

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</u>
學 號	<u>1170300217</u>
班 級	<u>1736101</u>
學 生	<u>于海波</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

摘要是论文内容的高度概括，应具有独立性和自含性，即不阅读论文的全文，就能获得必要的信息。摘要应包括本论文的目的、主要内容、方法、成果及其理论与实际意义。摘要中不宜使用公式、结构式、图表和非公知公用的符号与术语，不标注引用文献编号，同时避免将摘要写成目录式的内容介绍。

**关键词：**程序；生命周期；预处理；编译；汇编；链接；进程管理；存储管理；IO管理；P2P；020；；

（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

<b>第 1 章 概述</b>	<b>- 4 -</b>
1.1 HELLO 简介	- 4 -
1.2 环境与工具	- 4 -
1.3 中间结果	- 4 -
1.4 本章小结	- 4 -
<b>第 2 章 预处理</b>	<b>- 5 -</b>
2.1 预处理的概念与作用	- 5 -
2.2 在 UBUNTU 下预处理的命令	- 5 -
2.3 HELLO 的预处理结果解析	- 5 -
2.4 本章小结	- 5 -
<b>第 3 章 编译</b>	<b>- 7 -</b>
3.1 编译的概念与作用	- 7 -
3.2 在 UBUNTU 下编译的命令	- 7 -
3.3 HELLO 的编译结果解析	- 7 -
3.4 本章小结	- 12 -
<b>第 4 章 汇编</b>	<b>- 14 -</b>
4.1 汇编的概念与作用	- 14 -
4.2 在 UBUNTU 下汇编的命令	- 14 -
4.3 可重定位目标 ELF 格式	- 14 -
4.4 HELLO.O 的结果解析	- 16 -
4.5 本章小结	- 16 -
<b>第 5 章 链接</b>	<b>- 18 -</b>
5.1 链接的概念与作用	- 18 -
5.2 在 UBUNTU 下链接的命令	- 18 -
5.3 可执行目标文件 HELLO 的格式	- 18 -
5.4 HELLO 的虚拟地址空间	- 19 -
5.5 链接的重定位过程分析	- 20 -
5.6 HELLO 的执行流程	- 20 -
5.7 HELLO 的动态链接分析	- 21 -
5.8 本章小结	- 22 -
<b>第 6 章 HELLO 进程管理</b>	<b>- 24 -</b>
6.1 进程的概念与作用	- 24 -
6.2 简述壳 SHELL-BASH 的作用与处理流程	- 24 -
6.3 HELLO 的 FORK 进程创建过程	- 24 -

6.4 HELLO 的 EXECVE 过程 .....	- 24 -
6.5 HELLO 的进程执行 .....	- 25 -
6.6 HELLO 的异常与信号处理 .....	- 25 -
6.7 本章小结 .....	- 26 -
<b>第 7 章 HELLO 的存储管理 .....</b>	<b>- 28 -</b>
7.1 HELLO 的存储器地址空间 .....	- 28 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理 .....	- 28 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理 .....	- 28 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换 .....	- 29 -
7.5 三级 CACHE 支持下的物理内存访问 .....	- 30 -
7.6 HELLO 进程 FORK 时的内存映射 .....	- 31 -
7.7 HELLO 进程 EXECVE 时的内存映射 .....	- 31 -
7.8 缺页故障与缺页中断处理 .....	- 32 -
7.9 动态存储分配管理 .....	- 32 -
7.10 本章小结 .....	- 32 -
<b>第 8 章 HELLO 的 IO 管理 .....</b>	<b>- 35 -</b>
8.1 LINUX 的 IO 设备管理方法 .....	- 35 -
8.2 简述 UNIX IO 接口及其函数 .....	- 35 -
8.3 PRINTF 的实现分析 .....	- 35 -
8.4 GETCHAR 的实现分析 .....	- 36 -
8.5 本章小结 .....	- 38 -
<b>结论 .....</b>	<b>- 38 -</b>
<b>附件 .....</b>	<b>- 39 -</b>
<b>参考文献 .....</b>	<b>- 40 -</b>

# 第 1 章 概述

## 1.1 Hello 简介

1. P2P: 在 linux 中, `hello.c` 经过 `cpp` 的预处理、`ccl` 的编译、`ld` 的链接, 成为可执行目标程序 `hello`, 之后在 `shell` 中运行该程序, `fork` 产生子进程, 于是 `hello` 便从工程切分成为进程。

2. O2O: `shell` 程序使用 `execve` 函数载入程序并映射虚拟内存。进入程序入口时开始载入物理内存, 并从 `main` 函数开始执行目标函数, 之后处理器为该程序执行逻辑控制流, 执行程序。运行结束之后, `shell` 父进程回收 `hello` 进程。

## 1.2 环境与工具

硬件环境: Intel Core i5-6200U x64CPU, 8G RAM, 256G SSD

软件环境: Ubuntu 18.04.1 LTS

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

## 1.3 中间结果

`hello.i` `hello.c` 预处理的文本程序  
`hello.s` `hello.i` 编译的汇编文本程序  
`hello.o` `hello.s` 汇编的二进制文件  
`hello` `hello.o` 链接的可执行二进制文件  
`hello1.s` `hello` 使用 `objdump` 反汇编后的汇编文本程序  
`hello.elf` `hello` 的 `elf` 表  
`hello1.c` 用于测试 `sleepsecs` 值得文本程序  
`hello1` `hello1.c` 生成得可执行二进制文件

## 1.4 本章小结

概述了 `hello` 的 P2P 和 O2O 过程, 列举了实验的环境和工具以及产生的中间文件。

(第 1 章 0.5 分)

## 第 2 章 预处理

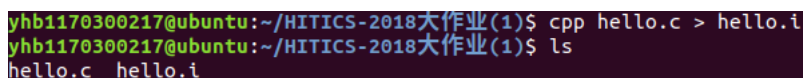
### 2.1 预处理的概念与作用

预处理的概念：预处理一般是指在程序源代码被翻译为目标代码的过程中，生成二进制代码之前的过程。预处理中会展开以#起始的行，试图解释为预处理指令 (preprocessing directive)，其中 ISO C/C++ 要求支持的包括 #if/#ifdef/#ifndef/#else/#elif/#endif（条件编译）、#define（宏定义）、#include（源文件包含）、#line（行控制）、#error（错误指令）、#pragma（和实现相关的杂注）以及单独的#（空指令）。预处理指令一般被用来使源代码在不同的执行环境中被方便的修改或者编译。

