

哈爾濱工業大學

計算機系統

大作業

題	目	<u>程序人生-Hello's P2P</u>		
專	業	<u>計算機科學與技術</u>		
學	號	<u>1170300418</u>		
班	級	<u>1736101</u>		
學	生	<u>于新蕊</u>		
指	導	教	師	<u>劉宏偉</u>

計算機科學與技術學院
2018 年 12 月

摘 要

本文通过运用与结合《深入理解计算机系统（第三版）》，在 linux 下，遍历 hello.c 的生命周期。通过运用 gcc、gdb、objdump、readelf 等工具，深入了解 hello.c 的程序本质。将实例与课本结合，深入地理解计算机系统的课程内容。

关键词：生命周期；编译；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.....	- 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.....	- 7 -
2.4 本章小结.....	- 7 -
第 3 章 编译.....	- 8 -
3.1 编译的概念与作用.....	- 8 -
3.2 在 UBUNTU 下编译的命令.....	- 8 -
3.3 HELLO 的编译结果解析.....	- 9 -
3.4 本章小结.....	- 14 -
第 4 章 汇编.....	- 15 -
4.1 汇编的概念与作用.....	- 15 -
4.2 在 UBUNTU 下汇编的命令.....	- 15 -
4.3 可重定位目标 ELF 格式.....	- 15 -
4.4 HELLO.O 的结果解析.....	- 19 -
4.5 本章小结.....	- 20 -
第 5 章 链接.....	- 21 -
5.1 链接的概念与作用.....	- 21 -
5.2 在 UBUNTU 下链接的命令.....	- 21 -
5.3 可执行目标文件 HELLO 的格式.....	- 21 -
5.4 HELLO 的虚拟地址空间.....	- 23 -
5.5 链接的重定位过程分析.....	- 24 -
5.6 HELLO 的执行流程.....	- 26 -
5.7 HELLO 的动态链接分析.....	- 26 -
5.8 本章小结.....	- 27 -
第 6 章 HELLO 进程管理.....	- 28 -
6.1 进程的概念与作用.....	- 28 -

6.2 简述壳 SHELL-BASH 的作用与处理流程.....	- 28 -
6.3 HELLO 的 FORK 进程创建过程.....	- 28 -
6.4 HELLO 的 EXECVE 过程.....	- 29 -
6.5 HELLO 的进程执行.....	- 30 -
6.6 HELLO 的异常与信号处理.....	- 30 -
6.7 本章小结.....	- 34 -
第 7 章 HELLO 的存储管理.....	- 36 -
7.1 HELLO 的存储器地址空间.....	- 36 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 36 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 37 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 38 -
7.5 三级 CACHE 支持下的物理内存访问.....	- 40 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 41 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 41 -
7.8 缺页故障与缺页中断处理.....	- 42 -
7.9 动态存储分配管理.....	- 42 -
7.10 本章小结.....	- 44 -
第 8 章 HELLO 的 IO 管理.....	- 45 -
8.1 LINUX 的 IO 设备管理方法.....	- 45 -
8.2 简述 UNIX IO 接口及其函数.....	- 45 -
8.3 PRINTF 的实现分析.....	- 46 -
8.4 GETCHAR 的实现分析.....	- 47 -
8.5 本章小结.....	- 47 -
结论.....	- 47 -
附件.....	- 49 -
参考文献.....	- 50 -

第 1 章 概述

1.1 Hello 简介

根据 Hello 的自白，利用计算机系统的术语，简述 Hello 的 P2P，020 的整个过程。
P2P : From Program to Process. `hello.c` 经过预处理器 `cpp` 预处理成 `hello.i`，再经过编译器 `ccl` 变成 `hello.s`，再经过汇编器 `as` 变成 `hello.o`，最后经过链接器 `ld` 与 C library 进行链接，最终变成 `hello`。此时的 `hello` 成为了一个 program。shell 通过键盘键入“`./hello`”，此时 `hello` 变成了一个 process，shell 为它 fork 一个子进程，并 `execve` `hello` 这个进程。这就是 P2P 的过程。

020 : From Zero-0 to Zero-0. Shell 执行可执行目标文件 `hello`，并创建一组新的代码、数据、堆和栈段。新的栈和堆段被初始化为零。执行 `hello` 的过程中，堆、栈段大小发生变化。在 `hello` 执行完成后，父进程 shell 回收子进程 `hello`，IO 管理与信号处理通过软硬结合，将其输出显示到屏幕。堆栈信息恢复到执行 `hello` 之前的状态，也就是执行 `hello` 前后堆栈信息没有改变，这就是 020 的过程。

1.2 环境与工具

硬件环境：X64CPU; 2.60GHz; 8GRAM;

软件环境：Windows10 64 位；Vmware 14；Ubuntu 18.04 LTS 64 位

开发与调试工具：gedit; vi; gcc; as; ld; gdb; readelf; hexedit

1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名称，文件的作用等。

文件	内容
<code>hello.i</code>	预处理过的源程序
<code>hello.s</code>	汇编程序
<code>hello.o</code>	可重定位目标程序
<code>hello</code>	可执行程序
<code>hello_o.elf</code>	<code>hello.o</code> 通过 <code>readelf</code> 查看的 elf 结构文本
<code>hello.elf</code>	<code>hello</code> 通过 <code>readelf</code> 查看的 elf 结构文本
<code>hello_o.asm</code>	<code>hello.o</code> 通过 <code>objdump</code> 查看的反汇编文本
<code>hello.asm</code>	<code>hello</code> 通过 <code>objdump</code> 查看的反汇编文本

1.4 本章小结

hello.c 经过预处理、编译、汇编、链接四个阶段变成可执行文件 hello，再通过 shell 执行可执行文件 hello，然后再被回收，体现了程序从无到有的过程。

(第 1 章 0.5 分)

第 2 章 预处理

2.1 预处理的概念与作用

预处理器（`cpp`）根据以字符`#`开头的命令（`directives`），修改原始的 C 程序。如 `hello.c` 中 `#include <stdio.h>` 指令告诉预处理器读系统头文件 `stdio.h` 的内容，并把它直接插入到程序文本中去。结果就得到另外一个 C 程序，通常是以 `.i` 作为文件扩

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

展名的。例如 `hello.c` 中的

预处理的作用有：

- ①将头文件的内容，直接插入到程序文本中。
- ②将符号常量替换为后边的文本，常见的有宏定义常数、宏定义符号、宏定义函数等等。
- ③删除所有注释。`/**/`，`//`。
- ④添加行号和文件标识符。用于显示调试信息：错误或警告的位置。
- ⑤处理条件预编译 `#if`, `#ifdef`, `#if`, `#elif`, `#endif`
- ⑥保留 `#pragma` 编译器指令。（1）设定编译器状态，（2）指示编译器完成一些特定的动作。

2.2 在 Ubuntu 下预处理的命令

`gcc -E hello.c -o hello.i`

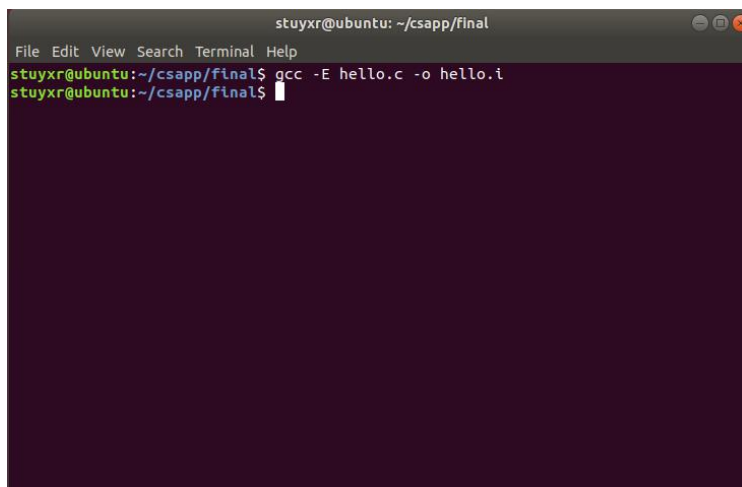
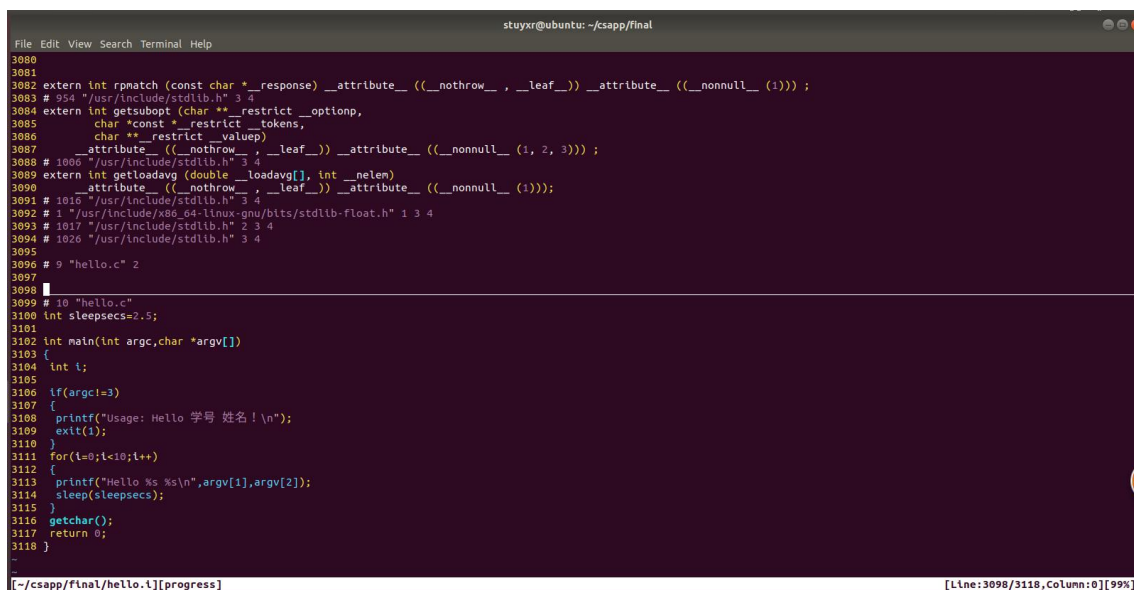


图 2-1 在 Ubuntu 下预处理的命令

2.3 Hello 的预处理结果解析



```
3080
3081
3082 extern int rmatch(const char *_response) __attribute__((__nothrow__ , __leaf__)) __attribute__((__nonnull__(1)));
3083 # 954 "/usr/include/stdlib.h" 3 4
3084 extern int getsubopt(char **_restrict __optionp,
3085 char *const *_restrict __tokens,
3086 char **_restrict __valuep)
3087 __attribute__((__nothrow__ , __leaf__)) __attribute__((__nonnull__(1, 2, 3)));
3088 # 1006 "/usr/include/stdlib.h" 3 4
3089 extern int getloadavg(double __loadavg[], int __nelem)
3090 __attribute__((__nothrow__ , __leaf__)) __attribute__((__nonnull__(1)));
3091 # 1010 "/usr/include/stdlib.h" 3 4
3092 # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
3093 # 1017 "/usr/include/stdlib.h" 2 3 4
3094 # 1026 "/usr/include/stdlib.h" 3 4
3095
3096 # 9 "hello.c" 2
3097
3098
3099 # 10 "hello.c"
3100 int sleepsecs=2.5;
3101
3102 int main(int argc,char *argv[])
3103 {
3104     int i;
3105
3106     if(argc!=3)
3107     {
3108         printf("Usage: Hello 学号 姓名!\n");
3109         exit(1);
3110     }
3111     for(i=0;i<10;i++)
3112     {
3113         printf("Hello %s %s\n",argv[1],argv[2]);
3114         sleep(sleepsecs);
3115     }
3116     getchar();
3117     return 0;
3118 }
```

