

# 从 OS 视角漫谈 HelloWorld

## 摘要：

本文详细介绍了 `hello.c` 程序在 Linux 系统下的生命周期，可以分为 P2P（从程序到进程）和 O2O（进程被分配内存和内存释放）两部分。

首先，`hello.c` 程序要经历由人能读懂的高级语言转化为机器能读懂的机器语言指令的阶段，即从源文件到目标文件的转化。主要体现在 P2P 阶段。在 Unix 系统上，这是通过编译器驱动程序完成的。`hello.c` 程序经历了预处理阶段、编译阶段、汇编阶段和链接阶段，转化为可执行目标文件“`hello`”，并被存放在磁盘上。

接下来是程序加载进内存并运行的过程，体现为 O2O 部分。在运行 `hello` 文件的过程中，`hello` 程序好像是独占了处理器和内存，程序中的代码和数据好像是系统内存中的唯一对象，这种程序运行的假象是依靠进程的概念实现的，涉及到了逻辑控制流和虚拟存储空间两部分。

通过在 `shell` 中输入可执行目标文件的名称，程序运行时 `shell` 会创建一个新的子进程，操作系统调用 `execve` 函数将 `hello` 程序加载到子进程的地址空间，开始在新进程的上下文中运行程序，执行指令。在程序运行过程中可能出现意外情况，程序需要对内部程序状态中的变化做出反应，现代系统通过异常控制流实现。常见的异常控制流的形式包括异常和信号两部分实现。

`Hello` 程序独占内存空间的假象是通过虚拟内存实现的。Linux 为每个进程维护了一个单独的虚拟地址空间，程序运行中，Linux 通过将虚拟内存区域与磁盘上的一个对象关联起来，进行内存映射。假设在翻译某个虚拟地址 A 时触发了缺页，控制将转移到内核的缺页处理程序。

`Hello` 程序的生命周期涉及了从程序到进程（P2P）和从零到零（O2O）两个角度，具有代表性，通过观察总结这一完整周期，可以更深刻的理解计算机运行程序的整个过程。

**关键词：**操作系统；Linux；进程；中断异常；存储；I/O 处理；文件系统

---

# 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 .....	- 5 -
1.2.1 硬件环境.....	- 5 -
1.2.2 软件环境.....	- 5 -
1.2.3 开发工具.....	- 5 -
1.3 中间结果 .....	- 6 -
1.4 本章小结 .....	- 6 -
<b>第 2 章 预处理 .....</b>	<b>- 7 -</b>
2.1 预处理的概念与作用 .....	- 7 -
2.2 在 UBUNTU 下预处理的命令 .....	- 7 -
2.3 HELLO 的预处理结果解析 .....	- 8 -
2.4 本章小结 .....	- 8 -
<b>第 3 章 编译 .....</b>	<b>- 9 -</b>
3.1 编译的概念与作用 .....	- 9 -
3.2 在 UBUNTU 下编译的命令 .....	- 9 -
3.3 HELLO 的编译结果解析 .....	- 9 -
3.4 本章小结 .....	- 15 -
<b>第 4 章 汇编 .....</b>	<b>- 16 -</b>
4.1 汇编的概念与作用 .....	- 16 -
4.2 在 UBUNTU 下汇编的命令 .....	- 16 -
4.3 可重定位目标 ELF 格式 .....	- 16 -
4.4 HELLO.o 的结果解析 .....	- 19 -
4.5 本章小结 .....	- 20 -
<b>第 5 章 链接 .....</b>	<b>- 21 -</b>
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 .....	- 24 -
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 .....	- 27 -
5.8 本章小结 .....	- 27 -
<b>第 6 章 OS 将程序加载到内存 .....</b>	<b>- 28 -</b>

6.1 引导部分:以 LINUX00 为例分析 .....	- 28 -
6.2 调试分析 LINUX 0.00 引导程序.....	- 28 -
<b>第 7 章 HELLO 进程管理 .....</b>	<b>- 37 -</b>
7.1 进程的概念与作用 .....	- 37 -
7.2 SHELL-BASH 的作用与处理流程.....	- 37 -
7.3 HELLO 的 FORK 进程创建过程.....	- 38 -
7.4 HELLO 的 EXECVE 过程.....	- 38 -
7.5 HELLO 的进程执行 .....	- 39 -
7.6 HELLO 的中断与系统调用 .....	- 40 -
7.7 HELLO 的异常与信号处理.....	- 42 -
7.8 本章小结 .....	- 47 -
<b>第 8 章 多任务与进程切换 .....</b>	<b>- 48 -</b>
8.1 前期调试 LINUX00 步骤 .....	- 48 -
8.2 IRET 前后栈的变化.....	- 49 -
8.3 模式切换 .....	- 50 -
8.4 任务切换 .....	- 52 -
8.5 进程切换: 调试分析 LINUX0.11 .....	- 58 -
<b>第 9 章 HELLO 的存储管理 .....</b>	<b>- 63 -</b>
9.1 HELLO 的存储器地址空间 .....	- 63 -
9.2 INTEL 逻辑地址到线性地址的变换-段式管理 .....	- 63 -
9.3 HELLO 的线性地址到物理地址的变换-页式管理 .....	- 64 -
9.4 TLB 与四级页表支持下的 VA 到 PA 的变换 .....	- 64 -
9.5 三级 CACHE 支持下的物理内存访问 .....	- 65 -
9.6 HELLO 进程 FORK 时的内存映射 .....	- 65 -
9.7 HELLO 进程 EXECVE 时的内存映射 .....	- 66 -
9.8 缺页故障与缺页中断处理 .....	- 66 -
9.9 本章小结 .....	- 67 -
<b>第 10 章 HELLO 的 I/O 管理.....</b>	<b>- 68 -</b>
10.1 LINUX 的 I/O 设备管理方法 .....	- 68 -
10.2 简述 UNIX I/O 接口及其函数.....	- 68 -
10.3 PRINTF 的实现分析.....	- 69 -
10.4 GETCHAR 的实现分析.....	- 70 -
10.5 本章小结 .....	- 70 -
<b>第 11 章 HELLO 的文件系统管理 .....</b>	<b>- 71 -</b>
11.1 文件路径解析.....	- 71 -
11.2 检查文件权限.....	- 71 -
11.3 获取文件元数据.....	- 71 -
11.4 分配磁盘缓存.....	- 71 -

---

11.5 缺页中断时处理.....	- 72 -
11.6 总结.....	- 72 -
结论 .....	- 73 -
附件 .....	- 74 -
参考文献.....	- 75 -

---

# 第 1 章 概述

## 1.1 Hello 简介

### (1) P2P: 程序变为进程

一个 C 语言程序变为进程的过程包括编写源代码、编译、加载、创建进程、设置环境、开始执行、进程执行、进程结束和进程退出等步骤。

**编写 C 语言程序：**首先使用文本编辑器编写 C 语言源代码 `hello.c`。

**编译源代码：**接下来使用 C 语言编译器（如 `gcc`）将源代码编译成机器代码。编译器会检查代码的语法错误，并将源代码转换成可执行文件。这一过程包括预处理生成新的 C 语言程序 `hello.i`，编译器编译形成汇编程序 `hello.s`，汇编器将汇编程序变为可重定位目标文件 `hello.o`，链接器将 `hello.o` 和其他库文件进行链接生成可执行目标文件 `hello`。

**加载可执行文件：**当操作系统启动时，它会加载可执行文件到内存中。操作系统会为每个进程分配一定的内存空间，并将可执行文件的内容复制到该内存空间中。

**创建进程：**操作系统会创建一个新的进程，为该进程分配一个进程标识符（PID）和进程控制块（PCB）。

**设置进程环境：**操作系统会为进程设置环境变量、文件描述符、信号处理函数等，以便进程可以正常运行。

此时，已经实现了从程序到进程的转化。

**开始执行：**操作系统将控制权交给进程，进程开始执行。当进程执行完所有指令或遇到异常时，进程会结束。

### (2) 020: 进程从无到有到无

程序从磁盘加载到内存，操作系统为其分配虚拟内存空间，并映射到物理内存。程序执行完毕后，操作系统回收资源，但内存并不会回到完全的零状态，而是保持一个动态的状态，随时准备为新的程序或数据分配。

**程序加载：**当操作系统决定运行一个程序，它会为该程序创建一个进程。这个过程包括为进程分配一个唯一的进程标识符（PID）。

**虚拟内存分配：**操作系统会为新创建的进程分配虚拟内存空间。虚拟内存是操作系统提供的一种机制，它允许每个进程拥有一个连续的、私有的地址空间，

---

这个空间在逻辑上是独立的，不受其他进程的影响。

**内存映射：**虚拟内存并不直接对应物理内存。操作系统使用内存管理单元（MMU）和页表来将虚拟地址映射到物理地址。当程序开始执行时，并不是所有的程序代码和数据都会立即加载到物理内存中。操作系统会根据需要（按需分页或按需换页）将程序的部分内容从磁盘加载到物理内存中。

**程序执行：**程序被加载到内存中，CPU 就可以开始执行程序的指令。程序的执行过程中，可能会动态分配内存，这些内存也会被映射到虚拟内存空间中。

**程序结束：**当程序执行完毕，操作系统会收到程序结束的信号。

**资源回收：**操作系统会回收为该程序分配的所有资源，包括虚拟内存和物理内存。这通常涉及到将物理内存中的数据写回磁盘，并更新页表以反映内存的释放。虚拟内存空间会被标记为可用，以便未来分配给其他进程。

**内存状态：**虽然内存中的数据被回收，但计算机的内存并不会回到“初始的零状态”。内存中可能仍然包含其他正在运行的程序的数据和代码。只有当所有程序都结束并且操作系统释放了所有内存，内存才会回到一个相对空闲的状态。

## 1.2 环境与工具

### 1.2.1 硬件环境

X64 CPU; 2.50GHz; 16.0G RAM; 256GHD Disk 以上

设备名称:	DESKTOP-3UE363C
处理器:	12th Gen Intel(R) Core(TM) i5-12500H 2.50 GHz
机带 RAM:	16.0 GB (15.7 GB 可用)

### 1.2.2 软件环境

Windows10 64 位; VirtualBox 11 以上;

### 1.2.3 开发工具

Visual Studio 2022; Bochs

---

## 1.3 中间结果

文件名	作用
hello.c	源程序
hello.i	预处理后的修改的 C 程序
hello.s	汇编程序
hello.o	可重定位目标文件
hello	可执行目标文件
objhello.o.txt	hello.o 的反汇编文件
objhello.txt	hello 的反汇编文件

## 1.4 本章小结

### **P2P：程序变为进程**

一个 C 语言程序变为进程的过程包括编写源代码、编译、加载、创建进程、设置环境、开始执行等步骤。

### **020：进程从无到有到无**

程序从磁盘加载到内存，操作系统为其分配虚拟内存空间，并映射到物理内存。程序执行完毕后，操作系统回收资源，但内存并不会回到完全的零状态，而是保持一个动态的状态，随时准备为新的程序或数据分配。

---

## 第 2 章 预处理

### 2.1 预处理的概述与作用

#### 2.1.1 概念

预处理是编译过程的第一步，它是由预处理器完成的。预处理器（CPP）根据以字符开头的命令，修改原始的 C 程序。比如 `hello.c` 中第 1 行的 `include` 命令告诉预处理器读取系统头文件 `stdio.h` 的内容，并把它直接插入程序文本中。结果就得到了另一个 C 程序，通常是以 `.i` 作为文件扩展名。

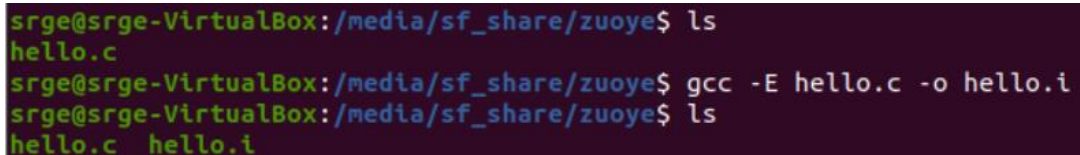
#### 2.1.2 作用

- （1）宏替换：将源代码中的宏调用替换为宏定义的内容。
- （2）头文件展开：预处理器读取头文件内容并直接插入程序文本中。
- （3）条件编译：根据条件编译指令（如 `if`, `ifdef`, `ifndef`, `elif`, `else`, `endif`）来决定是否编译某一部分代码。
- （4）注释处理：移除源代码中的注释。
- （5）预定义宏：预处理器定义了一些预定义的宏，这些宏提供了关于源代码的元数据，如当前行号、文件名、编译日期和时间等。

预处理的作用是让开发者能够编写更灵活、更易于维护的代码。通过宏定义和条件编译，开发者可以编写跨平台的代码，或者根据不同的编译选项生成不同的程序版本。头文件包含则允许代码的模块化和重用，提高了开发效率。预处理器是 C 语言编译过程中不可或缺的一环，它为编译器提供了处理后的源代码，以便进行后续的编译步骤。

### 2.2 在 Ubuntu 下预处理的命令

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



```
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello.c
srge@srge-VirtualBox:/media/sf_share/zuoye$ gcc -E hello.c -o hello.i
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello.c  hello.i
```



---

## 2.3 Hello 的预处理结果解析

```
3032 extern int getsubopt (char **__restrict __optionp,  
3033     char *const *__restrict __tokens,  
3034     char **__restrict __valuep)  
3035     __attribute__((__nothrow__ , __leaf__)) __attribute__((__nonnull__ (1, 2, 3))) ;  
3036 # 1003 "/usr/include/stdlib.h" 3 4  
3037 extern int getloadavg (double __loadavg[], int __nelem)  
3038     __attribute__((__nothrow__ , __leaf__)) __attribute__((__nonnull__ (1)));  
3039 # 1013 "/usr/include/stdlib.h" 3 4  
3040 # 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4  
3041 # 1014 "/usr/include/stdlib.h" 2 3 4  
3042 # 1023 "/usr/include/stdlib.h" 3 4  
3043  
3044 # 10 "hello.c" 2  
3045  
3046  
3047 # 11 "hello.c"  
3048 int main(int argc,char *argv[]){  
3049     int i;  
3050  
3051     if(argc!=5){  
3052         printf("用法: Hello 学号 姓名 手机号 秒数! \n");  
3053         exit(1);  
3054     }  
3055     for(i=0;i<10;i++){  
3056         printf("Hello %s %s %s\n",argv[1],argv[2],argv[3]);  
3057         sleep(atoi(argv[4]));  
3058     }  
3059     getchar();  
3060     return 0;  
3061 }
```

1. 预处理后的文件仍然是文本文件，但代码量显著增加达到了 3061 行。
2. 预处理将 `stdio.h`、`unistd.h`、`stdlib.h` 头文件的内容插入到源代码中。
3. 行号和文件名标记：预处理在源代码中插入了行号和文件名信息。

## 2.4 本章小结

本章主要介绍了预处理的概念与作用，以及在 Ubuntu 下预处理的命令。通过调用预处理命令对 `hello.c` 程序进行操作，分析预处理结果 `hello.i`，验证了预处理过程的功能。对于预处理这一部分有了更深刻的体会和理解。

