

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 計算機科學與技術

學 號 1180301009

班 級 1803010

學 生 閻嘉良

指 導 教 師 史先俊

計算機科學與技術學院

2019 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：**P2P；O2O；进程；计算机系统

本文主要是通过合理运用这个学期在计算机系统课程上学习的知识，分析研究hello程序在Linux下的P2P和O2O过程，通过熟练使用各种工具，学习Linux框架下整个程序的声明周期，加深对课本知识的印象。

# 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>- 6 -</b>
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	- 6 -
<b>第 3 章 编译</b>	<b>- 8 -</b>
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	- 8 -
3.4 本章小结	- 12 -
<b>第 4 章 汇编</b>	<b>- 13 -</b>
4.1 汇编的概念与作用	- 13 -
4.2 在 UBUNTU 下汇编的命令	- 13 -
4.3 可重定位目标 ELF 格式	- 13 -
4.4 HELLO.O 的结果解析	- 15 -
4.5 本章小结	错误！未定义书签。
<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	- 20 -
5.5 链接的重定位过程分析	- 20 -
5.6 HELLO 的执行流程	- 21 -
5.7 HELLO 的动态链接分析	- 22 -
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.....	- 28 -
<b>第 7 章 HELLO 的存储管理.....</b>	<b>- 29 -</b>
7.1 HELLO 的存储器地址空间.....	- 29 -
7.2 INTEL 逻辑地址到线性地址的变换-段式管理.....	- 29 -
7.3 HELLO 的线性地址到物理地址的变换-页式管理.....	- 30 -
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换.....	- 30 -
7.5 三级 CACHE 支持下的物理内存访问.....	- 31 -
7.6 HELLO 进程 FORK 时的内存映射.....	- 32 -
7.7 HELLO 进程 EXECVE 时的内存映射.....	- 32 -
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.....	- 33 -
<b>第 8 章 HELLO 的 IO 管理.....</b>	<b>- 34 -</b>
8.1 LINUX 的 IO 设备管理方法.....	- 34 -
8.2 简述 UNIX IO 接口及其函数.....	- 34 -
8.3 PRINTF 的实现分析.....	- 34 -
8.4 GETCHAR 的实现分析.....	- 37 -
8.5 本章小结.....	- 37 -
<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

### P2P: Program to Process

在 linux 中, `hello.c` 经过 `cpp` 的预处理、`ccl` 的编译、`as` 的汇编、`ld` 的连接最终成为可执行目标程序 `hello`, 在 shell 中键入启动命令后, shell 为其 `fork`, 产生子进程, 产生子进程后 shell 为 `hello` `execve`, 于是 `hello` 便从 Program (个人理解为程序项目) 变为 Process (进程), 这便是 P2P 的过程。

### 2.O2O: Zero-0 to Zero-0

这里的 020 应该指的是 Process 在内存中 From Zero to Zero。产生子进程后 shell 为 `hello` `execve`, 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` 函数执行目标代码, CPU 为运行的 `hello` 分配时间片执行逻辑控制流。当程序运行结束后, shell 父进程负责回收 `hello` 进程, 内核删除相关数据结构, 这便是 020 的过程。

## 1.2 环境与工具

软件环境: Windows10、Ubuntu 19.04

硬件环境: Core i7-8770U

开发和调试工具: Codeblocks、Visual Studio 2017、gdb、gedit

## 1.3 中间结果

文件名	作用
<code>hello.c</code>	源程序
<code>hello.i</code>	预处理文件
<code>hello.s</code>	根据 <code>hello.i</code> 编译得到的.s 文件
<code>hello.o</code>	可重定位目标程序
<code>hello</code>	连接后的可执行文件

## 1.4 本章小结

本章对 `hello` 进行了简单的介绍, 分析了其 P2P 和 020 的过程, 列出了本次任务的环境和工具, 并且阐明了任务过程中出现的中间产物及其作用。

(第 1 章 0.5 分)

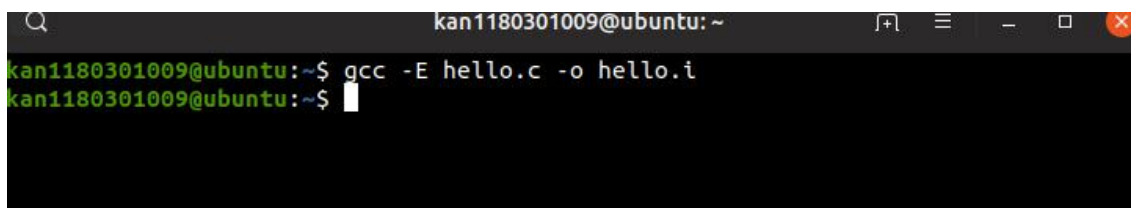
## 第 2 章 预处理

### 2.1 预处理的概念与作用

预处理器(cpp)根据以#开头的命令,修改原始的C程序。比如 hello.c 中第 6 行的#include<stdio.h>命令高速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。结果就得到了另一个C程序,通常是以.i 作为文件拓展名

### 2.2 在 Ubuntu 下预处理的命令

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



### 2.3 Hello 的预处理结果解析

修改得到的C程序 hello.i 已经从原来 hello.c 的 534 个字节增加到 63364 个字节,并且增加到 3110 行。再用 gedit 打开 hello.i,发现在 main 函数在文件的最后部分。

而在 main 函数之前,预处理器(cpp)读取头文件 stdio.h 、stdlib.h 、和 unistd.h 中的内容,三个系统头文件依次展开。比如 stdio.h 的展开,打开 usr/include/stdio.h 发现了其中还含有#开头的宏定义等,预处理器会对此继续递归展开,最终的.i 程序中没有#define,并且针对#开头的条件编译语句,cpp 根据#if 后面的条件决定需要编译的代码。

### 2.4 本章小结

本章介绍了 hello.c 的预处理阶段,根据预处理命令得到了修改后的 hello.i 程序,并且对 hello.i 程序进行了预处理结果解析,理解了预处理器读取系统头

文件中内容，并把它插入程序文本中的过程

**(第 2 章 0.5 分)**



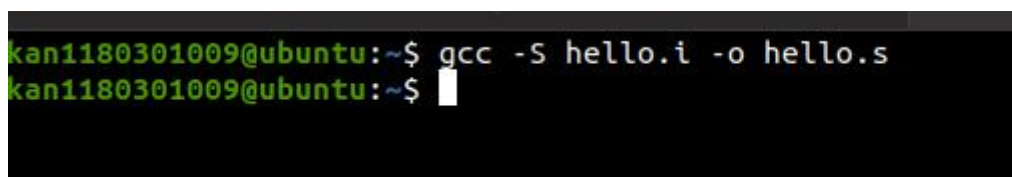
## 第 3 章 编译

### 3.1 编译的概念与作用

在这个阶段，编译器首先要检查代码的规范性，是否有语法错误等，以确定代码的实际要做的工作，再检查无误后，编译器（ccl）见文本文件 `hello.i` 翻译成文本文件 `hello.s`，它包含一个汇编语言程序。该程序包含函数 `main` 的定义，语句以一种文本格式描述了一条低级机器语言指令。汇编语言位不同高级语言的不同编译器提供了通用的输出语言。

### 3.2 在 Ubuntu 下编译的命令

```
gcc -S hello.i -o hello.s
```



```
kan1180301009@ubuntu:~$ gcc -S hello.i -o hello.s
kan1180301009@ubuntu:~$
```

### 3.3 Hello 的编译结果解析

`hello.s` 中的各种指令：

<code>.file</code>	源文件名
<code>.globl</code>	全局变量
<code>.data</code>	数据段
<code>.align</code>	对齐
<code>.type</code>	指定是对象类型或是函数类型
<code>.size</code>	大小
<code>.long</code>	长整型
<code>.section</code>	节头表
<code>.rodata</code>	只读数据段
<code>.string</code>	字符串