图 2-2 预处理结果 hello.i 文件（部分）

我们发现①注释被删除了②头文件信息已经被插入到了 hello.i 中，而且文件变成了 3118 行（由于插入了头文件的代码）。

gcc 先打开 stdio.h 然后发现里面还有#define，就继续打开，直到最后的文件中没有#define 为止。在这之间会有大量的 typedef 重命名，以及定义大量文件输入输出指针等等。

2.4 本章小结

本阶段完成了 hello.c 的预处理过程。预处理使程序在后序的操作中不受阻碍，可以进行下一阶段的汇编处理。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译的过程是将预处理好的高级语言程序文本翻译成能执行相同操作的汇编语言的过程。

编译的作用有：

- ①将源代码程序输入扫描器，将源代码的字符序列分割成一系列记号。
- ②基于词法分析得到的一系列记号，生成语法树。
- ③由语义分析器完成，指示判断是否合法，并不判断对错。又分静态语义：隐含浮点型到整形的转换，会报 `warning`。
- ④中间代码（语言）使得编译器分为前端和后端，前端产生与机器（或环境）无关的中间代码，编译器的后端将中间代码转换为目标机器代码，目的：一个前端对多个后端，适应不同平台。
- ⑤编译器后端主要包括：代码生成器：依赖于目标机器，依赖目标机器的不同字长，寄存器，数据类型等

3.2 在 Ubuntu 下编译的命令

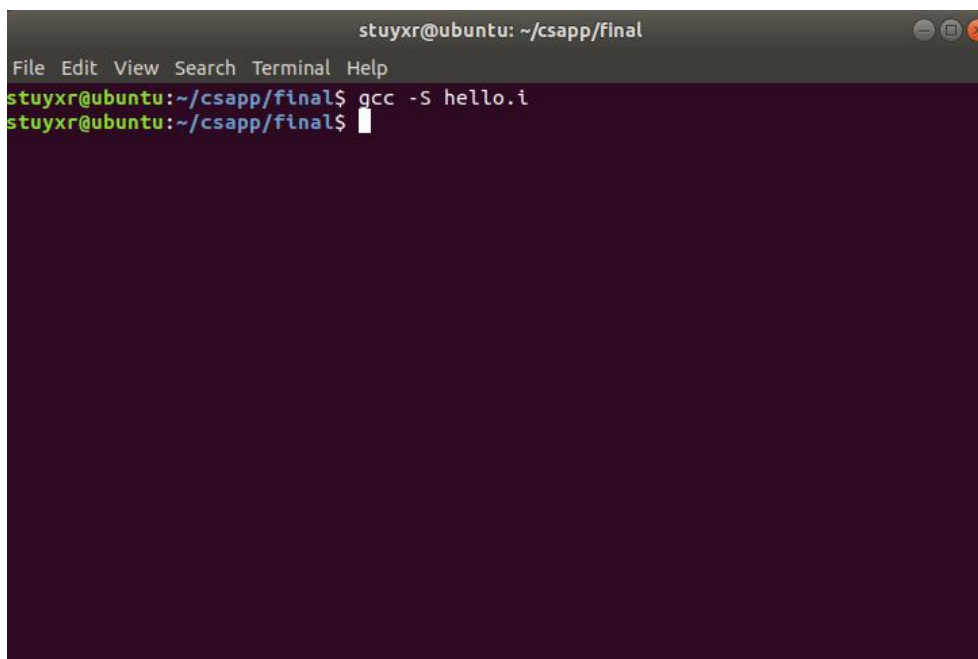
A screenshot of a terminal window titled 'stuyxr@ubuntu: ~/csapp/final'. The terminal shows the command 'gcc -S hello.i' being entered at the prompt 'stuyxr@ubuntu:~/csapp/final\$'. The output is not visible, only the prompt and the command are shown. The terminal has a dark background and standard window controls at the top.

图 3-1 在 Ubuntu 下编译的命令

3.3 Hello 的编译结果解析

3.3.1 数据

1、常量

对于常量，比如 `printf("%d", x);` 中的字符串 `"%d"`，或者 `const int x = 10;` 中的 `x`，都是常量。汇编器会将这样的常量定义在 `.rodata` 段中。在 `hello.c` 中，常量只有 `"Usage: Hello 学号 姓名! \n"` 和 `"Hello %s %s\n"`，它被存在了 `.rodata` 段中。

```
.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"
```

图 3-1 被存储在 `.rodata` 的字符串

2、全局变量

不同的变量被定义在不同的节中，初始化的全局变量和静态变量定义在 `.bss` 节；已初始化的全局和静态变量定义在只读代码区的 `.data` 节。

编译器在 `.text` 段中声明为全局变量(`.globl`)，全局变量被定义在 `.data` 节，`sleepsecs` 对齐要求(`.align`)是 4 字节，类型(`.type`)是 `object`(对象)，大小(`.size`)是 4 字节，`sleepsecs` 的初始值为 2(`.long`)。

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

图 3-2 全局变量 `sleepsecs` 在 `hello.s` 中的声明

3、局部变量

局部变量存放在堆栈中，如图 3-3 为 `for` 循环 `for(i = 0; i < 10; i++);` 对应的汇编代码。这个循环首先将局部变量 `i` 赋值为 0，我们看到 `L2` 中，将 `-4(%rbp)` 赋值为 0。因此我们得知，局部变量 `i` 保存在堆栈中。

```

.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
    .cfi_endproc

```

图 3-3 for 循环对应的汇编代码

3.3.2 赋值

1、int sleepsecs = 2.5; 这个全局变量 sleepsecs 的赋值在.data 节中直接声明为值 2 的 long 类型数据。

```

sleepsecs:
    .long   2

```

图 3-4 sleepsecs 的赋值

2、i = 0; 局部变量的赋值用汇编语言中的 mov 指令来赋值。对于 mov 指令 MOV S, D, 表示将 D 这个位置的数据赋值为 S。根据传送数据类型的不同, mov 指令最后一个字符也不同, 如下表。

指令	描述
movb S, D	传送 1Byte
movw S, D	传送 2Byte
movl S, D	传送 4Byte
movq S, D	传送 8Byte

同时 mov S, D 中的 S 和 D 可以有以下类型:

S	D
立即数	寄存器

寄存器	寄存器
内存	寄存器
立即数	内存
寄存器	内存

3.3.3 类型转换

`int sleepsecs = 2.5;`由于 `sleepsecs` 是 `int` 类型，2.5 是浮点类型，因此赋值时会发生隐式类型转换。程序改变数值和位模式的原则是：值会向零舍入。如图 3-4 所示，`sleepsecs` 的初始值被设为 2。

3.3.4 算术操作与逻辑运算

指令	效果	描述
<code>leaq S, D</code>	$D \leftarrow \&S$	加载有效地址
<code>INC D</code>	$D \leftarrow D + 1$	加 1
<code>DEC D</code>	$D \leftarrow D - 1$	减 1
<code>NEG D</code>	$D \leftarrow -D$	取负
<code>NOT D</code>	$D \leftarrow \sim D$	取补
<code>ADD S, D</code>	$D \leftarrow D + S$	加
<code>SUB S, D</code>	$D \leftarrow D - S$	减
<code>IMUL S, D</code>	$D \leftarrow D * S$	乘
<code>XOR S, D</code>	$D \leftarrow D \wedge S$	异或
<code>OR S, D</code>	$D \leftarrow D S$	或
<code>AND S, D</code>	$D \leftarrow D \& S$	与
<code>SAL k, D</code>	$D \leftarrow D \ll k$	左移
<code>SHL k, D</code>	$D \leftarrow D \gg k$	左移（等同于 SAL）
<code>SAR k, D</code>	$D \leftarrow D \gg (A)k$	算术右移
<code>SHR k, D</code>	$D \leftarrow D \gg (L)k$	逻辑右移

hello.s 用到的指令(部分)：

- 1、`addl $1, -4(%rbp)` 对应 c 语言的 `i++`。
- 2、`subq $32, %rsp` 栈帧减 32。

3.3.5 关系操作

指令	基于	描述
<code>CMP S1, S2</code>	<code>S2 - S1</code>	比较
<code>TEST S1, S2</code>	<code>S1 & S2</code>	测试

CMP 指令和 TEST 指令都只改变条件码的值，不改变寄存器或内存中数据的值。如果要比较 S1, S2 的大小关系，我们可以通过 CMP S1, S2，然后访问条件码的值（通常通过跳转操作来得知条件码的值）来获知。如果要比较 S1 是否等于 S2，俺么我们可以通过 TEST S1, S2 然后访问条件码的值来获知。

在 hello.s 中：

```
cmpl    $9, -4(%rbp)  这是在比较 i 和 9 的关系。对应语句 i < 10
cmpl    $3, -20(%rbp) 这是在比较 argv 和 3 的关系。对应语句 argv != 3
```

3.3.6 控制转移

我们通过访问条件码来判断下一步执行哪个语句。通常在 jmp 前一句都会是 TEST 或 CMP 改变条件码，然后根据这个条件码跳转。

在 hello.s 中：

```
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)
```

改变条件码

ZF为1则跳转

图 3-5 if(argv != 3)的条件跳转

```
movl    $0, -4(%rbp)
jmp     .L3  无条件跳转
```

图 3-6 无条件跳转

```
.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  小于等于则跳转
```

图 3-7 for(i = 0; i < 10; i++)的跳转

3.3.7 函数操作

1、调用函数

调用函数用 `call` 命令来实现。

```

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

```

图 3-8 hello.s 中的 `call` 指令

2、参数传递

64 位机器下的参数传递保存在寄存器中。其中第 1 个参数保存在 `%rdi` 中，第 2 个参数保存在 `%rsi` 中，第 3 个参数保存在 `%rdx` 中，第 4 个参数保存在 `%rcx` 中，第 5 个参数保存在 `%r8` 中，第 6 个参数保存在 `%r9` 中，剩下的参数保存在堆栈中。

```

movq    %rax, %rsi
leaq    .LC1(%rip), %rdi
movl    $0, %eax
call    printf@PLT
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep@PLT

```

图 3-8 hello.s 中的参数传递

3、返回值

返回值保存在 `%rax` 中。

```

.L3:
cmpl    $9, -4(%rbp)
jle     .L4
call    getchar@PLT
movl    $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc

```

图 3-8 hello.s 中的 `return 0`;

3.4 本章小结

编译器通过编译将修改了的源程序编译成汇编程序。本章对比 C 语言的语句和汇编语句，理解了汇编语言不同语句的具体含义，以及不同数据类型的操作与存储。

(第 3 章 2 分)