预处理的作用：预处理会从系统的头文件包中将这三个头文件插入到 hello.c 的文本中生成 hello.i 文件。在编译代码的第一时间，就把其设定的标识符，全部一一替代，成为了中间码后，再进行正式的编译工作，便于编译。

### 2.2 在 Ubuntu 下预处理的命令

命令：cpp hello.c > hello.i



```
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ cpp hello.c > hello.i
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ ls
hello.c  hello.i
```

### 2.3 Hello 的预处理结果解析

打开 hello.i 之后发现，整个 hello.i 程序已经拓展为 3188 行，main 函数出现在 hello.c 中的代码自 3099 行开始。

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

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

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

增加的文本其实是三个头文件的源码。

其中有两种形式：

(1) 用于描述运行库在计算机内的位置

```
3100 # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
3101 # 955 "/usr/include/stdlib.h" 2 3 4
3102 # 967 "/usr/include/stdlib.h" 3 4
3103
3104 # 9 "hello.c" 2
```

(2) 声明可能用的的函数名：

```
typedef int __io_close_fn (void *__cookie);
# 385 "/usr/include/libio.h" 3 4
extern int __underflow (_IO_FILE *);
extern int __uflow (_IO_FILE *);
extern int __overflow (_IO_FILE *, int);
# 429 "/usr/include/libio.h" 3 4
extern int _IO_getc (_IO_FILE *__fp);
extern int _IO_putc (int __c, _IO_FILE *__fp);
extern int _IO_feof (_IO_FILE *__fp) __attribute__((__nothrow__, __leaf__));
extern int _IO_ferror (_IO_FILE *__fp) __attribute__((__nothrow__, __leaf__));
```

## 2.4 本章小结

概括了预处理的概念和作用，详细说明了 linux 对 c 文本预处理的命令和过程，

分

析了预处理结果 hello.i 文件的结构。（第 2 章 0.5 分）

## 第3章 编译

### 3.1 编译的概念与作用

编译的概念：

把高级语言文本程序翻译成等价的汇编语言文本程序。

编译的作用：

生成二进制可执行程序 的必由之路。

### 3.2 在 Ubuntu 下编译的命令

命令：gcc -S hello.i -o hello.s

```
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ gcc -S hello.i -o hello.o
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ ls
hello.c  hello.i  hello.o
```

### 3.3 Hello 的编译结果解析

#### 3.30 伪指令：

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

#### 3.31 数据

hello.s 中用到的 C 数据类型有：整数、字符串、数组。



## 字符串

程序中的字符串分别是：

“Usage: Hello 学号 姓名!\n”，第一个 printf 传入的输出格式化参数，在 hello.s 中声明，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，一个\代表一个字节。

“Hello %s %s\n”，第二个 printf 传入的输出格式化参数 其中后两个字符串都声明在了.rodata 只读数据节。

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

## 整数

程序中涉及的整数有：

int sleepsecs: sleepsecs 在 C 程序中被声明为全局变量，且已经被赋值，编译器处理时在.data 节声明该变量，.data 节存放已经初始化的全局和静态 C 变量。在图 3.3 中，可以看到，编译器首先将 sleepsecs 在.text 代码段中声明为全局变量，其次在.data 段中，设置对齐方式为 4、设置类型为对象、设置大小为 4 字节、设置为 long 类型其值为 2（long 类型在 linux 下与 int 相同为 4B，将 int 声明为 long 应该是编译器偏好）。

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

1. int i: 编译器将局部变量存储在寄存器或者栈空间中，在 hello.s 中编译器将 i 存储在栈上空间-4(%rbp)中，可以看出 i 占据了栈中的 4B。
2. int argc: 作为第一个参数传入。
3. 立即数: 其他整形数据的出现都是以立即数的形式出现的，直接硬编码在汇编代码中。

## 数组

程序中涉及数组的是：char \*argv[] main，函数执行时输入的命令行，argv 作为存放 char 指针的数组同时是第二个参数传入。

argv 单个元素 char\* 大小为 8B，argv 指针指向已经分配好的、一片存放着字符指针的连续空间，起始地址为 argv，main 函数中访问数组元素 argv[1],argv[2]时，按照起始地址 argv 大小 8B 计算数据地址取数据，在 hello.s 中，使用两次(%rax)（两次 rax 分别为 argv[1]和 argv[2]的地址）取出其值。

```

.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)

```

### 3.32 赋值

程序中涉及的赋值操作有：

1. `int sleepsecs=2.5`：因为 `sleepsecs` 是全局变量，所以直接在 `.data` 节中将 `sleepsecs` 声明为值 2 的 `long` 类型数据。
2. `i=0`：整型数据的赋值使用 `mov` 指令完成，根据数据的大小不同使用不同后缀，分别为：

因为 `i` 是 4B 的 `int` 类型，所以使用 `movl` 进行赋值

```

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

```

#### 3.3.3 类型转换

程序中涉及隐式类型转换的是：`int sleepsecs=2.5`，将浮点数类型的 2.5 转换为 `int` 类型。

当在 `double` 或 `float` 向 `int` 进行类型转换的时候，程序改变数值和位模式的原则是：值会向零舍入。例如 1.999 将被转换成 1，-1.999 将被转换成 -1。进一步来讲，可能会产生值溢出的情况，与 Intel 兼容的微处理器指定位模式 `[10...000]` 为整数不确定值，一个浮点数到整数的转换，如果不能为该浮点数找到一个合适的整数近似值，就会产生一个整数不确定值。

浮点数默认类型为 `double`，所以上述强制转化是 `double` 强制转化为 `int` 类型。遵从向零舍入的原则，将 2.5 舍入为 2。

#### 3.3.4 算术操作

进行数据算术操作的汇编指令有：

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

程序中涉及的算数操作有：

1. `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。
2. 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`。

### 3.3.5 关系操作

进行关系操作的汇编指令有：

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

程序中涉及的关系运算为：

`argc!=3`：判断 `argc` 不等于 3。hello.s 中使用 `cmpl $3,-20(%rbp)`，计算 `argc-3` 然

后设置条件码，为下一步 `je` 利用条件码进行跳转作准备。

`i<10`: 判断 `i` 小于 10。hello.s 中使用 `cmpl $9,-4(%rbp)`，计算 `i-9` 然后设置条件码，为下一步 `jle` 利用条件码进行跳转做准备。