.text	代码段
-------	-----

## 1. 数据:

hello.s 中 C 语言的数据类型有: 局部变量, 指针数组, 无全局变量

int argc; argc 是函数传入的第一个 int 型参数, 存储在 %edi 中

int i; 局部变量, 通常保存在寄存器或是栈中。根据 movl \$0, -4(%rbp) 操作可知 i 的数据类型占用了 4 字节的栈空间。

argv[1]、argv[2] 和 argv[3] 都声明在 .rodata 只能读数据段中, 并且给出了字符串的编码

## 2. 赋值: i 在循环之前被赋值

```
0x000000000040117d <+11>: mov    %rsi,%rbp
0x0000000000401180 <+14>: mov    $0x0,%ebx
0x0000000000401185 <+19>: jmp    0x4011d4 <main+98>
```

在 main 函数第 41 行循环开始之前的第 14 行, i 被存储到 %ebx 中, 赋值为 0

## 3. 类型转换

这里涉及一处调用系统函数 atoi 函数将 argv[3] 转换成 int 型的操作

```
0x00000000004011b7 <+69>: mov    0x18(%rbp),%rdi
0x00000000004011bb <+73>: mov    $0xa,%edx
0x00000000004011c0 <+78>: mov    $0x0,%esi
0x00000000004011c5 <+83>: callq  0x401040 <strtol@plt>
--Type <RET> for more, q to quit, c to continue without paging--
```

## 4. 算数操作

这里有一处, 通过不断增加 i 的值, 并与 7 不断比较的例子

```
0x00000000004011cc <+90>: callq  0x401070 <sleep@plt>
0x00000000004011d1 <+95>: add     $0x1,%ebx
0x00000000004011d4 <+98>: cmp     $0x7,%ebx
0x00000000004011d7 <+101>: jle     0x40119b <main+41>
0x00000000004011d9 <+103>: mov     0x2e80(%rip),%rdi    # 0x4040
```

## 5. 逻辑操作/位操作

如果 cmp 按位比较算位操作/逻辑操作, 那么在 4 的图中已经有了, 并且已经加以说明了, 除此之外没有位操作

## 6. 关系操作

第一处: 看 argc 是否是 4 而进行的关系操作

```
0x0000000000401174 <+2>: sub     $0x8,%rsp
0x0000000000401178 <+6>: cmp     $0x4,%edi
0x000000000040117b <+9>: jne     0x401187 <main+21>
0x000000000040117d <+11>: mov     %rsi,%rbp
```

第二处: 在 for 循环中判断 i 是否到达临界条件而设计的

```

0x00000000004011cc <+90>:    callq  0x401070 <sleep@plt>
0x00000000004011d1 <+95>:    add     $0x1,%ebx
0x00000000004011d4 <+98>:    cmp     $0x7,%ebx
0x00000000004011d7 <+101>:   jle     0x40119b <main+41>
0x00000000004011d9 <+103>:   mov     0x2e80(%rip),%rdi    # 0x4040

```

## 7. 数组/指针/结构操作

这里有将参数导入 printf 函数和 atoi 函数的部分，是将 argv 字符串数组导入这两个函数的过程，如下图：

```

0x000000000040119b <+41>:    mov     0x10(%rbp),%rcx
0x000000000040119f <+45>:    mov     0x8(%rbp),%rdx
0x00000000004011a3 <+49>:    mov     $0x402030,%esi
0x00000000004011a8 <+54>:    mov     $0x1,%edi
0x00000000004011ad <+59>:    mov     $0x0,%eax
0x00000000004011b2 <+64>:    callq   0x401050 <__printf_chk@plt>
0x00000000004011b7 <+69>:    mov     0x18(%rbp),%rdi
0x00000000004011bb <+73>:    mov     $0xa,%edx
0x00000000004011c0 <+78>:    mov     $0x0,%esi
0x00000000004011c5 <+83>:    callq   0x401040 <strtol@plt>
-Type <RET> for more, q to quit, c to continue without paging--c
0x00000000004011ca <+88>:    mov     %eax,%edi
0x00000000004011cc <+90>:    callq   0x401070 <sleep@plt>

```

## 8. 控制转移

判断 argc 是否为 4 而进行的控制转移：jne

```

0x0000000000401174 <+2>:    sub     $0x8,%rsp
0x0000000000401178 <+6>:    cmp     $0x4,%edi
0x000000000040117b <+9>:    jne     0x401187 <main+21>
0x000000000040117d <+11>:   mov     %rsi,%rbp

```

判断 i 是否小于等于 7 而进行的控制转移：jle

```

0x00000000004011cc <+90>:    callq   0x401070 <sleep@plt>
0x00000000004011d1 <+95>:    add     $0x1,%ebx
0x00000000004011d4 <+98>:    cmp     $0x7,%ebx
0x00000000004011d7 <+101>:   jle     0x40119b <main+41>
0x00000000004011d9 <+103>:   mov     0x2e80(%rip),%rdi    # 0x4040

```

还有一处：jmp

这里是如果判断 argc 是 4，那么不必进行下面的部分代码，直接跳到 for 循环即可

```

0x0000000000401180 <+14>:    mov     $0x0,%ebx
0x0000000000401185 <+19>:    jmp     0x4011d4 <main+98>
0x0000000000401187 <+21>:    mov     $0x402008,%edi
0x000000000040118c <+26>:    callq   0x401030 <puts@plt>
0x0000000000401191 <+31>:    mov     $0x1,%edi

```

## 9. 函数操作

1.main 函数：

参数传递：传入参数 argc 和 argv，分别用寄存器 %rdi 和 %rsi 存储。



函数调用：被系统启动函数调用。

函数返回：设置%eax 为 0 并且返回，对应 return 0 。

```

0x0000000000401172 <+0>:    push    %rbp
0x0000000000401173 <+1>:    push    %rbx
0x0000000000401174 <+2>:    sub     $0x8,%rsp
0x0000000000401178 <+6>:    cmp     $0x4,%edi
0x000000000040117b <+9>:    jne     0x401187 <main+21>
0x000000000040117d <+11>:   mov     %rsi,%rbp
0x0000000000401180 <+14>:   mov     $0x0,%ebx
0x0000000000401185 <+19>:   jmp     0x4011d4 <main+98>
0x0000000000401187 <+21>:   mov     $0x402008,%edi
0x000000000040118c <+26>:   callq   0x401030 <puts@plt>
0x0000000000401191 <+31>:   mov     $0x1,%edi
0x0000000000401196 <+36>:   callq   0x401060 <exit@plt>
0x000000000040119b <+41>:   mov     0x10(%rbp),%rcx
0x000000000040119f <+45>:   mov     0x8(%rbp),%rdx
0x00000000004011a3 <+49>:   mov     $0x402030,%esi
0x00000000004011a8 <+54>:   mov     $0x1,%edi
0x00000000004011ad <+59>:   mov     $0x0,%eax
0x00000000004011b2 <+64>:   callq   0x401050 <__printf_chk@plt>
0x00000000004011b7 <+69>:   mov     0x18(%rbp),%rdi
0x00000000004011bb <+73>:   mov     $0xa,%edx
0x00000000004011c0 <+78>:   mov     $0x0,%esi
0x00000000004011c5 <+83>:   callq   0x401040 <strtol@plt>
--Type <RET> for more, q to quit, c to continue without paging--c
0x00000000004011ca <+88>:   mov     %eax,%edi
0x00000000004011cc <+90>:   callq   0x401070 <sleep@plt>
0x00000000004011d1 <+95>:   add     $0x1,%ebx
0x00000000004011d4 <+98>:   cmp     $0x7,%ebx
0x00000000004011d7 <+101>:  jle     0x40119b <main+41>
0x00000000004011d9 <+103>:  mov     0x2e80(%rip),%rdi      # 0x404060 <
stdin@@GLIBC_2.2.5>
0x00000000004011e0 <+110>:  callq   0x401080 <getc@plt>
0x00000000004011e5 <+115>:  mov     $0x0,%eax
0x00000000004011ea <+120>:  add     $0x8,%rsp
0x00000000004011ee <+124>:  pop     %rbx
0x00000000004011ef <+125>:  pop     %rbp
0x00000000004011f0 <+126>:  retq