第 4 章 汇编

4.1 汇编的概念与作用

把汇编语言翻译成机器语言的过程称为汇编。

作用：

汇编器是将汇编代码(.s)转变成机器可以识别的机器指令，并将这些指令打包成可重定位目标程序(.o)。`.o` 文件是一个二进制文件。

4.2 在 Ubuntu 下汇编的命令

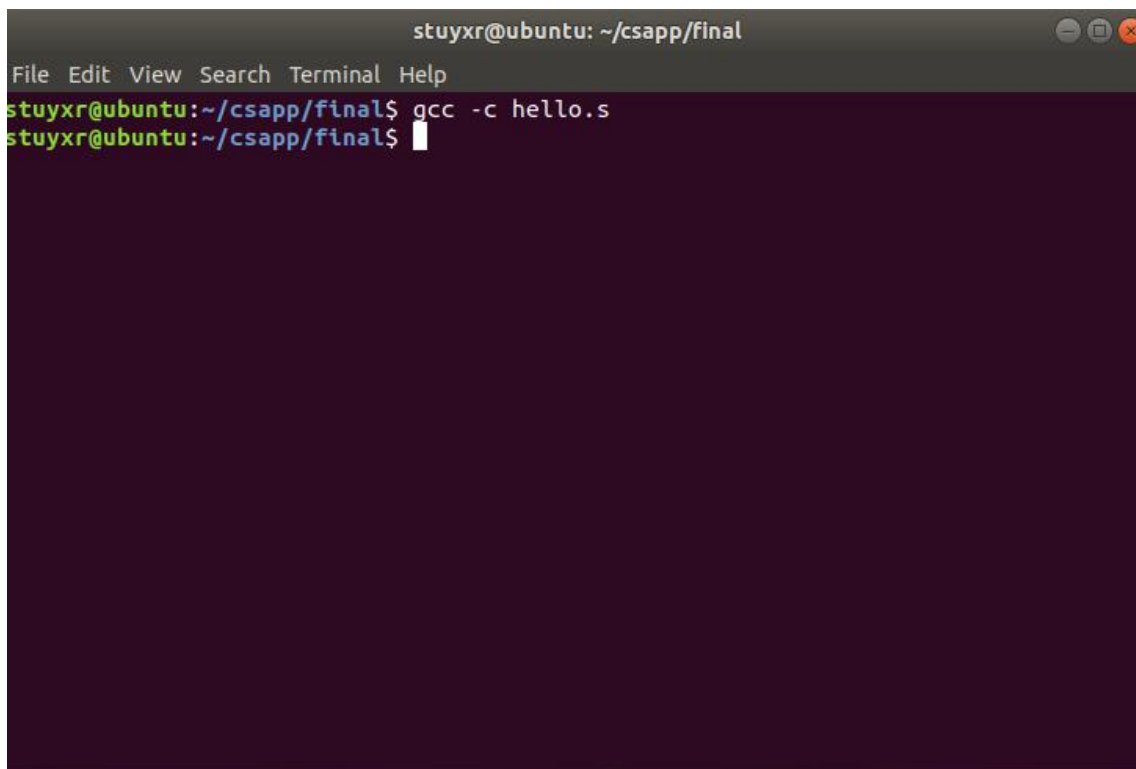
A screenshot of a terminal window titled 'stuyxr@ubuntu: ~/csapp/final'. The terminal shows the command 'gcc -c hello.s' being entered and executed. The prompt is 'stuyxr@ubuntu:~/csapp/final\$'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The background is dark purple.

图 4-1 在 Ubuntu 下的汇编的命令

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
-------	---------------------

段头部表	将连续的文件节映射到运行时内存段
.init	程序初始化代码需要调用的函数
.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
.symtab	存放程序中定义和引用的函数和全局变量信息
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

ELF 格式

键入命令 `readelf -a hello.o > hello_o.elf`

1、ELF header

magic: 给操作系统和编译器辨别此文件是 ELF 二进制库。0x45 0x4C 0x46 为 ELF 三个字母的 ASCII 码。

type: 文件类型

Relocatable file = 1 (.o, .a 可重定位文件, 静态库)

Executable file = 2 (可执行文件, a.out, exe, 运行库)

OS: 操作系统

machine: 架构

version: ELF 版本, 目前均为 1

entry: 程序的入口地址 (虚拟地址), .o 文件没有入口, 故为 0。可执行文件应该为 `_start` 的虚拟地址。

Size/Number of section headers: 节头部表中条目的大小和数量

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                           0
  Type:                                 REL (Relocatable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:              1152 (bytes into file)
  Flags:                                 0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               0 (bytes)
  Number of program headers:              0
  Size of section headers:               64 (bytes)
  Number of section headers:              13
  Section header string table index: 12

```

图 4-2 ELF header

2、Section Headers

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

```

Section Headers:
 [Nr] Name                Type           Address          Offset
      Size                EntSize          Flags    Link    Info    Align
 [ 0]                      NULL           0000000000000000 00000000
      0000000000000000    0000000000000000 0 0
 [ 1] .text                  PROGBITS       0000000000000000 00000040
      0000000000000081    0000000000000000 AX      0 0 1
 [ 2] .rela.text           RELA           0000000000000000 00000340
      00000000000000c0    0000000000000018 I      10 1 8
 [ 3] .data                 PROGBITS       0000000000000000 000000c4
      0000000000000004    0000000000000000 WA      0 0 4
 [ 4] .bss                  NOBITS         0000000000000000 000000c8
      0000000000000000    0000000000000000 WA      0 0 1
 [ 5] .rodata               PROGBITS       0000000000000000 000000c8
      000000000000002b    0000000000000000 A      0 0 1
 [ 6] .comment              PROGBITS       0000000000000000 000000f3
      000000000000002b    0000000000000001 MS      0 0 1
 [ 7] .note.GNU-stack        PROGBITS       0000000000000000 0000011e
      0000000000000000    0000000000000000 0 0 1
 [ 8] .eh_frame              PROGBITS       0000000000000000 00000120
      0000000000000038    0000000000000000 A      0 0 8
 [ 9] .rela.eh_frame         RELA           0000000000000000 00000400
      0000000000000018    0000000000000018 I      10 8 8
[10] .symtab                 SYMTAB         0000000000000000 00000158
      00000000000000198    0000000000000018 11 9 8
[11] .strtab                 STRTAB         0000000000000000 000002f0
      000000000000004d    0000000000000000 0 0 1
[12] .shstrtab              STRTAB         0000000000000000 00000418
      0000000000000061    0000000000000000 0 0 1

```

图 4-3 Section Headers

3、重定位节.rela.text

这一步生成的可重定向目标文件由于未和标准 C library 链接，因此有些信息需要

修改，如代码节、数据节中的对每个符号的引用。我们在 .rela.text 节中记录这些需要修改的地址。 .rela.text 中每个条目维护这些信息：

信息	含义
Offset	要修改的引用相对于 .text 或 .data 节头的偏移量
Info	8Byte。前 4Byte 是 symbol，代表重定位到的目标在 .symtab 中的偏移量。后 4Byte 是 type，是重定位类型（有 R_X86_64_PC32 和 R_X86_64_PLT32 两种）
Addend	计算重定位位置的辅助信息，共占 8 个字节
Name	重定向到的目标的名称

```
Relocation section '.rela.text' at offset 0x340 contains 8 entries:
  Offset          Info          Type          Sym. Value      Sym. Name + Addend
0000000000018    000500000002  R_X86_64_PC32  0000000000000000 .rodata - 4
000000000001d    000c00000004  R_X86_64_PLT32  0000000000000000 puts - 4
0000000000027    000d00000004  R_X86_64_PLT32  0000000000000000 exit - 4
0000000000050    000500000002  R_X86_64_PC32  0000000000000000 .rodata + 1a
000000000005a    000e00000004  R_X86_64_PLT32  0000000000000000 printf - 4
0000000000060    000900000002  R_X86_64_PC32  0000000000000000 sleepsecs - 4
0000000000067    000f00000004  R_X86_64_PLT32  0000000000000000 sleep - 4
0000000000076    001000000004  R_X86_64_PLT32  0000000000000000 getchar - 4
```

图 4-4 重定位节 .rela.text

hello.o 的 .rela.text 节中维护了这些条目：.L0（第一个 printf 中的字符串）、puts 函数、exit 函数、.L1（第二个 printf 中的字符串）、printf 函数、sleepsecs、sleep 函数、getchar 函数。

4、.symtab

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

```
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
```


图 4-5 .symtab 符号表

4.4 Hello.o 的结果解析

1、键入 `objdump -d -r hello.o > hello_o.asm` 将 `hello.o` 进行反汇编。

2、`hello_o.asm` 与 `hello.s` 的对比

```

.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
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
.cfi_offset 1, %rbp

```

```

Disassembly of section .text:
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)
13: 83 7d ec 03  cmpl   $0x3, -0x14(%rbp)
14: 74 16      je     2b <main+0x2b>
15: 48 8d 3d 00 00 00 00  lea     0x0(%rip), %rdi # 1c <main+0x1c>
16:          18: R_X86_64_PC32
17: 1c: e8 00 00 00 00  callq 21 <main+0x21>
18:          1d: R_X86_64_PLT32
19: 21: bf 01 00 00 00  mov     $0x1, %edi
20: 26: e8 00 00 00 00  callq 2b <main+0x2b>
21:          27: R_X86_64_PLT32
22: 2b: c7 45 fc 00 00 00 00  movl   $0x0, -0x4(%rbp)
23:          32: eb 3b      jmp     6f <main+0x6f>
24: 34: 48 8b 45 e0  mov     -0x20(%rbp), %rax
25: 38: 48 83 c0 10  add     $0x10, %rax
26: 3c: 48 8b 10      mov     (%rax), %rdx
27: 3f: 48 8b 45 e0  mov     -0x20(%rbp), %rax
28: 43: 48 83 c0 08  add     $0x8, %rax
29: 47: 48 8b 00      mov     (%rax), %rax
30: 4a: 48 89 c6      mov     %rax, %rsi
31: 4d: 48 8d 3d 00 00 00 00  lea     0x0(%rip), %rdi # 54 <main+0x54>
32:          50: R_X86_64_PC32
33: 54: b8 00 00 00 00  mov     $0x0, %eax
34: 59: e8 00 00 00 00  callq 5e <main+0x5e>
35:          5a: R_X86_64_PLT32
36: 5e: 8b 05 00 00 00 00  mov     0x0(%rip), %eax # 64 <main+0x64>
37:          60: R_X86_64_PC32
38: 64: 89 c7      mov     %eax, %edi
39: 66: e8 00 00 00 00  callq 6b <main+0x6b>
40:          67: R_X86_64_PLT32
41: 6b: c7 45 fc 00 00 00 00  movl   $0x1, -0x4(%rbp)

```

1) 分支转移

反汇编代码跳转指令的使用的是相对地址(相对.text段的地址),而 `hello.s` 中用的是段符号.L0,L1 等等。例如 `hello.s` 中的 `je .L2` 对应反汇编代码中的 `je 2b`。因为段名称只是在汇编语言中便于编写的助记符,所以在汇编成机器语言之后显然不存在,而是确定的地址。

2) 函数调用

在.s文件中,call指令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的地址。例如 `hello.s` 中的 `call exit@plt` 对应反汇编代码中的 `call 2b`。这是因为由于我们调用的函数是来自外部的函数,所以 `hello.o` 和标准库链接时需要重定位计算地址,我们现在无法知道最终运行时的地址,所以用 0 来填充。call指令对应的机器代码的编码是 `e8`,我们在后面填充 4 字节的 0。又由于 call 指令采用相对寻址(相对%rip),所以我们用 0 填充,相当于 call 下一条指令。在链接时根据.rela.text 中的内容再加以修正。

3) 全局变量的引用

在 `hello.s` 中，全局变量用全局变量名称(`%rip`)来引用，而在反汇编代码中，我们用 `0x0(%rip)`。和函数调用的原理类似，我们由于不知道全局变量在最终链接后的地址，我们暂时用 0 来填充。在链接时根据 `.rela.text` 中的内容再加以修正。

4) 二进制指令

`hello.s` 中只有汇编指令，而 `hello.o` 的反汇编文件中有机器指令。机器语言由二进制代码构成（图中反汇编结果用 16 进制表示），是计算机能够直接识别和执行的一种机器指令的集合。汇编指令和二进制指令存在一个映射关系。例如：`call` 指令对应 `0xe8`，占 1 个字节，通常后边会跟有 4 字节的相对地址。

4.5 本章小结

本章通过使用 `readelf` 和 `objdump` 辅助工具查看了可重定向目标文件 `hello.o` 的 `elf` 结构和反汇编代码。通过 `objdump` 还可以反汇编查看二进制文件的汇编代码和机器代码及其对应关系。

（第 4 章 1 分）

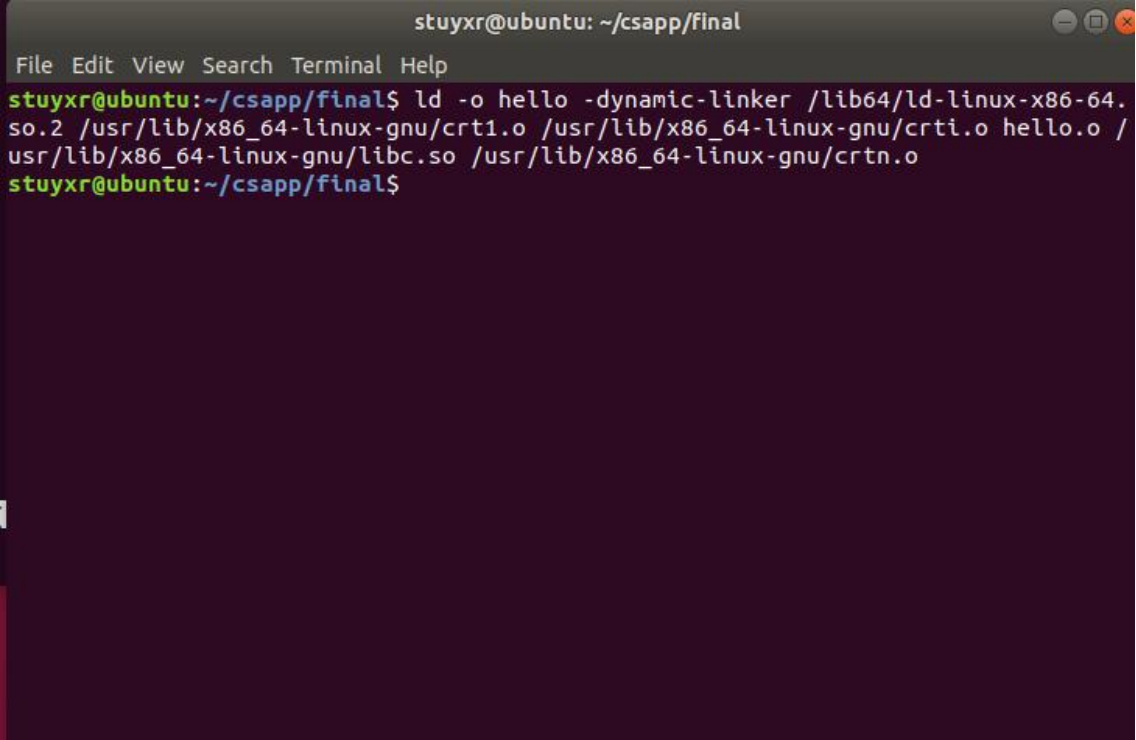
第 5 章 链接

5.1 链接的概念与作用

链接是将各种代码和数据片段收集并组合成一个单一文件的过程，这个文件可被加载到内存并执行。

链接的作用是将不可执行的可重定向目标文件变成可执行的可执行文件。

5.2 在 Ubuntu 下链接的命令



