12 调用函数前的 GOT 表)

根据.plt 中 exit@plt jmp 的引用地址 0x601030 可以得到其.got.plt 条目为 0x4004e6，正是其下条指令地址

调用_start 之后发生改变，0x601008 后的 16 字节发生改变，其中 GOT[0]（对应 0x600e28）和 GOT[1]（对应 0x7fb06087e168）包含动态链接器在解析函数地址时会使用的信息。GOT[2]（对应 0x7fb06066e870）是动态链接器在 ld-linux.so 模块中的入口点。

00000000:00601000	28 0e 60 00 00 00 00 00	68 e1 87 60 b0 7f 00 00	(.hc....
00000000:00601010	70 e8 66 60 b0 7f 00 00	a6 04 40 00 00 00 00 00	pjf.....
00000000:00601020	b6 04 40 00 00 00 00 00	c6 04 40 00 00 00 00 00@.....
00000000:00601030	d6 04 40 00 00 00 00 00	e6 04 40 00 00 00 00 00@.....
00000000:00601040	f6 04 40 00 00 00 00 00	00 00 00 00 00 00 00 00@.....

(图 5.13 调用函数后的 GOT 表)

GOT[2] 对应部分是正式动态链接库的入口地址，

00007fb0:6066e870	53	PUSH	RBX
00007fb0:6066e871	48 89 e3	MOV	RBX, RSP
00007fb0:6066e874	48 83 e4 e0	AND	RSP, 0xFFFFFFFFFFFFE0
00007fb0:6066e878	48 81 ec 80 01 0...	SUB	RSP, 0x180
00007fb0:6066e87f	48 89 84 24 40 0...	MOV	[RSP+0x140], RAX
00007fb0:6066e887	48 89 8c 24 48 0...	MOV	[RSP+0x148], RCX
00007fb0:6066e88f	48 89 94 24 50 0...	MOV	[RSP+0x150], RDX
00007fb0:6066e897	48 89 b4 24 58 0...	MOV	[RSP+0x158], RSI
00007fb0:6066e89f	48 89 bc 24 60 0...	MOV	[RSP+0x160], RDI
00007fb0:6066e8a7	4c 89 84 24 68 0...	MOV	[RSP+0x168], R8
00007fb0:6066e8af	4c 89 8c 24 70 0...	MOV	[RSP+0x170], R9
00007fb0:6066e8b7	c5 fd 7f 04 24	VMOVDQA	[RSP], YMM0
00007fb0:6066e8bc	c5 fd 7f 4c 24 20	VMOVDQA	[RSP+0x20], YMM1
00007fb0:6066e8c2	c5 fd 7f 54 24 40	VMOVDQA	[RSP+0x40], YMM2
00007fb0:6066e8c8	c5 fd 7f 5c 24 60	VMOVDQA	[RSP+0x60], YMM3
00007fb0:6066e8ce	c5 fd 7f a4 24 8...	VMOVDQA	[RSP+0x80], YMM4
00007fb0:6066e8d7	c5 fd 7f ac 24 a...	VMOVDQA	[RSP+0xA0], YMM5
00007fb0:6066e8e0	c5 fd 7f b4 24 c...	VMOVDQA	[RSP+0xC0], YMM6
00007fb0:6066e8e9	c5 fd 7f bc 24 e...	VMOVDQA	[RSP+0xE0], YMM7

(图 5.14 GOT[2] 对应部分)

成功完成链接。

5.8 本章小结

本章完成了对 `hello.o` 的链接操作。经过链接，ELF 可重定位的目标文件变成可执行的目标文件，链接器会将静态库代码写入程序中，以及动态库调用的相关信息，并且将地址进行重定位，从而保证寻址的正确进行，得到了可以执行的二进制文件。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

6.1.1 概念

进程的经典定义就是一个执行中的程序的实例。系统中的每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量、以及打开文件描述符的集合。

6.1.2 作用

给予应用程序关键抽象：

一个独立的逻辑流，它提供一个假象，好像我们的程序独占地使用处理器

一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统

6.2 简述壳 Shell-bash 的作用与处理流程

作用

shell 是一个交互型的应用级程序，它代表用户运行其他程序。shell 管理你与操作系统之间的交互，提供了你与操作系统之间通讯的方式。这种通讯可以以交互方式（从键盘输入，并且可以立即得到响应），或者以 shell script（非交互）方式执行。Shell 基本上是一个命令解释器。它接收用户命令（如 ls 等），然后调用相应的应用程序。

处理流程

shell 执行一系列的读 / 求值(read /evaluate) 步骤，然后终止。读步骤读取来自用户的一个命令行。求值步骤解析命令行，并代表用户运行程序。即等待你输入，向操作系统解释你的输入，并且处理各种各样的操作系统的输出结果。

6.3 Hello 的 fork 进程创建过程

shell 通过调用 fork 函数创建一个新的运行的子进程, 也就是 hello。hello 进程几乎但不完全与 shell 相同。hello 进程得到与 shell 用户级虚拟地址空间相同的 (但是独立的) 一份副本, 包括代码和数据段、堆、共享库以及用户栈。hello 进程还获得与 shell 任何打开文件描述符相同的副本, 这就意味着当 shell 调用 fork 时, hello 可以读写 shell 中打开的任何文件。shell 和 hello 进程之间最大的区别在于它们有不同的 PID。



(图 6.1 hello 的 fork 过程)

6.4 Hello 的 execve 过程

execve 函数加载并运行可执行目标文件 filename, 且带参数列表 argv 和环境变量列表 envp。只有当出现错误时, 例如找不到 filename, execve 才会返回到调用程序。所以, 与 fork 一次调用返回两次不同, execve 调用一次并从不返回。

参数列表如下图表示:

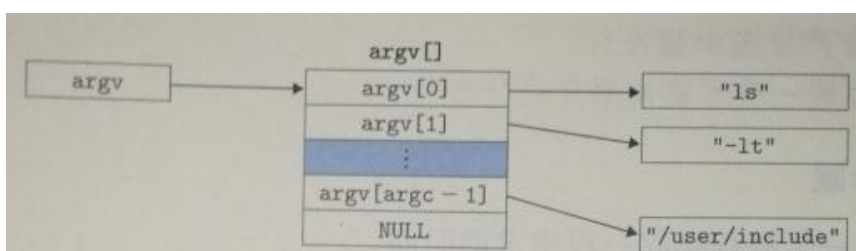


图 8-20 参数列表的组织结构

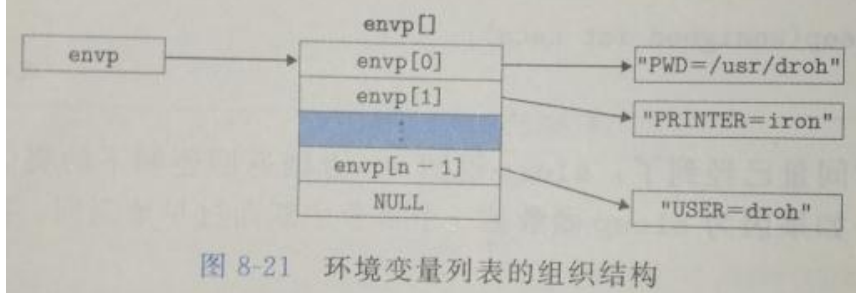


图 8-21 环境变量列表的组织结构

(图 6.2 参数列表、环境变量列表)

Execve 加载 filename 后, 调用启动代码, 启动代码设置栈, 并将控制传递给新程序的主函数。

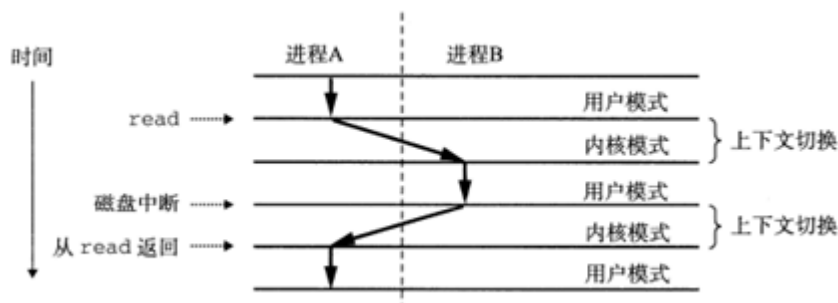
6.5 Hello 的进程执行

hello 在刚开始运行时内核为其保存一个上下文, 进程在用户状态下运行。如果没有异常或中断信号的产生, hello 正常运行。

若输入的参数个数不等于 3 个, 则输出一行字符串后调用 exit 函数正常退出, 程序产生一个终止异常, 程序结束。

否则执行 for 循环, 如果在调用 sleep 函数之前, 没有异常或系统中断, 则 hello 正常运行, 进行 sleep 系统调用, hello 显式地请求让其休眠, 进行上下文切换, 执行另一进程, 计时器开始计时, 记到 2.5s 时产生一个中断信号, 中断当前正在进行的进程, 再次进行上下文切换, 恢复 hello 休眠前的上下文信息, hello 继续执行。否则, 内核启用调度器休眠当前进程, 并在内核模式中进行上下文切换, 同时将控制权传递给这个被恢复的进程。