---

## 第 3 章 编译

### 3.1 编译的概念与作用

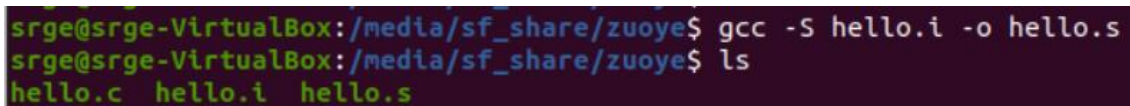
#### 3.1.1 概念

编译是指编译器(ccl)将高级程序语言翻译为汇编语言文本文件的过程，即将 hello.i 翻译成文本文件 hello.s 的过程。

#### 3.1.2 作用

将人类可读的高级语言转换为计算机可执行的机器语言，同时在这个过程中可以进行优化，以提高程序的运行效率。汇编语言作为编译过程的中间产物，为不同高级语言提供了通用的输出格式，有助于后续的链接和加载过程。

### 3.2 在 Ubuntu 下编译的命令

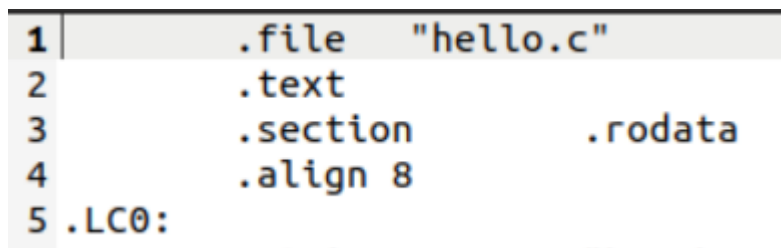


```
srge@srge-VirtualBox:/media/sf_share/zuoye$ gcc -S hello.i -o hello.s
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello.c hello.i hello.s
```

图 3.1 编译命令

### 3.3 Hello 的编译结果解析

#### 3.3.1 伪指令



1	.file	"hello.c"
2	.text	
3	.section	.rodata
4	.align	8
5	.LC0:	

图 3.2 hello.s 中伪指令

汇编代码中以 ‘.’ 开头的是伪指令，记录文件相关信息，用于汇编和链接阶段。例如.file 代表源文件名称，.text 代表代码段。

### 3.3.2 数据

(1) 参数: int argc, char argv[]

<pre> int main(int argc, char *argv[]) {     int i;      if(argc!=5) { </pre> <p>图 3.3 (a) 源代码中参数</p>	<p>Main 函数有两个参数: int 型 argc, 和 char 数组 argv。</p>
<pre> 22     movl    %edi, -20(%rbp) 23     movq    %rsi, -32(%rbp) </pre> <p>图 3.3 (b) 汇编代码中参数</p>	<p>参数分别存在寄存器 %rdi 和寄存器 %rsi 中。将参数 argc 传入 main 函数的值存储到 %rbp 的地址减去 20 字节的位置。将 argv 传入 main 函数的值存储到 -32(%rbp) 表示的位置</p>

(2) 局部变量: int i;

<pre> int i;  if(argc!=5){     printf("用法: Hello 学号 姓名 手机号 秒数! \n");     exit(1); } for(i=0;i&lt;10;i++){ </pre> <p>图 3.4 (a) 源代码中局部变量 i</p>	<p>源代码中局部变量 i 用于 for 循环中计数, 并且没有初值</p>
<pre> 30 .L2: 31     movl    \$0, -4(%rbp) 32     jmp     .L3 </pre> <p>图 3.4 (b) 汇编中局部变量</p>	<p>在汇编代码中 31 行体现, 代码将 i 存入栈中, 优化掉了设置局部变量的步骤, 直接将 0 存入栈中计数</p>

(3) 常量: 字符串常量

<pre> printf("用法: Hello 学号 姓名 手机号 秒数! \n"); exit(1);  (i=0;i&lt;10;i++){     printf("Hello %s %s %s\n", argv[1], argv[2], argv[3]); } </pre> <p>图 3.5 (a) 源代码中的字符串常量</p>	<pre> 3     .section      .rodata 4     .align 8 5 .LC0: 6     .string      "\347\224\250\346\263\225: Hello \345\255\246\345\217\267 \345\247\223\345\220\215 \346\211\213\346\234\272\345\217\267 \347\247\222\346\225\260\357\274\201" 7 .LC1: 8     .string      "Hello %s %s %s\n" </pre> <p>图 3.5 (b) 汇编中的字符串常量: 存在 rodata 只读代码段</p>
--	--

#### (4) 数据类型

汇编中通过寄存器类型、指令后缀等方式指明数据类型

```
49      movq    (%rax), %rax
50      movq    %rax, %rdi
51      call    atoi@PLT
52      movl    %eax, %edi
53      call    sleep@PLT
54      addl    $1, -4(%rbp)
55 .L3:
56      cmpl    $9, -4(%rbp)
57      jle     .L4
```

图 3.6 汇编中数据字节数表示

- A) 汇编语言通过在 mov、add 等指令后加后缀来表现数据类型的字节数，b 代表一字节，w 代表二字节，l 代表四字节，q 代表八字节。
- B) 通过寄存器表示数据类型区分整型和浮点型，%rax、%rdx 等寄存器均是整型寄存器，存放整数类型的数据；而%xmm0 等寄存器存放浮点类型。
- C) 57 行中的 jle 的后缀代表进行有符号数的比较，若无符号的比较则为 be。

#### 3.3.3 赋值

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

图 3.7 汇编中的赋值

汇编语言中的赋值通过 mov 指令实现数据的“复制”，例如 31 行将立即数 0 复制到 %rbp-4 的位置，实际上对应了源代码中将 i 初始化为 0 的操作

#### 3.3.4 强制类型转换

```
47      movq    -32(%rbp), %rax
48      addq    $32, %rax
49      movq    (%rax), %rax
50      movq    %rax, %rdi
51      call    atoi@PLT
```

图 3.8 汇编中的强制类型转换

通过调用 atoi 函数实现 atoi(argv[4])的操作

### 3.3.5 算术操作

```
38      addq    $16, %rax
```

图 3.9 汇编语言中的运算

汇编语言中的算术操作通过拆分成 add、sub、leaq 等指令实现。38 行将寄存器 %rax 中的数加上 16。

### 3.3.6 关系操作

```
24      cmpl    $5, -20(%rbp)
25      je      .L2
```

图 3.10 argc!=5 的汇编表达

```
56      cmpl    $9, -4(%rbp)
57      jle     .L4
```

图 3.11 i<10 的汇编表达

### 3.3.7 数组操作

```
33 .L4:
34      movq    -32(%rbp), %rax
35      addq    $24, %rax
36      movq    (%rax), %rcx
37      movq    -32(%rbp), %rax
38      addq    $16, %rax
39      movq    (%rax), %rdx
40      movq    -32(%rbp), %rax
41      addq    $8, %rax
42      movq    (%rax), %rax
```

图 3.12 数组 argv 存入寄存器

3.3.8 控制转移：hello.c 中涉及条件转移 if 和循环控制 for

```
if(argc!=5) {
```

图 3.13 (a) 源代码 if 语句

```
24      cmpl    $5, -20(%rbp)
25      je      .L2
26      leaq    .LC0(%rip), %rdi
27      call    puts@PLT
28      movl    $1, %edi
29      call    exit@PLT
30 .L2:
31      movl    $0, -4(%rbp)
32      jmp     .L3
```

图 3.13 (b) 汇编 if 语句实现

If 语句主要是通过 `cmp` 指令和 `jmp` 类指令实现跳转，24 行比较 `argc` 和 4，如果不满足跳转.L2,满足则执行下面语句。

```
for(i=0;i<10;i++) {
```

图 3.14 (a) 源代码循环控制

```
30 .L2:
31      movl    $0, -4(%rbp)
32      jmp     .L3
33 .L4:
34      movq    -32(%rbp), %rax
35      addq    $24, %rax
36      movq    (%rax), %rcx
37      movq    -32(%rbp), %rax
38      addq    $16, %rax
39      movq    (%rax), %rdx
40      movq    -32(%rbp), %rax
41      addq    $8, %rax
42      movq    (%rax), %rax
43      movq    %rax, %rsi
44      leaq    .LC1(%rip), %rdi
45      movl    $0, %eax
46      call    printf@PLT
47      movq    -32(%rbp), %rax
48      addq    $32, %rax
49      movq    (%rax), %rax
50      movq    %rax, %rdi
51      call    atoi@PLT
52      movl    %eax, %edi
53      call    sleep@PLT
54      addl    $1, -4(%rbp)
55 .L3:
56      cmpl    $9, -4(%rbp)
57      jle     .L4
58      call    getchar@PLT
59      movl    $0, %eax
```

图 3.14 (b) 汇编中的循环控制实现

For 语句的实现：.L2 循环遍量初始化、.L4 循环体循环、.L3 循环条件检查



### 3.3.6 函数调用

汇编语言中的函数调用通过 `call` 指令实现指令跳转，并且在调用 `call` 指令前将参数保存到对应的参数寄存器中。`Call` 指令包含 `push` 返回地址入栈和指令跳转的功能。

```
26      leaq    .LC0(%rip), %rdi
27      call    puts@PLT
```

图 3.15 调用 `puts` 函数

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

图 3.16 调用 `exit` 函数

```
34      movq    -32(%rbp), %rax
35      addq    $24, %rax
36      movq    (%rax), %rcx
37      movq    -32(%rbp), %rax
38      addq    $16, %rax
39      movq    (%rax), %rdx
40      movq    -32(%rbp), %rax
41      addq    $8, %rax
42      movq    (%rax), %rax
43      movq    %rax, %rsi
44      leaq    .LC1(%rip), %rdi
45      movl    $0, %eax
46      call    printf@PLT
```

图 3.17 调用 `printf` 函数

```
48      addq    $32, %rax
49      movq    (%rax), %rax
50      movq    %rax, %rdi
51      call    atoi@PLT
```

图 3.17 调用 `atoi` 函数

```
52      movl    %eax, %edi
53      call    sleep@PLT
54      addl    $1, -4(%rbp)
```

图 3.17 调用 `sleep` 函数

```
58      call    getchar@PLT
```

图 3.17 调用 `getchar` 函数

---

## 3.4 本章小结

本章主要分析了 `hello.i` 文件到 `hello.s` 文件的过程中编译的概念与作用，分析 `hello` 的汇编结果，更加深入理解了汇编语言的实现以及与 C 语言的对应。伪指令主要是前面有 `.` 标记的起解释说明作用的部分。数据的实现中，参数主要通过固定的参数寄存器传参，局部变量保存在栈中，字符串常量保存在 `.data` 只读代码段。数据类型主要通过寄存器类型、指令后缀等方式指明。变量的赋值在汇编语言中主要通过 `mov` 指令实现数据的复制。也涉及强制类型转换，主要通过调用系统函数来实现。算术操作由 `add` 等指令实现。关系的判断操作由 `cmp` 指令和条件跳转指令实现。数组存在栈中，对数组的操作通过栈顶指针来访问。在控制转移中，`hello.c` 中用到了 `if` 条件控制和 `for` 循环控制两种操作，通过 `cmp` 指令和条件跳转指令实现。汇编语言通过栈实现函数调用。主要通过 `call` 指令实现 `push` 返回地址入栈和修改 `rip` 的值实现跳转的作用。



---

## 第 4 章 汇编

### 4.1 汇编的概念与作用

#### 4.1.1 概念

汇编器(as)将 `hello.s` 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，并将结果保存在目标文件 `hello.o` 中。`hello.o` 文件是一个二进制文件，它包含的 17 个字节是函数 `main` 的指令编码。

#### 4.1.2 作用

**转换成机器代码：**汇编器将汇编语言中的指令转换成计算机的机器代码

**优化代码：**汇编器会对代码进行一些优化，比如去除无用的指令、优化跳转等，以提高程序的运行效率。

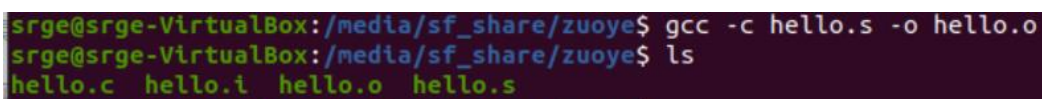
**生成目标文件：**汇编后的代码被保存为 `.o` 文件，这个文件包含了程序运行所需的全部机器代码和符号表信息，符号表包含了变量和函数的名称以及它们在内存中的地址。

**为链接做准备：**在程序编译过程中，不同的源文件可能会被汇编成不同的 `.o` 文件，这些 `.o` 文件在链接阶段会被合并成一个可执行文件。

**提供调试信息：**汇编器会生成调试信息，这些信息在程序调试时非常有用，可以帮助开发者找到程序中的错误。

**处理宏指令：**汇编器会处理汇编语言中的宏指令，这些宏指令在汇编时会被替换成实际的指令代码。

### 4.2 在 Ubuntu 下汇编的命令



```
srge@srge-VirtualBox:/media/sf_share/zuoye$ gcc -c hello.s -o hello.o
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello.c  hello.i  hello.o  hello.s
```

图 4.1 汇编命令

### 4.3 可重定位目标 elf 格式

#### 4.3.1 典型可重定位目标文件格式

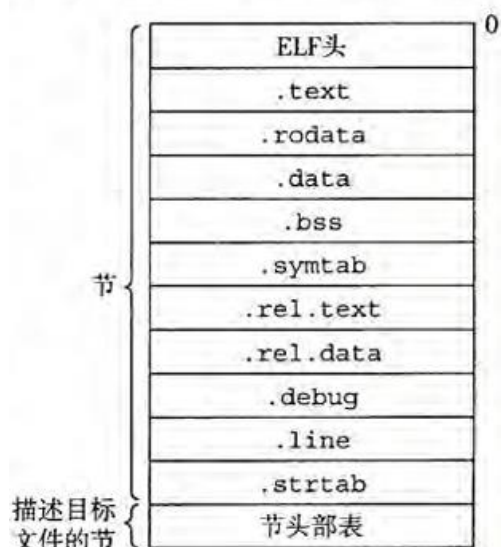


图 4.2 典型可重定位目标文件格式

### 4.3.2 以 hello.o 为例分析

#### (1) EFL 头: readelf -h hello.o

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，以及节头部表中条目的大小和数量。不同节的位置和大小是由节头部表描述的，其中目标文件中每个节都有一个固定大小的条目。

```
srge@srge-VirtualBox:/media/sf_share/zuoye$ readelf -h hello.o
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)
Version:    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: 1264 (bytes into file)
标志:      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: 14
Section header string table index: 13
```