### 3.3.6 控制转移

程序中涉及的控制转移有：

`if(argv!=3)`: 当 `argv` 不等于 3 的时候执行程序段中的代码。对于 `if` 判断，编译器使用跳转指令实现，首先 `cmpl` 比较 `argv` 和 3，设置条件码，使用 `je` 判断 ZF 标志位，如果为 0，说明 `argv-3=0` `argv==3`，则不执行 `if` 中的代码直接跳转到 `.L2`，否则顺序执行下一条语句，即执行 `if` 中的代码。

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

`for(i=0;i<10;i++)`: 使用计数变量 `i` 循环 10 次。编译器的编译逻辑是，首先无条件跳转到位于循环体 `.L4` 之后的比较代码，使用 `cmpl` 进行比较，如果 `i<=9`，则跳入 `.L4` `for` 循环体执行，否则说明循环结束，顺序执行 `for` 之后的逻辑。

```
.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
```

### 3.3.7 函数操作

函数是一种过程，过程提供了一种封装代码的方式，用一组指定的参数和可选的返回值实现某种功能。P 中调用函数 Q 包含以下动作：

传递控制: 进行过程 Q 的时候，程序计数器必须设置为 Q 的代码的起始地址，然后在返回时，要把程序计数器设置为 P 中调用 Q 后面那条指令的地址。

传递数据: P 必须能够向 Q 提供一个或多个参数，Q 必须能够向 P 中返回一个值。

分配和释放内存：在开始时，Q 可能需要为局部变量分配空间，而在返回前，又必须释放这些空间。

程序中涉及函数操作的有：

main 函数：

传递控制，main 函数因为被调用 call 才能执行（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，分别使用 %rdi 和 %rsi 存储，函数正常出口为 return 0，将 %eax 设置 0 返回。

分配和释放内存，使用 %rbp 记录栈帧的底，函数分配栈帧空间在 %rbp 之上，程序结束时，调用 leave 指令，leave 相当于 `mov %rbp,%rsp, pop %rbp`，恢复栈空间为调用之前的状态，然后 ret 返回，ret 相当 pop IP，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，设置 %rsi 为 argv[1]，%rdx 为 argv[2]。

控制传递：第一次 printf 因为只有一个字符串参数，所以 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

本章主要阐述了编译器是如何处理 C 语言的各个数据类型以及各类操作的，基本都是先给出原理然后结合 hello.c C 程序到 hello.s 汇编代码之间的映射关系作出合理解释。

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

**(第 3 章 2 分)**

## 第 4 章 汇编

### 4.1 汇编的概念与作用

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。这个过程称为汇编，亦即汇编的作用。

### 4.2 在 Ubuntu 下汇编的命令

指令：as hello.s -o hello.o

```

yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ as hello.s -o hello.o
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ ls
hello.c hello.i hello.o hello.s

```

### 4.3 可重定位目标 elf 格式

使用 readelf -a hello.o > hello.elf 指令获得 hello.o 文件的 ELF 格式。其组成如下：

ELF Header: 以 16B 的序列 Magic 开始，Magic 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、字节头部表（section header table）的文件偏移，以及节头部表中条目的大小和数量等信息。

```

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

```

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

节头:						
[号]	名称	类型	地址	偏移量		
	大小	全体大小	旗标	链接	信息	对齐
[ 0]		NULL	0000000000000000	00000000		
	0000000000000000	0000000000000000		0	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

重定位节.rela.text, 一个.text 节中位置的列表, 包含.text 节中需要进行重定位的信息, 当链接器把这个目标文件和其他文件组合时, 需要修改这些位置。图中 8 条重定位信息分别是对.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 函数进行重定位声明。

重定位节 '.rela.text' at offset 0x340 contains 8 entries:					
偏移量	信息	类型	符号值	符号名称 + 加数	
000000000018	000500000002	R_X86_64_PC32	0000000000000000	.rodata - 4	
00000000001d	000c00000004	R_X86_64_PLT32	0000000000000000	puts - 4	
000000000027	000d00000004	R_X86_64_PLT32	0000000000000000	exit - 4	
000000000050	000500000002	R_X86_64_PC32	0000000000000000	.rodata + 1a	
00000000005a	000e00000004	R_X86_64_PLT32	0000000000000000	printf - 4	
000000000060	000900000002	R_X86_64_PC32	0000000000000000	sleepsecs - 4	
000000000067	000f00000004	R_X86_64_PLT32	0000000000000000	sleep - 4	
000000000076	001000000004	R_X86_64_PLT32	0000000000000000	getchar - 4	

.rela 节的包含的信息有 (readelf 显示与 hello.o 中的编码不同, 以 hello.o 为准):



offset	需要进行重定向的代码在.text或.data节中的偏移位置，8个字节。
Info	包括symbol和type两部分，其中symbol占前4个字节，type占后4个字节，symbol代表重定位到的目标在.symtab中的偏移量，type代表重定位的类型
Addend	计算重定位位置的辅助信息，共占8个字节
Type	重定位到的目标的类型
Name	重定向到的目标的名称

.rela.eh\_frame : eh\_frame 节的重定位信息。

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

#### 4.4 Hello.o 的结果解析

使用 `objdump -d -r hello.o > hello.o.objdump` 获得反汇编代码。

主要差别如下：

分支转移：反汇编代码跳转指令的操作数使用的不是段名称如.L3，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。

函数调用：在.s 文件中，函数调用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。这是因为 hello.c 中调用的函数都是共享库中的函数，最终需要通过动态链接器才能确定函数的运行时执行地址，在汇编成为机器语言的时候，对于这些不确定地址的函数调用，将其 call 指令后的相对地址设置为全 0（目标地址正是下一条指令），然后在.rela.text 节中为其添加重定位条目，等待静态链接的进一步确定。

全局变量访问：在.s 文件中，访问 rodata（printf 中的字符串），使用段名称 + %rip，在反汇编代码中 0 + %rip，因为 rodata 中数据地址也是在运行时确定，故访问也需要重定位。所以在汇编成为机器语言时，将操作数设置为全 0 并添加重定位条目。

#### 4.5 本章小结

本章介绍了 hello 从 hello.s 到 hello.o 的汇编过程，通过查看 hello.o 的 elf 格式和使用 objdump 得到反汇编代码与 hello.s 进行比较的方式，间接了解到从汇编语言

映射到机器语言汇编器需要实现的转换。（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。链接可以执行于编译时，也就是在源代码被编译成机器代码时；也可以执行于加载时，也就是在程序被加载器加载到内存并执行时；甚至于运行时，也就是由应用程序来执行。链接是由叫做链接器的程序执行的。链接器使得分离编译成为可能。