当循环结束后, 程序执行 getchar 函数, 通过执行系统调用的 read 来完成。read 产生一个中断, 内核调度进行上下文切换, hello 陷入内核。内核中的陷阱处理程序请求磁盘控制器的 DMA 传输, 并且在磁盘控制器完成从磁盘到内存的数据传输后, 发出中断信号, 中断处理器, 内核接收到这个中断, 进行上下文切换回到 hello 进程, hello 重新在用户模式下继续运行。在 return 语句后, 程序从主程序返回。



(图 6.3 hello 的进程执行)

6.6 hello 的异常与信号处理

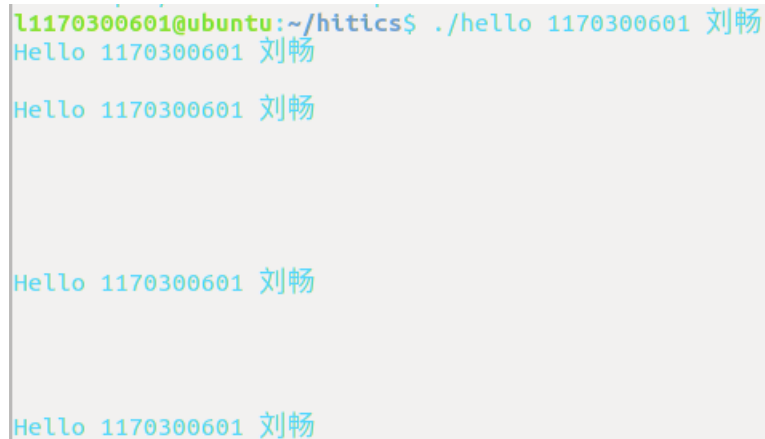
运行正常

A terminal window titled 'l1170300601@ubuntu: ~/hitics' with a menu bar (文件(F), 编辑(E), 查看(V), 搜索(S), 终端(T), 帮助(H)). The prompt is 'l1170300601@ubuntu:~/hitics\$'. The command './hello 1170300601 刘畅' has been executed, resulting in ten lines of output: 'Hello 1170300601 刘畅'.

(图 6.4 运行正常)

按下回车

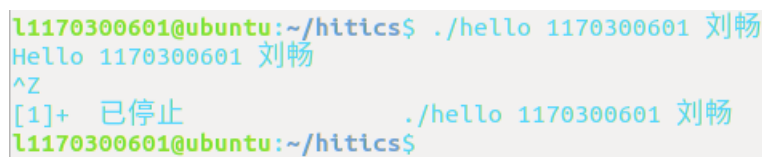
没啥反应，被忽略

A terminal window showing the same command './hello 1170300601 刘畅' as in Figure 6.4. The first line of output 'Hello 1170300601 刘畅' is visible. After pressing the carriage return, the prompt returns without any further output, indicating the process was ignored.

(图 6.5 按下回车)

Ctrl-Z

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。

A terminal window showing the command './hello 1170300601 刘畅' and the first line of output 'Hello 1170300601 刘畅'. Pressing Ctrl-Z results in '^Z' being displayed, followed by '[1]+ 已停止 ./hello 1170300601 刘畅'. The prompt then returns to 'l1170300601@ubuntu:~/hitics\$'.

(图 6.6 Ctrl-Z)

Ctrl-C

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。

```
l1170300601@ubuntu:~/hitics$ ./hello 1170300601 刘畅
Hello 1170300601 刘畅
Hello 1170300601 刘畅
^C
l1170300601@ubuntu:~/hitics$
```

(图 6.7 Ctrl-C)

Ctrl-Z+ps

hello 没有被回收

```
l1170300601@ubuntu:~/hitics$ ./hello 1170300601 刘畅
Hello 1170300601 刘畅
^Z
[1]+  已停止                  ./hello 1170300601 刘畅
l1170300601@ubuntu:~/hitics$ ps
  PID TTY          TIME CMD
  4271 pts/0        00:00:00 bash
  4280 pts/0        00:00:00 hello
  4281 pts/0        00:00:00 ps
l1170300601@ubuntu:~/hitics$
```

(图 6.8 Ctrl-Z+ps)

Ctrl-Z +jobs

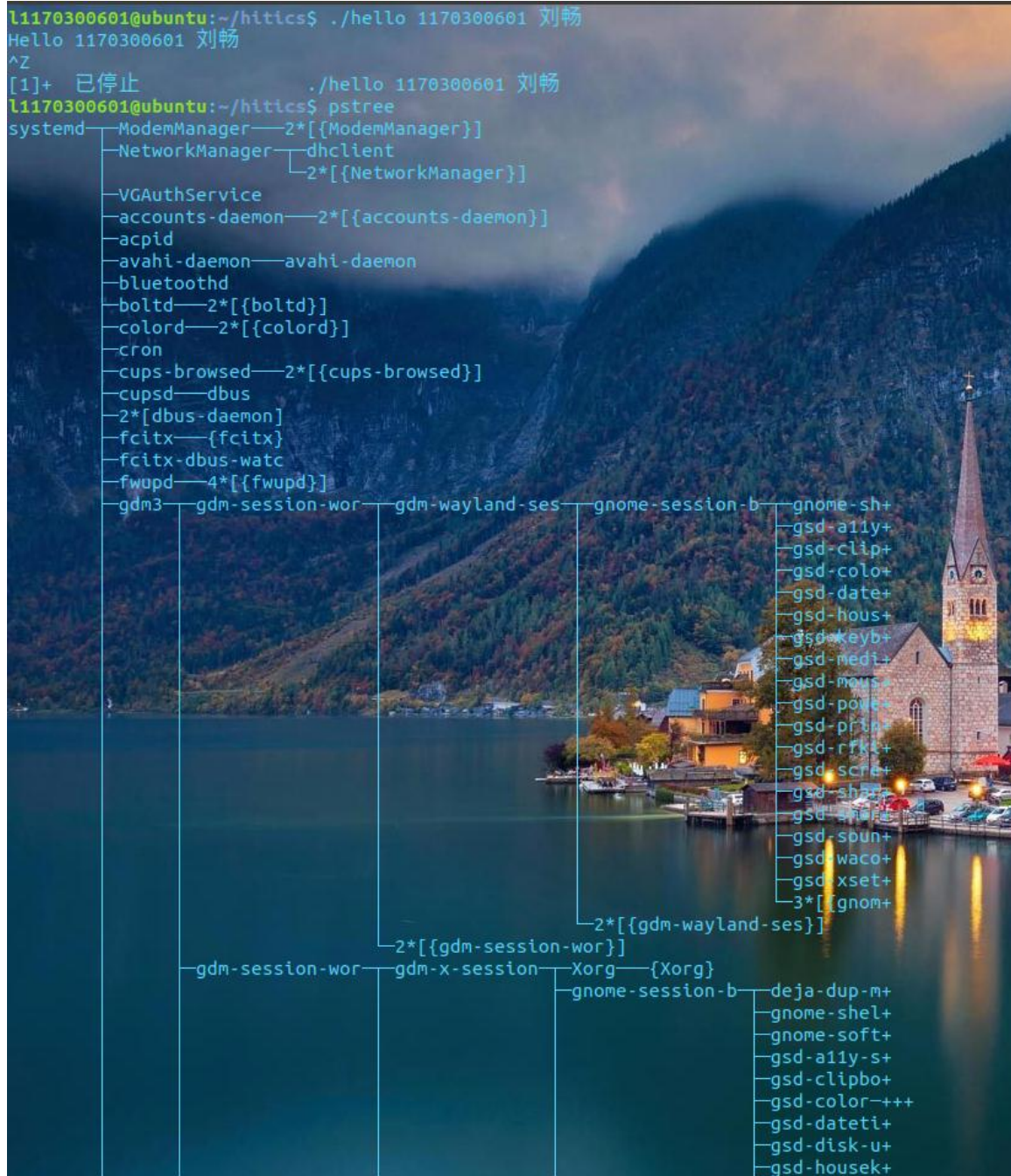
hello 被挂起

```
l1170300601@ubuntu:~/hitics$ ./hello 1170300601 刘畅
Hello 1170300601 刘畅
^Z
[1]+  已停止                  ./hello 1170300601 刘畅
l1170300601@ubuntu:~/hitics$ jobs
[1]+  已停止                  ./hello 1170300601 刘畅
l1170300601@ubuntu:~/hitics$
```