图 4.3 hello.o 文件 ELF 头

#### (2) 节头部表: readelf -S hello.o

描述不同节的名称、类型、地址、偏移量、大小、全体大小、旗标、链接、信息、对齐等信息。

```

srges@VirtualBox:/media/sf_share/zuoye$ readelf -S hello.o
There are 14 section headers, starting at offset 0x4f0:

节头:
[ 0] 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  0  0  0
     0000000000000000  0000000000000000
[ 1] .text      PROGBITS  0000000000000000  0  0  1
     000000000000009d  0000000000000000  AX  0  0
[ 2] .rela.text  RELA      0000000000000000  000003a0
     00000000000000c0  0000000000000018  I   11  1  8
[ 3] .data      PROGBITS  0000000000000000  000000dd
     0000000000000000  0000000000000000  WA  0  0  1
[ 4] .bss       NOBITS    0000000000000000  000000dd
     0000000000000000  0000000000000000  WA  0  0  1
[ 5] .rodata    PROGBITS  0000000000000000  000000e0
     0000000000000040  0000000000000000  A   0  0  8
[ 6] .comment   PROGBITS  0000000000000000  00000120
     000000000000002c  0000000000000001  MS  0  0  1
[ 7] .note.GNU-stack  PROGBITS  0000000000000000  0000014c
     0000000000000000  0000000000000000  0   0  0  1
[ 8] .note.gnu.property  NOTE      0000000000000000  00000150
     0000000000000020  0000000000000000  A   0  0  8
[ 9] .eh_frame  PROGBITS  0000000000000000  00000170
     0000000000000038  0000000000000000  A   0  0  8
[10] .rela.eh_frame  RELA      0000000000000000  00000460
     0000000000000018  0000000000000018  I   11  9  8
[11] .symtab     SYMTAB     0000000000000000  000001a8
     00000000000001b0  0000000000000018  12  10  8
[12] .strtab     STRTAB     0000000000000000  00000358
     0000000000000048  0000000000000000  0   0  1
[13] .shstrtab   STRTAB     0000000000000000  00000478
     0000000000000074  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 4.4 hello.o 文件节头部表

### (3) 节符号表: readelf -s hello.o

节符号表中存储了程序中定义和引用的函数和全局变量的信息。包含的信息主要包括符号名称、符号类型（表示符号的类型，如函数、数据、文件等）、符号绑定、符号地址、符号大小、符号所在节和符号索引等信息。

```

srges@VirtualBox:/media/sf_share/zuoye$ readelf -s hello.o

Symbol table '.symtab' contains 18 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 9
 9: 0000000000000000 0 SECTION LOCAL DEFAULT 6
10: 0000000000000000 157 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 atoi
16: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND sleep
17: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND getchar

```

图 4.5 hello.o 文件节符号表

#### (4) 重定位节: readelf -r hello.o

重定位节包含了重定位信息, 这些信息告诉链接器如何修改目标文件中的某些部分, 以便在生成最终的可执行文件时, 符号引用能够正确地指向它们的定义。

偏移量: 目标文件中需要重定位的字节的偏移量。

符号索引: 指向符号表中符号的索引, 该符号的地址被用来计算重定位的值。

重定位类型: 指定了如何进行重定位, 例如 PC 相对重定位、绝对重定位等。

添加数: 一个可选的数值, 用于在计算最终地址时添加到符号地址上。

```
srge@srge-VirtualBox:/media/sf_share/zuoye$ readelf -r hello.o

重定位节 '.rela.text' at offset 0x3a0 contains 8 entries:
  偏移量      信息      类型      符号值      符号名称 + 加数
00000000001c 000500000002 R_X86_64_PC32 0000000000000000 .rodata - 4
000000000021 000c00000004 R_X86_64_PLT32 0000000000000000 puts - 4
00000000002b 000d00000004 R_X86_64_PLT32 0000000000000000 exit - 4
00000000005f 000500000002 R_X86_64_PC32 0000000000000000 .rodata + 2c
000000000069 000e00000004 R_X86_64_PLT32 0000000000000000 printf - 4
00000000007c 000f00000004 R_X86_64_PLT32 0000000000000000 atoi - 4
000000000083 001000000004 R_X86_64_PLT32 0000000000000000 sleep - 4
000000000092 001100000004 R_X86_64_PLT32 0000000000000000 getchar - 4

重定位节 '.rela.eh_frame' at offset 0x460 contains 1 entry:
  偏移量      信息      类型      符号值      符号名称 + 加数
000000000020 000200000002 R_X86_64_PC32 0000000000000000 .text + 0
```

图 4.6 hello.o 文件重定位节

## 4.4 Hello.o 的结果解析

### 4.4.1 反汇编命令

```
srge@srge-VirtualBox:/media/sf_share/zuoye$ objdump -r -d hello.o >objhello.o.txt
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello.c  hello.i  hello.o  hello.s  objhello.o.txt
```

图 4.7 反汇编命令

### 4.4.2 反汇编结果解析

将 hello.o 反汇编的结果 objhello.o.txt 与 hello.s 对比

- (1) hello.s 中有大量的以“.”开头的伪指令, 反汇编代码中则没有。
- (2) hello.o 的反汇编中多了机器指令和地址, 地址从 main 函数开始从 0 增加。可以看出机器指令和汇编指令是一一对应的。
- (3) hello.s 中十进制表示的立即数, 在 hello.o 的反汇编中为十六进制表示
- (4) hello.o 反汇编的跳转指令中, 是对应的地址的跳转, 而 hello.s 中的跳转则是以.Lx 命名的代码块为单位的跳转



<pre> .align 8 .LC0: .string "\347\224\250\346\263\225: Hello \345\255\246\345\21 .LC1: .string "Hello %s %s %s\n" .text .globl main .type main, @function main: .LFB6: .cfi_startproc endbr64 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 \$5, -20(%rbp) je .L2 leaq .LC0(%rip), %rdi call puts@PLT movl \$1, %edi call exit@PLT .L2: movl \$0, -4(%rbp) jmp .L3 .L4: movq -32(%rbp), %rax addq \$24, %rax movq (%rax), %rcx 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 movq -32(%rbp), %rax addq \$32, %rax movq (%rax), %rax movq %rax, %rdi call atoi@PLT 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 .LFE6: .size main, .-main .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0" .section .note.GNU-stack,"",@progbits .section .note.gnu.property,"a" .align 8 .long 1f - 0f .long 4f - 1f .long 5 0: .string "GNU" 1: .align 8 .long 0xc0000002 .long 3f - 2f 2: .long 0x3 3: .align 8 4: </pre>	<pre> hello.o: 文件格式 elf64-x86-64  Disassembly of section .text:  0000000000000000 &lt;main&gt;: 0: f3 0f 1e fa      endbr64 4: 55              push %rbp 5: 48 89 e5        mov %rsp,%rbp 8: 48 83 ec 20     sub \$0x20,%rsp c: 89 7d ec        mov %edi,-0x14(%rbp) f: 48 89 75 e0     mov %rsi,-0x20(%rbp) 13: 83 7d ec 05     cmpl \$0x5,-0x14(%rbp) 17: 74 16          je 2f &lt;main+0x2f&gt; 19: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 20 &lt;main+0x20&gt; 1c: R_X86_64_PC32 .rodata-0x4 20: e8 00 00 00 00 callq 25 &lt;main+0x25&gt; 21: R_X86_64_PLT32 puts-0x4 25: bf 01 00 00 00 mov \$0x1,%edi 2a: e8 00 00 00 00 callq 2f &lt;main+0x2f&gt; 2b: R_X86_64_PLT32 exit-0x4 2f: c7 45 fc 00 00 00 00 movl \$0x0,-0x4(%rbp) 36: eb 53          jmp 8b &lt;main+0x8b&gt; 38: 48 8b 45 e0     mov -0x20(%rbp),%rax 3c: 48 83 c0 18     add \$0x18,%rax 40: 48 8b 08        mov (%rax),%rcx 43: 48 8b 45 e0     mov -0x20(%rbp),%rax 47: 48 83 c0 10     add \$0x10,%rax 4b: 48 8b 10        mov (%rax),%rdx 4e: 48 8b 45 e0     mov -0x20(%rbp),%rax 52: 48 83 c0 08     add \$0x8,%rax 56: 48 8b 00        mov (%rax),%rax 59: 48 89 c6        mov %rax,%rsi 5c: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 63 &lt;main+0x63&gt; 5f: R_X86_64_PC32 .rodata+0x2c 63: b8 00 00 00 00 mov \$0x0,%eax 68: e8 00 00 00 00 callq 6d &lt;main+0x6d&gt; 69: R_X86_64_PLT32 printf-0x4 6d: 48 8b 45 e0     mov -0x20(%rbp),%rax 71: 48 83 c0 20     add \$0x20,%rax 75: 48 8b 00        mov (%rax),%rax 78: 48 89 c7        mov %rax,%rdi 7b: e8 00 00 00 00 callq 80 &lt;main+0x80&gt; 7c: R_X86_64_PLT32 atoi-0x4 80: 89 c7          mov %eax,%edi 82: e8 00 00 00 00 callq 87 &lt;main+0x87&gt; 83: R_X86_64_PLT32 sleep-0x4 87: 83 45 fc 01     addl \$0x1,-0x4(%rbp) 8b: 83 7d fc 09     cmpl \$0x9,-0x4(%rbp) 8f: 7e a7          jle 38 &lt;main+0x38&gt; 91: e8 00 00 00 00 callq 96 &lt;main+0x96&gt; 92: R_X86_64_PLT32 getchar-0x4 96: b8 00 00 00 00 mov \$0x0,%eax 9b: c9            leaveq 9c: c3            retq </pre>
--	---

图 4.8 (a) hello.s 文件

图 4.8 (b) hello.o 反汇编

## 4.5 本章小结

本章主要对程序运行的汇编阶段展开介绍，包括汇编的概念与作用、在 Ubuntu 下汇编的命令、可重定位目标 elf 格式、hello.o 的结果解析等内容。

---

## 第 5 章 链接

### 5.1 链接的概念与作用

#### 5.1.1 概念

链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。这个过程包括解析符号引用、重定位地址、合并段等操作。

链接可以执行于编译时也就是在源代码被翻译成机器代码时；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；甚至执行于运行时，也就是由应用程序来执行。在早期的计算机系统中，链接是手动执行的。在现代系统中，链接是由叫做链接器的程序自动执行的。

#### 5.1.2 作用

链接使得分离编译成为可能。我们不用将一个大型的应用程序组织为一个巨大的源文件，而是可以把它分解为更小、更好管理的模块，可以独立地修改和编译这些模块。当我们改变这些模块中的一个时，只需简单地重新编译它，并重新链接应用，而不必重新编译其他文件。


将多个源文件编译生成的目标文件合并成一个可执行文件。

解析目标文件中的符号引用，确保所有函数和变量都能正确调用。

重定位地址，将目标文件中的相对地址转换为绝对地址。

合并段，将多个目标文件中相同类型的段（代码段、数据段）合成一个段。

### 5.2 在 Ubuntu 下链接的命令



```
srge@srge-VirtualBox:/media/sf_share/zuoye$ 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
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello hello.c hello.i hello.o hello.s objhello.o.txt
```

图 5.1 链接命令

### 5.3 可执行目标文件 hello 的格式

#### 5.3.1 通用的格式

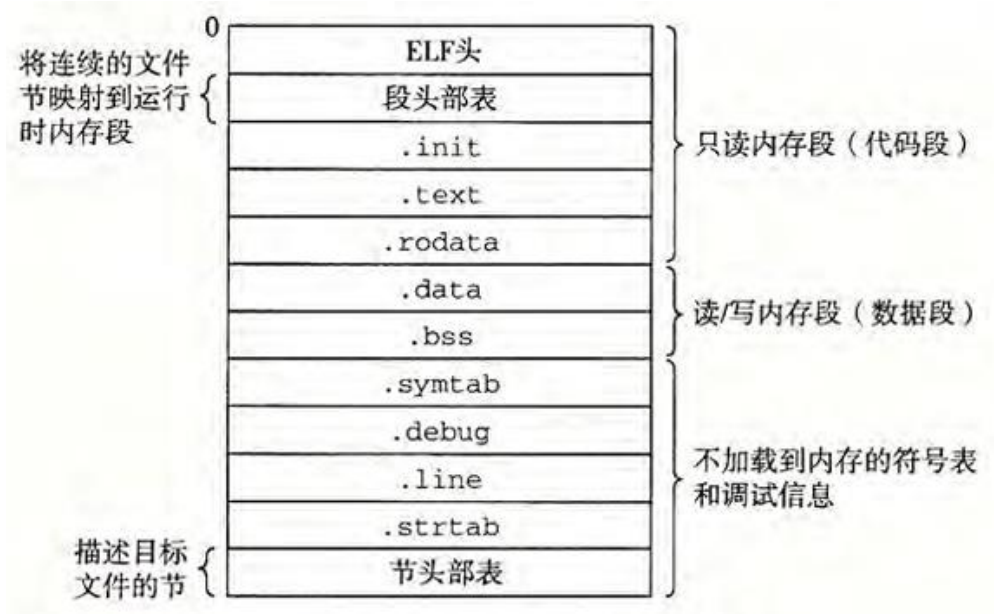


图 5.2 可执行目标文件通用格式

### 5.3.2 hello 的 ELF 分析

#### (1) ELF 头: readelf -h hello

能够得到 hello 文件的类型 (可执行文件)、程序入口点地址 (0x4010f0)、程序头起点、节总数 (27 个) 等信息。

```
srges@srge-VirtualBox: /media/sf_share/zuoye$ readelf -h hello
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)
  Version:                               1 (current)
  OS/ABI:                               UNIX - System V
  ABI 版本:                               0
  类型:                               EXEC (可执行文件)
  系统架构:                               Advanced Micro Devices X86-64
  版本:                               0x1
  入口点地址:                               0x4010f0
  程序头起点:                               64 (bytes into file)
  Start of section headers:               14208 (bytes into file)
  标志:                               0x0
  Size of this header:                     64 (bytes)
  Size of program headers:                 56 (bytes)
  Number of program headers:               12
  Size of section headers:                 64 (bytes)
  Number of section headers:               27
  Section header string table index:      26
```

图 5.3 hello 的 ELF 头

## (2) 节头部表

```

srge@srge-VirtualBox:/media/sf_share/zuoye$ readelf -S hello
There are 27 section headers, starting at offset 0x3780:

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] .interp      PROGBITS 00000000004002e0 000002e0
     000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.gnu.proper NOTE      0000000000400300 00000300
     0000000000000020 0000000000000000 A 0 0 8
[ 3] .note.ABI-tag NOTE      0000000000400320 00000320
     0000000000000020 0000000000000000 A 0 0 4
[ 4] .hash        HASH      0000000000400340 00000340
     0000000000000038 0000000000000004 A 6 0 8
[ 5] .gnu.hash     GNU_HASH 0000000000400378 00000378
     000000000000001c 0000000000000000 A 6 0 8
[ 6] .dynsym       DYNSYM   0000000000400398 00000398
     00000000000000d8 0000000000000018 A 7 1 8
[ 7] .dynstr       STRTAB   0000000000400470 00000470
     000000000000005c 0000000000000000 A 0 0 1
[ 8] .gnu.version  VERSYM   00000000004004cc 000004cc
     0000000000000012 0000000000000002 A 6 0 2
[ 9] .gnu.version_r VERNEED  00000000004004e0 000004e0
     0000000000000020 0000000000000000 A 7 1 8
[10] .rela.dyn      RELA     0000000000400500 00000500
     0000000000000030 0000000000000018 A 6 0 8
[11] .rela.plt      RELA     0000000000400530 00000530
     0000000000000090 0000000000000018 AI 6 21 8
[12] .init         PROGBITS 0000000000401000 00001000
     000000000000001b 0000000000000000 AX 0 0 4
[13] .plt          PROGBITS 0000000000401020 00001020
     0000000000000070 0000000000000010 AX 0 0 16
[14] .plt.sec      PROGBITS 0000000000401090 00001090
     0000000000000060 0000000000000010 AX 0 0 16
[15] .text         PROGBITS 00000000004010f0 000010f0
     0000000000000155 0000000000000000 AX 0 0 16
[16] .fini         PROGBITS 0000000000401248 00001248
     000000000000000d 0000000000000000 AX 0 0 4
-----
[17] .rodata       PROGBITS 0000000000402000 00002000
     0000000000000048 0000000000000000 A 0 0 8
[18] .eh_frame     PROGBITS 0000000000402048 00002048
     00000000000000fc 0000000000000000 A 0 0 8
[19] .dynamic      DYNAMIC  0000000000403e50 00002e50
     00000000000001a0 0000000000000010 WA 7 0 8
[20] .got          PROGBITS 0000000000403ff0 00002ff0
     0000000000000010 0000000000000008 WA 0 0 8
[21] .got.plt      PROGBITS 0000000000404000 00003000
     0000000000000048 0000000000000008 WA 0 0 8
[22] .data         PROGBITS 0000000000404048 00003048
     0000000000000004 0000000000000000 WA 0 0 1
[23] .comment      PROGBITS 0000000000000000 0000304c
     000000000000002b 0000000000000001 MS 0 0 1
[24] .symtab       SYMTAB   0000000000000000 00003078
     00000000000000c8 0000000000000018 25 30 8
[25] .strtab       STRTAB   0000000000000000 00003540
     0000000000000158 0000000000000000 0 0 1
[26] .shstrtab     STRTAB   0000000000000000 00003698
     00000000000000e1 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 5.4 hello 的节头部表



## 5.4 hello 的虚拟地址空间

使用 edb 加载 hello，查看本进程的虚拟地址空间各段信息，并与 5.3 对照分析说明。

由下图可知，虚拟地址空间的起始位置是 0x401000。可以查询各个节的信息，起始地址，大小等信息。

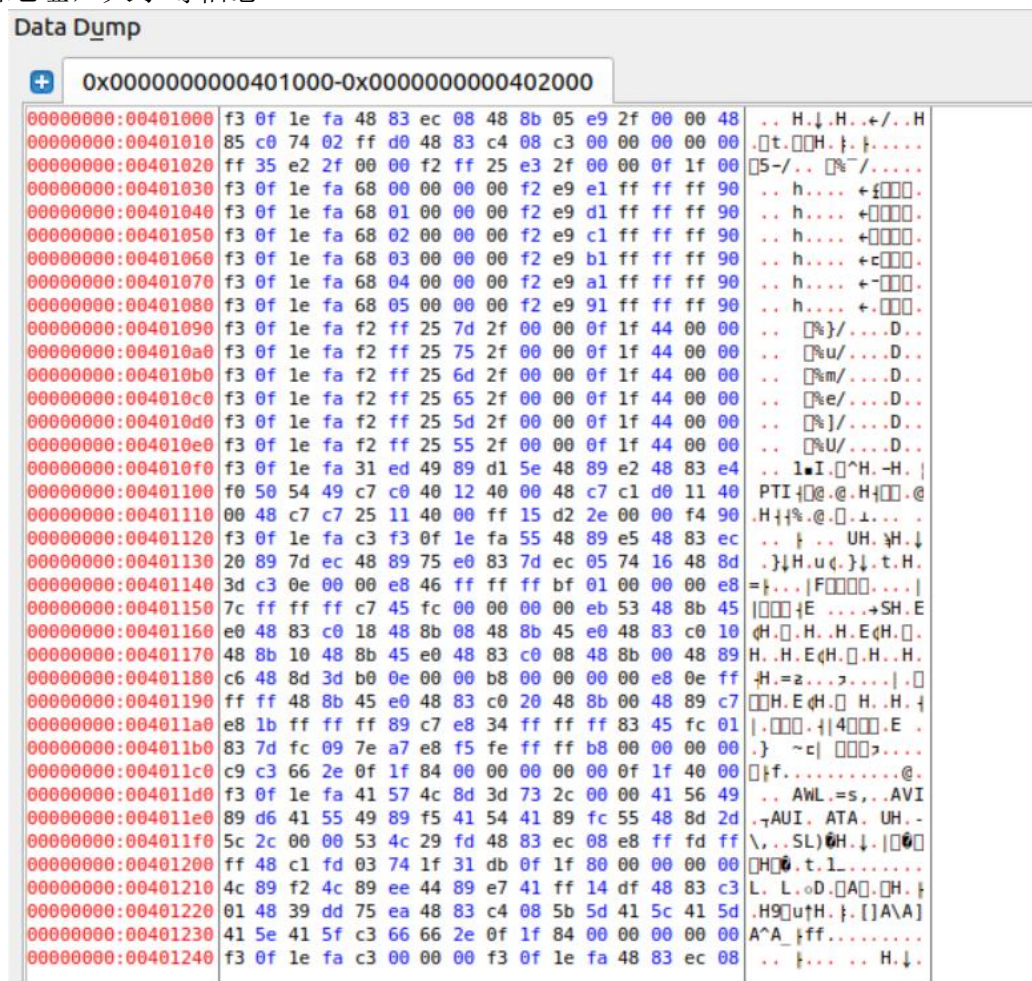


图 5.5 hello 虚拟地址空间起始位置

## 5.5 链接的重定位过程分析

```
srge@srge-VirtualBox:/media/sf_share/zuoye$ objdump -r -d hello >objhello.txt
srge@srge-VirtualBox:/media/sf_share/zuoye$ ls
hello hello.c hello.i hello.o hello.s objhello.o.txt objhello.txt
```

图 5.6 反汇编指令

```

0000000000401125 <main>:
401125: f3 0f 1e fa      endbr64
401129: 55              push %rbp
40112a: 48 89 e5        mov %rsp,%rbp
40112d: 48 83 ec 20     sub $0x20,%rsp
401131: 89 7d ec        mov %edi,-0x14(%rbp)
401134: 48 89 75 e0     mov %rsi,-0x20(%rbp)
401138: 83 7d ec 05     cmpl $0x5,-0x14(%rbp)
40113c: 74 16          je 401154 <main+0x2f>
40113e: 48 8d 3d c3 0e 00 00 lea 0xec3(%rip),%rdi # 402008 <_IO_stdin_used+0x8>
401145: e8 46 ff ff ff  callq 401090 <puts@plt>
40114a: bf 01 00 00 00  mov $0x1,%edi
40114f: e8 7c ff ff ff  callq 4010d0 <exit@plt>
401154: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)
40115b: eb 53          jmp 4011b0 <main+0x8b>
40115d: 48 8b 45 e0     mov -0x20(%rbp),%rax
401161: 48 83 c0 18     add $0x18,%rax
401165: 48 8b 08        mov (%rax),%rcx
401168: 48 8b 45 e0     mov -0x20(%rbp),%rax
40116c: 48 83 c0 10     add $0x10,%rax
401170: 48 8b 10        mov (%rax),%rdx
401173: 48 8b 45 e0     mov -0x20(%rbp),%rax
401177: 48 83 c0 08     add $0x8,%rax
40117b: 48 8b 00        mov (%rax),%rax
40117e: 48 89 c6        mov %rax,%rsi
401181: 48 8d 3d b0 0e 00 00 lea 0xeb0(%rip),%rdi # 402038 <_IO_stdin_used+0x38>
401188: b8 00 00 00 00  mov $0x0,%eax
40118d: e8 0e ff ff ff  callq 4010a0 <printf@plt>
401192: 48 8b 45 e0     mov -0x20(%rbp),%rax
401196: 48 83 c0 20     add $0x20,%rax
40119a: 48 8b 00        mov (%rax),%rax
40119d: 48 89 c7        mov %rax,%rdi
4011a0: e8 1b ff ff ff  callq 4010c0 <atoi@plt>
4011a5: 89 c7          mov %eax,%edi
4011a7: e8 34 ff ff ff  callq 4010e0 <sleep@plt>
4011ac: 83 45 fc 01     addl $0x1,-0x4(%rbp)
4011b0: 83 7d fc 09     cmpl $0x9,-0x4(%rbp)
4011b4: 7e a7          jle 40115d <main+0x38>
4011b6: e8 f5 fe ff ff  callq 4010b0 <getchar@plt>
4011bb: b8 00 00 00 00  mov $0x0,%eax
4011c0: c9             leaveq
4011c1: c3            retq
4011c2: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)
4011c9: 00 00 00       nopl
4011cc: 0f 1f 40 00    nopl 0x0(%rax)

```

图 5.7 hello 的反汇编代码

### (1) 对比 hello.o 和 hello 的重定位项目：

**Hello.o** 文件是编译器生成的目标文件，包含机器代码和数据，但其中的地址是相对的或未定义的。符号引用和定义都保留在符号表中，但尚未完全解析。含有重定位信息，指示链接器需要在哪些地方进行地址修正。

**Hello** 文件是链接器输出的文件，所有符号已经解析，所有地址已经重定位。包含最终的机器码和数据，可以直接加载到内存中运行。包含额外的节，如.init（初始化代码），.plt（过程链接表）等

总结：hello 比 hello.o 中多出.init 节和.plt 节。hello 中有确定的虚拟地址，完成了重定位，hello.o 中虚拟地址均为 0，未完成重定位。Hello 中增加了外部链接的函数，如 printf 函数。

### (2) 分析重定位的过程

重定位是链接器完成符号解析后进行的步骤，为每个符号和节分配准确的内存地址，并调整代码中的地址引用。在这个过程中：

**节重定位：**链接器确定每个合并节在最终可执行文件中的位置和大小。为每个合并后的节分配一个连续的虚拟内存地址空间。例如，合并后的.text 节可能从

某个地址开始，紧接着是合并后的.data 节。

**符号重定位：**每个输入模块的符号（如函数和全局变量）被分配一个全局地址。链接器更新每个符号的引用，使其指向正确的内存地址。

**修改代码和数据中的引用：**链接器扫描代码节和数据节中的每个重定位条目，修改指令或数据中的地址引用。例如，一个调用外部函数 `printf` 的指令在编译时可能只是一个占位符，链接器会更新这个占位符为 `printf` 的实际地址。

## 5.6 hello 的执行流程

使用 `gdb/edb` 执行 `hello`，下面是从加载 `hello` 到 `_start`，到 `call main`，以及程序终止的所有过程。

```
srge@srge-VirtualBox:/media/sf_share/zuoye$ edb --run hello
[load_plugins] Cannot load library /usr/lib/x86_64-linux-gnu/edb/libDebuggerErrorConsole.so: (/usr/lib/
x86_64-linux-gnu/edb/libDebuggerErrorConsole.so: undefined symbol: _ZN26DebuggerErrorConsolePlugin20Deb
uggerErrorConsole20compareDisassemblersEv)
Starting edb version: 1.0.0
Please Report Bugs & Requests At: https://github.com/eteran/edb-debugger/issues
Running Terminal: "/usr/bin/xterm"
Terminal Args: ("-title", "edb output", "-hold", "-e", "sh", "-c", "tty > /tmp/edb_temp_file_134715781
3_26761;trap \"\\\" INT QUIT TSTP;exec<&-; exec>&-;while ;; do sleep 3600; done")
Terminal process has TTY: "/dev/pts/1"
Loading session file
Loading plugin-data
restoreComments
comparing versions: [0] [4096]
[Analyzer] Region name: "/usr/lib/x86_64-linux-gnu/ld-2.31.so"
[Analyzer] identifying executable headers...
[Analyzer] adding entry points to the list...
[Analyzer] attempting to add 'main' to the list...
[Analyzer] attempting to add functions with symbols to the list...
[Analyzer] attempting to add marked functions to the list...
[Analyzer] attempting to collect functions with fuzzy analysis...
[Analyzer] collecting basic blocks...
[Analyzer] determining function types...
[Analyzer] complete
[Analyzer] elapsed: 139 ms
```

图 5.8 Edb 执行 hello

程序名	地址
init	0x401000
.plt	0x401020
puts@plt	0x401090
printf@plt	0x4010a0
getchar@plt	0x4010b0
atoi@plt	0x4010c0
exit@plt	0x4010d0
sleep@plt	0x4010e0
_start	0x4010f0
_dl_relocate_static_pie	0x401120
main	0x401125
_libc_csu_init	0x4011c0
_libc_csu_fini	0x401230
_fini	0x401238





---

## 第 6 章 os 将程序加载到内存

### 6.1 引导部分:以 Linux00 为例分析

#### (1) 引导程序和操作系统头写入引导盘

```
boot.s(512B) head.s ->Image ->boot a:
```

(2) 从引导盘移动引导程序到内存: 开机后硬件将 BIOS 程序放入内存, PC 指向程序执行。BIOS 自检、移动引导盘 512B(55AA 花码结尾)到指定内存区域 0x7c00, 将 PC 指向 0x7c00。

#### (3) 引导程序初始化段寄存器:

```
CS:IP 0x0:0x7c00 (jump i)——> CS:IP 0x7c00:0x05
```

#### (4) 移动剩余 head.s 到内存

- 中断向量表: BIOS 已经将中断向量表拷贝到内存 0x00
- BIOS 已经将中断程序放入内存 0x13:输入参数, 把一定扇区放入内存
- INT 0x13,调用 0x13 中断, 设置参数, 将 head.s 移动到内存
- 此时内存: 0x0: 中断向量表; 0x7c00: boot.s; 0x10000:head.s

#### (5) 中断调用过程

利用中断门描述符, 找到中断处理函数起始地址(CS 左移 2 字节, 加 IP); 高两个字节为 CS, 低两个字节为 IP(注意小端序: 地址越大, 字节位越高);

```
0x00-0x04 FE E3 00 F0 -> CS:F000 IP:E3 FE
```

把代码要执行的地址压栈, 执行结束后弹栈, 继续执行

#### (6) 把 0x10000 处的 head.s 拷贝到 0x0 位置

- 调用完中断程序后, BIOS 残留的程序没有用了, 可以随意覆盖
- 先关中断, 避免出错
- 再循环移动

#### (7) 执行 head.s

### 6.2 调试分析 Linux 0.00 引导程序

#### (1) head.s 的工作原理

head.s 的主要功能是初始化系统环境、实现多任务切换

- 初始设置: 进行段寄存器、栈初始化;
- 设置 IDT、GDT 表, 其中 IDT 表设置成所有项指向 ignore\_int 默认中断门, GDT 表设置代码段、数据段、两个任务的 LDT 和 TSS;

- 设置好 GDT 表后重新加载所有寄存器；
- 设置中断门和陷阱门，中断门实现任务的切换，陷阱门实现对系统函数 `write_char` 函数的调用；
- 通过 `push` 与 `iret` 跳转到 `task0` 执行。
- 任务切换：通过 `timer_interrupt` 程序和 TSS LDT 支持，实现在 `task0`、`task1` 之间的任务切换。