```

2.printf 函数：

参数传递：call puts 时只传入了字符串参数首地址；for 循环中 call printf 时传入了 argv[1]和 argc[2]的地址。

函数调用：for 循环中被调用

```

0x00000000004011ad <+59>:    mov     $0x0,%eax
0x00000000004011b2 <+64>:    callq   0x401050 <__printf_chk@plt>
0x00000000004011b7 <+69>:    mov     0x18(%rbp),%rdi

```

3.exit 函数：

参数传递：传入的参数为 1，再执行退出命令

函数调用：if 判断条件满足后被调用

```

0x0000000000401191 <+31>:  mov    $0x1,%edi
0x0000000000401196 <+36>:  callq  0x401060 <exit@plt>
0x000000000040119b <+41>:  mov     0x10(%rbp),%rcx

```

#### 4.sleep 函数:

参数传递: 传入参数 atoi 函数返回值, 传递控制 call sleep

函数调用: for 循环下被调用

```

0x00000000004011c8 <+88>:  mov     %eax,%edi
0x00000000004011cc <+90>:  callq  0x401070 <sleep@plt>
0x00000000004011d1 <+95>:  add     $0x1,%ebx

```

#### 5.getchar

传递控制: call getchar

函数调用: 在 main 中被调用

```

0x00000000004011d9 <+103>:  mov     0x2e80(%rip),%rdi
0x00000000004011e0 <+110>:  callq  0x401080 <getc@plt>
0x00000000004011e5 <+115>:  mov     $0x0,%eax

```

#### 6. atoi 函数:

参数传递: argv[3]

函数调用: 在 main 函数中被调用

```

0x00000000004011c0 <+78>:  mov     $0x0,%esi
0x00000000004011c5 <+83>:  callq  0x401040 <strtol@plt>
Type <RET> for more, q to quit, c to continue without paging--c

```

### 3.4 本章小结

本章我们主要介绍了编译器是如何将文本编译成汇编代码的。可以发现, 编译器并不是死板的按照我们原来文本的顺序, 逐条语句进行翻译下来的。编译器在编译的过程中, 不近会对我们的代码做一些隐式的优化, 而且会将原来代码中用到的跳转, 循环等操作操作用控制转移等方法进行解析。最后生成我们需要的 hello.s 文件。

(第3章2分)

## 第 4 章 汇编

### 4.1 汇编的概念与作用

概念：

汇编器（as）将 hello.s 文件翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保存在 hello.o 中。这里的 hello.o 是一个二进制文件。

作用：

我们知道，汇编代码也只是我们人可以看懂的代码，而机器并不能读懂，真正机器可以读懂并执行的是机器代码，也就是二进制代码。汇编的作用就是将我们之前再 hello.s 中保存的汇编代码翻译成可以攻机器执行的二进制代码，这样机器就可以根据这些 01 代码，真正的开始执行我们写的程序了。

### 4.2 在 Ubuntu 下汇编的命令

```
gcc -c hello.s -o hello.o
```

```
kan1180301009@ubuntu:~$ gcc -c hello.s -o hello.o
kan1180301009@ubuntu:~$
```

### 4.3 可重定位目标 elf 格式

1. ELF 头：

ELF 头以一个 16 字节的序列开始，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，节头部表中条目的大小和数量等。

```
kan1180301009@ubuntu:~$ readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 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
kan1180301009@ubuntu:~$
```



## 2. 节头表

记录了每个节的名称、类型、属性（读写权限）、在 ELF 文件中所占的长度、对齐方式和偏移量

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[ 0]	0000000000000000	NULL	0000000000000000	00000000
[ 1]	.text	PROGBITS	0000000000000000	00000040
[ 2]	.rela.text	RELA	0000000000000000	00000340
[ 3]	.data	PROGBITS	0000000000000000	000000ce
[ 4]	.bss	NOBITS	0000000000000000	000000ce
[ 5]	.rodata	PROGBITS	0000000000000000	000000d0
[ 6]	.comment	PROGBITS	0000000000000000	00000102
[ 7]	.note.GNU-stack	PROGBITS	0000000000000000	00000126
[ 8]	.eh_frame	PROGBITS	0000000000000000	00000128
[ 9]	.rela.eh_frame	RELA	0000000000000000	00000400
[10]	.symtab	SYMTAB	0000000000000000	00000160
[11]	.strtab	STRTAB	0000000000000000	000002f8
[12]	.shstrtab	STRTAB	0000000000000000	00000418

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),

## 3. 重定位节（如下图）

重定位条目告诉链接器在将目标文件合并成可执行文件时如何修改这个引用。如图，偏移量是需要被修改的引用的节偏移，符号标识被修改引用应该指向的符号。类型告知链接器如何修改新的引用，加数是一个有符号常数，一些类型的重定位要用它对被修改引用的值做偏移调整。

**r\_offset:**

此成员指定应用重定位操作的位置。不同的目标文件对于此成员的解释会稍有不同。

**r\_info:**

此成员指定必须对其进行重定位的符号表索引以及要应用的重定位类型。

重定位类型特定于处理器。重定位项的重定位类型或符号表索引是将 ELF32\_R\_TYPE 或 ELF32\_R\_SYM 分别应用于项的 r\_info 成员所得的结果。

**r\_addend:**

此成员指定常量加数，用于计算将存储在可重定位字段中的值。

**重定位类型:**

**R\_X86\_64\_PC32:** 重定位一个使用 32 位 PC 相对地址的引用。在指令中编码的 32 位值加上 PC 的当前运行时值，得到有效地址。

R\_X86\_64\_32: 重定位一个使用 32 位 PC 绝对地址的引用。直接使用在指令中编码的 32 位值作为有效地址。

```
kan1180301009@ubuntu:~$ readelf -r hello.o

Relocation section '.rela.text' at offset 0x340 contains 8 entries:
   Offset             Info             Type             Sym. Value          Sym. Name + Addend
00000000000018      0005000000002 R_X86_64_PC32      0000000000000000   .rodata - 4
0000000000001d      000b000000004 R_X86_64_PLT32      0000000000000000   puts - 4
00000000000027      000c000000004 R_X86_64_PLT32      0000000000000000   exit - 4
00000000000050      0005000000002 R_X86_64_PC32      0000000000000000   .rodata + 21
0000000000005a      000d000000004 R_X86_64_PLT32      0000000000000000   printf - 4
0000000000006d      000e000000004 R_X86_64_PLT32      0000000000000000   atoi - 4
00000000000074      000f000000004 R_X86_64_PLT32      0000000000000000   sleep - 4
00000000000083      0010000000004 R_X86_64_PLT32      0000000000000000   getchar - 4

Relocation section '.rela.eh_frame' at offset 0x400 contains 1 entry:
   Offset             Info             Type             Sym. Value          Sym. Name + Addend
00000000000020      0002000000002 R_X86_64_PC32      0000000000000000   .text + 0
kan1180301009@ubuntu:~$
```

#### 4. 符号表

它存放在程序中定义和引用的函数和全局变量的信息,.symtab 符号表不包含局部变量的条目。

```
kan1180301009@ubuntu:~$ readelf -s hello.o

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        142 FUNC     GLOBAL   DEFAULT    1 main
  10: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND _GLOBAL_OFFSET_TABLE_
  11: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND puts
  12: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND exit
  13: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND printf
  14: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND atoi
  15: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND sleep
  16: 0000000000000000          0 NOTYPE   GLOBAL   DEFAULT   UND getchar
kan1180301009@ubuntu:~$
```