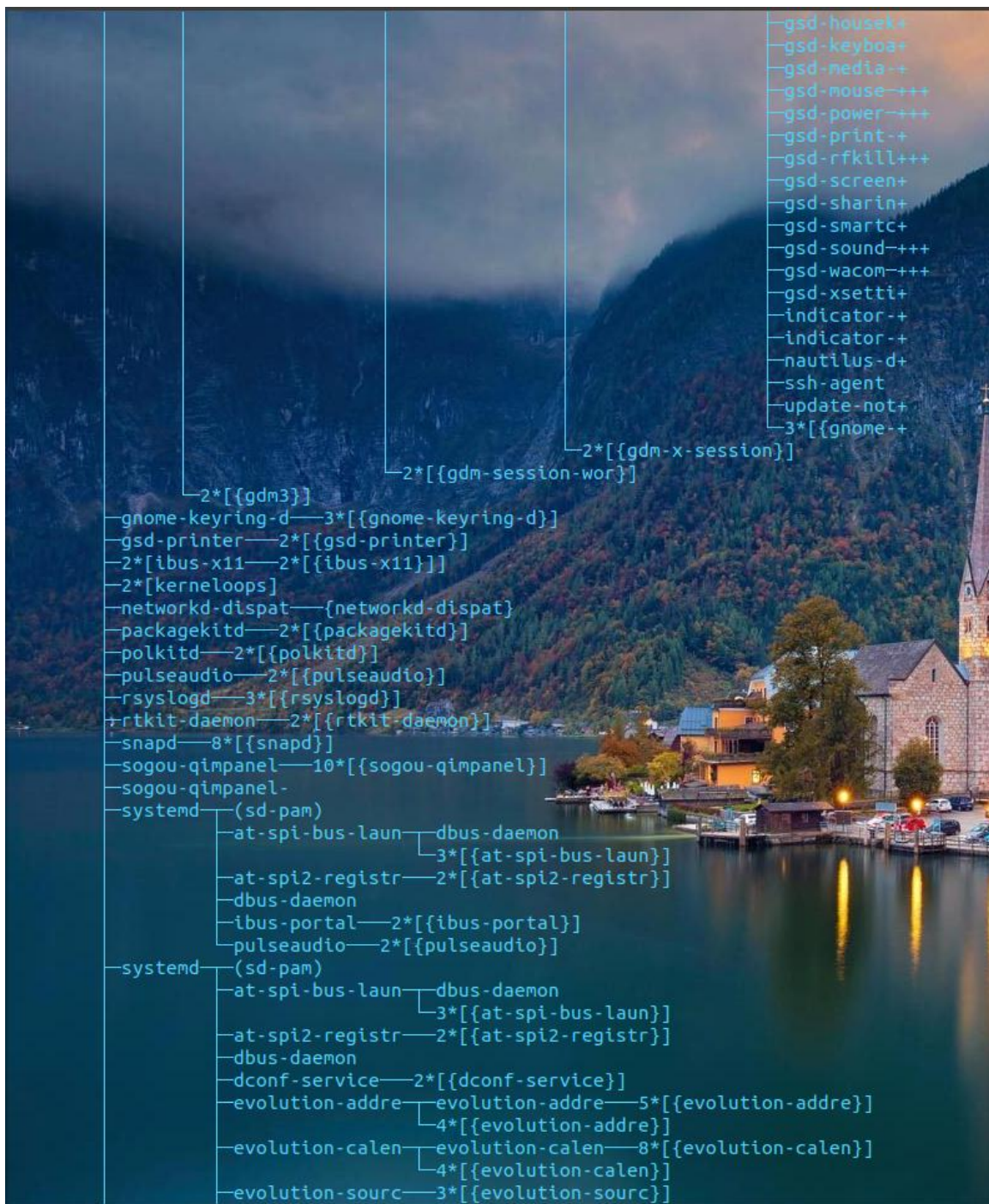
(图 6.9 Ctrl-Z +jobs)

Ctrl-Z +pstree

打印进程树



(图 6.10 Ctrl-Z +pstree)



(图 6.10 续)

```

gnome-shell-cal—5*[{gnome-shell-cal}]
gnome-terminal—bash—hello
                  |—pstree
                  |—4*[{gnome-terminal-}]
goa-daemon—4*[{goa-daemon}]
goa-identity-se—3*[{goa-identity-se}]
gvfs-afc-volume—3*[{gvfs-afc-volume}]
gvfs-goa-volume—2*[{gvfs-goa-volume}]
gvfs-gphoto2-vo—2*[{gvfs-gphoto2-vo}]
gvfs-mtp-volume—2*[{gvfs-mtp-volume}]
gvfs-udisks2-vo—2*[{gvfs-udisks2-vo}]
gvfsd—gvfsd-dnssd—2*[{gvfsd-dnssd}]
      |—gvfsd-network—3*[{gvfsd-network}]
      |—gvfsd-recent—2*[{gvfsd-recent}]
      |—gvfsd-trash—2*[{gvfsd-trash}]
      |—2*[{gvfsd}]
gvfsd-fuse—5*[{gvfsd-fuse}]
gvfsd-metadata—2*[{gvfsd-metadata}]
ibus-portal—2*[{ibus-portal}]
nautilus—4*[{nautilus}]
zeitgeist-daemo—2*[{zeitgeist-daemo}]
zeitgeist-fts—2*[{zeitgeist-fts}]

systemd-journal
systemd-logind
systemd-resolve
systemd-timesyn—{systemd-timesyn}
systemd-udev
udisksd—4*[{udisksd}]
upowerd—2*[{upowerd}]
vmhgfs-fuse—11*[{vmhgfs-fuse}]
vmtoolsd—{vmtoolsd}
vmtoolsd—3*[{vmtoolsd}]
vmware-vmblock—2*[{vmware-vmblock-}]
whoopsie—2*[{whoopsie}]
wpa_supplicant

l1170300601@ubuntu:~/hitics$

```

(图 6.10 续)

Ctrl-Z+fg

发送 SIGCONT 信号继续执行停止的进程

```

l1170300601@ubuntu: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
l1170300601@ubuntu:~/hitics$ ./hello 1170300601 刘畅
Hello 1170300601 刘畅
^Z
[1]+  已停止                  ./hello 1170300601 刘畅
l1170300601@ubuntu:~/hitics$ fg
./hello 1170300601 刘畅
Hello 1170300601 刘畅
Hello 1170300601 刘畅
Hello 1170300601 刘畅

```

(图 6.11 Ctrl-Z+fg)

Ctrl-Z +kill

杀死进程



```
l1170300601@ubuntu: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
l1170300601@ubuntu:~/hitics$ ./hello 1170300601 刘畅
Hello 1170300601 刘畅
Hello 1170300601 刘畅
^Z
[1]+  已停止                  ./hello 1170300601 刘畅
l1170300601@ubuntu:~/hitics$ ps
  PID TTY          TIME CMD
  4447 pts/0        00:00:00 bash
  4455 pts/0        00:00:00 hello
  4456 pts/0        00:00:00 ps
l1170300601@ubuntu:~/hitics$ kill -9 4455
[1]+  已杀死                  ./hello 1170300601 刘畅
l1170300601@ubuntu:~/hitics$ ps
  PID TTY          TIME CMD
  4447 pts/0        00:00:00 bash
  4457 pts/0        00:00:00 ps
l1170300601@ubuntu:~/hitics$
```

(图 6.12 Ctrl-Z +kill)

不停乱按

没啥反应，被忽略



```
l1170300601@ubuntu: ~/hitics
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
l1170300601@ubuntu:~/hitics$ ./hello 1170300601 刘畅
Hello 1170300601 刘畅
6848Hello 1170300601 刘畅
scbsiaHello 1170300601 刘畅
jls olnHello 1170300601 刘畅
s lsacn slHello 1170300601 刘畅
js nlHello 1170300601 刘畅
Hello 1170300601 刘畅
```

(图 6.13 不停乱按)

6.7 本章小结

本章简单介绍了进程的概念和作用。简述了 shell 的作用与处理流程，同时陈述了 fork 以及 execve 函数的具体流程。最后通过实例展现 hello 的异常与信号处理过程。

(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

7.1.1 逻辑地址

hello.c 经过编译后出现在 hello.o 中地址。分为两个部分，一个部分为段基址，另一个部分为段偏移量

7.1.2 线性地址

地址空间是一个非负整数地址的有序集合，而如果此时地址空间中的整数是连续的，则我们称这个地址空间为线性地址空间，这里指 hello 里面的虚拟内存地址。

7.1.3 虚拟地址

与物理地址相似，虚拟内存被组织为一个存放在磁盘上的 N 个连续的字节大小的单元组成的数组，其每个字节对应的地址成为虚拟地址。虚拟地址包括 VPO（虚拟页面偏移量）、VPN（虚拟页号）、TLBI（TLB 索引）、TLBT（TLB 标记），这里也是 hello 里面的虚拟内存地址。