```
stuyxr@ubuntu: ~/csapp/final
File Edit View Search Terminal Help
stuyxr@ubuntu:~/csapp/final$ 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
stuyxr@ubuntu:~/csapp/final$
```

5.3 可执行目标文件 hello 的格式

指令：readelf -a hello > hello.elf

1、ELF header

和 hello.o 的 ELF header 结构类似，不过 section 的个数不同。

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                          0
  Type:                                 EXEC (Executable file)
  Machine:                              Advanced Micro Devices X86-64
  Version:                              0x1
  Entry point address:                  0x400500
  Start of program headers:             64 (bytes into file)
  Start of section headers:             5928 (bytes into file)
  Flags:                                0x0
  Size of this header:                  64 (bytes)
  Size of program headers:              56 (bytes)
  Number of program headers:            8
  Size of section headers:              64 (bytes)
  Number of section headers:            25
  Section header string table index:    24

```

图 5-1 ELF Header

2、Section Headers

```

Section Headers:
[Nr] Name                Type           Address          Offset
     Size                EntSize        Flags Link Info  Align
[ 0] 0000000000000000    NULL          0000000000000000 00000000
     0000000000000000    0000000000000000 0 0 0
[ 1] .interp              PROGBITS       0000000000400200 00000200
     0000000000000001c    0000000000000000 A 0 0 1
[ 2] .note.ABI-tag        NOTE           000000000040021c 0000021c
     00000000000000020    0000000000000000 A 0 0 4
[ 3] .hash                HASH           0000000000400240 00000240
     00000000000000034    0000000000000004 A 5 0 8
[ 4] .gnu.hash            GNU_HASH       0000000000400278 00000278
     0000000000000001c    0000000000000000 A 5 0 8
[ 5] .dynsym              DYNSYM         0000000000400298 00000298
     000000000000000c0    0000000000000018 A 6 1 8
[ 6] .dynstr              STRTAB         0000000000400358 00000358
     00000000000000057    0000000000000000 A 0 0 1
[ 7] .gnu.version          VERSYM         00000000004003b0 000003b0
     00000000000000010    0000000000000002 A 5 0 2
[ 8] .gnu.version_r        VERNEED        00000000004003c0 000003c0
     00000000000000020    0000000000000000 A 6 1 8
[ 9] .rela.dyn             RELA           00000000004003e0 000003e0
     00000000000000030    0000000000000018 A 5 0 8
[10] .rela.plt             RELA           0000000000400410 00000410
     00000000000000078    0000000000000018 AI 5 19 8
[11] .init                PROGBITS       0000000000400488 00000488
     00000000000000017    0000000000000000 AX 0 0 4
[12] .plt                 PROGBITS       00000000004004a0 000004a0
     00000000000000060    0000000000000010 AX 0 0 16

```

图 5-2 Section Headers（部分）

Size: 对应的节的大小

Address: 虚拟地址

Offset: 这个节在程序中的偏移量

3、.symtab


```

Symbol table '.symtab' contains 49 entries:
  Num:      Value              Size Type      Bind      Vis      Ndx Name
  0: 0000000000000000      0 NOTYPE   LOCAL   DEFAULT   UND
  1: 0000000000400200      0 SECTION LOCAL   DEFAULT    1
  2: 000000000040021c      0 SECTION LOCAL   DEFAULT    2
  3: 0000000000400240      0 SECTION LOCAL   DEFAULT    3
  4: 0000000000400278      0 SECTION LOCAL   DEFAULT    4
  5: 0000000000400298      0 SECTION LOCAL   DEFAULT    5
  6: 0000000000400358      0 SECTION LOCAL   DEFAULT    6
  7: 00000000004003b0      0 SECTION LOCAL   DEFAULT    7
  8: 00000000004003c0      0 SECTION LOCAL   DEFAULT    8
  9: 00000000004003e0      0 SECTION LOCAL   DEFAULT    9
 10: 0000000000400410      0 SECTION LOCAL   DEFAULT   10
 11: 0000000000400488      0 SECTION LOCAL   DEFAULT   11
 12: 00000000004004a0      0 SECTION LOCAL   DEFAULT   12
 13: 0000000000400500      0 SECTION LOCAL   DEFAULT   13
 14: 0000000000400634      0 SECTION LOCAL   DEFAULT   14
 15: 0000000000400640      0 SECTION LOCAL   DEFAULT   15
 16: 0000000000400670      0 SECTION LOCAL   DEFAULT   16
 17: 0000000000600e50      0 SECTION LOCAL   DEFAULT   17
 18: 0000000000600ff0      0 SECTION LOCAL   DEFAULT   18
 19: 0000000000601000      0 SECTION LOCAL   DEFAULT   19
 20: 0000000000601040      0 SECTION LOCAL   DEFAULT   20
 21: 0000000000000000      0 SECTION LOCAL   DEFAULT   21
 22: 0000000000000000      0 FILE     LOCAL   DEFAULT   ABS hello.c
 23: 0000000000000000      0 FILE     LOCAL   DEFAULT   ABS
 24: 0000000000600e50      0 NOTYPE   LOCAL   DEFAULT   17 __init_array_end
 25: 0000000000600e50      0 OBJECT   LOCAL   DEFAULT   17 __DYNAMIC
 26: 0000000000600e50      0 NOTYPE   LOCAL   DEFAULT   17 __init_array_start
 27: 0000000000601000      0 OBJECT   LOCAL   DEFAULT   19 _GLOBAL_OFFSET_TABLE_
 28: 0000000000400630      2 FUNC     GLOBAL   DEFAULT   13 __libc_csu_fini
 29: 0000000000601040      0 NOTYPE   WEAK     DEFAULT   20 data_start
 30: 0000000000000000      0 FUNC     GLOBAL   DEFAULT   UND puts@@GLIBC_2.2.5
 31: 0000000000601044      4 OBJECT   GLOBAL   DEFAULT   20 sleepsecs
 32: 0000000000601048      0 NOTYPE   GLOBAL   DEFAULT   20 _edata
 33: 0000000000400634      0 FUNC     GLOBAL   DEFAULT   14 _fini
 34: 0000000000000000      0 FUNC     GLOBAL   DEFAULT   UND printf@@GLIBC_2.2.5
 35: 0000000000000000      0 FUNC     GLOBAL   DEFAULT   UND __libc_start_main@@GLIBC_
 36: 0000000000601040      0 NOTYPE   GLOBAL   DEFAULT   20 data_start

```

图 5-3 可执行文件 hello 的符号表

5.4 hello 的虚拟地址空间

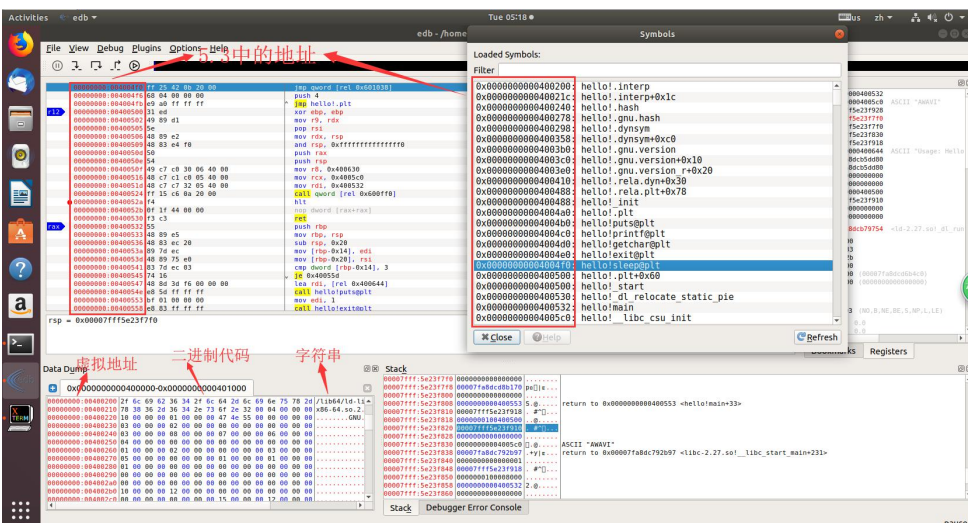


图 5-4 edb 打开 hello

在 0x400000~0x401000 段中，程序被载入，自虚拟地址 0x400000 开始，自 0x400fff 结束，这之间每个节（开始 ~.eh_frame 节）的排列即开始结束同图 5.2 中 Address 中声明。