### 4.4 Hello.o 的结果解析

Objdump 后的汇编代码:



```

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 04       cmpl    $0x4,-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 48            jmp     7c <main+0x7c>
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+0x21
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: 48 8b 45 e0       mov     -0x20(%rbp),%rax
62: 48 83 c0 18       add     $0x18,%rax
66: 48 8b 00          mov     (%rax),%rax
69: 48 89 c7          mov     %rax,%rdi
6c: e8 00 00 00 00    callq   71 <main+0x71>
                        71: R_X86_64_PLT32      atoi-0x4
71: 89 c7            mov     %eax,%edi
73: e8 00 00 00 00    callq   78 <main+0x78>
                        74: R_X86_64_PLT32      sleep-0x4
78: 83 45 fc 01       addl    $0x1,-0x4(%rbp)
7c: 83 7d fc 07       cmpl    $0x7,-0x4(%rbp)
80: 7e b2            jle     34 <main+0x34>
82: e8 00 00 00 00    callq   87 <main+0x87>
                        83: R_X86_64_PLT32      getchar-0x4
87: b8 00 00 00 00    mov     $0x0,%eax
8c: c9              leaveq  %rax
8d: c3              retq

```

通过与原文件对比后发现：

- 1) 过程调用的符号，被替换为相对寻址。
- 2) 条件转移中的.L 标记被准确的相对 main 的开始的偏移所代替。
- 3) 全局变量的符号，被替换为相对寻址。

值得注意的是，由于还未进行链接，上述的相对寻址全都暂时被 0x0(%rip)代替。

此外，可以发现，`.text` 中的各个汇编指令的作用效果没有太大变化。

**(第 4 章 1 分)**

## 第 5 章 链接

### 5.1 链接的概念与作用

定义：

链接是通过链接器（ld）将各种代码和数据片断收集并组合成一个单一文件的过程。这个文件可以被加载（复制）到内存并执行。

作用：

因为有了链接这个概念的存在，所以我们的代码才回变得比较方便和简洁，同时可移植性强，模块化程度比较高。因为链接的过程可以使我们将程序封装成很多的模块，我们在变成的过程中只用写主程序的部分，对于其他的部分我们有些可以直接调用模块，就像 C 中的 printf 一样。

作为编译的多后一步链接，就是处理当前程序调用其他模块的操作，将该调用的模块中的代码组合到相应的可执行文件中。

### 5.2 在 Ubuntu 下链接的命令

```
ld -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  
/usr/lib/x86_64-linux-gnu/crtn.o hello.o -lc -z relro -o hello
```

```
kan1180301009@ubuntu:~$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-li  
nux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o hello.o  
-lc -z relro -o hello  
kan1180301009@ubuntu:~$
```

### 5.3 可执行目标文件 hello 的格式

```
kan1180301009@ubuntu:~$ readelf -S hello
There are 25 section headers, starting at offset 0x3728:

Section Headers:
[Nr] Name           Type              Address            Offset
     Size           EntSize          Flags Link Info  Align
[ 0]                  NULL              0000000000000000  00000000
     0000000000000000 0000000000000000 0 0 0
[ 1] .interp            PROGBITS          0000000000400270  00000270
     000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag      NOTE              000000000040028c  0000028c
     0000000000000020 0000000000000000 A 0 0 4
[ 3] .hash              HASH              00000000004002b0  000002b0
     0000000000000038 0000000000000004 A 5 0 8
[ 4] .gnu.hash           GNU_HASH           00000000004002e8  000002e8
     000000000000001c 0000000000000000 A 5 0 8
[ 5] .dynsym             DYNSYM            0000000000400308  00000308
     00000000000000d8 0000000000000018 A 6 1 8
[ 6] .dynstr             STRTAB            00000000004003e0  000003e0
     000000000000005c 0000000000000000 A 0 0 1
[ 7] .gnu.version         VERSYM            000000000040043c  0000043c
     0000000000000012 0000000000000002 A 5 0 2
[ 8] .gnu.version_r       VERNEED           0000000000400450  00000450
     0000000000000020 0000000000000000 A 6 1 8
[ 9] .rela.dyn            RELA              0000000000400470  00000470
     0000000000000030 0000000000000018 A 5 0 8
[10] .rela.plt            RELA              00000000004004a0  000004a0
     0000000000000090 0000000000000018 AI 5 19 8
[11] .init                PROGBITS          0000000000401000  00001000
     0000000000000017 0000000000000000 AX 0 0 4
[12] .plt                 PROGBITS          0000000000401020  00001020
     0000000000000070 0000000000000010 AX 0 0 16
[13] .text                PROGBITS          0000000000401090  00001090
     0000000000000121 0000000000000000 AX 0 0 16
[14] .fini                PROGBITS          00000000004011b4  000011b4
     0000000000000009 0000000000000000 AX 0 0 4
[15] .rodata              PROGBITS          0000000000402000  00002000
     000000000000003a 0000000000000000 A 0 0 8
[16] .eh_frame            PROGBITS          0000000000402040  00002040
     00000000000000fc 0000000000000000 A 0 0 8
[17] .dynamic              DYNAMIC           0000000000403e50  00002e50
     00000000000001a0 0000000000000010 WA 6 0 8
[18] .got                  PROGBITS          0000000000403ff0  00002ff0
     0000000000000010 0000000000000008 WA 0 0 8
[19] .got.plt              PROGBITS          0000000000404000  00003000
     0000000000000048 0000000000000008 WA 0 0 8
[20] .data                 PROGBITS          0000000000404048  00003048
     0000000000000004 0000000000000000 WA 0 0 1
[21] .comment              PROGBITS          0000000000000000  0000304c
     0000000000000023 0000000000000001 MS 0 0 1
[22] .symtab               SYMTAB            0000000000000000  00003070
     0000000000000498 0000000000000018 23 28 8
[23] .strtab               STRTAB            0000000000000000  00003508
     0000000000000158 0000000000000000 0 0 1
[24] .shstrtab             STRTAB            0000000000000000  00003660
     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)
```

图中保存了可执行文件 hello 中的各个节的信息。可以看到 hello 文件中的节的数目比 hello.o 中多了很多，说明在链接过后有很多文件有添加进来。下面列出每一节中各个信息条目的含义：



名称和大小这个条目中存储了每一个节的名称和这个节在重定位文件中所占的大小。

地址这个条目中，保存了各个节在重定位文件中的具体位置也就是地址。偏移量这一栏中保存的是这个节在程序里面的地址的偏移量，也就是相对地址。

## 5.4 hello 的虚拟地址空间

1.连接后的 hello 属于 elf 可执行目标文件，所包含的各类信息如下：

```
kan1180301009@ubuntu:~$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 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:                    0x401090
  Start of program headers:               64 (bytes into file)
  Start of section headers:              14120 (bytes into file)
  Flags:                                  0x0
  Size of this header:                     64 (bytes)
  Size of program headers:                 56 (bytes)
  Number of program headers:               10
  Size of section headers:                 64 (bytes)
  Number of section headers:               25
  Section header string table index:      24
kan1180301009@ubuntu:~$
```

2.虚拟地址空间各段信息：

- (1) .PDHR:起始位置为 0x400040,大小为 0x1c0
- (2) .INTERP:起始位置为 0x400200,大小为 0x1c
- (3) .LOAD: 起始位置为 0x400000, 大小为 0x81c
- (4) .LOAD: 起始位置为 0x600e00,大小为 0x258

## 5.5 链接的重定位过程分析

1.hello 中增加了许多节和被调用的函数。

2.对 rodata 的引用：在 hello.o 的反汇编文件中对 printf 参数字符串的引用使用全 0 替代。在 hello 中则使用确定地址，这是因为链接后全局变量的地址能够确定。

3.hello.o 中 main 的地址从 0 开始，hello 中 main 的地址不再是 0.库函数的代码都链接到了程序中。

Hello.o 的反汇编：

```

1: 83 7d ec 04      cmpl    $0x4,-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 48            jmp     7c <main+0x7c>
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+0x21
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: 48 8b 45 e0      mov     -0x20(%rbp),%rax
62: 48 83 c0 18      add     $0x18,%rax
66: 48 8b 00         mov     (%rax),%rax
69: 48 89 c7         mov     %rax,%rdi