7.1.4 物理地址

CPU 通过地址总线的寻址，找到真实的物理内存对应地址。CPU 对内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址。这里是 hello 在运行时虚拟内存地址对应的物理地址。

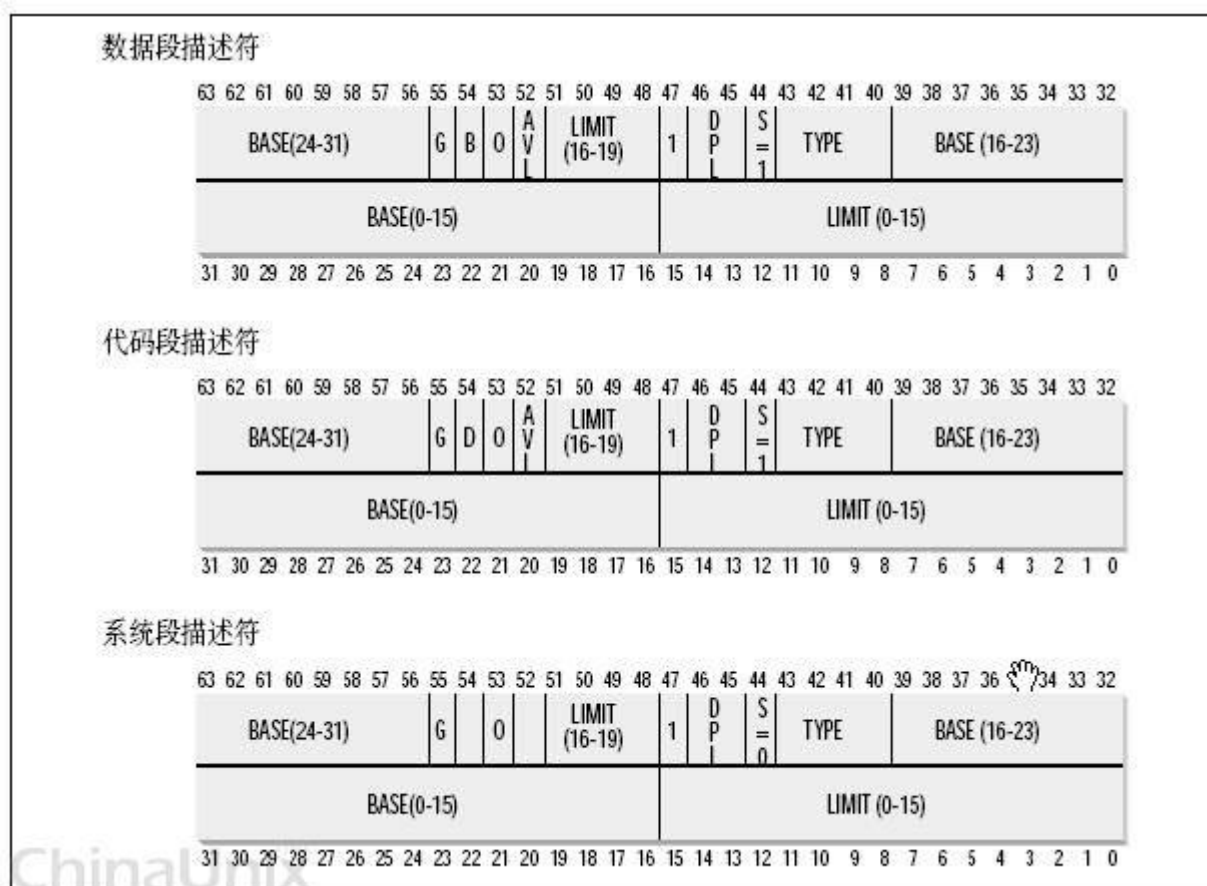
7.2 Intel 逻辑地址到线性地址的变换-段式管理

逻辑地址如何转换为线性地址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：



(图 7.1 段标识符)

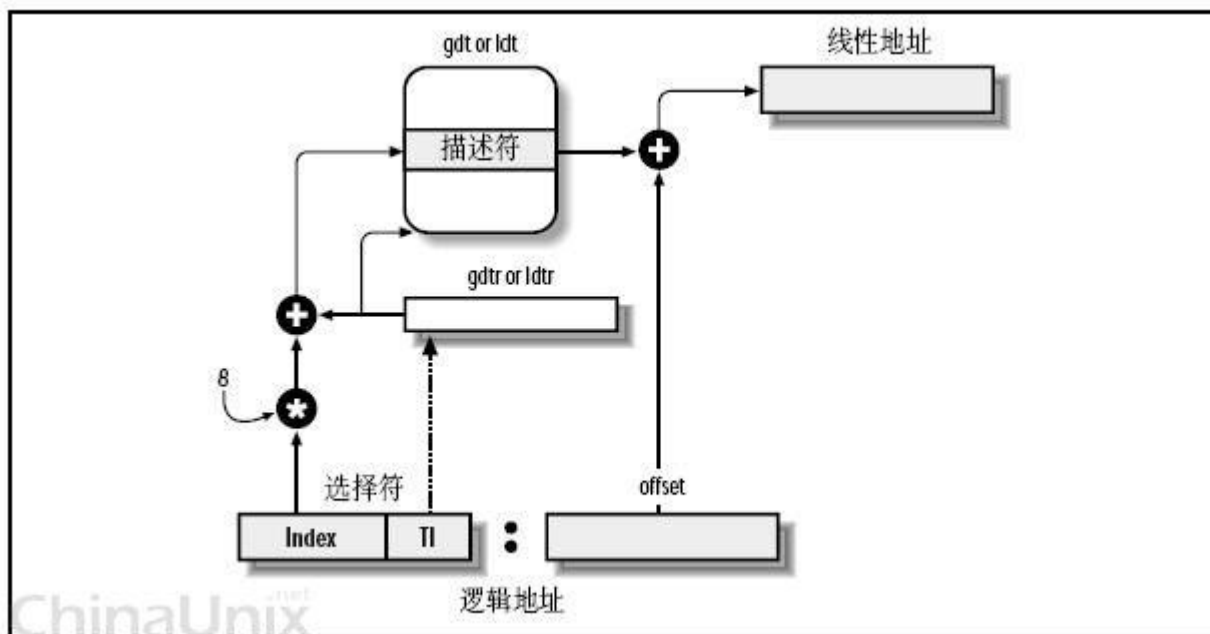
索引号，或者直接理解成数组下标——那它总要对应一个数组吧，它又是什么索引呢？这是“段描述符”，段描述符具体地址描述了一个段。这样，很多个段描述符，就组了一个数组，叫“段描述符表”，这样，可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，这个描述符就描述了一个段，由 8 个字节组成，如下图：



(图 7.2 段描述符表)

图示比较复杂，可以利用一个数据结构来定义它，不过，在此只关心一样，就是 Base 字段，它描述了一个段的开始位置的线性地址。Intel 设计的本意是，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。那究竟什么时候该用 GDT，什么时候该用 LDT 呢？这是由段选择符中的 T1 字段表示的，=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。再看这张图比起来要直观些：



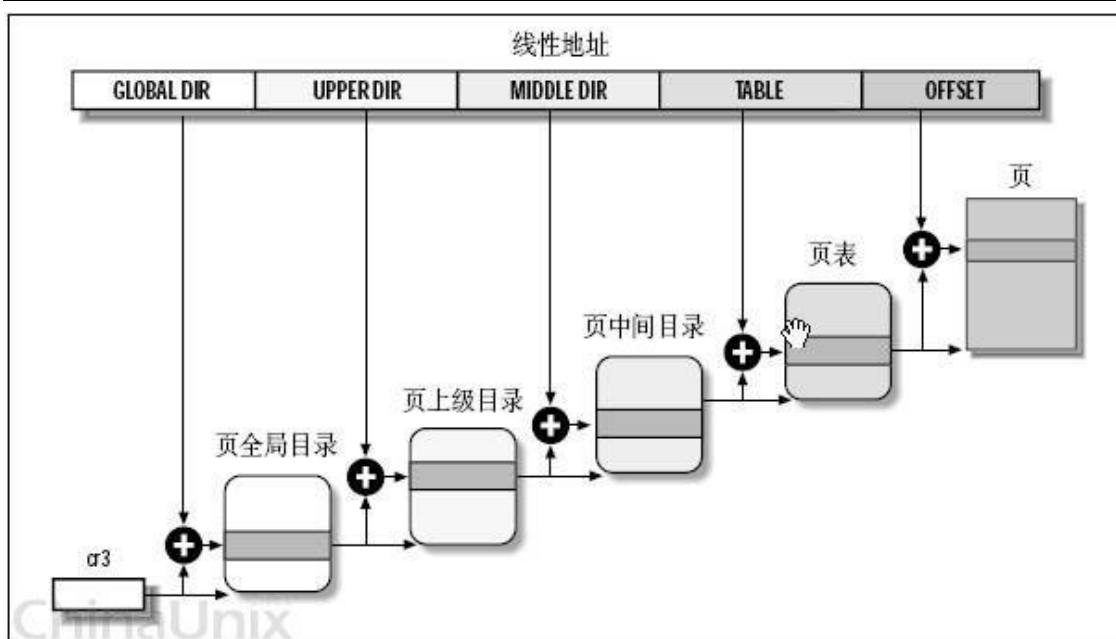
(图 7.3 翻译流程)