## (2) 系统初始化

- **段寄存器初始化：**设置数据段寄存器（DS）为内核数据段（0x10），为内核和任务代码提供一致的内存视图。

```
movl $0x10,%eax
mov %ax,%ds
```

- **栈初始化：**将栈指针初始化为 `init_stack`。

```
lss init_stack,%esp
```

```
init_stack:                                user stack for task0.
.long init_stack
.word 0x10
```

- **设置 IDT：**将中断描述符表 `idt` 设置成具有 256 个项，并都指向 `ignore_int` 中断门。加载中断描述符表寄存器(用 `lidt` 指令)。真正实用的中断门以后再安装。

```
call setup_idt
setup_idt:
    lea ignore_int,%edx    / 将 ignore_int 的有效地址（偏移值）存
入 %edx /
    movl $0x00080000,%eax  / 完成目标代码段选择符和偏移地址的高位部分。
    /
    movw %dx,%ax           / 完成描述符的 offset_low 字段。 /
    movw $0x8E00,%dx       / 设置为中断门（P=1, DPL=0, Type=E, D=1）
    /
    lea idt,%edi           / 将 IDT 表的起始地址加载到 %edi /
    mov $256,%ecx          / 需要初始化 IDT 的 256 项条目。 /
rp_sidt:
    movl %eax,(%edi)        / 将门描述符的低 4 字节（offset_low,
selector）写入 IDT 表当前项。 /
    movl %edx,4(%edi)       / 将门描述符的高 4 字节（attributes,
offset_high）写入 IDT 表当前项。 /
    addl $8,%edi            / 指向下一条门描述符位置（每个描述符 8 字
节）。 /
    dec %ecx
    jne rp_sidt            / 循环，初始化 IDT 项。 /

lidt lidt_opcode           / 加载 IDT 表地址到 CPU 的 IDT 寄存器 /
ret                        / 返回调用点，IDT 初始化完成。 /

lidt_opcode:
```

```
.word 2568-1
.long idt
```

```
idt: .fill 256,8,0    idt is uninitialized
```

- **设置 GDT:** 定义内存段和任务状态段 (TSS) 的布局。  
设置代码段、数据段、和代码初始定义的各段描述符。

```
SCRN_SEL  = 0x18
TSS0_SEL  = 0x20
LDT0_SEL  = 0x28
TSS1_SEL  = 0x30
LDT1_SEL  = 0x38
call setup_gdt
setup_gdt:
    lgdt lgdt_opcode
    ret
lgdt_opcode:
    .word (end_gdt-gdt)-1
    .long gdt
gdt:
    .quad 0x0000000000000000    / NULL descriptor /
    .quad 0x00c09a00000007ff    / 8Mb 代码段, 选择符 0x08 /
    .quad 0x00c09200000007ff    / 8Mb 数据段, 选择符 0x10 /
    .quad 0x00c0920b80000002    / 屏幕段, 选择符 0x18 /
    .word 0x0068, tss0, 0xe900, 0x0 / TSS0 描述符, 选择符 0x20 /
    .word 0x0040, ldt0, 0xe200, 0x0 / LDT0 描述符, 选择符 0x28 /
    .word 0x0068, tss1, 0xe900, 0x0 / TSS1 描述符, 选择符 0x30 /
    .word 0x0040, ldt1, 0xe200, 0x0 / LDT1 描述符, 选择符 0x38 /
end_gdt:
    .fill 128,4,0
```

- 设置好 gdt 后重新加载所有段寄存器

```
movl $0x10,%eax
mov %ax,%ds
mov %ax,%es
mov %ax,%fs
mov %ax,%gs
```

```
lss init_stack,%esp
```

- **8253 定时器中断设置:** 配置定时器为 100 Hz 的频率, 用于周期性触发定时器中断。

```
//setup up timer 8253 chip.
movb $0x36, %al
movl $0x43, %edx
outb %al, %dx
movl $11930, %eax    timer frequency 100 HZ
movl $0x40, %edx
outb %al, %dx
movb %ah, %al
```

```
outb %al, %dx
```

```
//setup timer & system call interrupt descriptors.
```

```
movl $0x00080000, %eax
```

```
movw $timer_interrupt, %ax
```

```
movw $0x8E00, %dx
```

```
movl $0x08, %ecx
```

The PC default timer int.

```
lea idt(,%ecx,8), %esi
```

```
movl %eax, (%esi)
```

```
movl %edx, 4(%esi)
```

```
movw $system_interrupt, %ax
```

```
movw $0xef00, %dx
```

```
movl $0x80, %ecx
```

```
lea idt(,%ecx,8), %esi
```

```
movl %eax, (%esi)
```

```
movl %edx, 4(%esi)
```

➤ 设置陷阱门：调用 write\_char 输出字符

```
/ system call handler /
```

```
.align 2
```

```
system_interrupt:
```

```
push %ds
```

保存数据段寄存器的值

```
pushl %edx
```

保存 EDX 寄存器的值

```
pushl %ecx
```

保存 ECX 寄存器的值

```
pushl %ebx
```

保存 EBX 寄存器的值

```
pushl %eax
```

保存 EAX 寄存器的值

```
movl $0x10, %edx
```

设置数据段选择符为 0x10（内核数据

段）

```
mov %dx, %ds
```

将 DS 设置为内核数据段选择符

（0x10）

```
call write_char
```

调用系统调用的处理函数，这里假设是

写字符操作

```
恢复寄存器值
```

```
popl %eax
```

恢复 EAX 寄存器

```
popl %ebx
```

恢复 EBX 寄存器

```
popl %ecx
```

恢复 ECX 寄存器

```
popl %edx
```

恢复 EDX 寄存器

```
pop %ds
```

恢复数据段寄存器

```
iret
```

返回中断，恢复现场

(3) 定时器驱动的多任务切换

➤ 任务说明：定义两个 L3 任务 task0 和 task1，它们分别通过系统调用打印字符 'A' 和 'B'，并进入空转循环。

```
task0:
```

```
movl $0x17, %eax
```



```

    movw %ax, %ds
    movb $65, %al          / print 'A' /
    int $0x80
    movl $0xffff, %ecx 空转一会
1:  loop 1b
    jmp task0

task1:
    movl $0x17, %eax
    movw %ax, %ds
    movb $66, %al          / print 'B' /
    int $0x80
    movl $0xffff, %ecx
1:  loop 1b
    jmp task1

```

➤ 设置局部描述符表和任务状态段:

```

ldt0: .quad 0x0000000000000000
      .quad 0x00c0fa00000003ff  0x0f, base = 0x00000
      .quad 0x00c0f200000003ff  0x17

tss0: .long 0          / back link /
      .long krn_stk0, 0x10  / esp0, ss0 /
      .long 0, 0, 0, 0, 0  / esp1, ss1, esp2, ss2, cr3 /
      .long 0, 0, 0, 0, 0  / eip, eflags, eax, ecx, edx /
      .long 0, 0, 0, 0, 0  / ebx esp, ebp, esi, edi /
      .long 0, 0, 0, 0, 0, 0  / es, cs, ss, ds, fs, gs /
      .long LDT0_SEL, 0x8000000 / ldt, trace bitmap /

      .fill 128,4,0
krn_stk0:

ldt1: .quad 0x0000000000000000
      .quad 0x00c0fa00000003ff  0x0f, base = 0x00000
      .quad 0x00c0f200000003ff  0x17

tss1: .long 0          / back link /
      .long krn_stk1, 0x10  / esp0, ss0 /
      .long 0, 0, 0, 0, 0  / esp1, ss1, esp2, ss2, cr3 /
      .long task1, 0x200  / eip, eflags /
      .long 0, 0, 0, 0  / eax, ecx, edx, ebx /
      .long usr_stk1, 0, 0, 0  / esp, ebp, esi, edi /
      .long 0x17,0x0f,0x17,0x17,0x17,0x17 / es, cs, ss, ds, fs, gs /
      .long LDT1_SEL, 0x8000000 / ldt, trace bitmap /

      .fill 128,4,0
krn_stk1:

```

- 移动到任务 0: 设置必要的初始化标志, 如嵌套标志、ltr 指向任务 0, current 变量设为 0, 打开中断。通过 iret 指令操作堆栈, 手动降低特权级, 修改 CS 从 08 到 0f

```
pushfl  复位标志寄存器 EFLAGS 中的嵌套任务标志
andl $0xffffbfff, (%esp)
popfl   把那一位变为 0
```

```
    tr ldt 只需要初始化一次, 以后 CPU 自动处理
movl $TSS0_SEL, %eax  TSS 段选择符
ltr %ax  指向任务 0
movl $LDT0_SEL, %eax
lldt %ax
```

```
movl $0, current
```

```
sti  打开中断
```

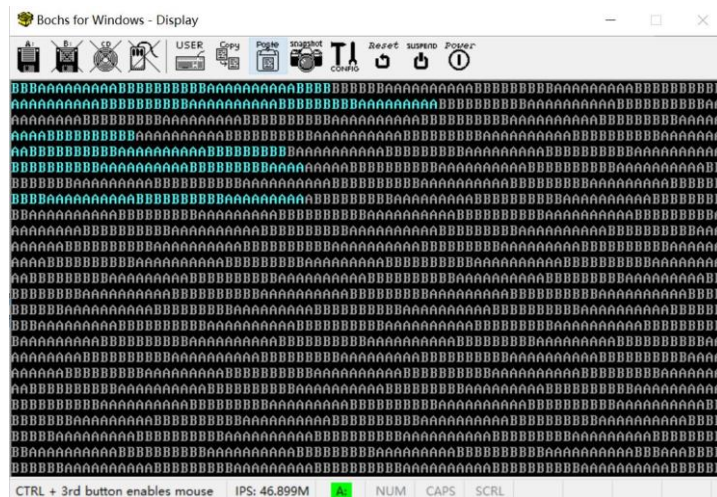
00 进入 11 特权级, 只有调用返回指令可以降低特权级, 执行出栈操作, 手动压栈出栈

```
pushl $0x17  任务 0 段选择符
pushl $init_stack  堆栈指针入栈
pushfl
pushl $0x0f  1111, “11”, 查 ldt, 同时表示描述符的特权级
    ldt 此时指向了任务 0 的描述符, 指向了任务 0 的 ldt
    修改 CS 从 08 到 0f, 基地址不变, EIP 不变, 但特权级改变
    内核模式: 08:task0
    用户模式: 0f:task0
pushl $task0  代码指针入栈
iret
```

通过中断门进行任务切换: 使用全局变量 current 记录当前任务号 (0 或 1)。定时器中断处理程序根据 current 的值切换到另一个任务。任务切换通过 ljmp 指令跳转到对应任务的 TSS 段选择符 (TSS0\_SEL 或 TSS1\_SEL)。

```
timer_interrupt:
    push %ds
    pushl %eax
    movl $0x10, %eax
    mov %ax, %ds
    movb $0x20, %al
    outb %al, $0x20
    movl $1, %eax
    cmpl %eax, current
    je 1f
    movl %eax, current
    ljmp $TSS1_SEL, $0
    jmp 2f
1:  movl $0, current
    ljmp $TSS0_SEL, $0
```

- ```
2: popl %eax
   pop %ds
   iret
```
- 任务执行展示:



## 6.2 head.s 的内存分布状况

- 在 head.s 将内存分布好后，在切换到任务 0 进行前，查看 GDT 表如下

| Bochs Enhanced Debugger   |                 |          |               |             |                                         |                      |              |            |                      |
|---------------------------|-----------------|----------|---------------|-------------|-----------------------------------------|----------------------|--------------|------------|----------------------|
| Command View Options Help |                 |          |               |             |                                         |                      |              |            |                      |
| Continue [c]              |                 | Step [s] | Step N [s ##] |             | Refresh                                 |                      | Break [^C]   |            |                      |
| Reg N...                  | Hex Value       | Decimal  | Address       | Bytes       | Mnemonic                                | Index                | Base Address | Size       | DPL Info             |
| eax                       | 00080166        | 524646   | 00000079      | (1) 9C      | pushf                                   | 00 (Selector 0x0000) | 0x0          | 0x0        | 0 Unused             |
| ebx                       | 00000000        | 0        | 0000007a      | (7) 8124... | and dword ptr ss[esp], 0xffffbf         | 01 (Selector 0x0008) | 0x0          | 0x7FFFFFFF | 0 32-bit code        |
| ecx                       | 00000080        | 128      | 00000081      | (1) 9D      | popf                                    | 02 (Selector 0x0010) | 0x0          | 0x7FFFFFFF | 0 32-bit data        |
| edx                       | 0000ef00        | 61184    | 00000082      | (5) B820... | mov eax, 0x00000020                     | 03 (Selector 0x0018) | 0xB8000      | 0x2FFF     | 0 32-bit data        |
| esi                       | 00000598        | 1432     | 00000087      | (3) 0F00... | ltr ax                                  | 04 (Selector 0x0020) | 0xBF8        | 0x68       | 3 Available 32bit TS |
| edi                       | 00000998        | 2456     | 0000008a      | (5) B828... | mov eax, 0x00000028                     | 05 (Selector 0x0028) | 0xBE0        | 0x40       | 3 LDT                |
| ebp                       | 00000000        | 0        | 0000008f      | (3) 0F00... | ltdt ax                                 | 06 (Selector 0x0030) | 0xE78        | 0x68       | 3 Available 32bit TS |
| esp                       | 00000bd8        | 3032     | 00000092      | (10) C70... | mov dword ptr ds:0x0000017d, 0x00000000 | 07 (Selector 0x0038) | 0xE60        | 0x40       | 3 LDT                |
| eip                       | 00000079        | 121      | 0000009c      | (1) FB      | sti                                     |                      |              |            |                      |
| eflags                    | 00000046        |          | 0000009d      | (2) 6A17... | push 0x00000017                         |                      |              |            |                      |
| cs                        | 0008            |          | 0000009f      | (5) 68D8... | push 0x00000bd8                         |                      |              |            |                      |
| ds                        | 0010            |          | 000000a4      | (1) 9C      | pushf                                   |                      |              |            |                      |
| es                        | 0010            |          | 000000a5      | (2) 6A0F... | push 0x0000000f                         |                      |              |            |                      |
| ss                        | 0010            |          | 000000a7      | (5) 68E0... | push 0x000001e0                         |                      |              |            |                      |
| fs                        | 0010            |          | 000000ac      | (1) CF      | iret                                    |                      |              |            |                      |
| gs                        | 0010            |          | 000000ad      | (7) 0F01... | lgdt ds:0x0000018c                      |                      |              |            |                      |
| gdt                       | 00000998 ( ...) |          | 000000b4      | (1) C3      | ret                                     |                      |              |            |                      |
| idt                       | 00000198 ( ...) |          | 000000b5      | (6) 8D15... | lea edx, ds:0x00000114                  |                      |              |            |                      |
| ldtr                      | 0000            |          | 000000bb      | (5) B800... | mov eax, 0x00080000                     |                      |              |            |                      |
| tr                        | 0000            |          | 000000c0      | (3) 6689... | mov ax, dx                              |                      |              |            |                      |
| cr0                       | 60000011        |          | 000000c3      | (4) 66BA... | mov dx, 0x8e00                          |                      |              |            |                      |