```

Hello 的反汇编:

```

4010f3: eb 48            jmp     40113d <main+0x7c>
4010f5: 48 8b 45 e0      mov     -0x20(%rbp),%rax
4010f9: 48 83 c0 10      add     $0x10,%rax
4010fd: 48 8b 10         mov     (%rax),%rdx
401100: 48 8b 45 e0      mov     -0x20(%rbp),%rax
401104: 48 83 c0 08      add     $0x8,%rax
401108: 48 8b 00         mov     (%rax),%rax
40110b: 48 89 c6         mov     %rax,%rsi
40110e: 48 8d 3d 18 0f 00 00 lea     0xf18(%rip),%rdi      # 40202d <IO_stdin>
401115: b8 00 00 00 00    mov     $0x0,%eax
40111a: e8 21 ff ff ff    callq  401040 <printf@plt>
40111f: 48 8b 45 e0      mov     -0x20(%rbp),%rax
401123: 48 83 c0 18      add     $0x18,%rax
401127: 48 8b 00         mov     (%rax),%rax
40112a: 48 89 c7         mov     %rax,%rdi
40112d: e8 2e ff ff ff    callq  401060 <atoi@plt>
401132: 89 c7            mov     %eax,%edi
401134: e8 47 ff ff ff    callq  401080 <sleep@plt>
401139: 83 45 fc 01      addl    $0x1,-0x4(%rbp)
40113d: 83 7d fc 07      cmpl    $0x7,-0x4(%rbp)
401141: 7e b2            jle     4010f5 <main+0x34>
401143: e8 88 ff ff ff    callq  401050 <printf@plt>

```

Hello.o 中空白的函数处全部都在可执行文件 hello 中有了填补

## 5.6 hello 的执行流程

(以下格式自行编排, 编辑时删除)

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

程序名称	程序地址
ld-2.27.so!_dl_start	0x7fe7a8177030
ld-2.27.so!_dl_init	0x7fe7a81859e0
hello!_start	0x400500
libc-2.27.so!_libc_start_main	0x7fe7a7f98a80
-libc-2.27.so!_cxa_atexit	0x7fe7a7f9ac43
-libc-2.27.so!_libc_csu_init	0x4005c0
hello!_init	0x400488

libc-2.27.so!_setjmp	0x7fe7a7884c10
-libc-2.27.so!_sigsetjmp	0x7fe7a7884b70
--libc-2.27.so!_sigjmp_save	0x7fe7a7884bd0
hello!main	0x400532
hello!puts@plt	0x4004b0
hello!puts@plt	0x4004e0
*hello!printf@plt	—
*hello!sleep@plt	—
*hello!getchar@plt	—
ld-2.27.so!_dl_runtime_resolve_xsave	0x7fe7a784e680
-ld-2.27.so!_dl_fixup	0x7fe7a7846df0
--ld-2.27.so!_dl_lookup_symbol_x	0x7fe7a78420b0
libc-2.27.so!exit	0x7fe7a7889128

## 5.7 Hello 的动态链接分析

程序调用一个由共享库定义的函数时，编译器没有办法预测这个函数的运行时地址，因为定义它的共享模块在运行时可以加载到任意位置。GNU 编译系统使用延迟绑定的技术解决这个问题，将过程地址的延迟绑定推迟到第一次调用该过程时。

延迟绑定要用到全局偏移量表（GOT）和过程链接表（PLT）两个数据结构。如果一个目标模块调用定义在共享库中的任何函数，那么它就有自己的 GOT 和 PLT。

**PLT:** PLT 是一个数组，其中每个条目是 16 字节代码。PLT[0]是一个特殊条目，跳转到动态链接器中。每个条目都负责调用一个具体的函数。PLT[[1]]调用系统启动函数（`__libc_start_main`）。从 PLT[[2]]开始的条目调用用户代码调用的函数。

**GOT:** GOT 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会使用的信息。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，其地址需要在运行时被解析。

## 5.8 本章小结

链接的过程，是将原来的只保存了你写的函数的代码与代码用所用的库函数合并的一个过程。在这个过程中链接器会为每个符号、函数等信息重新分配虚拟内存地址，方法就是用每个.o 文件中的重定位节与其它的节想配合，算出正确的地址。同时，将你会用到的库函数加载（复制）到可执行文件中。这些信息一同构成了一个完整的计算机可以运行的文件。链接让我们的程序做到了很好的模块化，我们只需要写我们的主要代码，对于读入、IO 等操作，可以直接与封装的模块相链接，这样大大的简化了代码的书写难度。

（第 5 章 1 分）





## 第 6 章 hello 进程管理

### 6.1 进程的概念与作用

进程是一个执行中程序的实例。是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

作用：进程的概念为我们提供这样一种假象，就好像我们的程序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地使用处理器和内存，处理器好像是无间断地一条接一条地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 俗称壳，是指“为使用者提供操作界面”的软件（命令解析器）。它接收用户命令，然后调用相应的应用程序。

1.功能：命令解释。Linux 系统中的所有可执行文件都可以作为 Shell 命令来执行。

2.处理流程：

1) 当用户提交了一个命令后，Shell 首先判断它是否为内置命令，如果是就通过 Shell 内部的解释器将其解释为系统功能调用并转交给内核执行。

2) 若是外部命令或应用程序就试图在硬盘中查找该命令并将其调入内存，再将其解释为系统功能调用并转交给内核执行。

### 6.3 Hello 的 fork 进程创建过程

在终端中输入./hello 学号 姓名，shell 判断它不是内置命令，于是会加载并运行当前目录下的可执行文件 hello.此时 shell 通过 fork 创建一个新的子进程。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，包括代码和数据段、堆、共享库和用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，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。子进程与父进程有不同的 pid。fork 被调用一次，返回两次。在父进程中 fork 返回子进程的 pid，在子进程中 fork 返回 0.父进程与子进程是并发运行的独立进程。

### 6.4 Hello 的 execve 过程

execve 函数在新创建的子进程的上下文中加载并运行 hello 程序。execve 函数加载并运行可执行目标文件 filename，且带参数列表 argv 和环境变量列表 envp。

只有发生错误时 `execve` 才会返回到调用程序。所以，`execve` 调用一次且从不返回。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已存在的区域结构。

2. 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。栈和堆区域也是请求二进制零的，初始长度为零。

3. 映射共享区域。如果 `hello` 程序与共享对象链接，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。设置当前进程上下文中的程序计数器，使之指向代码区域的入口点。下一次调度这个进程时，它将从这个入口点开始执行。

## 6.5 Hello 的进程执行

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。

一个进程执行它的控制流的一部分的每一时间段叫做时间片。

处理器通常用某个控制寄存器的一个模式位来提供用户模式和内核模式的功能。设置了模式位时，进程就运行在内核模式中，该进程可以执行指令集中的任何指令，可以访问系统中的任何内存位置。没有设置模式位时，进程就运行在用户模式中，用户模式中的进程不允许执行特权指令。

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了进程的进程的决定叫做调度。

程序在执行 `sleep` 函数时，`sleep` 函数首先会调用 `atoi` 函数将 `argv[3]` 中的字符或字符串转换为整型数字，此时 `sleep` 函数调用 `wait` 或 `waitpid` 函数挂起直到 `atoi` 函数执行完成并返回一个数字，此时 `sleep` 函数作为父进程回收执行 `atoi` 函数的子进程。

此时，在 `sleep` 函数获取参数后，`sleep` 系统调用显式地请求让调用进程休眠，调度器抢占当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程。`sleep` 的倒计时结束后，控制会回到 `hello` 进程中。程序调用 `getchar()` 时，内核可以执行上下文切换，将控制转移到其他进程。`getchar()` 的数据传输结束之后，引发一个中断信号，控制回到 `hello` 进程中

## 6.6 hello 的异常与信号处理

hello 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

程序运行过程中可以按键盘，如不停乱按，包括回车，Ctrl-Z，Ctrl-C 等，Ctrl-z 后可以运行 ps jobs pstree fg kill 等命令，请分别给出各命令及运行截图屏，说明异常与信号的处理。

1. 正常退出：

```
kan1180301009@ubuntu:~$ ./hello 1180301009 阚嘉良 2
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
o
kan1180301009@ubuntu:~$ ps
  PID TTY          TIME CMD
  3470 pts/1        00:00:00 bash
  3978 pts/1        00:00:00 ps
kan1180301009@ubuntu:~$
```