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。我们就有了一个数组了。
- 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，这样，它了 Base，即基地址就知道了。
- 3、把 Base + offset，就是要转换的线性地址了。还是挺简单的，对于软件来讲，原则上就需要把硬件转换所需的信息准备好，就可以让硬件来完成这个转换了。

7.3 Hello 的线性地址到物理地址的变换-页式管理

原理上来讲，Linux 只需要为每个进程分配好所需数据结构，放到内存中，然后在调度进程的时候，切换寄存器 cr3，剩下的就交给硬件来完成了（事实上要复杂得多，在此只分析最基本的流程）。前面说了 i386 的二级页管理架构，不过有些 CPU，还有三级，甚至四级架构，Linux 为了在更高层次提供抽象，为每个 CPU 提供统一的界面。提供了一个四层页管理架构，来兼容这些二级、三级、四级管理架构的 CPU。这四级分别为：页全局目录 PGD、页上级目录 PUD、页中间目录 PMD、页表 PT。整个转换依据硬件转换原理，只是多了二次数组的索引罢了，如下图：



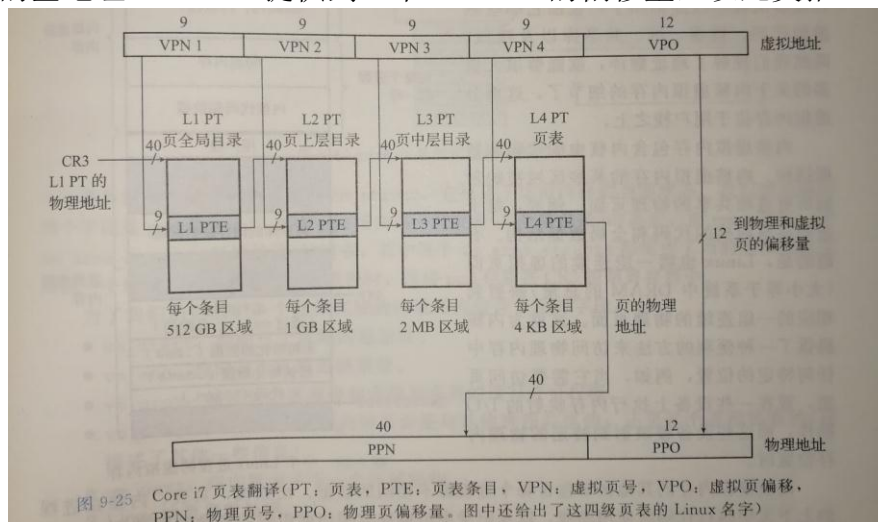
(图 7.4 Hello 的线性地址到物理地址的变换-页式管理)

那么,对于使用二级管理架构 32 位的硬件,四级转换怎么能够协调地工作呢? 嗯,来看这种情况下,怎么来划分线性地址吧!从硬件的角度,32 位地址被分成了三部份;从软件的角度,由于多引入了两部份,也就是说,共有五部份。——要让二层架构的硬件认识五部份也很容易,在地址划分的时候,将页上级目录和页中间目录的长度设置为 0 就可以了。这样,操作系统见到的是五部份,硬件还是按它死板的三部份划分,也就共建了和谐计算机系统。这样,虽说是多此一举,但是考虑到 64 位地址,使用四层转换架构的 CPU,此时不再把中间两个设为 0 了,这样,软件与硬件再次共建了和谐计算机系统!

现在来理解 Linux 高招,因为硬件根本看不到所谓 PUD, PMD, 所以,本质上要求 PGD 索引,直接就对应了 PT 的地址。而不是再到 PUD 和 PMD 中去查数组(虽然它们两个在线性地址中,长度为 0, $2^0=1$, 也就是说,它们都是有一个数组元素的数组),那么,内核如何合理安排地址呢?从软件的角度上来讲,因为它的项只有一个,32 位,刚好可以存放与 PGD 中长度一样的地址指针。那么所谓先到 PUD, 到到 PMD 中做映射转换,就变成了保持原值不变,——转手就可以了。这样,就实现了“逻辑上指向一个 PUD,再指向一个 PDM,但在物理上是直接指向相应的 PT 的这个抽象,因为硬件根本不知道有 PUD、PMD 这个东西”。然后交给硬件,硬件对这个地址进行划分,看到的是: 页目录= 0000100000 PT = 0101000111 offset = 001001011000 嗯,先根据 0000100000(32),在页目录数组中索引,找到其元素中的地址,取其高 20 位,找到页表的地址,页表的地址是由内核动态分配的,接着,再加一个 offset,就是最终的物理地址了。

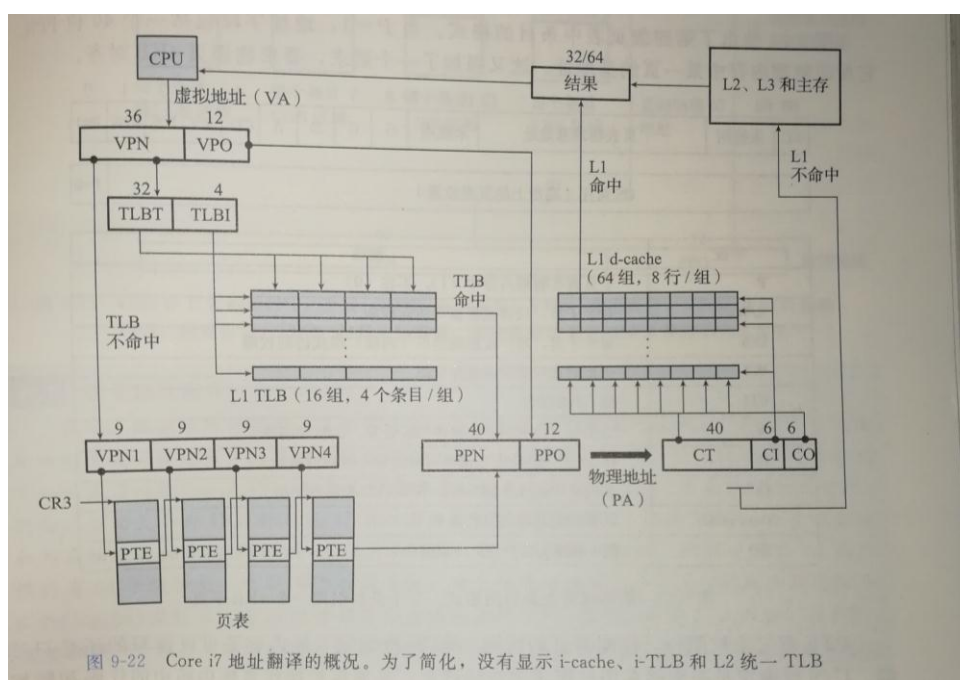
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

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 7.5 TLB 与四级页表支持下的 VA 到 PA 的变换)

7.5 三级 Cache 支持下的物理内存访问



(图 7.6 三级 Cache 支持下的物理内存访问)

首先 CPU 发出一个虚拟地址，在 TLB 里面寻找。如果命中，那么将 PTE 发送给 L1Cache，否则先在页表中更新 PTE。然后再进行 L1 根据 PTE 寻找物理地址，检测是否命中的工作。这样就能完成 Cache 和 TLB 的配合工作。