### 5.2 在 Ubuntu 下链接的命令

命令：

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

注意：因为需要生成的是 64 位的程序，所以，使用的动态链接器和链接的目标文件都应该是 64 位的。

```
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ 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
yhb1170300217@ubuntu:~/HITICS-2018大作业(1)$ ls
hello hello.c hello.elf hello.i hello.o hello.s
```

### 5.3 可执行目标文件 hello 的格式

使用 `readelf -a hello > hello.elf` 命令生成 hello 程序的 ELF 格式文件。

在 ELF 格式文件中，Section Headers 对 hello 中所有的节信息进行了声明，其中包括大小 Size 以及在程序中的偏移量 Offset，因此根据 Section Headers 中的信息我们就可以用 HexEdit 定位各个节所占的区间（起始位置，大小）。其中 Address 是程序被载入到虚拟地址的起始地址。

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

## 5.4 hello 的虚拟地址空间

使用 edb 打开 hello 程序，通过 edb 的 Data Dump 窗口查看加载到虚拟地址中的 hello 程序。

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。

查看 ELF 格式文件中的 Program Headers，程序头表在执行的时候被使用，它告诉链接器运行时加载的内容并提供动态链接的信息。每一个表项提供了各段在虚拟地址空间和物理地址空间的大小、位置、标志、访问权限和对齐方面的信息。在下面可以看出，程序包含 8 个段：

PHDR 保存程序头表。

INTERP 指定在程序已经从可执行文件映射到内存之后，必须调用的解释器（如动态链接器）。

LOAD 表示一个需要从二进制文件映射到虚拟地址空间的段。其中保存了常量

数据（如字符串）、程序的目标代码等。

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: 指定在重定位结束之后那些内存区域是需要设置只读。

程序头:

Type	Offset	VirtAddr	PhysAddr
	FileSiz	MemSiz	Flags Align
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040
	0x00000000000001c0	0x00000000000001c0	R 0x8
INTERP	0x0000000000000200	0x0000000000400200	0x0000000000400200
	0x00000000000001c	0x00000000000001c	R 0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]			
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000
	0x000000000000076c	0x000000000000076c	R E 0x200000
LOAD	0x0000000000000e50	0x0000000000600e50	0x0000000000600e50
	0x00000000000001f8	0x00000000000001f8	RW 0x200000
DYNAMIC	0x0000000000000e50	0x0000000000600e50	0x0000000000600e50
	0x00000000000001a0	0x00000000000001a0	RW 0x8
NOTE	0x000000000000021c	0x000000000040021c	0x000000000040021c
	0x0000000000000020	0x0000000000000020	R 0x4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000	RW 0x10
GNU_RELRO	0x0000000000000e50	0x0000000000600e50	0x0000000000600e50
	0x00000000000001b0	0x00000000000001b0	R 0x1

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

## 5.5 链接的重定位过程分析

使用 `objdump -d -r hello > hello1.objdump` 获得 hello 的反汇编代码。与 hello.o 反汇编文本 hello.objdump 相比, 在 hello1.objdump 中多的节如下:

1. .interp 保存 ld.so 的路径
2. .note.ABI-tag Linux 下特有的 section
3. .hash 符号的哈希表
4. .gnu.hash GNU 拓展的符号的表
5. .dynstr 存放.dynsym 节中的符号名称
6. .dynsym 运行时/动态符号表
7. .rela.dyn 运行时/动态重定位表
8. .rela.plt .plt 节的重定位条目
9. .init 程序初始化需要执行的代码
10. .plt 动态链接-过程链接表

11. .fini 当程序正常终止时执行的代码
12. .dynamic 存放被 ld.so 使用的动态链接信息
13. .got 动态链接-全局偏移量表-存放变量
14. .got.plt 动态链接-全局偏移量表-存放函数
15. .data 初始化了的数据
16. .comment 一串包含编译器的 NULL-terminated 的字符串

一些理解：使用 ld 命令链接的时候，指定了动态链接器为 64 的 /lib64/ld-linux-x86-64.so.2, crt1.o、crti.o、crtm.o 中主要定义了程序入口 \_start、\_init, \_start 程序调用 hello.c 中的 main 函数。在 rodata 中，链接器解析重定条目时发现两个类型为 R\_X86\_64\_PC32 的对.rodata 的重定位，.rodata 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。

## 5.6 hello 的执行流程

使用 edb 执行 hello，观察函数执行流程，将过程中执行的主要函数列在下面：

程序名称	程序地址
ld-2.27.so!_dl_start	0x7fce 8cc38ea0
ld-2.27.so!_dl_init	0x7fce 8cc47630
hello!_start	0x400500
libc-2.27.so!__libc_start_main	0x7fce 8c867ab0
-libc-2.27.so!__cxa_atexit	0x7fce 8c889430
-libc-2.27.so!__libc_csu_init	0x4005c0
hello!_init	0x400488
libc-2.27.so!_setjmp	0x7fce 8c884c10
-libc-2.27.so!_sigsetjmp	0x7fce 8c884b70
--libc-2.27.so!_sigjmp_save	0x7fce 8c884bd0
hello!main	0x400532
hello!puts@plt	0x4004b0
hello!exit@plt	0x4004e0
*hello!printf@plt	--



*hello!sleep@plt	--
*hello!getchar@plt	--
ld-2.27.so!_dl_runtime_resolve_xsave	0x7fce 8cc4e680
-ld-2.27.so!_dl_fixup	0x7fce 8cc46df0
--ld-2.27.so!_dl_lookup_symbol_x	0x7fce 8cc420b0
libc-2.27.so!exit	0x7fce 8c889128

## 5.7 Hello 的动态链接分析

动态共享链接库中的函数，编译器没有办法预测函数的运行时地址，所以需添加重定位记录，等待动态链接器处理。动态链接器使用过程链接表 PLT+全局偏移量表 GOT 实现函数的动态链接。其中 GOT 中存放函数目标地址，PLT 使用 GOT 中地址跳转到目标函数。

dl\_init 调用之前，对于每一条 PIC 函数调用，调用的目标地址都实际指向 PLT 中的代码。调用之后，如下图 2，两个 8B 数据分别发生改变，如下图 3。重定位表（.plt 节需要重定位的函数的运行时地址）用来确定函数地址。

Address	Hex Data	ASCII Data
00000000:00601000	50 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00	P. ....
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	.....@...
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00	.....@...