2. Ctrl+z 中断进程后运行 fg，使后台进程继续运行，信号：SIGCONTIN (fg)

```
kan1180301009@ubuntu:~$ ./hello 1180301009 阚嘉良 2
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
^Z
[1]+  Stopped                  ./hello 1180301009 阚嘉良 2
kan1180301009@ubuntu:~$ ps
  PID TTY          TIME CMD
  2228 pts/0        00:00:00 bash
  4060 pts/0        00:00:00 hello
  4106 pts/0        00:00:00 ps
kan1180301009@ubuntu:~$ fg
./hello 1180301009 阚嘉良 2
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
C
kan1180301009@ubuntu:~$
```

3. Ctrl+z 后运行 jobs

```
kan1180301009@ubuntu:~$ ./hello 1180301009 阚嘉良 1
Hello 1180301009 阚嘉良
al lo 1180301009 阚嘉良
ello 1180301009 阚嘉良
^Z
[1]+  Stopped                  ./hello 1180301009 阚嘉良 1
kan1180301009@ubuntu:~$ jobs
[1]+  Stopped                  ./hello 1180301009 阚嘉良 1
kan1180301009@ubuntu:~$
```

4. Ctrl+z 后运行 pstree(这里我找到的是 hello 所在处,上一级进程是 systemd)

```
gnome-terminal- bash- hello
                  |
                  | pstree
                  |
                  bash
                  |
                  4*[{gnome-terminal-}]
gnome-terminal- 3*[{gnome-terminal-}]
```

5. Ctrl+z 后运行 kill 命令

```
kan1180301009@ubuntu:~$ ps
  PID TTY          TIME CMD
 2228 pts/0        00:00:01 bash
 4113 pts/0        00:00:00 hello
 4119 pts/0        00:00:00 ps
kan1180301009@ubuntu:~$ kill -9 4113
kan1180301009@ubuntu:~$ ps
  PID TTY          TIME CMD
 2228 pts/0        00:00:01 bash
 4120 pts/0        00:00:00 ps
[1]+  Killed                  ./hello 1180301009 阚嘉良 1
kan1180301009@ubuntu:~$
```

如图,平时显示进程已经成功被杀死

6. 不停乱按:

```
kan1180301009@ubuntu:~$ ./hello 1180301009 阚嘉良 1
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
lskjflklHello 1180301009 阚嘉良
lsfioeHello 1180301009 阚嘉良
jsfifeHello 1180301009 阚嘉良
ljsfisflHello 1180301009 阚嘉良
sljfleifHello 1180301009 阚嘉良
skjffjHello 1180301009 阚嘉良
lskjfk
kan1180301009@ubuntu:~$ ./hello 1180301009 阚嘉良 1
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良

Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良

sfHello 1180301009 阚嘉良
e
fs
Hello 1180301009 阚嘉良
f
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
kan1180301009@ubuntu:~$
```

7. Ctrl+c 结束进程：终止  
信号：SIGINT

```
kan1180301009@ubuntu:~$ ./hello 1180301009 阚嘉良 1
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
Hello 1180301009 阚嘉良
^C
kan1180301009@ubuntu:~$ ps
  PID TTY          TIME CMD
 2228 pts/0        00:00:01 bash
 4180 pts/0        00:00:00 ps
kan1180301009@ubuntu:~$
```

## 6.7 本章小结

本章中阐述了进程的概念以及他在计算机中具体是如何在使用的。其次，还介绍了如何利用 shell 这个平台来对进程进行监理调用或发送信号等一系列操作。

(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：在有地址变换功能的计算机中，访内指令给出的地址（操作数）叫逻辑地址，也叫相对地址。要经过寻址方式的计算或变换才得到内存存储器中的实际有效地址，即物理地址。是 `hello.o` 中的相对偏移地址。

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

虚拟地址：程序访问存储器所使用的逻辑地址称为虚拟地址。是 `hello` 里的虚拟内存地址。

物理地址：在存储器里以字节为单位存储信息，为正确地存放或取得信息，每一个字节单元给以一个唯一的存储器地址，称为物理地址。是 `hello` 里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

#### 1. 基本原理：

在段式存储管理中，将程序的地址空间划分为若干个段(segment)，这样每个进程有一个二维的地址空间。在段式存储管理系统中，为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。

在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法等方法。

在回收某个段所占用的空间时，要注意将收回的空间与其相邻的空间合并。

段式存储管理也需要硬件支持，实现逻辑地址到物理地址的映射。

程序通过分段划分为多个模块，如代码段、数据段、共享段：

- 可以分别编写和编译
- 可以针对不同类型的段采取不同的保护
- 可以按段为单位来进行共享，包括通过动态链接进行代码共享

这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。

总的来说，段式存储管理的优点是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。缺点与页式存储管理的缺点相同，进程必须全部装入内存。

#### 2. 段式管理的数据结构：

为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物



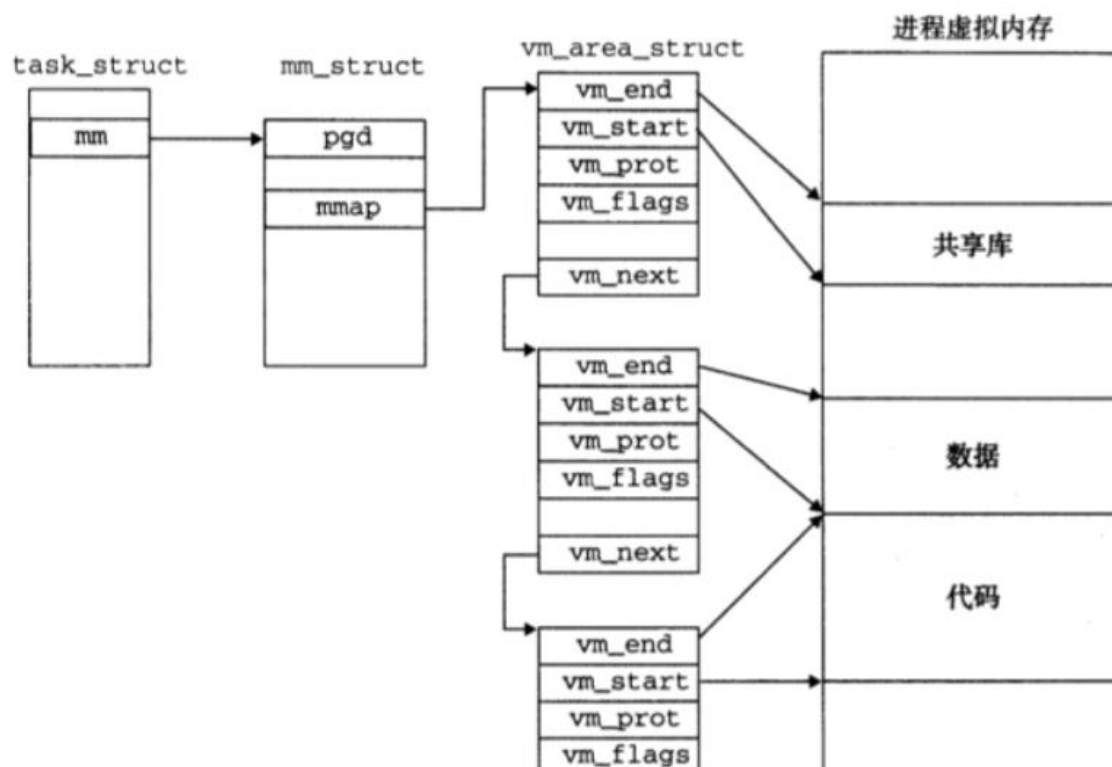
理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