接着我们用打开之前的 hello.elf，查看 Program Headers

```
Program Headers:
```

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040 0x00000000000001c0	0x000000000000400040 0x00000000000001c0	0x000000000000400040 R 0x8
INTERP	0x0000000000000200 0x000000000000001c	0x000000000000400200 0x000000000000001c	0x000000000000400200 R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000 0x000000000000076c	0x000000000000400000 0x000000000000076c	0x000000000000400000 R E 0x200000
LOAD	0x0000000000000e50 0x00000000000001f8	0x000000000000600e50 0x00000000000001f8	0x000000000000600e50 RW 0x200000
DYNAMIC	0x0000000000000e50 0x00000000000001a0	0x000000000000600e50 0x00000000000001a0	0x000000000000600e50 RW 0x8
NOTE	0x000000000000021c 0x0000000000000020	0x00000000000040021c 0x0000000000000020	0x00000000000040021c R 0x4
GNU_STACK	0x0000000000000000 0x0000000000000000	0x0000000000000000 0x0000000000000000	0x0000000000000000 RW 0x10
GNU_RELRO	0x0000000000000e50 0x00000000000001b0	0x000000000000600e50 0x00000000000001b0	0x000000000000600e50 R 0x1

图 5-5 hello 的 Program Headers

从中可以看出，程序包含 8 个段：

PHDR 保存程序头表。

INTERP 保存程序执行前需要调用的解释器。

LOAD 表示程序目标代码和常量信息。

DYNAMIC 保存了由动态链接器使用的信息。

NOTE 保存辅助信息。

GNU_STACK：权限标志，标志栈是否是可执行的。

GNU_RELRO：保存在重定位之后只读信息的位置。

5.5 链接的重定位过程分析

链接的重定位过程由两步组成。第一步，重定位节和符号定义。在这一步，链接器将所有相同类型的节合并为同一类型的新的聚合节。第二步，重定位节的富豪引用，使得代码节和数据节中的符号指向正确的地址。

下面键入 `objdump -d -r hello > hello.asm`，查看 `hello.asm` 与 `hello_o.asm` 的差异。

图 5-6 hello.asm 与 hello.o.asm 对比

- 1、增加了 .init, .plt, .fini 节。 .init 是程序初始化， .plt 是动态链接表， .fini 是程序终止时需要的执行的指令。
- 2、hello 中的地址是虚拟内存地址，从 0x400000 开始，而 hello.o 的地址时相对 .text 节的偏移地址。
- 3、全局变量的引用。链接器解析重定条目时发现两个类型为 R_X86_64_PC32 或 R_X86_64_PC32 的进行重定位。 .rodata 或 .data 与 .text 节之间的相对距离确定，因此链接器直接修改 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。

重定位地址计算方法如下：

```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset; /* ptr to reference to be relocated */

        /* Relocate a PC-relative reference */
        if (r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset; /* ref's run-time address */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
        }

        /* Relocate an absolute reference */
        if (r.type == R_X86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
    }
}
```

图 5-7 重定位地址计算方法

5.6 hello 的执行流程

使用 edb 跟踪，以下是从加载到结束过程中调用的函数：

程序名称	程序地址
ld-2.27.so! dl_init	
ld-2.27.so! dl_init	
hello! start	0x400500
libc-2.27.so! libc_start_main	
-libc-2.27.so! cxa_atexit	
-libc-2.27.so! libc_csu_init	0x4005c0
hello! init	0x400488
libc-2.27.so! setjmp	
-libc-2.27.so! sigsetjmp	
--libc-2.27.so! sigjmp_save	
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	
-ld-2.27.so! dl_fixup	
--ld-2.27.so! dl_lookup_symbol_x	
libc-2.27.so!exit	

5.7 Hello 的动态链接分析

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

```

00000000:00601000 10 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 b6 04 40 00 00 00 00 00 .@
00000000:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00 f.@
00000000:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 h.@
00000000:00601040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000:00601050 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

图 5-8 调用之前的 got.plt

```

00000000:00601000 10 0e 60 00 00 00 00 00 70 11 a2 ea b7 7f 00 00 .p.mto
00000000:00601010 50 f7 80 ea b7 7f 00 00 b6 04 40 00 00 00 00 00 P.to..@
00000000:00601020 c6 04 40 00 00 00 00 00 d6 04 40 00 00 00 00 00 f.@
00000000:00601030 e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00 h.@
00000000:00601040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

图 5-9 调用之前的 got.plt

5.8 本章小结

这一章我们分析了从可重定位目标文件到可执行文件中间的过程，即链接。分析可执行文件的 `hello` 的 `elf` 格式，分析 `hello` 的虚拟地址，逐步理解链接的过程和原理。

(第 5 章 1 分)

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程是一个执行中的程序的实例，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；堆栈区域存储区着活动过程调用的指令和本地变量。

作用：通过上下文切换，可以让 `cpu` 有一种同时运行很多可执行文件的假象，然而实际上是几个程序来回切换着占着 `cpu`。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用：Shell 是一个用高级语言编写的交互型的应用级程序。用户可以通过键入命令来和 linux 系统互动。

Shell 的处理流程：

- 1、读取用户的输入
- 2、分析输入内容，获得参数
- 3、如果是内核命令则直接执行，否则调用相应的程序执行命令
- 4、在程序运行期间，shell 需要监视键盘的输入内容，并且做出相应的反应
- 5、如果是前台程序，程序结束需要回收

6.3 Hello 的 fork 进程创建过程

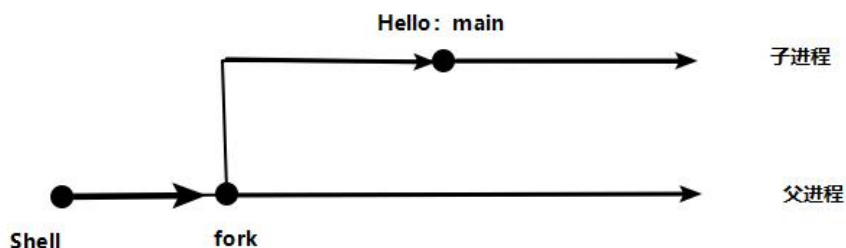


图 6-1 Hello 之创建进程

如图 6-1 所示，为 shell 创建 hello 进程的示意图。shell 首先 fork 一个子进程，这

个子进程是父进程的一个副本，有着和父进程相同的堆栈结构、堆栈存储信息、寄存器信息等等，虚拟地址相同但独立。

父进程与子进程是并发运行的独立进程，内核能够以任意方式交替执行它们的逻辑控制流的指令。如果通过分析命令行键入信息得知是前台程序，则在子进程执行期间，父进程（shell）等待子进程的完成，回收子进程。如指定为后台程序，则无需等待子进程完成。

（以下格式自行编排，编辑时删除）

6.4 Hello 的 execve 过程

fork 之后，我们就要在子进程中 `execve`(加载) `hello` 这个进程了。`execve` 时还要传入命令行参数。就是 `hello` 中的学号、姓名。

加载并运行 `hello` 需要以下几个步骤：

- 1、删除已存在的用户区域
- 2、映射私有区域
- 3、映射贡献区域
- 4、设置程序计数器

`execve` 函数加载并运行可执行文件 `hello`，且带参数列表 `argv` 和环境变量列表 `envp`。`argv` 变量指向一个以 `null` 结尾的指针数组，每一个指针都指向一个参数字符串。`envp` 变量指向一个以变量指向一个以 `null` 结尾的指针数组，每个指针指向一个环境变量字符串。

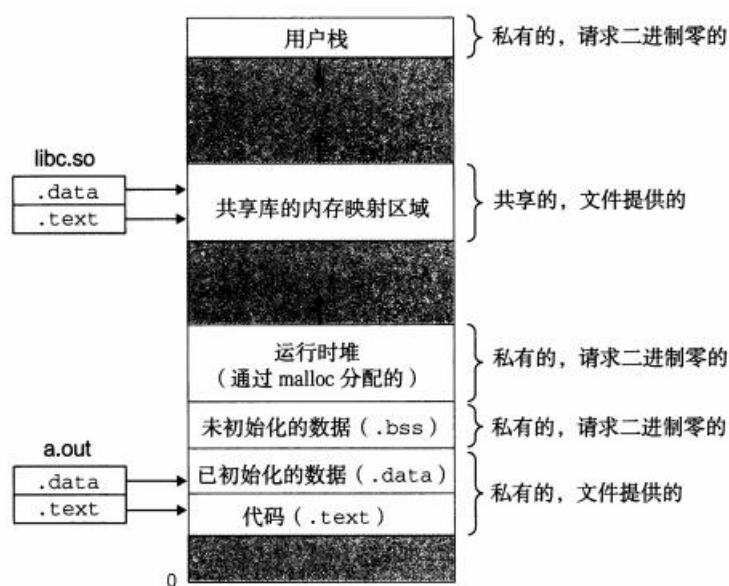


图 6-2 加载器是如何映射用户地址空间的

（以下格式自行编排，编辑时删除）

6.5 Hello 的进程执行

1、sleep()

sleep()方法是线程类(Thread)的静态方法,让调用的线程进入指定时间睡眠状态,使得当前线程进入阻塞状态,告诉系统至少在指定时间内不需要为线程调度器为该线程分配执行时间片,不再占用cpu,给执行机会给其他线程,但是监控状态依然保持,到时后会自动恢复。

2、上下文切换

如图所示,简单理解就是两个进程来回切换的过程,这样才会给我们一个程序独占cpu的错觉。两个程序需要内核的介入完成上下文切换。

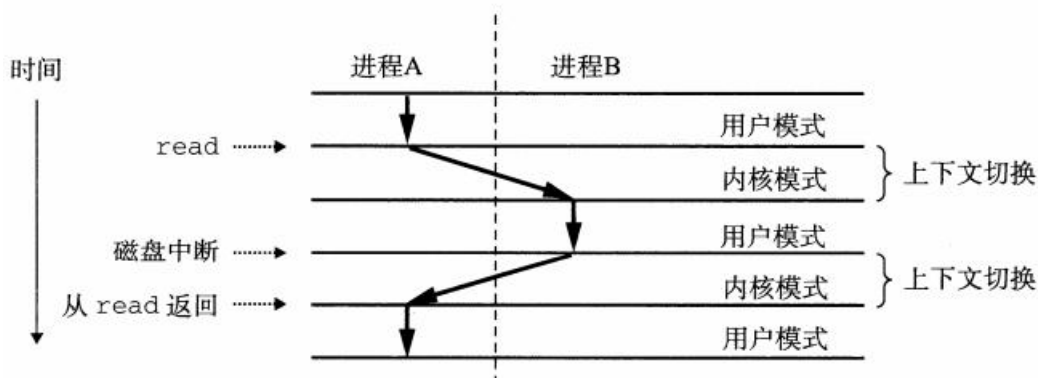


图 6-3 上下文的切换

3、进程 hello

在hello加载完成之后,hello就要开始执行了。但是cpu只有一个,还有其他进程也在运行,比如再打开一个shell等等。这是我们就要运用上下文切换。就在这种来回切换的过程中,hello执行到了sleep()命令,系统得知2s内不需要分配时间片给hello了,此时进入内核状态执行中断处理,将hello进程从等待队列中移出重新加入到运行队列,成为就绪状态,hello进程就可以继续进行自己的控制逻辑流了。

6.6 hello 的异常与信号处理

6.6.1 异常的种类

- 1、中断: SIGSTP: 挂起程序
- 2、终止: SIGINT: 终止程序

6.6.2 命令的运行

1、ctrl+Z 操作

```
stuyxr@ubuntu:~/csapp/final$ ./hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
^Z
[1]+  Stopped                  ./hello 1170300418 yxr
stuyxr@ubuntu:~/csapp/final$ ps
  PID TTY          TIME CMD
  6232 pts/1        00:00:00 bash
  7259 pts/1        00:00:00 hello
  7260 pts/1        00:00:00 ps
```