Address	Hex Data	ASCII Data
00000000:00601000	50 0e 60 00 00 00 00 00 70 51 92 d3 d9 7f 00 00	P. ....pQ.r...
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	.....@...
00000000:00601030	e6 04 40 00 00 00 00 00 f6 04 40 00 00 00 00 00	.....@...

Address	Hex Data	ASCII Data
7fd9:d3925150	a0 cd 70 d3 d9 7f 00 00 00 00 00 00 00 00 00 00	#pt. ....
7fd9:d3925160	00 c0 6f d3 d9 7f 00 00 00 00 00 00 00 00 00 00	..ot. 重定位表
7fd9:d3925170	00 00 00 00 00 00 00 00 00 57 92 d3 d9 7f 00 00	.....W.r...
7fd9:d3925180	50 0e 60 00 00 00 00 00 10 57 92 d3 d9 7f 00 00	P. ....W.r...
7fd9:d3925190	00 00 00 00 00 00 00 00 70 51 92 d3 d9 7f 00 00	.....pQ.r...
7fd9:d39251a0	00 00 00 00 00 00 00 00 e8 56 92 d3 d9 7f 00 00	.....V.r...

在函数调用时，程序先执行.plt 中的逻辑，再将函数序号压栈，跳转到 PLT 表的第一位，在 PLT[0]中将重定位表地址压栈，然后在动态链接器中使用函数序号和重定位表确定函数运行时地址，之后重写全局偏移量表，并将控制传递给目标函数。之后对同一函数的调用直接跳转到目标函数。最后，目标函数执行完后，返回地址为调用指令的下一条指令。

## 5.8 本章小结

在本章中主要介绍了链接的概念与作用、hello 的 ELF 格式，分析了 hello 的虚拟地址空间、重定位过程、执行流程、动态链接过程。（第 5 章 1 分）



## 第 6 章 hello 进程管理

### 6.1 进程的概念与作用

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。

进程为用户提供了以下假象：我们的程序好像是系统中当前运行的唯一程序一样，我们的程序好像是独占的使用处理器和内存，处理器好像是无间断的执行我们程序中的指令，我们程序中的代码和数据好像是系统内存中唯一的对象。

### 6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用：Shell 是一个用 C 语言编写的程序，他是用户使用 Linux 的桥梁。Shell 是指一种应用程序，Shell 应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。

处理流程：

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

### 6.3 Hello 的 fork 进程创建过程

在终端 Gnome-Terminal 中键入 `./hello 1170300217 于海波`，运行的终端程序会对输入的命令进行解析，因为 `hello` 不是一个内置的 shell 命令所以解析之后终端程序判断 `./hello` 的语义为执行当前目录下的可执行目标文件 `hello`，之后终端程序首先会调用 `fork` 函数创建一个新的运行的子进程，新创建的子进程几乎但不完全与父进程相同，子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一

份副本，这就意味着，当父进程调用 `fork` 时，子进程可以读写父进程中打开的任何文件。父进程与子进程之间最大的区别在于它们拥有不同的 `PID`。

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。在子进程执行期间，父进程默认选项是显示等待子进程的完成。

## 6.4 Hello 的 `execve` 过程

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

## 6.5 Hello 的进程执行

### 1. 重要概念

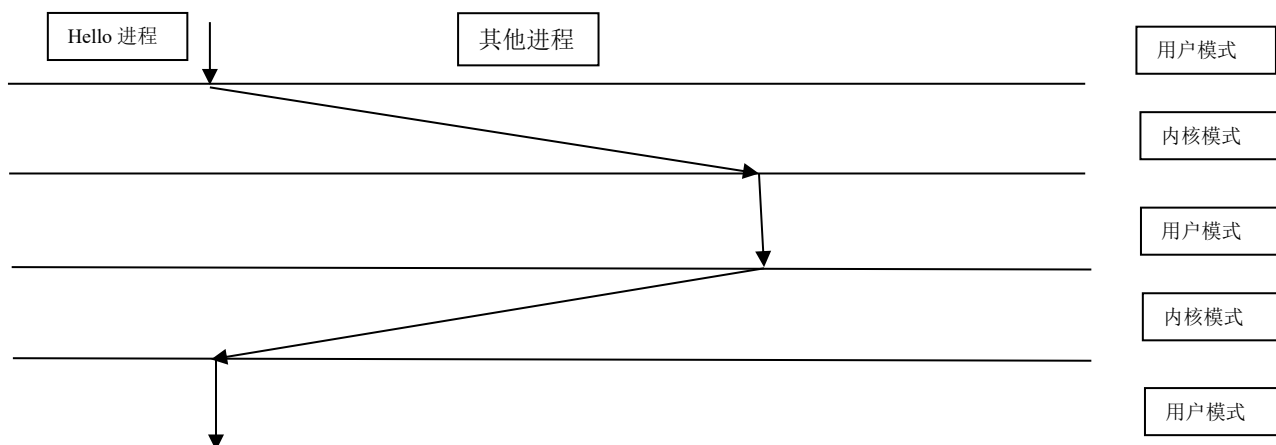
**逻辑控制流：**一系列程序计数器 `PC` 的值的序列叫做逻辑控制流。由于进程是轮流使用处理器的，同一个处理器每个进程执行它的流的一部分后被抢占，然后轮到其他进程。

**用户模式和内核模式：**处理器使用一个寄存器提供两种模式的区分。用户模式的进程不允许执行特殊指令，不允许直接引用地址空间中内核区的代码和数据；内核模式进程可以执行指令集中的任何命令，并且可以访问系统中的任何内存位置。

**上下文：**上下文就是内核重新启动一个被抢占的进程所需要恢复的原来的状态，由寄存器、程序计数器、用户栈、内核栈和内核数据结构等对象的值构成。

### 2. `sleep` 进程的调度过程

假如 `hello` 程序发生被抢占的情况，则进行上下文切换。内核调度新的进程运行后就会抢占当前进程，并保存以前进程的上下文并恢复当前进程被保存的上下文，将控制传递给当前进程。



如上图，hello 初始运行在用户模式，调用 `sleep` 之后陷入内核模式，内核处理休眠请求并释放当前进程，将 `hello` 进程从运行队列中加入等待队列，内核进行上下文切换，将当前进程的控制权交给其他进程。当 `sleep` 定时器到时发送一个中断信号，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，`hello` 进程继续进行自己的控制逻辑流。`hello` 调用 `getchar` 的时候基本步骤也如上图。执行输入流 `stdin` 的系统调用 `read`，`hello` 陷入内核，内核中的陷阱处理程序请求来自键盘缓冲区的传输，并且完成从键盘缓冲区到内存的数据传输后，中断处理器。与此同时进入内核模式，执行上下文切换，切换到其他进程。当完成键盘缓冲区到内存的数据传输时，引发一个中断信号，此时内核上下文切换回 `hello` 的进程。