通过 GDT 表可知各个段的起始地址和限长，对于 IDT、GDT 等有寄存器保存起始地址的段，可通过寄存器值计算得出起始位置和终止位置，例如 `gdt` 可知 GDT 初始位置为 `0x998`，经过计算得结束位置 `0x9d7`。

对于其他段，首先可以通过 `bochs` 找到代码段各部分的地址，即可以计算出相邻的数据段、栈段的起始地址。通过加上数据段、栈段的偏移量，即可得出各部分内存分布。

- 代码段

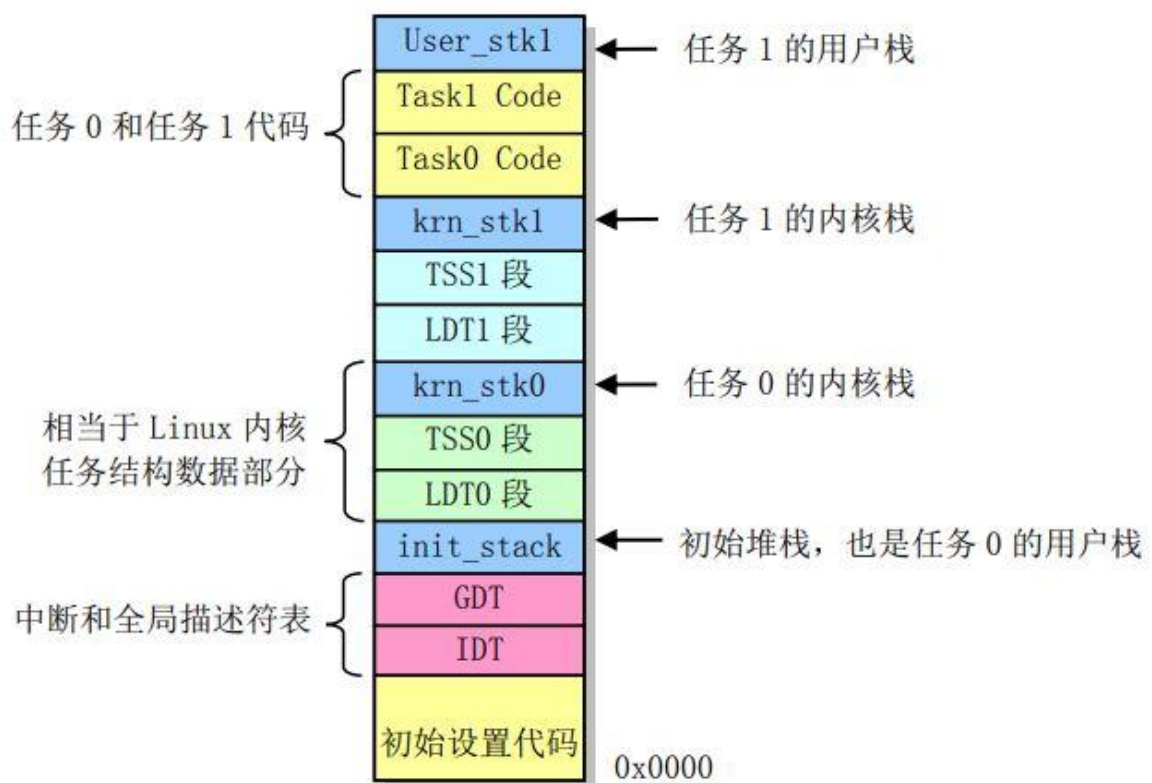
可通过 `bochs` 找到第一个指令的地址，和最后一个指令的地址

`startup_32:0x0-0xac`

| L.Address | Bytes              | Mnemonic               |
|-----------|--------------------|------------------------|
| 00000000  | (5) B810000000     | mov eax, 0x00000010    |
| 00000005  | (2) 8ED8           | mov ds, ax             |
| 00000007  | (7) 0FB225D80B0000 | lss esp, ds:0x00000bd8 |
| 0000000a  | (5) 000000000000   | push 0x00000000        |
| 000000ac  | (1) CF             | iret                   |
| 000000ad  | (7) 0F01158C010000 | lgdt ds:0x00000018c    |
| 000000ad  | (7) 0F01158C010000 | lgdt ds:0x00000018c    |
| 000000b4  | (1) C3             | ret                    |

setup\_gdt:0x0d-0xb4

- 数据段：根据代码地址和给定的偏移计算得出
- 栈段：根据代码地址和偏移计算：栈段的偏移量固定，均为 512 字节  
得到 Linux 内存各部分分布如下



| 段类型 | 名称               | 起始地址   | 终止地址   |
|-----|------------------|--------|--------|
| 代码段 | startup_32       | 0x00   | 0xAC   |
| 代码段 | setup_gdt        | 0xAD   | 0xB4   |
| 代码段 | setup_idt        | 0xB5   | 0xCD   |
| 代码段 | rp_sidt          | 0xD2   | 0xE4   |
| 代码段 | write_char       | 0xE5   | 0x113  |
| 代码段 | ignore_int       | 0x114  | 0x129  |
| 代码段 | timer_interrupt  | 0x12A  | 0x165  |
| 代码段 | system_interrupt | 0x166  | 0x17C  |
| 数据段 | current          | 0x17D  | 0x180  |
| 数据段 | scr_loc          | 0x181  | 0x184  |
| 数据段 | lidt_opcode      | 0x186  | 0x18B  |
| 数据段 | lgdt_opcode      | 0x18C  | 0x191  |
| 数据段 | idt              | 0x198  | 0x997  |
| 数据段 | gdt              | 0x998  | 0x9D7  |
| 栈段  | init_stack       | 0x9D8  | 0xBD8  |
| 数据段 | ldt0             | 0xBE0  | 0xBF7  |
| 数据段 | tss0             | 0xBF8  | 0xC5F  |
| 栈段  | krn_stk0         | 0xC60  | 0xE60  |
| 数据段 | ldt1             | 0xC60  | 0xE77  |
| 数据段 | tss1             | 0xE78  | 0xEDF  |
| 栈段  | krn_stk1         | 0xEE0  | 0x10E0 |
| 代码段 | task0            | 0x10E0 | 0x10F3 |
| 代码段 | task1            | 0x10F4 | 0x1107 |
| 栈段  | usr_stk1         | 0x1108 | 0x1308 |

---

## 第 7 章 hello 进程管理

### 7.1 进程的概念与作用

#### 7.1.1 概念

进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文（context）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。

#### 7.1.2 作用

一个独立的逻辑控制流，它提供一个假象，好像我们的程序独占地使用处理器。

一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统。

**多任务处理：**操作系统通过进程管理，可以在多个进程之间切换执行，实现多任务处理。

**资源分配：**操作系统为每个进程分配 CPU 时间、内存空间等资源，确保进程能够正常运行。

**保护机制：**进程之间的隔离可以防止一个进程影响其他进程的运行，提高系统的稳定性和安全性。

**并发执行：**进程可以并发执行，提高计算机的利用率和效率。

### 7.2 Shell-bash 的作用与处理流程

Shell（壳）是操作系统的用户界面，它允许用户与操作系统交互。Bash 是 Linux 系统中最常用的 Shell 之一，它提供了命令行界面，允许用户输入命令来执行程序、管理文件和执行各种系统操作。Bash 是 Linux 系统中最常用的 Shell 之一，它提供了命令行界面，允许用户输入命令来执行程序、管理文件和执行各种系统操作。Bash 不仅是一个命令解释器，还是一个脚本语言，允许用户编写脚本来自自动化重复性任务。Bash 的处理流程包括启动、输入命令、解析命令、执行命令、输出结果和等待下一个命令。

#### 7.2.1 作用

shell 应用程序提供了一个界面，用户通过访问这个界面访问操作系统内核的服务。具有文件管理、进程管理、用户管理、脚本编程等功能。

（1）命令解释器：Bash 解释用户输入的命令，并将其转换为操作系统可以理解的形式。

（2）脚本语言：Bash 不仅是一个命令解释器，还是一个脚本语言，允许用户编写脚本来自自动化重复性任务。

（3）环境配置：Bash 可以配置用户的环境变量，如 PATH、HOME 等，影响程序的执行和用户的工作环境。

---

(4) 程序执行：Bash 可以执行外部程序和脚本，是用户与计算机交互的主要方式。

### 7.2.2 处理流程

启动：当用户登录到 Linux 系统时，Bash 会启动并显示命令提示符。

输入命令：用户在命令提示符下输入命令。

解析命令：Bash 解析用户输入的命令，识别命令和参数。

执行命令：Bash 执行命令，可能涉及调用外部程序或执行内部命令。

输出结果：命令执行的结果会显示在屏幕上。

等待下一个命令：Bash 等待用户输入下一个命令。

## 7.3 Hello 的 fork 进程创建过程

父进程调用 fork()：父进程执行 fork() 系统调用，请求内核创建一个新的子进程。

内核分配资源：内核为子进程分配一个新的进程 ID，并复制父进程的内存空间、文件描述符表、环境变量、信号处理函数等资源。

返回值：fork()调用返回两次，一次在父进程，一次在子进程。在父进程，fork() 返回子进程的 PID。在子进程，fork()返回 0。如果 fork()失败，则返回-1。

执行流程分离：父进程和子进程从 fork()调用之后的下一条指令开始并行执行。父进程和子进程可以执行不同的代码路径，但它们都从 fork()调用的下一条指令开始执行。

进程独立执行：父进程和子进程各自执行自己的代码，它们是独立的实体，拥有自己的执行序列。

资源管理：子进程在创建时继承了父进程的大部分资源，但它们拥有独立的内存空间，因此对父进程的内存修改不会影响到子进程，反之亦然。

退出和回收：子进程可以正常退出或被父进程终止。当子进程退出时，内核通知父进程，父进程可以通过 wait()或 waitpid() 系统调用来回收子进程的资源。

错误处理：如果 fork() 调用失败，通常是因为系统资源不足，如进程数达到上限或内存不足。

## 7.4 Hello 的 execve 过程

### 7.4.1 execve 函数介绍

execve 是一个系统调用，用于在当前进程中替换掉当前执行的程序，加载并开始执行一个新的程序。

execve 的基本原型如下：

```
int execve(const char filename, char const argv[], char const envp[]);
```



---

**filename:** 要执行的程序文件的路径。

**argv:** 指向参数数组的指针，数组中的第一个元素是程序名，后面跟着传递给程序的参数。

**envp:** 指向环境变量数组的指针，数组中的每个元素都是一个以空字符终止的字符串。

#### 7.4.2execve 执行过程

**指定程序文件:** 调用 `execve` 时，需要指定要执行的程序文件的路径。

**参数传递:** `execve` 允许传递参数给新程序。这些参数包括程序名、参数列表和环境变量。

**环境变量:** 调用 `execve` 时，可以指定一个环境变量数组，新程序将使用这个环境变量数组。

**执行新程序:** `execve` 系统调用执行后，当前进程的内存空间、代码段、数据段等将被新程序的映像替换。这意味着当前进程的执行状态被完全丢弃，新的程序开始执行。

**返回值:** 如果 `execve` 成功执行，它不会返回到调用者，因为当前进程已经被新程序替换了。如果 `execve` 失败，它将返回到调用者，并且返回-1。

**错误处理:** `execve` 可能会因为多种原因失败，如指定的程序文件不存在、没有执行权限、路径名太长、参数列表过长等。

### 7.5 Hello 的进程执行

**(1) 进程调度:** 操作系统负责管理计算机中的多个进程，确保它们能够公平地共享 CPU 资源。当一个进程需要执行时，操作系统会根据一定的调度算法选择一个进程来运行。

**(2) 上下文切换:** 当操作系统决定从一个进程切换到另一个进程时，它需要保存当前进程的状态（如寄存器值、程序计数器等），并加载新进程的状态。这个过程称为上下文切换。

**(3) 用户态和核心态:** 操作系统通常运行在两种模式：用户态和核心态。用户态是普通应用程序的运行模式，而核心态是操作系统代码的运行模式。当应用程序需要执行某些特权操作时（如访问硬件、管理内存等），它必须通过系统调用请求操作系统服务，这时操作系统会将应用程序从用户态切换到核心态。

**(4) 系统调用:** 系统调用是应用程序请求操作系统服务的接口。例如，`sleep` 和 `exit` 都是系统调用。当应用程序执行这些调用时，它会陷入内核，内核处理这些请求，然后返回结果给应用程序。



(5) **时间片**：操作系统通常会将 CPU 时间分割成小的时间片，每个进程在一个时间片内运行。如果一个进程在一个时间片内没有完成，它会被挂起，等待下一次调度。

(6) **进程切换**：操作系统通过上下文切换来在进程之间切换，使得每个进程都有机会运行。这样，即使多个进程同时运行，它们也能轮流使用 CPU。

(7) **异常控制流**：当发生异常（如系统调用、中断等）时，CPU 会从用户态切换到核心态，执行异常处理程序。处理完成后，CPU 会返回到用户态，继续执行用户程序。