图 6-4 ctrl+z

键入 ctrl+z 向进程发送了一个 sigtstp 信号，让进程暂时挂起，输入 ps 命令发现 hello 进程依旧在运行。

2、ctrl+C 操作

```
stuyxr@ubuntu:~/csapp/final$ ./hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
^C
[1]+  Killed                    ./hello 1170300418 yxr
stuyxr@ubuntu:~/csapp/final$ ps
  PID TTY          TIME CMD
  6232 pts/1        00:00:00 bash
  7262 pts/1        00:00:00 ps
```

图 6-5 ctrl+c

键入 ctrl+c 向进程发送了一个 sigint 信号，让进程直接结束，输入 ps 命令发现当前 hello 进程已经被终止了。

3、fg 命令

```
stuyxr@ubuntu:~/csapp/final$ ./hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
^Z
[1]+  Stopped                  ./hello 1170300418 yxr
stuyxr@ubuntu:~/csapp/final$ fg 1
./hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
s
```


计算机系统课程报告

键入 **fg** 可以使后台挂起的进程继续运行。先输出了 4 次，然后 **ctrl+Z** 挂起之后，**fg** 命令又可以让他继续进行，然后把剩下的 6 次输出完。

4、jobs 命令

```
stuyxr@ubuntu:~/csapp/final$ ./hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
^Z
[1]+  Stopped                  ./hello 1170300418 yxr
stuyxr@ubuntu:~/csapp/final$ jobs
[1]+  Stopped                  ./hello 1170300418 yxr
```

图 6-7 jobs

`jobs` 命令可以查看当前的关键命令（`ctrl+Z/ctrl+C` 这类）内容，比如这时候就会返回 `ctrl+Z` 表示暂停命令

5、pstree 命令

```
stuyxr@ubuntu:~/csapp/final$ pstree
systemd├─ModemManager─2*[{ModemManager}]
│   └─NetworkManager─dhclient
│                       2*[{NetworkManager}]
│
│   └─VGAAuthService
│
│   └─accounts-daemon─2*[{accounts-daemon}]
│
│   └─acpid
│
│   └─avahi-daemon─avahi-daemon
│
│   └─bluetoothd
│
│   └─boltd─2*[{boltd}]
│
│   └─colord─2*[{colord}]
│
│   └─cron
│
│   └─cups-browsed─2*[{cups-browsed}]
│
│   └─cupsd
│
│   └─2*[dbus-daemon]
│
│   └─fcitx─{fcitx}
│
│   └─fcitx-dbus-watc
│
│   └─fwupd─4*[{fwupd}]
│
│   └─gdm3├─gdm-session-work─gdm-wayland-ses─gnome-session-b─gnome-sh+
│       │   │                   │               │               │   gsd-a11y+
│       │   │                   │               │               │   gsd-clip+
│       │   │                   │               │               │   gsd-colo+
│       │   │                   │               │               │   gsd-date+
│       │   │                   │               │               │   gsd-hous+
│       │   │                   │               │               │   gsd-keyb+
│       │   │                   │               │               │   gsd-medi+
│       │   │                   │               │               │   gsd-mous+
│       │   │                   │               │               │   gsd-powe+
│       │   │                   │               │               │   gsd-prin+
│       │   │                   │               │               │   gsd-rfki+
│       │   │                   │               │               │   gsd-scre+
│       │   │                   │               │               │   gsd-shar+
│       │   │                   │               │               │   gsd-smar+
│       │   │                   │               │               │   gsd-soun+
│       │   │                   │               │               │   gsd-waco+
│       │   │                   │               │               │   gsd-xset+
│       │   │                   │               │               └─3*[{gnom+
│       │   │                   │               └─2*[{gdm-wayland-ses}]
│       │   └─2*[{gdm-session-work}]
│       └─gdm-session-work─gdm-x-session─Xorg─{Xorg}
│           │                   │   └─gnome-session-b─deja-dup.+
│           │                   │                   │   gnome-shell+
│           │                   │                   └─gnome-soft+

```

图 6-8 pstree

pstree 是用进程树的方法把各个进程用树状图的方式连接起来

6、kill 指令

```
stuyxr@ubuntu:~/csapp/final$ ./hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
Hello 1170300418 yxr
^Z
[1]+  Stopped                  ./hello 1170300418 yxr
stuyxr@ubuntu:~/csapp/final$ ps
  PID TTY          TIME CMD
  6232 pts/1        00:00:00 bash
  7302 pts/1        00:00:00 hello
  7303 pts/1        00:00:00 ps
stuyxr@ubuntu:~/csapp/final$ kill -9 7302
stuyxr@ubuntu:~/csapp/final$ ps
  PID TTY          TIME CMD
  6232 pts/1        00:00:00 bash
  7304 pts/1        00:00:00 ps
[1]+  Killed                  ./hello 1170300418 yxr
stuyxr@ubuntu:~/csapp/final$ ps
  PID TTY          TIME CMD
  6232 pts/1        00:00:00 bash
  7305 pts/1        00:00:00 ps
```

图 6-9 kill -9 pid

kill 指令向固定进程发送某些信号，比如 kill -9 7302，就表示向 PID 为 7302 的进程，发送了一个 SIGKILL 的信号，然后用 ps 显示发现 hello 进程已经被终结了。

7、不停乱按

```
stuyxr@ubuntu:~/csapp/final$ ./hello 1170300418 yxr
Hello 1170300418 yxr
dsfs
Hello 1170300418 yxr
f

sdf

Hello 1170300418 yxr
d
Hello 1170300418 yxr

d
Hello 1170300418 yxr
d
s
Hello 1170300418 yxr
dfsdfsffagdfgHello 1170300418 yxr
dgdgfdgfdgfdgfdgHello 1170300418 yxr
gHello 1170300418 yxr
sHello 1170300418 yxr
fdf55655stuyxr@ubuntu:~/csapp/final$ f
f: command not found
stuyxr@ubuntu:~/csapp/final$
stuyxr@ubuntu:~/csapp/final$
stuyxr@ubuntu:~/csapp/final$
stuyxr@ubuntu:~/csapp/final$ sdf

Command 'sdf' not found, but can be installed with:

sudo apt install sdf

stuyxr@ubuntu:~/csapp/final$
stuyxr@ubuntu:~/csapp/final$
stuyxr@ubuntu:~/csapp/final$ d
d: command not found
stuyxr@ubuntu:~/csapp/final$
stuyxr@ubuntu:~/csapp/final$ d
d: command not found
```

图 6-10 不停乱按的效果

这是因为在 `hello` 执行时随便乱按，但是程序只有再 `sleep20s` 之后才会读入字符，所以现在键入字符对程序不会有任何影响，直到 20s 之后，读了一个字符，程序结束了。我们之前键入的乱序字符串会被视为命令行。我们就和在命令行键入了随机字符串的效果一样了。

6.7 本章小结

本章介绍进程的概念与作用，简述了壳 Shell 的作用与处理流程，介绍 `hello` 的 `fork` 进程创建过程、`hello` 的 `execve` 过程和 `hello` 的进程执行。还解析了在执行 `hello` 时各种异常与信号处理。

(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

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。hello.o 里面的相对偏移地址就是逻辑地址。

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

虚拟地址：CPU 通过生成一个虚拟地址。就是 hello 里面的虚拟内存地址。CSAPP 上讲的虚拟地址就是线性地址。

物理地址：用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。计算机系统的主存被组织成一个由 M 个连续的字节大小的单元组成的数组。每字节都有一个唯一的物理地址。就是 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

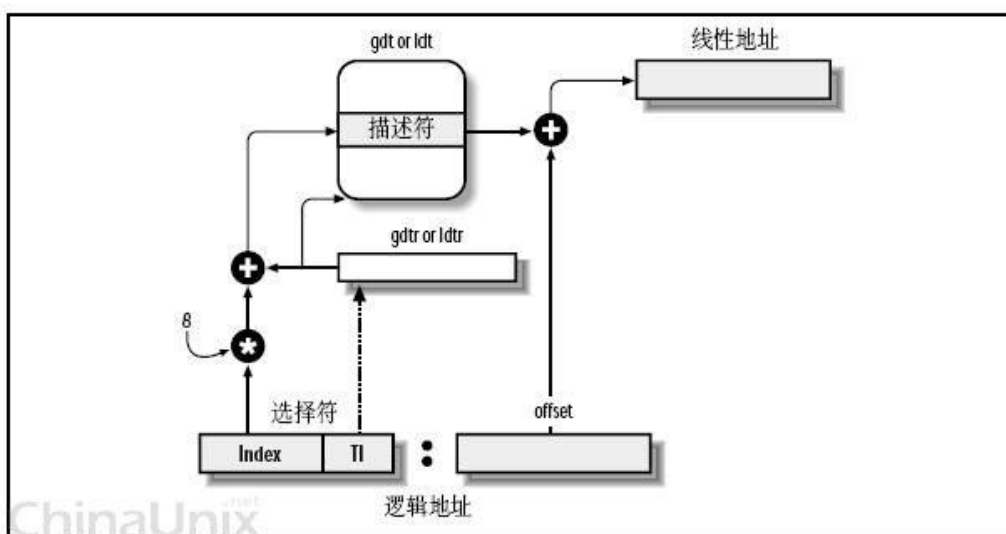


图 7-1 段式管理方法

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，段选择符的结构如下：

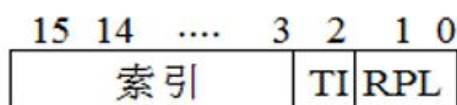


图 7-2 段选择符的结构

- 1、看段选择符的 T1 (段描述符的判别符)=0 还是 1，知道当前要转换是 GDT (全局段描述符) 中的段，还是 LDT (局部段描述符) 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。

7.3 Hello 的线性地址到物理地址的变换-页式管理

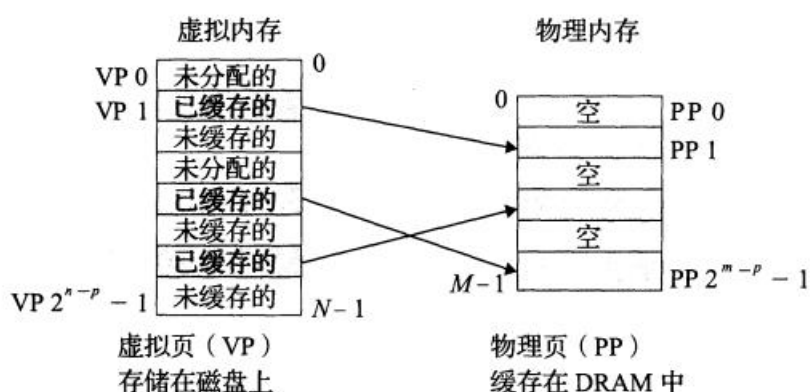


图 7-3 VM 是如何使用主存作为缓存的

从线性地址到物理地址的变换通过页式管理实现。

虚拟内存被组织为一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组。每字节都有一个唯一的虚拟地址。磁盘上数组的内容被缓存在主存中。虚拟页是带虚拟内存系统将虚拟内存分割为大小固定的块，作为磁盘和主存（较高层）之间的传输单元。VM 系统通过将虚拟内存分割成虚拟虚拟页的大小固定的块来解决。任意时刻，虚拟页面的集合分为三个不相交的子集：未分配的、缓存的、未缓存的，如图 7-3 所示。

页表就是一个页表条目的数组。虚拟地址空间中的每个页在页表中一个固定偏移量出都有一个 PTE。每个 PTE 由一个有效位和一个 n 位地址字段组成。有效位表面该虚拟页是否缓存在 DRAM 中，如果设置了有效位，那么地址字段就表示 DRAM 中相应的物理页的起始位置，这个物理页中缓存了该虚拟页，如果没有有效位，若是空地址就表明未分配，否则就执行该虚拟页在磁盘上的起始位置。如图 7-4 为页表结构，如图 7-5、7-6 为虚拟内存地址、物理地址的结构图。