## 6.6 hello 的异常与信号处理

当程序执行完成之后，进程被回收。

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

当按下 `ctrl-c` 之后，`shell` 父进程收到 `SIGINT` 信号，信号处理函数的逻辑是结束 `hello`，并回收 `hello` 进程。

乱按只是将屏幕的输入缓存到 `stdin`，当 `getchar` 的时候读出一个 `'\n'` 结尾的字符串（作为一次输入），其他字符串会当做 `shell` 命令行输入。

## 6.7 本章小结

在本章中，阐明了进程的定义与作用，介绍了 `Shell` 的一般处理流程，调用 `fork` 创建新进程，调用 `execve` 执行 `hello`，`hello` 的进程执行，`hello` 的异常与信号处理。（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

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。

逻辑地址：程序代码经过编译后出现在汇编程序中地址。逻辑地址由选择符（在实模式下是描述符，在保护模式下是用来选择描述符的选择符）和偏移量（偏移部分）组成。

线性地址：逻辑地址经过段机制后转化为线性地址，为描述符:偏移量的组合形式。分页机制中线性地址作为输入。

至于虚拟地址，只关注 CSAPP 课本中提到的虚拟地址，实际上就是这里的线性地址。

### 7.2 Intel 逻辑地址到线性地址的变换-段式管理

在保护模式下，分段机制可以描述为：通过解析段寄存器中的段选择符在段描述符表中根据 Index 选择目标描述符条目 Segment Descriptor，从目标描述符中提取出目标段的基地址，最后加上偏移量共同构成线性地址。CPU 位于 32 位模式时，内存 4GB，寄存器和指令都可以寻址整个线性地址空间，所以这时候不再需要使用基地址，将基地址设置为 0，此时逻辑地址=描述符=线性地址。保护模式时分段机制图示如下：

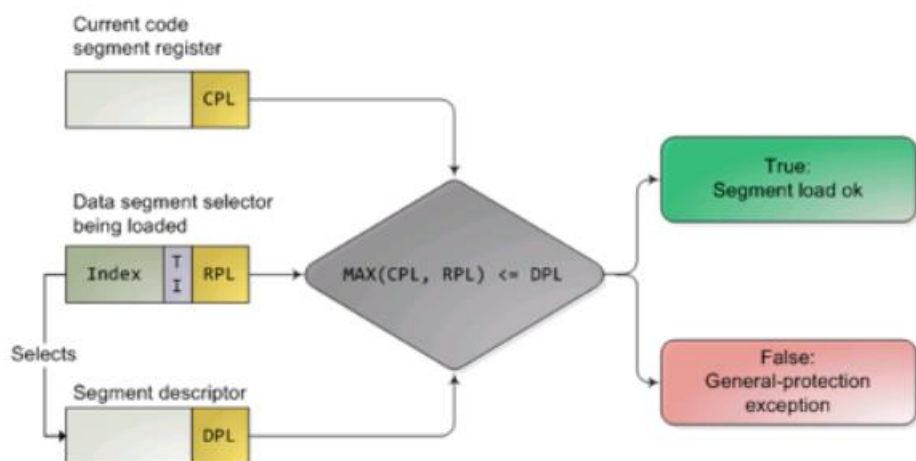


图5

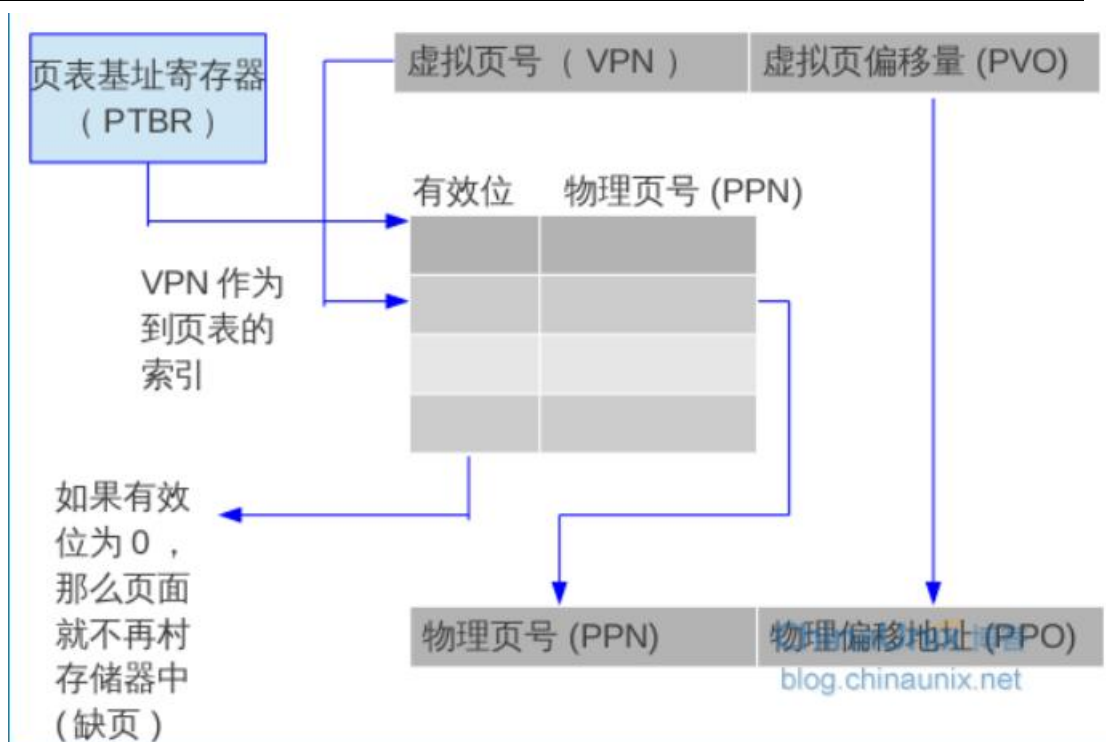
### 7.3 Hello 的线性地址到物理地址的变换-页式管理

线性地址到物理地址之间的转换通过分页机制完成。

分页机制指的是对虚拟地址内存空间进行分页。Linux 虚拟内存组织形式如下图，Linux 将虚拟内存组织成一些段的集合。内核为 hello 进程维护一个段的任务结构，其中每个条目指向一个次级结构体，它描述了虚拟内存的当前状态，而每个次级结构体指向一个三级结构体的链表，一个链表条目对应一个段，所以链表相连指出了 hello 进程虚拟内存中的所有段。