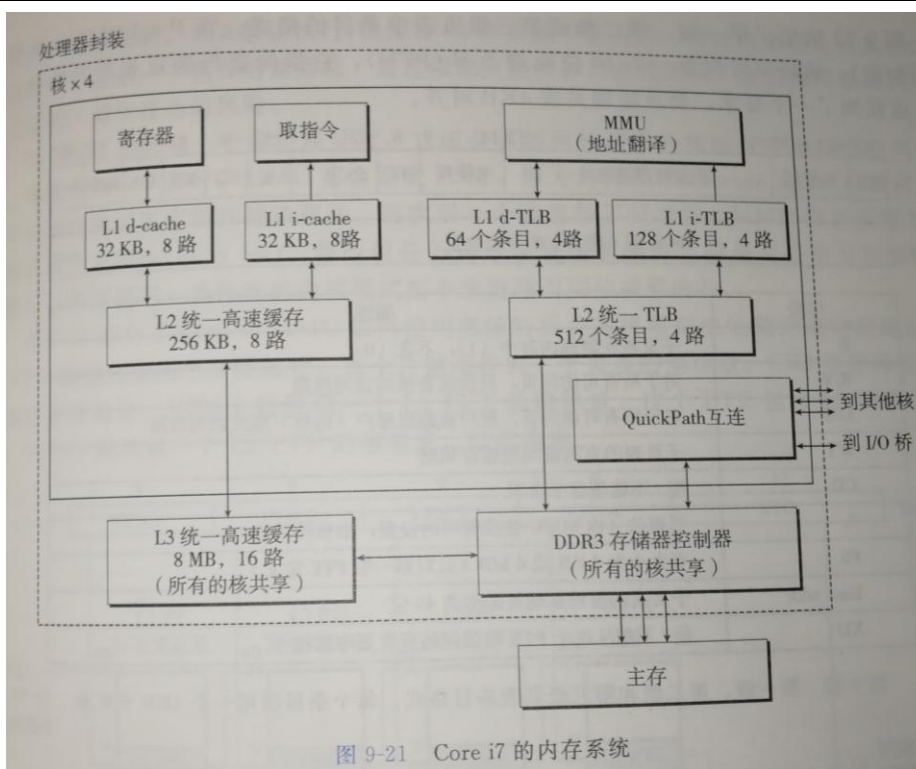


图 9-21 Core i7 的内存系统

(图 7.7 内存系统)

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 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。

当 fork 在 hello 进程中返回时，hello 进程现在的虚拟内存刚好和调用 fork 时存在的虚拟内存相同。当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间的抽象概念。

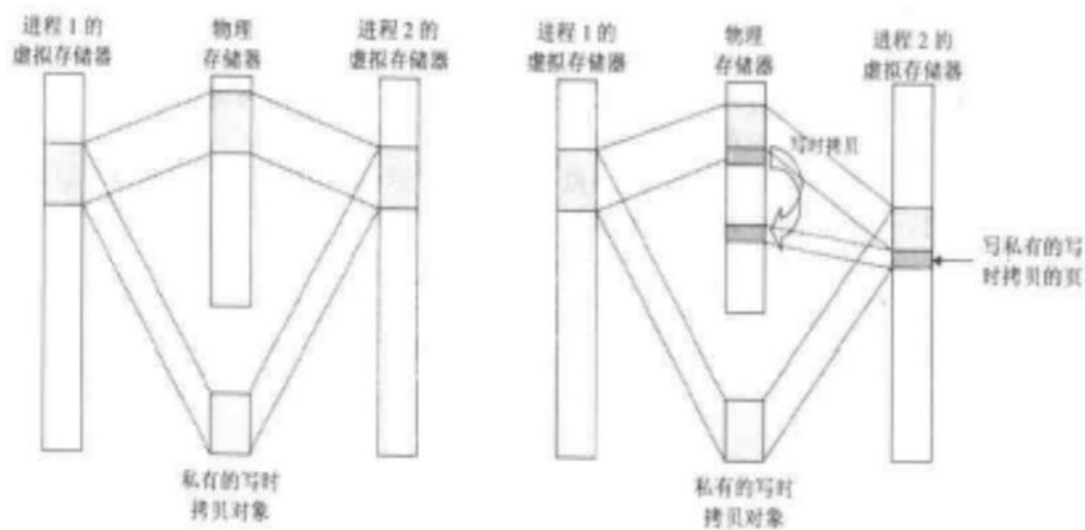


图 10.32 一个私有的写时拷贝对象

(图 7.8 hello 进程 fork 时的内存映射)

7.7 hello 进程 execve 时的内存映射

execve 函数在当前进程中加载并运行新程序 hello.out 的步骤：首先删除已存在的用户区域，也就是将 shell 与 hello 都有的区域结构删除。然后映射私有区域，即为新程序的代码、数据、bss 和栈区域创建新的区域结构，代码和初始化数据映射到 .text 和 .data 区（目标文件提供），.bss 和栈映射到匿名文件。下一步是映射共享区域，将一些动态链接库映射到 hello 的虚拟地址空间，最后设置 PC，使之指向 hello 程序的代码入口。



图 10.33 加载器是如何映射用户地址空间的区域的

(图 7.9 hello 进程 execve 时的内存映射)

7.8 缺页故障与缺页中断处理

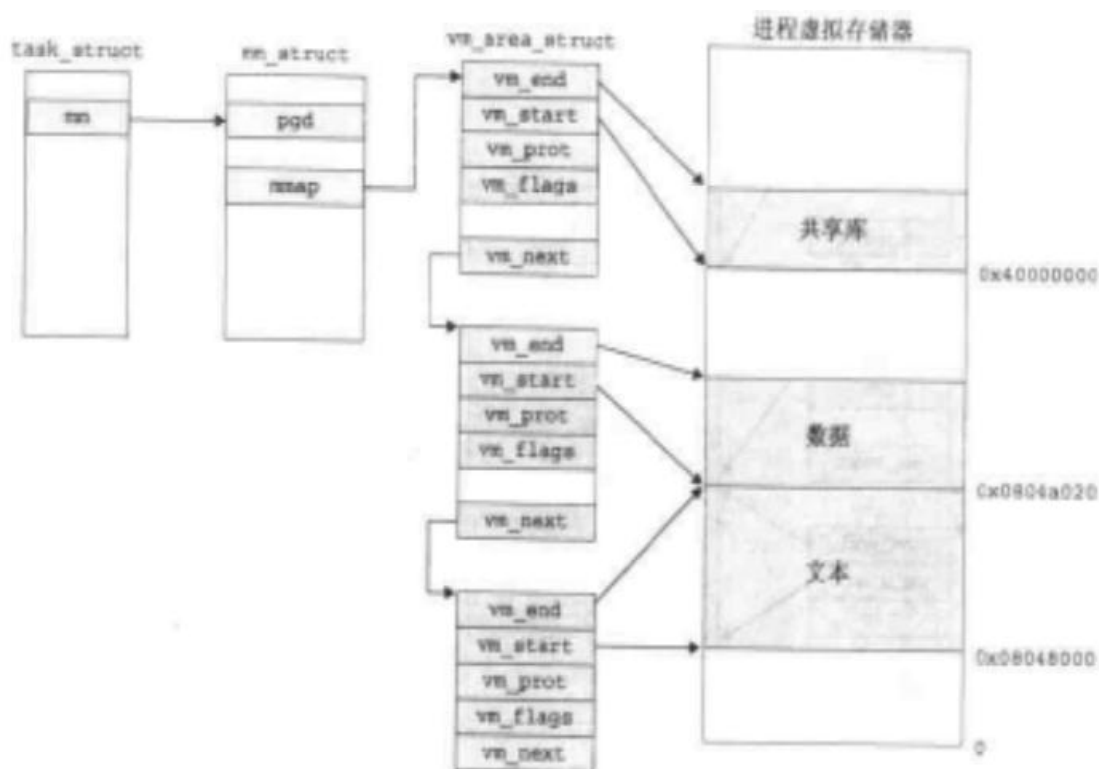


图 10.29 Linux 是如何组织虚拟存储器的

(图 7.10 linux 下虚拟存储器)

假设 MMU 在试图翻译某个虚拟地址 A 时，触发了一个缺页。这个异常导致控制转移到内核的缺页处理程序，处理程序随后就执行下面的步骤：

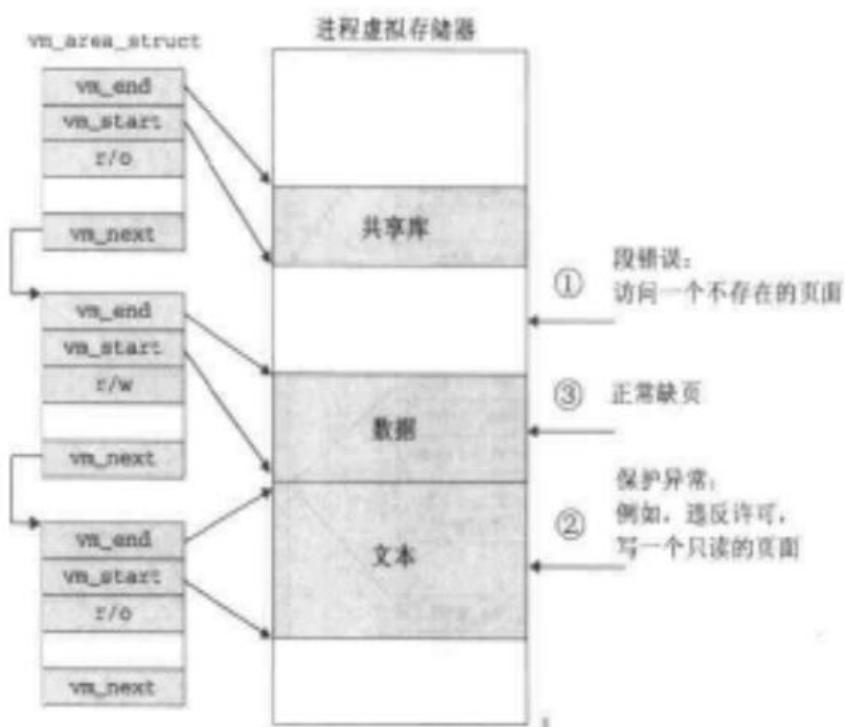
1. 虚拟地址 A 是合法的吗？换句话说，A 在某个区域结构定义的区域吗？为了回答这个问题，缺页处理程序搜索区域结构的链表，把 A 和每个区域结构中的 `vm_start` 和 `vm_end` 做比较。如果这个指令是不合法的，那么缺页处理程序就触发一个段错误，从而终止这个进程。