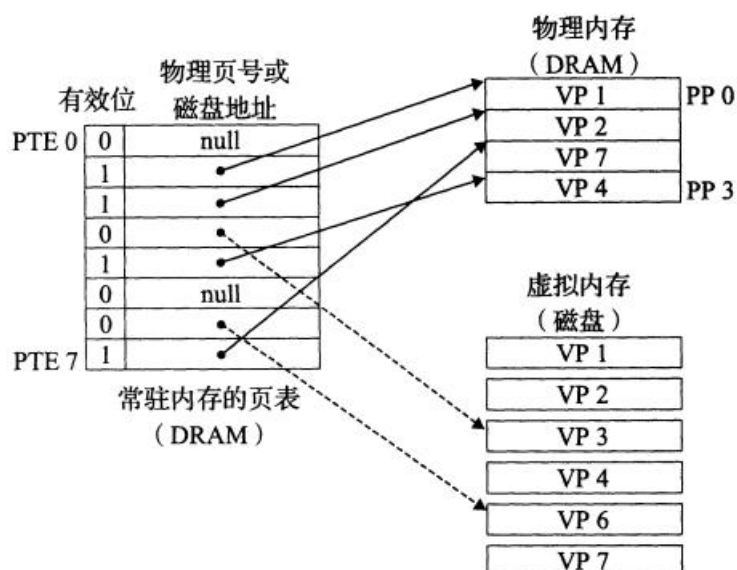


图 7-4 页表结构

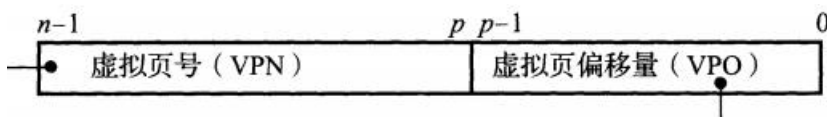


图 7-5 虚拟地址结构

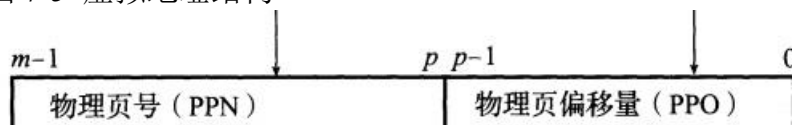


图 7-6 物理地址结构

其中 $VPO = PPO$ ，也就是一个引用相对于虚拟页的偏移量和相对于物理页的偏移量相等。

由虚拟地址变为物理地址需要 MMU 来进行翻译。

当有效位为 1 时，我们直接在页表找到对应的物理页号 PPN， $PPN + VPO$ 就是 $PPN + PPO$ 也就是物理地址了。

当有效位为 0 时，说明内存中不存在当前页，此时 MMU 会选择一个内存中的页为牺牲页，用当前页代替这个牺牲页，更改页表信息，完成这次读取。

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

1、TLB

每次 CPU 产生一个虚拟地址，MMU 就必须查阅一个 PTE，这样我们就需要从内存中读取数据。为了减少时间开销，设计了 TLB。TLB 是一个关于 PTE 的缓存，是 CPU 的一部分。

TLB 是一个小的、虚拟寻址的缓存，其每一行都保存着一个由单个 PTE 组成的块。结构如下图：其中 TLB 是标记位，TLB 是组索引，VPO 是虚拟页偏移量

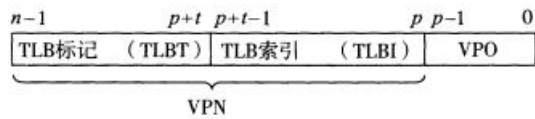


图 7-7 TLB 结构

图 7-7 位组数 $T=2^t$ ，页表大小 $P=2^p$ ，虚拟地址 n 位的一个 TLB 的结构。
 下图是 TLB 命中与不命中的两种情况。

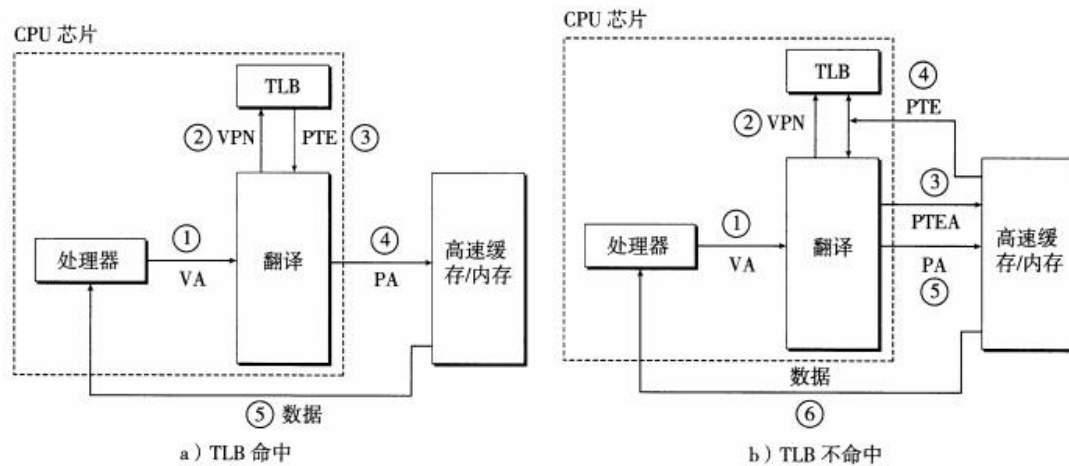


图 7-8 TLB 命中与不命中的操作图

TLB 命中时，我们从硬件结构 TLB 中读取页表信息就可以了。TLB 不命中时，我们只能从内存中读取，然后更新 TLB 即可。

2、四级页表

图 7-9 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。

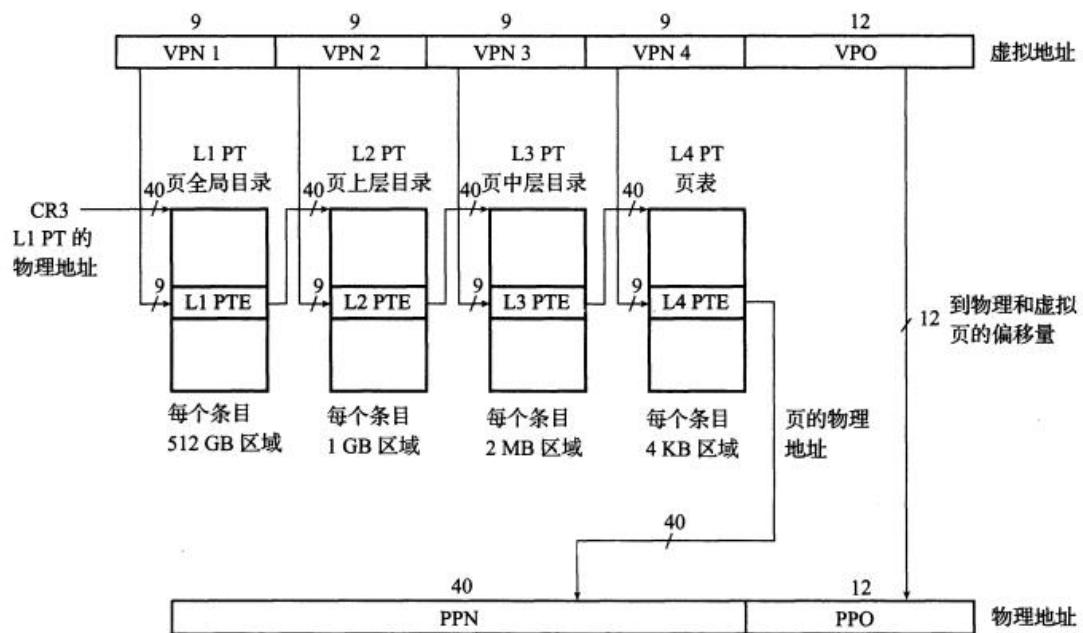


图 7-9 Core i7 页表翻译

7.5 三级 Cache 支持下的物理内存访问

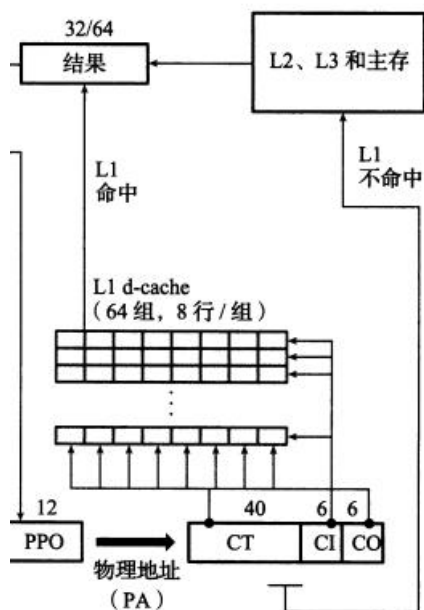


图 7-10 物理内存的访问

在上一步中我们获得了物理地址 VA, 如图 7-10, 使用 CI 进行组索引, 每组 8 路, 对 8 路的块分别匹配 CT 如果匹配成功且块的 valid 标志位为 1, 则命中, 根据数据偏移量 CO 取出数据返回。

如果没有匹配成功或者匹配成功但是标志位是 1，则不命中，向下一级缓存中查询数据。查询到数据之后，一种简单的放置策略如下：如果映射到的组内有空闲块，则直接放置，否则组内都是有效块，产生冲突，则采用某种策略进行替换。

7.6 hello 进程 fork 时的内存映射

当 fork 被调用时，内核会为新进程创建各种数据结构，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，它创建了当前进程的 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

加载并运行 hello 需要以下几个步骤：

- 1、删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。
- 2、映射私有区域。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。代码和数据区域被映射为 hello.out 文件中的 .text 和 .data 区。bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello.out 中，栈和堆地址也是请求二进制零的，初始长度为零。图 7.9 概括了私有区域的不同映射。
- 3、映射贡献区域。如果 hello.out 程序与共享对象（或目标）链接，比如标准 C 库 libc.so，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。
- 4、设置程序计数器。execve 做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

execve 函数加载并运行可执行文件 hello，且带参数列表 argv 和环境变量列表 envp。argv 变量指向一个以 null 结尾的指针数组，每一个指针都指向一个参数字符串。envp 变量指向一个以变量指向一个以 null 结尾的指针数组，每个指针指向一个环境变量字符串。

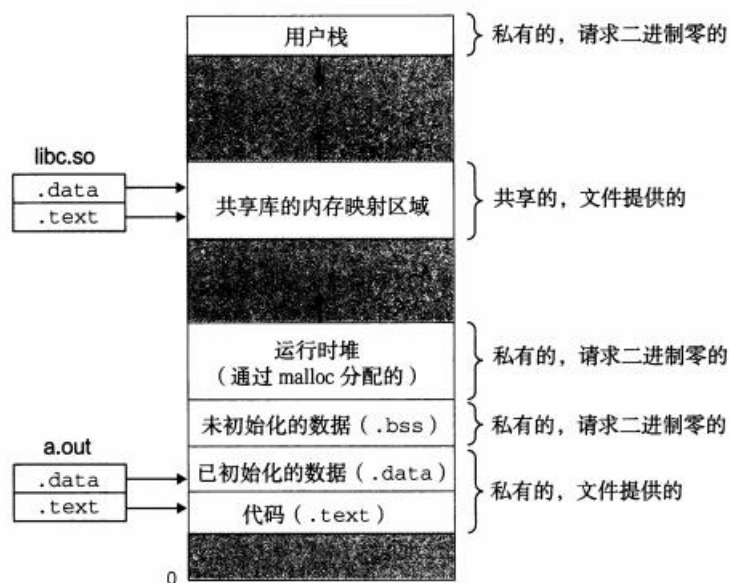


图 7-11 加载器是如何映射用户地址空间区域的

7.8 缺页故障与缺页中断处理

(以下格式自行编排, 编辑时删除)