- 进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(baseaddress)，即段内地址。

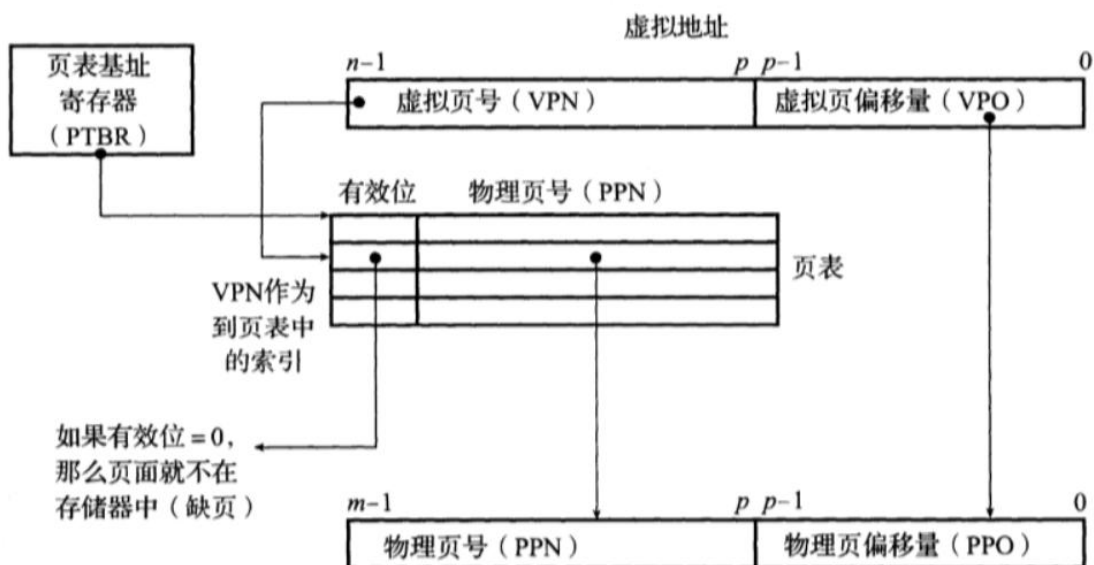
### 7.3 Hello 的线性地址到物理地址的变换-页式管理

线性地址到物理地址的转换是通过页的这个概念完成的。线性地址被分为以固定长度为单位的组，称为页。

首先 Linux 系统有自己的虚拟内存系统，其虚拟内存组织形式如图 7-4 所示，Linux 将虚拟内存组织成一些段的集合，段之外的虚拟内存不存在因此不需要记录。内核为 hello 进程维护一个段的任务结构即图中的 task\_struct，其中条目 mm 指向一个 mm\_struct，它描述了虚拟内存的当前状态，pgd 指向第一级页表的基地址（结合一个进程一串页表），mmap 指向一个 vm\_area\_struct 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。

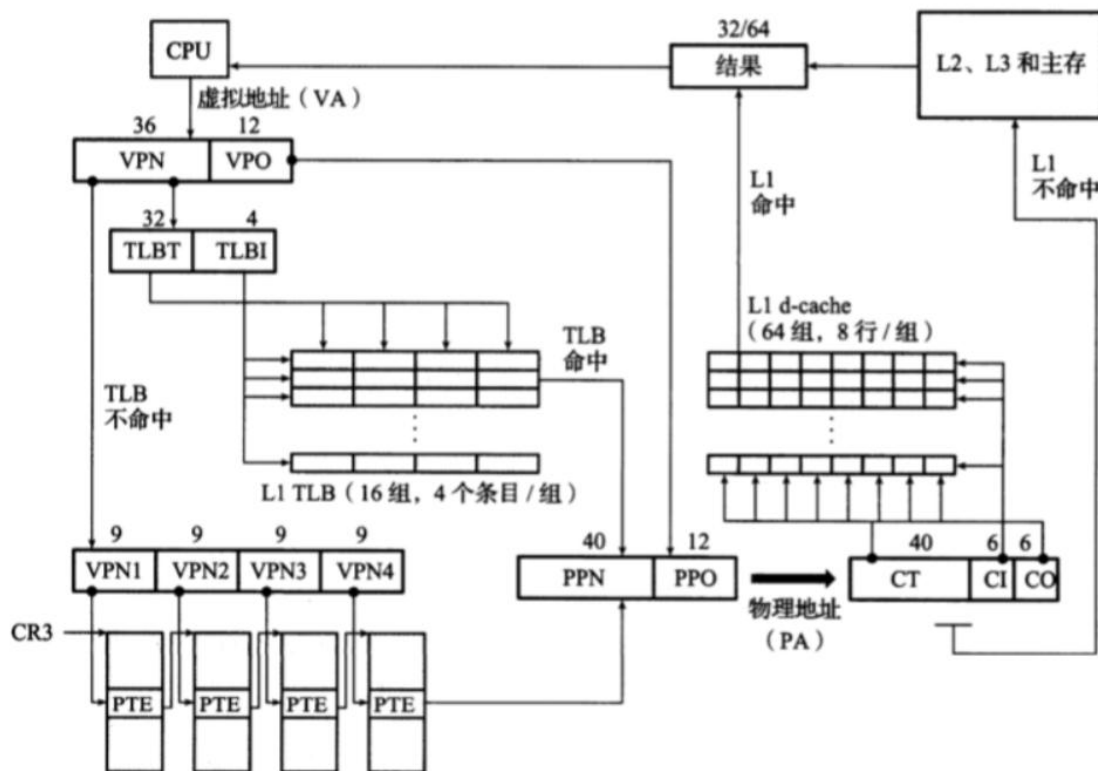


CPU 芯片上有一个专门的硬件叫做内存管理单元 (MMU)，这个硬件的功能就是动态的将虚拟地址翻译成物理地址的。这个表示如何工作的呢，如图 7-5 所示。N 为的虚拟地址包含两个部分，一个 p 位的虚拟页面偏移 (VPO) 和一个 (n-p) 位的虚拟页号 (VPN)。MMU 利用 VPN 来选择适当的 PTE (页表条目)。接下来在对应的 PTE 中获得 PPN (物理页号)，将 PPN 与 VPO 串联起来，就得到了相应的物理地址



## 7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

36 位的虚拟地址被分割成 4 个 9 位的片。CR3 寄存器包含 L1 页表的物理地址。VPN1 有一个到 L1 PTE 的偏移量，找到这个 PTE 以后又会包含到 L2 页表的基础地址；VPN2 包含一个到 L2PTE 的偏移量，找到这个 PTE 以后又会包含到 L3 页表的基础地址；VPN3 包含一个到 L3PTE 的偏移量，找到这个 PTE 以后又会包含到 L4 页表的基础地址；VPN4 包含一个到 L4PTE 的偏移量，找到这个 PTE 以后就是相应的 PPN（物理页号）。





## 7.5 三级 Cache 支持下的物理内存访问

在上面，我们已经获得了物理地址 VA，我们接着图 7-6 的右侧部分进行说明。使用 CI（后六位再后六位）进行组索引，每组 8 路，对 8 路的块分别匹配 CT（前 40 位）如果匹配成功且块的 valid 标志位为 1，则命中（hit），根据数据偏移量 CO（后六位）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（miss），向下一级缓存中查询数据（L2 Cache->L3 Cache->主存）。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（evict），则采用最近最少使用策略 LFU 进行替换。

## 7.6 hello 进程 fork 时的内存映射

在 7.3 节中我们已经提到过了 mm\_struct 和 vm\_area\_struct 这两个标记符，这里我们就需要用到他们。先来介绍一下：

mm\_struct（内存描述符）：描述了一个进程的整个虚拟内存空间。

vm\_area\_struct（区域结构描述符）：描述了进程的虚拟内存空间的一个区间。

在用 fork 创建内存的时候，我们需要以下三个步骤：

- 1：创建当前进程的 mm\_struct，vm\_area\_struct 和页表的原样副本。
- 2：两个进程的每个页面都标记为只读页面。
- 3：两个进程的每个 vm\_area\_struct 都标记为私有，这样就只能在写入时复制。