因为一个进程可以创建任意数量的新虚拟存储器区域，所以顺序搜索区域结构的链表开销可能会很大。因此在实际中，使用某些我们没有显示出来的字段，Linux 在链表中添加了一棵树，并在这棵树上进行查找。

2. 试图进行的对存储器的访问是否合法？换句话说，进程是否有读或者写这个区域内页面的权限？例如，这个缺页是不是由一条试图对这个代码段里的只读页面进行写操作的存储指令造成的？这个缺页是不是因为一个运行在用户模式中的进程试图从内核虚拟存储器中读取字造成的？如果试图进行的访问是不合法的，那么缺页处理程序会触发一个保护异常，从而终止这个进程。

3. 此刻，内核知道了这个缺页是由于对合法的虚拟地址进行合法的操作造成的。它选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，

换入新的页面，并更新页表，从而处理这次缺页。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令将再次发送 A 到 MMU，这一回，MMU 就能正常地翻译 A，而不会再产生一个缺页中断了。



(图 7.11 linux 缺页处理)

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)。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，它指向堆的顶部。

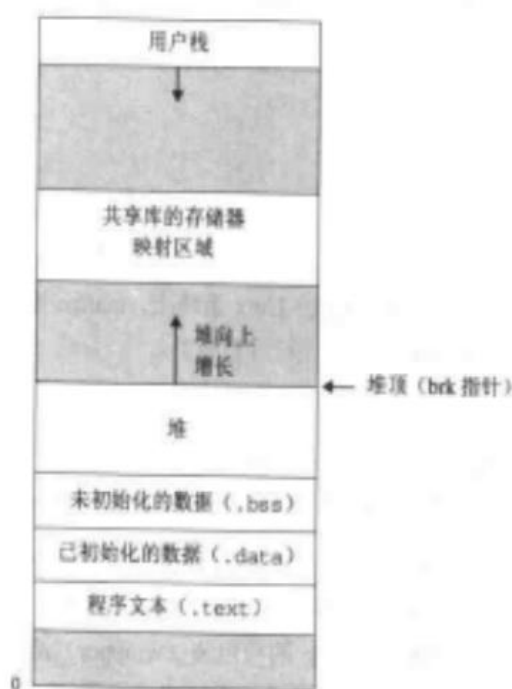


图 10.35 堆

(图 7.12 一个堆的基本形式)

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。

显式分配器: 要求应用显式地释放任何已分配的块。

隐式分配器: 要求分配器检测一个已分配块何时不再使用, 那么就释放这个块, 自动释放未使用的已经分配的块的过程叫做垃圾收集。而自动释放未使用的已分配的块的过程叫做垃圾收集。

内存分配方式

在操作系统中, 内存分配主要以下面三种方式存在:

(1) 静态存储区域分配。

内存存在程序编译的时候或者在操作系统初始化的时候就已经分配好, 这块内存存在程序的整个运行期间都存在, 而且其大小不会改变, 也不会被重新分配。例如全局变量, static 变量等。

(2) 栈上的内存分配。

栈是系统数据结构, 对于进程/线程是唯一的, 它的分配与释放由操作系统来维护, 不需要开发者来管理。在执行函数时, 函数内局部变量的存储单元都可以在栈上创建, 函数执行结束时, 这些存储单元会被自动释放。栈内存分配运算内

置于处理器的指令集中，效率很高，不同的操作系统对栈都有一定的限制。

(3) 堆上的内存分配，亦称动态内存分配。

程序在运行的期间用 malloc 申请的内存，这部分内存由程序员自己负责管理，其生存期由开发者决定：在何时分配，分配多少，并在何时用 free 来释放该内存。这是唯一可以由开发者参与管理的内存。使用的好坏直接决定系统的性能和稳定。

动态内存管理方法

在 Linux 下，glibc 的 malloc 提供了下面两种动态内存管理的方法：堆内存分配和 mmap 的内存分配，此两种分配方法都是通过相应的 Linux 系统调用来进行动态内存管理的。具体使用哪一种方式分配，根据 glibc 的实现，主要取决于所需分配内存的大小。一般情况下，应用层面的内存从进程堆中分配，当进程堆大小不够时，可以通过系统调用 brk 来改变堆的大小，但是在以下情况，一般由 mmap 系统调用来实现应用层面的内存分配：A、应用需要分配大于 1M 的内存，B、在没有连续的内存空间能满足应用所需大小的内存时。

(1)、调用 brk 实现进程里堆内存分配

在 glibc 中，当进程所需要的内存较小时，该内存会从进程的堆中分配，但是堆分配出来的内存空间，系统一般不会回收，只有当进程的堆大小到达最大限额时或者没有足够连续大小的空间来为进程继续分配所需内存时，才会回收不用的堆内存。在这种方式下，glibc 会为进程堆维护一些固定大小的内存池以减少内存碎片。

(2)、使用 mmap 的内存分配

在 glibc 中，一般在比较大的内存分配时使用 mmap 系统调用，它以页为单位来分配内存的（在 Linux 中，一般一页大小定义为 4K），这不可避免会带来内存浪费，但是当进程调用 free 释放所分配的内存时，glibc 会立即调用 unmmap，把所分配的内存空间释放回系统。

注意：这里我们讨论的都是虚拟内存的分配（即应用层面上的内存分配），主要由 glibc 来实现，它与内核中实际物理内存的分配是不同的层面，进程所分配到的虚拟内存可能没有对应的物理内存。如果所分配的虚拟内存没有对应的物理内存时，操作系统会利用缺页机制来为进程分配实际的物理内存。

带边界标签的隐式空闲链表分配器原理：

1. 通用分配器设置

最小块的大小为 16 字节，空闲链表组织成为一个隐式空闲链表，第一个字是一个双字边界对齐的不使用的填充字。填充字后面紧跟着一个特殊的序言块。序言块是在初始化时创建的，并且永不释放。在序言块后紧跟的是零个或者多个有 malloc 或者 free 调用创建的普通块。堆总以一个特殊的结尾块来结束。序言块和结尾块是一种消除合并时边界条件的技巧。分配器使用一个单独的私有全局变量，它总是指向序言块。

2. 操作空闲链表的基本常数和宏

在空闲链表中操作头部和脚部可能是很麻烦的，因为它要求大量使用强制类型转换和指针运算。我们发现定义一小组宏来访问和遍历空闲链表是很有帮助的。

3. 创建初始空闲链表

在调用 `mm_malloc` 与 `mm_free` 之前，应用必须通过调用 `mm_init` 函数来初始化堆。分配器初始化之后，准备好接受来自应用的分配和释放请求。

4. 释放和合并块

应用通过调用 `mm_free` 函数来释放一个以前分配的块，这个函数释放所请求的块，然后使用边界标记合并技术将与之邻接的空闲块合并起来。

5. 分配块

一个应用通过调用 `mm_malloc` 函数来向内存请求大小为 `size` 字节的块。检查完请求的真假之后，分配器必须调整请求块的大小，从而为头部和脚部留有空间，并满足双字对齐的要求。

一旦分配器调整了请求的大小，它就会搜索空闲链表，寻找一个合适的空闲块。如果有合适的，那么分配器就放置这个请求块，并可选的分割出多余的部分，然后返回新分配块的地址。

如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆，把请求块放置在这个新的空闲块里，可选的分割这个块然后返回一个指针，指向这个新分配的块。

显示空间链表的基本原理

隐式空闲链表为我们提供了一种简单的介绍一些基本分配器概念的方法。然而，因为块分配与堆块的总数呈线性关系，所以对于通用的分配器隐式空闲链表是不适合的（尽管对于堆块数量预先就知道是很小的特殊的分配器来说，它是比较好的）。