7.9 动态存储分配管理

7.9.1 动态内存分配器

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。

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。

7.9.2 动态内存管理的基本方法与策略

1、隐式空闲链表

首先, 这里用简单的隐式空闲链表来组织堆, 它的空闲块通过头部中的大小字段隐含地连接着, 可以通过遍历堆中所有的块来间接地遍历整个空闲块的集合。并且, 我们需要一个某种特殊标记的结束块。分配器的合并方法是搜索整个链表, 记住前面块的位置, 直到我们到达当前块。使用隐式空闲链表意味着每次 `free` 需

要的时间开销与堆的大小成线性关系。

Knuth 提出了一种聪明而通用的技术,叫做边界标记,允许在常数时间内进行对前面块的合并。他通过在每个块的结尾处添加一个脚部,其中脚部就是头部的一个副本。如果每个块包括这样一个脚部,那么分配器就可以通过检查它的脚部,判断前面一个块的起始位置和状态,这个脚部总是在距当前块开始位置一个字的距离。

考虑当分配器释放当前块时所有可能存在的情况:前空后空、前空后不空、前不空后空、前不空后不空。

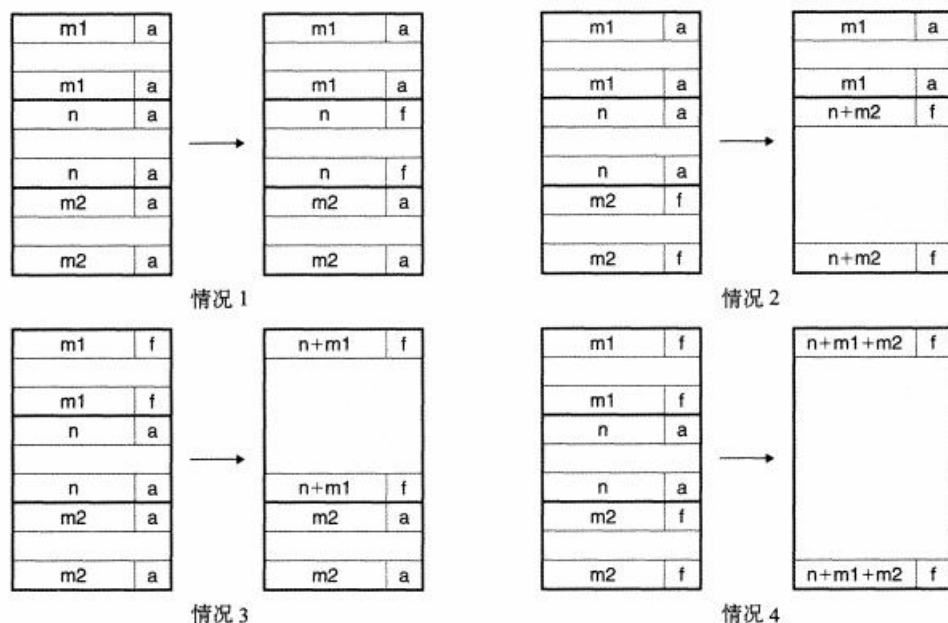


图 7-12 使用边界标记的合并

2、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使线性,也可能是个常数,这取决于我们选择的排序策略:

1)使用先进后出的顺序维护链表,将新释放的块放置在链表的最开始处。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的块。在这种情况下,释放一个块可以在常数时间内完成。如果使用边界标记,合并也可以在常数时间内完成。

2)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 排序的首次适配由更高的内存利用率,接近最佳适配的利用率。

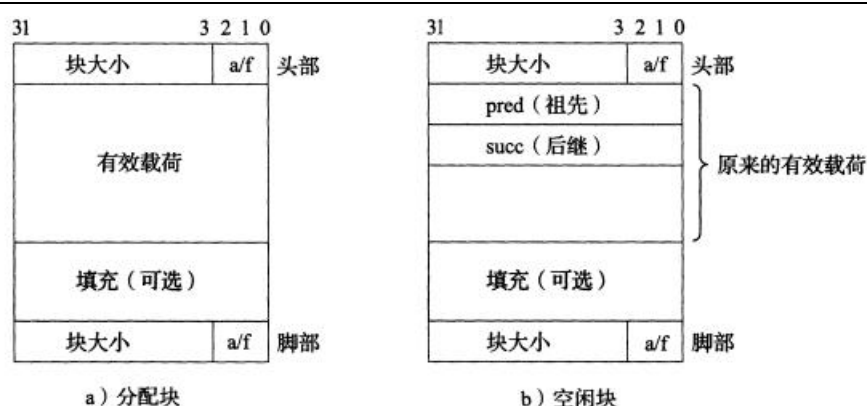


图 7-13 显式空闲链表的堆块结构

3、分离的空闲链表

为了减少分配时间，采用分离存储，就是维护多个链表，其中每个链表中的块有大致相等的大小。一般的思路是将可能的块大小分成一些等价类。以下是两种基本的方法：

1) 简单分离存储：每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。分配块时，检查相应的空闲链表，如果不空，简单分配其中第一块的全部，如果空，就像操作系统请求一个固定大小的额外内存片，将这个片分成大小相等的块。释放一个块，分配器只需要简单地将这个块插入到相应的空闲链表的前部

2) 分离适配：每个空闲链表时和一个大小类相关联的，并且组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，这些块的大小时大小类的成员。分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个空闲块。找到了，就分割，将剩余部分插入到合适的空闲链表中，否则继续查找空闲链表。若空闲链表中没有合适的块，就申请额外的内存。释放一个块时，要进行合并然后插入。

7.10 本章小结

这一章描述了 hello 的存储的管理机制和异常处理机制，让我们了解了一个系统内核如何为进程 hello 分配资源、空间、堆栈等等。虚拟内存的引入，让资源管理与分配变得简单很多。在此基础上，还增加了 TLB，多级页表等优化方式。

(第 7 章 2 分)

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

所有的 I/O 设备都被模型化为文件，内核也被映射为文件，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，这使得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

1、打开文件。

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

返回一个小的非负整数，即描述符。用描述符来标识文件。返回：若成功则为新文件描述符，若出错为-1。

2、改变当前文件位置。

```
off_t lseek(int filedes, off_t offset, int whence);
```

返回值：新的偏移量（成功），-1（失败）

从文件开头起始的字节偏移量。系统内核保持一个文件位置 k ，对于每个打开的文件，起始值为 0。应用程序执行 `seek`，设置当前位置 k ，通过调用 `lseek` 函数，显示地修改当前文件位置。

3、读写文件。

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

返回：若成功则为读的字节数，若 EOF 则为 0，若出错为-1。

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

返回：若成功则为写的字节数，若出错则为-1。

读操作：从文件拷贝 n 个字节到存储器，从当前文件位置 k 开始，将 k 增加到 $k+n$ ，对于一个大小为 m 字节的文件，当 $k \geq m$ 时，读操作触发一个 EOF 的条件。写操作：

从存储器拷贝 n 个字节到文件， k 更新为 $k+n$ 。

4、关闭文件。

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

返回：若成功则为 0，若出错则为-1。

内核释放文件打开时创建的数据结构，并恢复描述符到描述符池中，进程通过调

用 close 函数关闭一个打开的文件。关闭一个已关闭的描述符会出错。

8.3 printf 的实现分析

```
int printf(const char *fmt, ...)
{
    int i;
    char buf[256];
    va_list arg = (va_list)((char*)&fmt + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
    return i;
}
```

图 8-1 printf 代码

如上图所示，printf 调用了两个外部函数，一个是 vsprintf，还有一个是 write。arg 还获得第二个参数，即输出的时候格式化串对应的值。

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;
    for (p = buf; *fmt; fmt++)
    {
        if (*fmt != '%') //忽略无关字符
        {
            *p++ = *fmt;
            continue;
        }
        fmt++;
        switch (*fmt)
        {
            case 'x': //只处理%x一种情况
                itoa(tmp, *((int*)p_next_arg)); //将输入参数值转化为字符串保存在tmp
                strcpy(p, tmp); //将tmp字符串复制到p处
                p_next_arg += 4; //下一个参数值地址
                p += strlen(tmp); //放下一个参数值的地址
                break;
            case 's':
                break;
            default:
                break;
        }
    }
    return (p - buf); //返回最后生成的字符串的长度
}
```

图 8-2 vsprintf 代码

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

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

图 8-3 write 函数

write 函数是将 buf 中的 i 个元素写到终端的函数。

所以 Printf 的运行过程是从 vsprintf 生成显示信息，显示信息传送到 write 系统函数，write 函数再陷阱-系统调用 int 0x80 或 syscall. 字符显示驱动子程序。从 ASCII 到字模库到显示 vram。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。

8.4 getchar 的实现分析

异步异常-键盘中断的处理：键盘中断处理子程序。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

本章节讲述了一下 linux 的 I/O 设备管理机制，简单分析了 printf 函数和 getchar 函数原理。了解 Unix I/O 将帮助我们更深刻地理解其他的系统概念。

(第 8 章 1 分)

结论

我们俩总结一下 hello 的一生。

- 1、编写代码，通过 I/O 设备写入到磁盘中的 hello.c 中。
- 2、预处理，将 hello.c 经过预处理器 cpp 预处理为 hello.i，依然是一个 c 语言陈旭。
- 3、编译，将 hello.i 经过编译器 ccl 翻译成 hello.s，内容为汇编语言。
- 4、汇编，将 hello.s 经过汇编器 as 汇编成 hello.o，现在的 hello 是一个二进制文件了，不过还能执行。
- 5、链接，将 hello.s 经过汇编器 ld 链接成 hello，现在 hello 是一个可执行二进制文件了。
- 6、shell 通过 fork 创建子进程，子进程是父进程的副本。
- 7、子进程 execve 加载 hello，使 hello 加载到内存，改变 pc 的值，堆栈的状态等等。
- 8、磁盘读取、虚拟内存映射、CPU 执行指令、内核调度、缓存加载数据、信号处理、Unix I/O 输入与输出；
- 9、hello 进程结束，父进程 shell 回收之。

一个简单的 `hello` 程序，大概是我们接触 `c` 语言第一个写的程序了，它却蕴含着如此深刻的知识。通过理解“`hello` 的一生”，我也完成了一次系统的知识梳理，构造了一个成体系的计算机知识结构，并且对这样一系列的命令行操作更加熟悉了。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

文件	内容
hello.i	预处理过的源程序
hello.s	汇编程序
hello.o	可重定位目标程序
hello	可执行程序
hello_o.elf	hello.o 通过 readelf 查看的 elf 结构文本
hello.elf	hello 通过 readelf 查看的 elf 结构文本
hello_o.asm	hello.o 通过 objdump 查看的反汇编文本
hello.asm	hello 通过 objdump 查看的反汇编文本

列出所有的中间产物的文件名，并予以说明起作用。

(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] 百度百科. 预处理命令. <https://baike.baidu.com/item/预处理命令/10204389>.
- [2] Linux 内核中的 printf 实现
<https://blog.csdn.net/u012158332/article/details/78675427>
- [3] Linux/Unix 中系统级 IO. <https://www.cnblogs.com/whc-uestc/p/4365507.html>
- [4] 浅谈 sleep()和 wait(). https://blog.csdn.net/qq_34490018/article/details/81609147
- [5] (美) 布赖恩特 (Bryant,R.E.). 深入理解计算机系统. 北京: 机械工业出版社
- [6] lseek 函数的用法. <https://blog.csdn.net/songyang516/article/details/6779950>

(参考文献 0 分, 缺失 -1 分)

个人网站:

<https://stuyxr.com>