系统将每个段分割为被称为虚拟页的大小固定的块来作为进行数据传输的单元，在 Linux 下每个虚拟页大小为 4KB，物理内存也被分割为物理页。虚拟内存中 MMU 负责地址翻译，MMU 使用存放在物理内存中的被称为页表的数据结构将虚拟页映射到物理页。

地址翻译的基本原理如下图。虚拟地址分为虚拟页号 VPN 和虚拟页偏移量 VPO，根据位数限制的推理可以确定 VPN 和 VPO 分别占多少位。通过页表基址寄存器 PTBR+VPN 获得页表条目 PTE，一条 PTE 中包含有效位，权限信息和物理页号。如果是有效位 0 和 NULL 则代表没有在虚拟内存空间中分配该内存，如果是有效位 0 和非 NULL，则代表在虚拟内存空间中分配了但是没有被缓存到物理内存中，如果有效位是 1 则代表该内存已经缓存在了物理内存中。可以得到其物理页号 PPN，与 VPO 构成物理地址。



## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

条件：：由一个页表大小为 4KB，一个 PTE 条目 8B，共 512 个条目，使用  $\log(512)=9$  位二进制索引，4 个页表共使用 36 位二进制索引，VPN 共 36 位，因为 VA 一共 48 位，所以 VPO 12 位；因为 TLB 共 16 组，所以 TLBI 需  $\log(16)=4$  位，VPN 36 位，所以 TLBT 32 位。

访问过程：CPU 产生虚拟地址 VA，VA 传送给 MMU，MMU 使用 VPN 作为 TLBT(前 32 位)+TLBI(后 4 位)向 TLB 中进行匹配，若命中，则得到 PPN 与 VPO 组合成 PA。若不命中，MMU 向页表中查询，确定第一级页表的起始地址，VPN1 (9bit) 确定在第一级页表中的偏移量，查询出 PTE，如果在物理内存中，确定第二级页表的起始地址，以此类推，最终在第四级页表中查询到 PPN，与 VPO 组合成 PA，并且向 TLB 中添加条目。如果查询 PTE 的时候发现不在物理内存中，则引发缺页故障。其基本模式如下图左半部分：





`execve` 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件 `hello` 中的程序，用 `hello` 程序有效地替代了当前程序。加载并运行 `hello` 需要以下几个步骤：

删除已存在的用户区域，删除当前进程虚拟地址的用户部分中的已存在的区域结构。

映射私有区域，为新程序的代码、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` 文件中的 `.text` 和 `.data` 区，`bss` 区域是请求二进制零的，映射到匿名文件，其大小包含在 `hello` 中，栈和堆地址也是请求二进制零的，初始长度为零。

映射共享区域，`hello` 程序与共享对象 `libc.so` 链接，`libc.so` 是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。

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

## 7.8 缺页故障与缺页中断处理

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中，因此必须从磁盘中取出的时候就会发生故障。其处理流程遵循图 所示的故障处理流程。

缺页中断处理：缺页处理程序是系统内核中的代码，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。当缺页处理程序返回时，CPU 重新启动引起缺页的指令，这条指令再次发送 VA 到 MMU，这次 MMU 就能正常翻译 VA 了。

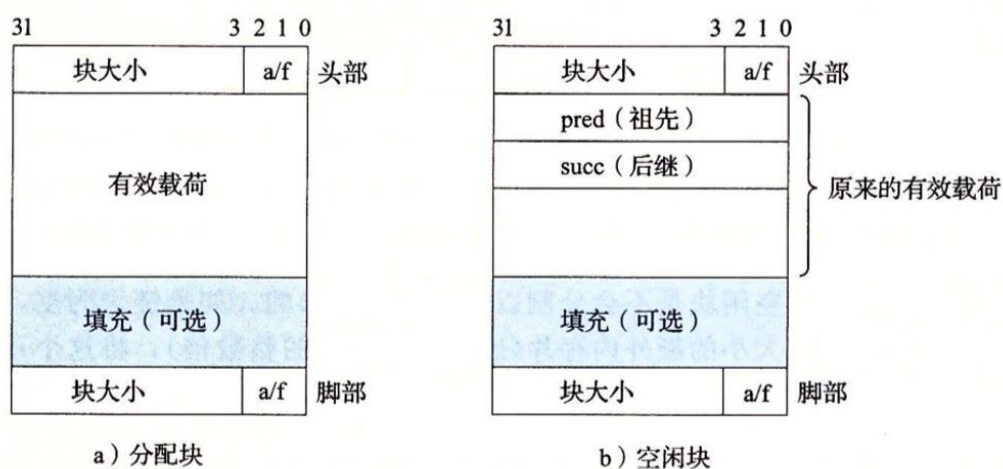
## 7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。分配器将堆视为一组不同大小的块来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块用来分配。空闲块保持空闲，直到显式地被分配。一个已分配的块保持已分配状态，直到被释放。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。

1. 显式执行：显式地释放任何已分配的块。
2. 隐式执行：检测一个已分配块何时不再使用，那么就释放这个块。该过程又被称为垃圾收集。

显示空间链表：将空闲块组织成链表形式的数据结构。堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个前驱和后继指针。使用双向链表而不

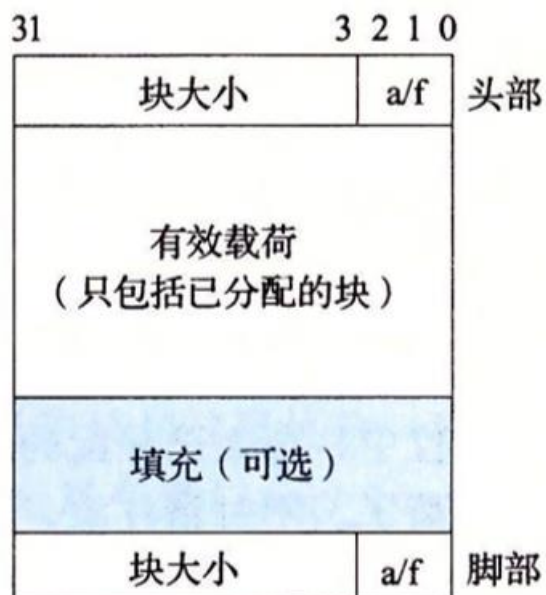
是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。已分配块和空闲块的基本结构如下：