一种更好的方法是将空闲块组织为某种形式的显式数据结构。因为根据定义，程序是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` 和 `suec` 指针。如图

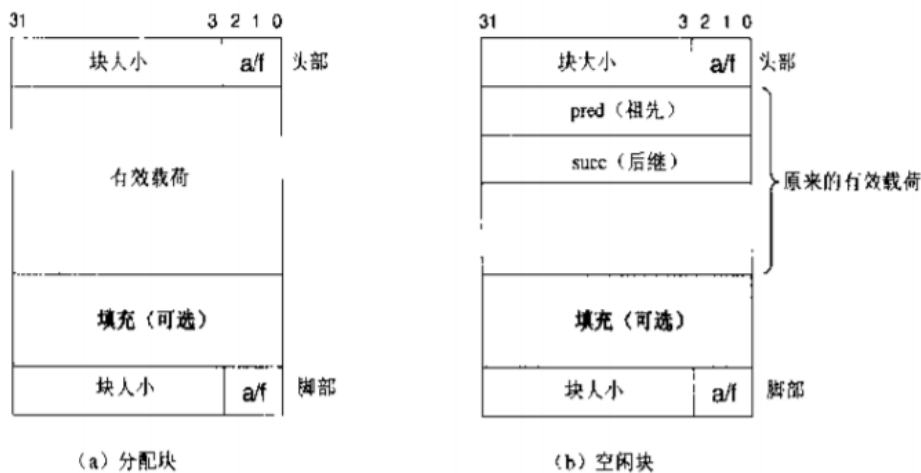


图 10.50 使用双向空闲链表的堆块的格式

(图 7.13 显式空闲链表)

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。这取决于我们在空闲链表中对块排序所选择的策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以存常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索，来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的存储器利用率，接近最佳适配的利用率。

一般而言，显式链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部。这就导致更大的最小块大小，也潜在地提高了内部碎片的程度。

7.10 本章小结

本章简单介绍了 hello 的存储器地址空间，阐述了 Intel 的段式管理和页式管理机制，以及 TLB 与多级页表支持下的 VA 到 PA 的转换，同时对 cache 支持下的物理内存访问做了说明。对 hello 的 fork 与 execve 做了探讨，介绍了有关缺页故障与缺页中断处理及动态存储分配管理方面的问题。

(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 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。

8.2 简述 Unix IO 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 $k \sim m$ 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。

Unix I/O 函数：

1. Open

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);
```

`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。

返回：若成功则为新文件描述符，若出错为-1。

2. close

进程通过调用 `close` 函数关闭一个打开的文件。

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

关闭一个已关闭的描述符会出错。

返回：若成功则为 0，若出错则为-1。

3. read 与 write

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);
```

`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。

返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1。

```
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 `fd` 的当前文件位置。

返回：若成功则为写的字节数，若出错则为-1。

8.3 printf 的实现分析

要分析 `printf` 函数的实现那么就需要了解 Linux 下 `printf` 函数的实现：


```
static int printf(const char *fmt, ...)
{
    va_list args;
    int i;
    va_start(args, fmt);
    write(1, printbuf, i=vsprintf(printbuf, fmt, args));
    va_end(args);
    return i;
}
```

(图 8.1 printf 函数原型)

然后查看

```
void va_start( va_list arg_ptr, prev_param );
```

```
type va_arg( va_list arg_ptr, type );
```

```
void va_end( va_list arg_ptr );
```

再查看 vsprintf 函数，发现 vsprintf 的作用是格式化。即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，从而产生格式化输出。

然后查看 write 函数，

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

(图 8.2 write 函数原型)

可知内核向寄存器传递几个参数后，调用了 syscall 函数。

最后查看 syscall 函数，

```
sys_call:
    call save
    push dword [p_proc_ready]
    sti
    push ecx
    push ebx
    call [sys_call_table + eax * 4]
    add esp, 4 * 3
    mov [esi + EAXREG - P_STACKBASE], eax
    cli
    ret
```

(图 8.3 sys_call 函数原型)

可知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）。

8.4 getchar 的实现分析

查看 getchar 源代码

```
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;
}
```

（图 8.4 getchar 函数原型）

getchar 函数通过调用 read 函数返回字符。其中 read 函数的第一个参数是描述符，0 代表标准输入。第二个参数输入内容的指针，这里也就是字符的地址。read 函数的返回值是读入的字符数，如果为 1 说明读入成功，那么直接返回字符，否则说明读到了 buf 的最后。

read 函数同样通过 sys_call 中断来调用内核中的系统函数。键盘中断处理程序会接受按键扫描码并将其转换为 ASCII 码后保存在缓冲区。然后 read 函数调用的系统函数可以对缓冲区 ASCII 码进行读取，直到接受回车键返回。

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

本章简要介绍了 linux 的 I/O 设备管理方法，概述了 I/O 对接口以及操作方法，对 printf 和 getchar 函数进行了实现分析。

(第 8 章 1 分)

结论

编写 hello.c	通过 ide 编写 hello.c 。
预处理过程	对 hello.c 中的预处理指令进行处理，将 hello.c 调用的所有外部的库展开合并到一个 hello.i 文件中。
编译过程	将 hello.i 翻译成 hello.s。
汇编过程	将 hello.s 翻译为机器语言指令保存至二进制可重定位目标文件 hello.o。
链接过程	将 hello.o 与动态链接库进行动态链接成为可执行目标程序 hello
创建子进程	shell 进程调用 fork 为其创建子进程
运行程序	shell 调用 execve，execve 调用启动加载器，重构映射虚拟内存，进入程序入口后程序开始载入物理内存，然后进入 main 函数。
执行指令	CPU 为其分配时间片，hello 在时间片中顺序执行。
访问内存	MMU 处理虚拟内存地址并通过多级页表以及 TLB 等将其翻译为物理地址，再根据物理地址访问内存取出内容
动态内存申请	程序调用 malloc 函数向动态内存分配器申请堆中的内存
接收信号	调用 shell 的信号处理函数处理 hello 运行过程中接收到的各种信号。
回收	shell 父进程回收子进程，删除与 hello 有关的所有数据结构以及占用的内存资源

(附表 1: 结论)

感悟：hello 的成功运行及终止是许多软硬件共同工作结果，是许多大师智慧的结晶，hello 经过了重重考验才得以和我们见面，其中任何一个环节出错，都有可能致 hello 无法正确运行，失之毫厘，谬以千里，所以我们平时做事情一定要细心，降低出错的概率，如果做小事都能出错的话，大事是由小事凝结而成的，我们就无法取得更大的成就。

(结论 0 分，缺失 -1 分，根据内容酌情加分)

附件

hello.i:	hello.c 预编译的结果，用于研究预编译的作用以及进行编译器的下一步编译操作。
hello.s:	hello.i 编译后的结果，用于研究汇编语言以及编译器的汇编操作，可以与 hello.c 对应，分析底层的实现。
hello.o:	hello.s 汇编后的结果，可重定位目标程序，没有经过链接，用于链接器或编译器链接生成最终可执行程序。
hello.out:	hello.o 链接后生成的可执行目标文件，可以用来反汇编或者通过 EDB、GDB 等工具分析链接过程以及程序运行过程，包括进入 main 函数前后发生的过程。
hello:	链接之后的可执行目标文件，可直接运行。
hello.o.s:	对可重定位目标文件反汇编得到，可以与对可执行目标文件反汇编得到的代码对比来分析链接过程。
a.s:	对可执行目标文件反汇编得到，可以用来分析链接过程与寻址过程。
hello.elf	hello 的 elf 格式，用来分析各段的基本信息

（附表二：附件表）

（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] 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. 2018. 4.
- [2] Linux 下进程的睡眠唤醒:
<https://blog.csdn.net/shengjin/article/details/21530337>
- [3] 进程的睡眠、挂起和阻塞: <https://www.zhihu.com/question/42962803>
- [4] 不游泳的鱼. 编译入门: 传说中的编译是在做什么.
<http://www.cnblogs.com/li--chao/p/9229927.html>. 2018-09-01.
- [5] ELF 文件-段和程序头:
<https://blog.csdn.net/u011210147/article/details/54092405>
- [6] 动态链接过程延迟绑定的实现 (PLT)
https://blog.csdn.net/Virtual_Func/article/details/48789947
- [7] shell _百度百科. <https://baike.baidu.com/item/shell/99702#3>
- [8] 逻辑地址、线性地址、物理地址和虚拟地址理解
<https://blog.csdn.net/bat67/article/details/52071829>
- [9] TLB 的作用及工作原理
<https://www.cnblogs.com/alantu2018/p/9000777.html>
- [10] printf 函数实现的深入剖析
<https://www.cnblogs.com/pianist/p/3315801.html>

(参考文献 0 分, 缺失 -1 分)