## 7.7 hello 进程 execve 时的内存映射

execve 函数在 shell 中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效的替代了当前程序。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 shell 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。为 hello 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。栈和堆区域也是请求二进制零的，初始长度为零。图 7.7 概括了私有区域的不同映射。
- 3：映射共享区域。如果 hello 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(PC)。execve 做的最后一件事情就是设置当前进程上下文中的程序计数器，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

在虚拟内存的习惯说法中，DRAM 缓存不命中称为缺页。例如：CPU 引用了 VP3 中的一个字，VP3 并未缓存在 DRAM 中。地址翻译硬件从内存中读取 PTE3，从

有效位推断出 VP3 未被缓存，并且触发一个缺页异常。缺页异常调用内核中的缺页异常处理程序，该程序会选择一个牺牲页，在此例中就是存放在 PP3 中的 VP4。如果 VP4 已经被修改了，那么内核就会将它复制回磁盘。无论哪种情况，内核都会修改 VP4 的页表条目，反映出 VP4 不再缓存在主存中这一事实

## 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的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。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。

### 1. 隐式空闲链表：

空闲块通过头部中的大小字段隐含地连接着。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。

（1）放置策略：首次适配、下一次适配、最佳适配。

首次适配从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配从上一次查询结束的地方开始。最佳适配检查每个空闲块，选择适合所需请求大小的最小空闲块。

（2）合并策略：立即合并、推迟合并。

立即合并就是在每次一个块被释放时，就合并所有的相邻块；推迟合并就是等到某个稍晚的时候再合并空闲块。

带边界标记的合并：

在每个块的结尾添加一个脚部，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，从而使得对前面块的合并能够在常数时间之内进行。

### 2. 显式空闲链表

每个空闲块中，都包含一个 `pred`（前驱）和 `succ`（后继）指针。使用双向链表使首次适配的时间减少到空闲块数量的线性时间。

空闲链表中块的排序策略：一种是用后进先出的顺序维护链表，将新释放的块放置在链表的开始处，另一种方法是按照地址顺序来维护链表，链表中每个块的地址都小于它后继的地址。

分离存储：维护多个空闲链表，每个链表中的块有大致相等的大小。将所有可能的块大小分成一些等价类，也叫做大小类。

分离存储的方法：简单分离存储和分离适配。

## 7.10 本章小结

本章讨论了存储器地址空间，段式管理、页式管理，TLB 与四级页表支持下的 VA 到 PA 的变换，三级 Cache 支持下的物理内存访问，hello 进程 fork 时和 execve 时的内存映射，缺页故障与缺页中断处理和动态存储分配管理。

**（第 7 章 2 分）**

## 第 8 章 hello 的 IO 管理

### 8.1 Linux 的 IO 设备管理方法

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

I/O 接口操作

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、标准输出和标准错误。

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  $k\geq m$  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。

函数：

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

进程通过调用 `open` 函数来打开一个已存在的文件或者创建一个新文件。`open` 函数将 `filename` 转换为一个文件描述符，而且返回描述符数字。`flags` 参数指明了进程打算如何访问这个文件。`mode` 参数指定了新文件的访问权限位。

2.`int close(int fd)`

进程通过调用 `close` 函数关闭一个打开的文件。

### 3. ssize\_t read(int fd, void \*buf, size\_t n)

应用程序通过调用 read 函数来执行输入。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，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。

### 4. ssize\_t write(int fd, const void \*buf, size\_t n)

应用程序通过调用 write 函数来执行输出。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。

## 8.3 printf 的实现分析

<https://www.cnblogs.com/pianist/p/3315801.html>

从 vsprintf 生成显示信息，到 write 系统函数，到陷阱-系统调用 int 0x80 或 syscall。

字符显示驱动子程序：从 ASCII 到字模库到显示 vram（存储每一个点的 RGB 颜色信息）。

显示芯片按照刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

Printf 代码：

```
1. int printf(const char *fmt, ...)
2. {
3.     int i;
4.     char buf[256];
5.
6.     va_list arg = (va_list)((char*)&fmt + 4);
7.     i = vsprintf(buf, fmt, arg);
8.     write(buf, i);
9.
10.    return i;
11. }
```

其中，va\_list 是一个字符指针，arg 表示函数的第二个参数。

vsprintf 的代码：

```
1. int vsprintf(char *buf, const char *fmt, va_list args)
2. {
3.     char* p;
4.     char tmp[256];
5.     va_list p_next_arg = args;
6.
7.     for (p=buf; *fmt; fmt++) {
```

```
8.     if (*fmt != '%') {
9.         *p++ = *fmt;
10.        continue;
11.    }
12.
13.    fmt++;
14.
15.    switch (*fmt) {
16.        case 'x':
17.            itoa(tmp, *((int*)p_next_arg));
18.            strcpy(p, tmp);
19.            p_next_arg += 4;
20.            p += strlen(tmp);
21.            break;
22.        case 's':
23.            break;
24.        default:
25.            break;
26.    }
27. }
28.
29.     return (p - buf);
30. }
```

vsprintf 的作用是格式化。它接受确定输出格式的格式字符串 fnt。用格式字符串对个数变化的参数进行格式化，产生格式化输出，并返回要打印的字符串的长度。

write 的代码

```
mov eax, _NR_write
```

```
mov ebx, [esp + 4]
```

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

```
int INT_VECTOR_SYS_CALL
```

先给寄存器传了几个参数，然后通过系统调用 sys\_call

```
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：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

```

1.int getchar(void)
2. {
3.     static char buf[BUFSIZ];
4.     static char *bb = buf;
5.     static int n = 0;
6.     if(n == 0)
7.     {
8.         n = read(0, buf, BUFSIZ);
9.         bb = buf;
10.    }
11.    return(--n >= 0)?(unsigned char) *bb++ : EOF;
12. }

```

getchar 函数调用 read 函数，将整个缓冲区都读到 buf 里，并将缓冲区的长度赋值给 n。返回时返回 buf 的第一个元素，除非 n<0。

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

系统级 I/O 完成了 Hello 一生的最后一步。

以文件为单位，保证了较高的读写速度。

在各个层次上，这样的分块抽象，极大地提高了计算机的处理性能。

**(第 8 章 1 分)**

## 结论

预处理阶段：hello.c 引用的所有外部的库和宏定义展开合并到一个 hello.i 的文本文件中。

编译阶段：hello.i 文件编译成为汇编文件 hello.s，形成汇编指令。

汇编阶段：将 hello.s 文件汇编成为可重定位目标文件 hello.o，这个文件是二进制文件，但仍需要重定位。

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，此时的 hello 可以被 shell 执行。

Shell：在 shell 程序中输入 ./hello 1180301009 阚嘉良，shell 解析命令行，调用 fork 为其创建一个子进程，调用 execve 函数，启动加载器，将 hello 加载进内存，此时的 hello 成为了一个进程。

执行阶段：hello 按照时间片进行运行，相应各种信号和异常。在这个过程中：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。通过 L1L2L3cache 加快数据访问。

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：hello 进程结束，成为僵尸进程，shell 父进程回收子进程，内核删除这个进程的所有数据。

在整个过程中，我们的存储管理以及页表的使用，都离不开局部性对我们进行支持，可以说，局部性是现代计算机系统的构建基础。。

**(结论 0 分，缺失 -1 分，根据内容酌情加分)**

## 附件

<b>hello.i</b>	预编译处理文件
<b>Hello.s</b>	汇编文件
<b>Hello.o</b>	由 <b>hello.c</b> 生成的可重定位文件
<b>Hello</b>	链接后的可执行文件
<b>Hello.c</b>	源文件

(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] 深入理解计算机系统教材
- [2] <https://www.cnblogs.com/pianist/p/3315801.html>
- [3] <https://www.cnblogs.com/wangcp-2014/p/5146343.html>
- [4] [https://blog.csdn.net/ky\\_heart/article/details/51865526](https://blog.csdn.net/ky_heart/article/details/51865526)
- [5] [https://blog.csdn.net/weixin\\_43836778/article/details/90903213](https://blog.csdn.net/weixin_43836778/article/details/90903213)
- [6] <https://github.com/dylanmckay/vsprintf>

(参考文献 0 分，缺失 -1 分)