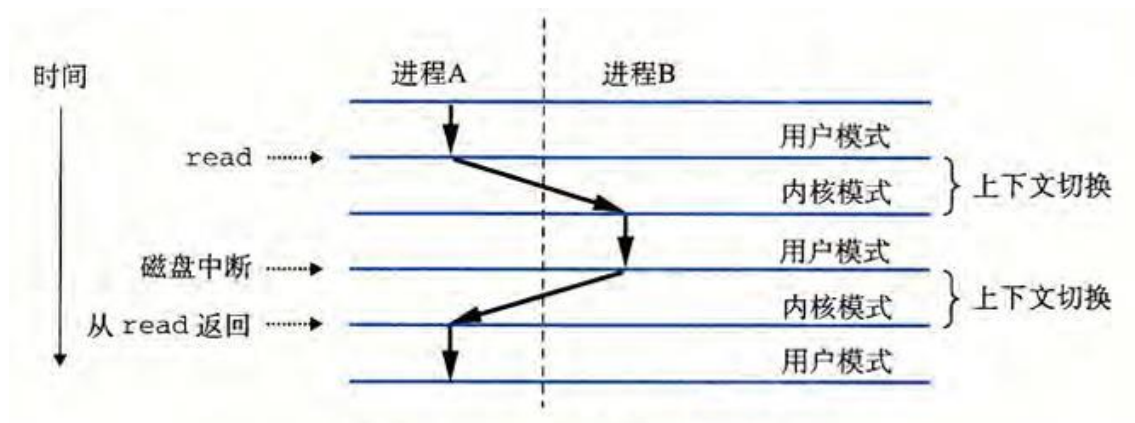


图 6.1 进程执行的上下文切换

## 7.6 Hello 的中断与系统调用

hello.c 程序调用系统调用的过程可以分为以下几个详细的阶段，包括用户态与内核态之间的交互：

### (1) 程序启动与用户态代码执行

当运行 hello.c 时，操作系统将可执行文件加载到内存中并初始化运行环境。程序从 main 函数开始执行，其中调用 printf 函数输出字符串：

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

- printf 是 C 标准库中的一个函数，用于格式化输出数据。
- 调用 printf 会将数据写入标准输出缓冲区，并最终通过系统调用将数据传递给操作系统。

### (2) printf 内部实现

#### ➤ 字符串格式化

printf 接受输入字符串 "Hello, World!\n"，并进行格式化处理。如果有占位符（如 %d），printf 会替换这些占位符为实际参数值。

---

### ➤ 缓冲区管理

`printf` 会将格式化后的字符串存入标准输出缓冲区，以减少直接调用系统调用的次数，从而提高性能。

默认情况下，标准输出（`stdout`）是行缓冲模式，遇到换行符 `\n` 或缓冲区满时才会刷新。

### ➤ 系统调用准备

当缓冲区刷新时，`printf` 内部会调用 `write` 函数，这是一个底层的 POSIX 标准 I/O 函数。

## (3) 用户态到内核态的系统调用

### ➤ 触发 `write` 系统调用

`write` 是一个系统调用，用于将数据写入文件或设备。在用户态，`write` 函数原型如下：

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

参数意义：

- `fd` 是文件描述符，标准输出为 1。
- `buf` 是要写入的数据缓冲区地址。
- `count` 是缓冲区中数据的字节数。

`printf` 将这些参数传递给 `write`，请求将数据写入标准输出（屏幕）。

### ➤ 系统调用号传递

为了执行系统调用，程序需要将系统调用号和参数传递给内核：系统调用号标识具体的系统调用，例如 `write` 的系统调用号为 1（Linux x86\_64）。

参数通过寄存器传递，例如：系统调用号存入 `rax` 寄存器。文件描述符、缓冲区地址、字节数分别存入 `rdi`、`rsi`、`rdx` 寄存器。

### ➤ 执行 `syscall` 指令

- 用户态通过 `syscall` 指令触发中断，进入内核态。
- CPU 切换到内核模式，跳转到内核中负责处理系统调用的入口。

## (4) 内核态处理：进入内核态后，操作系统执行以下操作：

### ➤ 验证参数

检查文件描述符 `fd` 是否有效，确保缓冲区地址 `buf` 和字节数 `count` 合法。

### ➤ 资源管理

确定文件描述符对应的资源（例如标准输出设备）。

### ➤ 数据传递

将用户缓冲区中的数据拷贝到内核缓冲区，以确保数据不会因用户态程序修改而出错。

### ➤ 设备驱动交互

调用对应的设备驱动程序，将数据从内核缓冲区写入终端设备（屏幕）。

### ➤ 返回用户态

返回写入的字节数，并将结果存入 `rax` 寄存器中。

## (5) 用户态返回与后续操作

### ➤ 返回到 `printf`

系统调用完成后，内核将控制权交回用户态，`printf` 获取返回值（字节数）以检查写入是否成功。

### ➤ 完成程序逻辑

如果系统调用成功，printf 继续执行剩余逻辑；否则，可能触发错误处理代码。

### ➤ 程序退出

在 main 函数执行结束后，程序通过 exit 系统调用通知操作系统释放资源并终止进程。

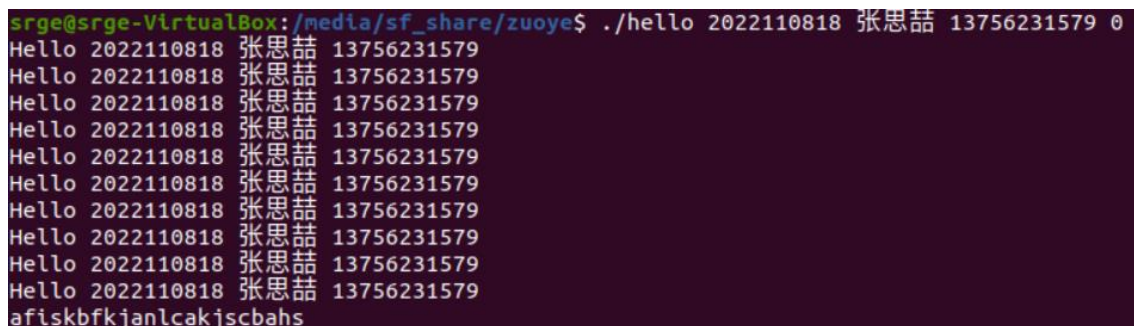
### (6) 系统调用过程总结

- 用户态准备参数：用户态代码调用 write 函数，准备好系统调用号和参数。
  - 触发系统调用：执行 syscall 指令进入内核态，操作系统根据系统调用号定位对应的处理函数（如 sys\_write）。
  - 内核处理逻辑：验证参数并完成实际的写操作，将数据写入目标设备。
  - 返回用户态：系统调用完成后，返回用户态，继续执行程序的后续逻辑。
- 通过以上过程，hello.c 程序成功将 "Hello, World!\n" 输出到屏幕。

## 7.7 Hello 的异常与信号处理

在程序运行过程中，通过按键盘可以运行各种命令，下面是命令运行的结果。

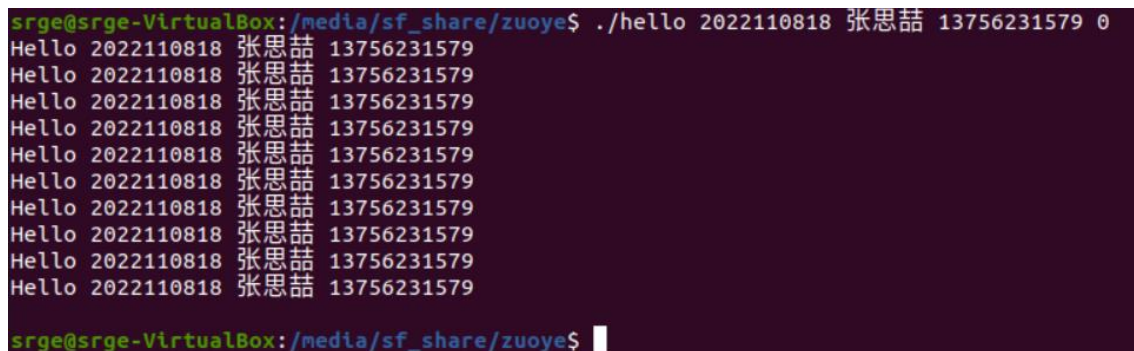
### 乱按：对进程没有影响



```
srge@srge-VirtualBox: /media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
afiskbfkjanlcakjscbahs
```

图 6.2 乱按

### 回车：程序停止运行



```
srge@srge-VirtualBox: /media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
srge@srge-VirtualBox: /media/sf_share/zuoye$
```

图 6.3 回车

Ctrl-C：前台作业终止 内核发送一个 SIGINT 信号到前台进程组的每个进程

```

srge@srge-VirtualBox:/media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^C
srge@srge-VirtualBox:/media/sf_share/zuoye$

```

图 6.4 ctrlc

**Ctrl-Z:** 前台作业停止内核会发送 SIGSTP 默认挂起前台 hello 作业

```

srge@srge-VirtualBox:/media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^Z
[1]+ 已停止                  ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox:/media/sf_share/zuoye$

```

图 6.5 ctrlz

**Ctrl-Z+ps:** 内核会发送 SIGSTP 默认挂起前台 hello 作业，但 hello 进程并没有回收，而是运行在后台下，通过 ps 指令可以对其进行查看。

```

srge@srge-VirtualBox:/media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^Z
[2]+ 已停止                  ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox:/media/sf_share/zuoye$ ps
  PID TTY          TIME CMD
  21560 pts/0        00:00:00 bash
  25572 pts/0        00:00:00 hello
  25579 pts/0        00:00:00 hello
  25580 pts/0        00:00:00 ps
srge@srge-VirtualBox:/media/sf_share/zuoye$

```

图 6.6 ctrlz ps

**Ctrl-Z+jobs:**



```
srge@srge-VirtualBox:/media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^Z
[3]+ 已停止                  ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox:/media/sf_share/zuoye$ jobs
[1] 已停止                  ./hello 2022110818 张思喆 13756231579 0
[2]- 已停止                  ./hello 2022110818 张思喆 13756231579 0
[3]+ 已停止                  ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox:/media/sf_share/zuoye$
```

图 6.7 ctrlz jobs

## Ctrl-Z+pstree

```
srge@srge-VirtualBox: /media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^Z
[4]+ 已停止                  ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox: /media/sf_share/zuoye$ pstree
systemd--ModemManager--2*[{ModemManager}]
--NetworkManager--2*[{NetworkManager}]
--3*[VBoxClient--VBoxClient--3*[{VBoxClient}]]
--VBoxClient--VBoxClient--4*[{VBoxClient}]
--VBoxDRMClient--4*[{VBoxDRMClient}]
--VBoxService--8*[{VBoxService}]
--accounts-daemon--2*[{accounts-daemon}]
--acpid
--avahi-daemon--avahi-daemon
--colord--2*[{colord}]
--cron
--cups-browsed--2*[{cups-browsed}]
--cupsd
--dbus-daemon
--gdm3--gdm-session-wor--gdm-x-session--Xorg--{Xorg}
--gnome-session-b--ssh-agent
--2*[{gnome-session-b}]
--2*[{gdm-x-session}]
--2*[{gdm3}]
--2*[{gdm-session-wor}]
--gnome-keyring-d--3*[{gnome-keyring-d}]
--2*[{kerneloops}]
--networkd-dispat
--polkitd--2*[{polkitd}]
--rsyslogd--3*[{rsyslogd}]
--rtkit-daemon--2*[{rtkit-daemon}]
--snapd--8*[{snapd}]
--switcheroo-cont--2*[{switcheroo-cont}]
--systemd--(sd-pam)
--at-spi-bus-laun--dbus-daemon
--3*[{at-spi-bus-laun}]
--at-spi2-registr--2*[{at-spi2-registr}]
--dbus-daemon
--dconf-service--2*[{dconf-service}]
--evolution-addre--5*[{evolution-addre}]
--evolution-calen--8*[{evolution-calen}]
--evolution-sourc--3*[{evolution-sourc}]
--gjs--4*[{gjs}]
--gnome-session-b--evolution-alarm--5*[{evolution-alarm}]
--gsd-disk-utilit--2*[{gsd-disk-utilit}]
--update-notifier--4*[{update-notifier}]
--3*[{gnome-session-b}]
--gnome-session-c--{gnome-session-c}
```



```

gnome-session-c {gnome-session-c}
gnome-shell-ibus-daemon-ibus-engine-lib-3*[{ibus-engine-lib}]
gnome-shell-ibus-daemon-ibus-engine-sim-2*[{ibus-engine-sim}]
gnome-shell-ibus-daemon-ibus-extension-3*[{ibus-extension-}]
gnome-shell-ibus-daemon-ibus-memconf-2*[{ibus-memconf}]
gnome-shell-ibus-daemon-2*[{ibus-daemon}]
gnome-shell-7*[{gnome-shell}]
gnome-shell-cal-5*[{gnome-shell-cal}]
gnome-terminal-bash-4*[{hello}]
gnome-terminal-bash-pstree
gnome-terminal-4*[{gnome-terminal-}]
goa-daemon-3*[{goa-daemon}]
goa-identity-se-2*[{goa-identity-se}]
gsd-a11y-settin-3*[{gsd-a11y-settin}]
gsd-color-3*[{gsd-color}]
gsd-datetime-3*[{gsd-datetime}]
gsd-housekeepin-3*[{gsd-housekeepin}]
gsd-keyboard-3*[{gsd-keyboard}]
gsd-media-keys-3*[{gsd-media-keys}]
gsd-power-3*[{gsd-power}]
gsd-print-notif-2*[{gsd-print-notif}]
gsd-printer-2*[{gsd-printer}]
gsd-rfkill-2*[{gsd-rfkill}]
gsd-screensaver-2*[{gsd-screensaver}]
gsd-sharing-3*[{gsd-sharing}]
gsd-smartcard-4*[{gsd-smartcard}]
gsd-sound-3*[{gsd-sound}]
gsd-usb-protect-3*[{gsd-usb-protect}]
gsd-wacom-2*[{gsd-wacom}]
gsd-wwan-3*[{gsd-wwan}]
gsd-xsettings-3*[{gsd-xsettings}]
gvfs-afc-volume-3*[{gvfs-afc-volume}]
gvfs-goa-volume-2*[{gvfs-goa-volume}]
gvfs-gphoto2-vo-2*[{gvfs-gphoto2-vo}]
gvfs-mtp-volume-2*[{gvfs-mtp-volume}]
gvfs-udisks2-vo-3*[{gvfs-udisks2-vo}]
gvfsd-gvfsd-trash-2*[{gvfsd-trash}]
gvfsd-gvfsd-2*[{gvfsd}]
gvfsd-fuse-5*[{gvfsd-fuse}]
gvfsd-metadata-2*[{gvfsd-metadata}]
ibus-portal-2*[{ibus-portal}]
ibus-x11-2*[{ibus-x11}]
nautilus-5*[{nautilus}]
pulseaudio-3*[{pulseaudio}]
snap-store-4*[{snap-store}]
tracker-miner-f-4*[{tracker-miner-f}]
xdg-desktop-por-4*[{xdg-desktop-por}]
xdg-desktop-por-3*[{xdg-desktop-por}]
xdg-document-po-5*[{xdg-document-po}]
xdg-permission-2*[{xdg-permission-}]
systemd-journal
systemd-logind
systemd-resolve
systemd-timesyn-{systemd-timesyn}
systemd-udev
udisksd-4*[{udisksd}]
unattended-upgr-{unattended-upgr}
upowerd-2*[{upowerd}]
whoopsie-2*[{whoopsie}]
wpa_supplicant
srge@srge-VirtualBox:/media/sf_share/zuoye$

```

图 6.8 ctrlz pstree

## Ctrl-Z+fg: 进程继续运行

```
srge@srge-VirtualBox: /media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^Z
[5]+ 已停止 ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox: /media/sf_share/zuoye$ fg
./hello 2022110818 张思喆 13756231579 0
```

图 6.9 ctrlz fg

**Ctrl-Z+kill: 通过 kill -9 命令，杀死 hello 程序**内核会发送 SIGKILL 信号给指定的 pid (hello 程序)，杀死 hello 程序

```
srge@srge-VirtualBox: /media/sf_share/zuoye$ ./hello 2022110818 张思喆 13756231579 0
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
Hello 2022110818 张思喆 13756231579
^Z
[6]+ 已停止 ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox: /media/sf_share/zuoye$ ps
  PID TTY          TIME CMD
 21560 pts/0    00:00:00 bash
 25572 pts/0    00:00:00 hello
 25579 pts/0    00:00:00 hello
 25581 pts/0    00:00:00 hello
 25582 pts/0    00:00:00 hello
 25606 pts/0    00:00:00 hello
 25607 pts/0    00:00:00 hello
 25608 pts/0    00:00:00 ps
srge@srge-VirtualBox: /media/sf_share/zuoye$ kill -9 25572
srge@srge-VirtualBox: /media/sf_share/zuoye$ ps
  PID TTY          TIME CMD
 21560 pts/0    00:00:00 bash
 25579 pts/0    00:00:00 hello
 25581 pts/0    00:00:00 hello
 25582 pts/0    00:00:00 hello
 25606 pts/0    00:00:00 hello
 25607 pts/0    00:00:00 hello
 25609 pts/0    00:00:00 ps
[1] 已杀死 ./hello 2022110818 张思喆 13756231579 0
srge@srge-VirtualBox: /media/sf_share/zuoye$
```