维护链表顺序包括：

1. 后进先出，将新释放的块放置在链表的开始处。
2. 按照地址顺序来维护链表，其中链表中的每个块的地址都小于它的后继块的地址。

带边界标签的隐式空闲链表：在内存块中增加 4 字节的头部和 4 字节的脚部。其中头部用于寻找下一个块，脚部用于寻找上一个块。脚部的设计是专门为了合并空闲块的。其基本结构如下图：



## 7.10 本章小结

本章主要介绍了 hello 的存储器地址空间、intel 的段式管理、hello 的页式管理，以 intel Core7 在指定环境下介绍了 VA 到 PA 的变换、物理内存访问，还介绍了 hello 进程 fork 时的内存映射、execve 时的内存映射、缺页故障与缺页中断处理、动态存储分配管理。（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：所有的 IO 设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

### 8.2 简述 Unix IO 接口及其函数

Unix I/O 接口统一操作：

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。

Shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。

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`，显式地将改变当前文件位置  $k$ 。

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  $m$  字节的而文件，当  $k \geq m$  时，触发 EOF。类似一个写操作就是从内存中复制  $n>0$  个字节到一个文件，从当前文件位置  $k$  开始，然后更新  $k$ 。

关闭文件，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去。

Unix I/O 函数：

`int open(char* filename,int flags,mode_t mode)`，进程通过调用 `open` 函数来打开一个存在的文件或是创建一个新文件的。`open` 函数将 `filename` 转换为一个文件描述符，并且返回描述符数字，返回的描述符总是在进程中当前没有打开的最小描述符，`flags` 参数指明了进程打算如何访问这个文件，`mode` 参数指定了新文件的访问权限位。

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

`ssize_t read(int fd,void *buf,size_t n)`, `read` 函数从描述符为 `fd` 的当前文件位置赋值最多 `n` 个字节到内存位置 `buf`。返回值-1 表示一个错误, 0 表示 EOF, 否则返回值表示的是实际传送的字节数量。

`ssize_t write(int fd,const void *buf,size_t n)`, `write` 函数从内存位置 `buf` 复制至多 `n` 个字节到描述符为 `fd` 的当前文件位置。

### 8.3 printf 的实现分析

查看 `printf` 代码:

```
int printf(const char *fmt, ...)
{
    int i;

    char buf[256];

    va_list arg = (va_list)((char*)&fmt + 4);

    i = vsprintf(buf, fmt, arg);

    write(buf, i);

    return i;
}
```

首先 `arg` 获得第二个不定长参数, 即输出的时候格式化串对应的值。

在 `printf` 中调用系统函数 `write(buf,i)` 将长度为 `i` 的 `buf` 输出。`write` 函数如下:

`write`:

```
mov eax, _NR_write

mov ebx, [esp + 4]
```

```
mov ecx, [esp + 8]
```

```
int INT_VECTOR_SYS_CALL
```

在 `write` 函数中，将栈中参数放入寄存器，`ecx` 是字符个数，`ebx` 存放第一个字符地址，`int INT_VECTOR_SYS_CALL` 代表通过系统调用 `syscall`，查看 `syscall` 的实现：

`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
```

`syscall` 将字符串中的字节“Hello 1170300217 于海波”从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的 ASCII 码。

字符显示驱动子程序将通过 ASCII 码在字模库中找到点阵信息将点阵信息存储到 `vram` 中。

显示芯片会按照一定的刷新频率逐行读取 `vram`，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

于是我们的打印字符串“Hello 1170300217 于海波”就显示在了屏幕上。

## 8.4 getchar 的实现分析

异步异常-键盘中断的处理：当用户按键时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求，中断请求抢占当前进程运行键盘中断子程序，键盘中断子程序先从键盘接口取得该按键的扫描码，然后将该按键扫描码转换成 ASCII 码，保存到系统的键盘缓冲区之中。

getchar 函数落实到底层调用了系统函数 read，通过系统调用 read 读取存储在键盘缓冲区中的 ASCII 码直到读到回车符然后返回整个字串，getchar 进行封装，大体逻辑是读取字符串的第一个字符然后返回。

## 8.5 本章小结

本章主要介绍了 Linux 的 IO 设备管理方法、Unix IO 接口及其函数，分析了 printf 函数和 getchar 函数。（第 8 章 1 分）

## 结论

Hello 程序在运行过程中，主要经历了以下几个阶段：

预处理阶段：hello.c 引用的所有外部的库展开合并到一个 hello.i 的文本文件中
编译阶段：将 hello.i 文件编译成为汇编文件 hello.s
汇编阶段：将 hello.s 文件汇编成为可重定位目标文件 hello.o
链接阶段：将 hello.o 与可重定位目标文件和动态链接库链接成为可执行程序 hello
开始运行：在 shell 程序中输入 ./hello 1170300709 Kailai
Fork 阶段：shell 进程调用 fork 为其创建一个子进程
Execve 阶段：shell 程序调用 execve 函数，启动加载器，进入 hello 的 main 函数。
执行阶段：在一个时间段中，hello 独享 CPU 资源。
访存阶段：MMU 将程序中的虚拟内存地址通过页表映射成物理地址。
动态内存申请阶段：printf 调用 malloc 函数向动态内存分配器申请堆中的内存。
回收阶段：shell 父进程回收子进程，内核删除这个进程的所有数据。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

## 附件

列出所有的中间产物的文件名，并予以说明起作用。

hello.i hello.c 预处理的文本程序

hello.s hello.i 编译的汇编文本程序

hello.o hello.s 汇编的二进制文件

hello hello.o 链接的可执行二进制文件

hello.o.s hello 使用 `objdump` 反汇编后的汇编文本程序

hello.elf hello 的 elf 表

hello1.c 用于测试 `sleepsecs` 值得文本程序

hello1 hello1.c 生成得可执行二进制文件

(附件 0 分，缺失 -1 分)



## 参考文献

### 为完成本次大作业你翻阅的书籍与网站等

- [1] 林来兴. 空间控制技术[M]. 北京: 中国宇航出版社, 1992: 25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集: A 集[C]. 北京: 中国科学出版社, 1999.
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北: 天下文化出版社, 1998 [1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 湛颖. 空间交会控制理论与方法研究[D]. 哈尔滨: 哈尔滨工业大学, 1992: 8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359): 2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science, 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.

(参考文献 0 分, 缺失 -1 分)