图 6.10 ctrlz kill

## 7.8 本章小结

本章主要介绍了进程的概念与作用、Shell-bash 的作用与处理流程、Hello 的 fork 进程创建过程、Hello 的 execve 过程、Hello 的进程执行、Hello 的异常与信号处理。

## 第 8 章 多任务与进程切换

### 8.1 前期调试 Linux00 步骤

#### ➤ 进入 head.s

|          |                    |                      |
|----------|--------------------|----------------------|
| 000000a7 | (5) 68E0100000     | push 0x000010e0      |
| 000000ac | (1) CF             | iret                 |
| 000000ad | (7) 0F01158C010000 | laddt ds:0x00000018c |

#### ➤ iret 进入 task0

| L.Address | Bytes              | Mnemonic                |
|-----------|--------------------|-------------------------|
| 00000000  | (5) B810000000     | mov eax, 0x00000010     |
| 00000005  | (2) 8ED8           | mov ds, ax              |
| 00000007  | (7) 0FB225D80B0000 | lss esp, ds:0x000000bd8 |
| 0000000a  | (5) 58A2000000     | call 1462 (0x000000b5)  |

#### ➤ task0 中找到系统中断的入口 int 0x80

Bochs Enhanced Debugger

Command View Options Help

Continue [c]

| Reg N... | Hex Value       | Decimal | L.Address | Bytes          | Mnemonic                  | L.Address | Value    | (dec.)   | Break [^C] |
|----------|-----------------|---------|-----------|----------------|---------------------------|-----------|----------|----------|------------|
| eax      | 00000041        | 65      | 00000166  | (1) 1E         | push ds                   | 00000E4C  | 000010eb | 4331     |            |
| ebx      | 00000000        | 0       | 00000167  | (1) 52         | push edx                  | 00000E50  | 0000000f | 15       |            |
| ecx      | 00000080        | 128     | 00000168  | (1) 51         | push ecx                  | 00000E54  | 00000246 | 582      |            |
| edx      | 0000ef00        | 61184   | 00000169  | (1) 53         | push ebx                  | 00000E58  | 000000b8 | 3032     |            |
| esi      | 00000598        | 1432    | 0000016a  | (1) 50         | push eax                  | 00000E5C  | 00000017 | 23       |            |
| edi      | 00000998        | 2456    | 0000016b  | (5) BA10000000 | mov edx, 0x00000010       | 00000E60  | 00000000 | 0        |            |
| ebp      | 00000000        | 0       | 00000170  | (2) 8EDA       | mov ds, dx                | 00000E64  | 00000000 | 0        |            |
| esp      | 00000e4c        | 3660    | 00000172  | (5) E86EFFFFFF | call -146 (0x000000e5)    | 00000E68  | 000003ff | 1023     |            |
| eip      | 00000166        | 358     | 00000177  | (1) 58         | pop eax                   | 00000E6C  | 00c0fa00 | 12646912 |            |
| eflags   | 00000246        |         | 00000178  | (1) 5B         | pop ebx                   | 00000E70  | 000003ff | 1023     |            |
| cs       | 0008            |         | 00000179  | (1) 59         | pop ecx                   | 00000E74  | 00c0f200 | 12644864 |            |
| ds       | 0017            |         | 0000017a  | (1) 5A         | pop edx                   | 00000E78  | 00000000 | 0        |            |
| es       | 0000            |         | 0000017b  | (1) 1F         | pop ds                    | 00000E7C  | 000010e0 | 4320     |            |
| ss       | 0010            |         | 0000017c  | (1) CF         | iret                      | 00000E80  | 00000010 | 16       |            |
| fs       | 0000            |         | 0000017d  | (2) 0000       | add byte ptr ds:[eax], al | 00000E84  | 00000000 | 0        |            |
| gs       | 0000            |         | 0000017f  | (2) 0000       | add byte ptr ds:[eax], al | 00000E88  | 00000000 | 0        |            |
| gdt      | 00000998 ( ...) |         | 00000181  | (2) 0000       | add byte ptr ds:[eax], al | 00000E8C  | 00000000 | 0        |            |
| idt      | 00000198 ( ...) |         | 00000183  | (2) 0000       | add byte ptr ds:[eax], al | 00000E90  | 00000000 | 0        |            |
| ldtr     | 0be0            |         | 00000185  | (1) 90         | nop                       | 00000E94  | 00000000 | 0        |            |
| tr       | 0bf8            |         | 00000186  | (2) FF07       | inc dword ptr ds:[edi]    | 00000E98  | 000010f4 | 4340     |            |
| cr0      | 60000011        |         | 00000188  | (1) 98         | cwde                      | 00000E9C  | 00000200 | 512      |            |

bx\_dbg\_read\_linear: physical memory read error (phy=0x000100000000, lin=0x0000000100000000)

(0) Breakpoint 1, 0x0000000000000000 in ?? ()

bx\_dbg\_read\_pnode\_descriptor: selector 0x0030 points to a system descriptor and is not supported!

bx\_dbg\_read\_pnode\_descriptor: selector 0x0020 points to a system descriptor and is not supported!

(0) Breakpoint 2, 0x0000000000000000 in ?? ()

(0) Breakpoint 3, 0x00000000000010e0 in ?? ()

Break CPU: Protected Mode 32 i= 14577641 IOPL=0 id vip vif ac vm if nt of df IF if st ZF af PF cf

#### ➤ 单步运行 int 0x80 进入系统中断函数



| Continue [c] |                 |         | Step [s] |                | Step N [s ###]           | Refresh  |          |             | Break [^C] |
|--------------|-----------------|---------|----------|----------------|--------------------------|----------|----------|-------------|------------|
| Reg N...     | Hex Value       | Decimal | LAddress | Bytes          | Mnemonic                 | LAddress | Value    | (dec.)      |            |
| eax          | 00000041        | 65      | 000010e0 | (5) B817000000 | mov eax, 0x00000017      | 00000BD8 | 00000bd8 | 3032        |            |
| ebx          | 00000000        | 0       | 000010e5 | (2) 8ED8       | mov ds, ax               | 00000BDC | 90660010 | -1872363504 |            |
| ecx          | 00000080        | 128     | 000010e7 | (2) B041       | mov al, 0x41             | 00000BE0 | 00000000 | 0           |            |
| edx          | 0000ef00        | 61184   | 000010e9 | (2) CD80       | int 0x80                 | 00000BE4 | 00000000 | 0           |            |
| esi          | 00000598        | 1432    | 000010eb | (5) B9FF0F0000 | mov ecx, 0x00000fff      | 00000BE8 | 000003ff | 1023        |            |
| edi          | 00000998        | 2456    | 000010fd | (2) E2FE       | loop -2 (0x000010fd)     | 00000BEC | 00c0fb00 | 12647168    |            |
| ebp          | 00000000        | 0       | 000010ff | (2) EBEC       | jmp -20 (0x000010e0)     | 00000BF0 | 000003ff | 1023        |            |
| esp          | 00000bd8        | 3032    | 00001101 | (5) B817000000 | mov eax, 0x00000017      | 00000BF4 | 00c0f300 | 12645120    |            |
| eip          | 000010e9        | 4329    | 00001103 | (2) 8ED8       | mov ds, ax               | 00000BF8 | 00000000 | 0           |            |
| eflags       | 00000246        |         | 00001105 | (2) B042       | mov al, 0x42             | 00000BFC | 00000e60 | 3680        |            |
| cs           | 000f            |         | 00001107 | (2) CD80       | int 0x80                 | 00000C00 | 00000010 | 16          |            |
| ds           | 0017            |         | 00001109 | (5) B9FF0F0000 | mov ecx, 0x00000fff      | 00000C04 | 00000000 | 0           |            |
| es           | 0000            |         | 0000110b | (2) E2FE       | loop -2 (0x00001104)     | 00000C08 | 00000000 | 0           |            |
| ss           | 0017            |         | 0000110d | (2) EBEC       | jmp -20 (0x000010f4)     | 00000C0C | 00000000 | 0           |            |
| fs           | 0000            |         | 0000110f | (2) 0000       | add byte ptr ds[edx], al | 00000C10 | 00000000 | 0           |            |
| gs           | 0000            |         | 00001111 | (2) 0000       | add byte ptr ds[edx], al | 00000C14 | 00000000 | 0           |            |
| gdt          | 00000998 ( ...) |         | 00001113 | (2) 0000       | add byte ptr ds[edx], al | 00000C18 | 00000000 | 0           |            |
| idt          | 00000198 ( ...) |         | 00001115 | (2) 0000       | add byte ptr ds[edx], al | 00000C1C | 00000000 | 0           |            |
| ldtr         | 0be0            |         | 00001117 | (2) 0000       | add byte ptr ds[edx], al | 00000C20 | 00000000 | 0           |            |
| tr           | 0bf8            |         | 00001119 | (2) 0000       | add byte ptr ds[edx], al | 00000C24 | 00000000 | 0           |            |
| cr0          | 60000011        |         | 0000111b | (2) 0000       | add byte ptr ds[edx], al | 00000C28 | 00000000 | 0           |            |
| cr2          | 00000000        |         | 0000111d | (2) 0000       | add byte ptr ds[edx], al | 00000C2C | 00000000 | 0           |            |
| cr3          | 00000000        |         | 0000111f | (2) 0000       | add byte ptr ds[edx], al | 00000C30 | 00000000 | 0           |            |
| cr4          | 00000000        |         | 00001121 | (2) 0000       | add byte ptr ds[edx], al | 00000C34 | 00000000 | 0           |            |
| cr8          | 00000000        |         | 00001123 | (2) 0000       | add byte ptr ds[edx], al | 00000C38 | 00000000 | 0           |            |

## 8.2 iret 前后栈的变化

### ➤ system 函数中执行 iret 前的栈

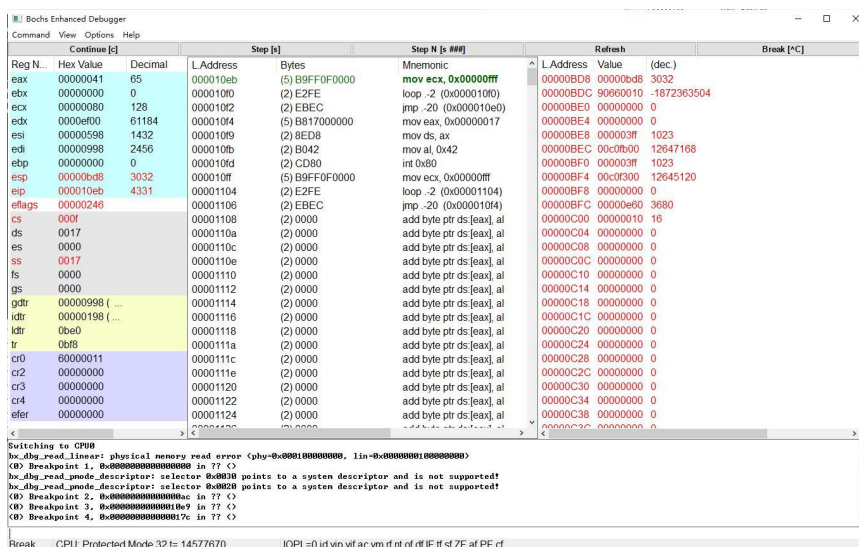
- 栈为任务 0 的内核栈：CS = 0x08，特权级为 00,在内核模式。
- 栈底为 krnl\_stk0=0xE60，栈顶为 SS:ESP=0x10:0x0E4C。
- 栈中有五个元素分别为：SS=0x17, ESP=0x0BD8, EFLAGS, CS=0xF, EIP=0x10EB。
- 栈中数值与前面 task0 执行 int 0x80 前的相关数值相符，说明此时栈中保存的是 task0 在执行系统中中断前压栈的内容。系统中断时发生了栈切换，从用户栈切换到了内核栈。

| Continue [c] |                 |         | Step [s] |                  | Step N [s ###]               | Refresh  |          |          | Break [^C] |
|--------------|-----------------|---------|----------|------------------|------------------------------|----------|----------|----------|------------|
| Reg N...     | Hex Value       | Decimal | LAddress | Bytes            | Mnemonic                     | LAddress | Value    | (dec.)   |            |
| eax          | 00000041        | 65      | 0000017c | (1) CF           | iret                         | 00000E4C | 000010eb | 4331     |            |
| ebx          | 00000000        | 0       | 0000017d | (2) 0000         | add byte ptr ds[edx], al     | 00000E50 | 0000000f | 15       |            |
| ecx          | 00000080        | 128     | 0000017f | (2) 0000         | add byte ptr ds[edx], al     | 00000E54 | 00000246 | 582      |            |
| edx          | 0000ef00        | 61184   | 00000181 | (2) 0100         | add dword ptr ds[edx], eax   | 00000E58 | 00000bd8 | 3032     |            |
| esi          | 00000598        | 1432    | 00000183 | (2) 0000         | add byte ptr ds[edx], al     | 00000E5C | 00000017 | 23       |            |
| edi          | 00000998        | 2456    | 00000185 | (1) 90           | nop                          | 00000E60 | 00000000 | 0        |            |
| ebp          | 00000000        | 0       | 00000186 | (2) FF07         | inc dword ptr ds[edi]        | 00000E64 | 00000000 | 0        |            |
| esp          | 00000e4c        | 3680    | 00000188 | (1) 98           | cwde                         | 00000E68 | 000003ff | 1023     |            |
| eip          | 0000017c        | 380     | 00000189 | (2) 0100         | add dword ptr ds[edx], eax   | 00000E6C | 00c0fa00 | 12646912 |            |
| eflags       | 00000283        |         | 0000018b | (2) 003F         | add byte ptr ds[edi], bh     | 00000E70 | 000003ff | 1023     |            |
| cs           | 0008            |         | 0000018d | (6) 00980900008D | add byte ptr ds[edx], bh     | 00000E74 | 00c0f200 | 12644864 |            |
| ds           | 0017            |         | 0000018f | (2) B600         | mov dh, 0x00                 | 00000E78 | 00000000 | 0        |            |
| es           | 0000            |         | 00000191 | (2) 0000         | add byte ptr ds[edx], al     | 00000E7C | 000010e0 | 4320     |            |
| ss           | 0010            |         | 00000193 | (3) 001401       | add byte ptr ds[ecx+eax], dl | 00000E80 | 00000010 | 16       |            |
| fs           | 0000            |         | 00000195 | (2) 0800         | or byte ptr ds[edx], al      | 00000E84 | 00000000 | 0        |            |
| gs           | 0000            |         | 00000197 | (6) 008E00001401 | add byte ptr ds[esi+180879]  | 00000E88 | 00000000 | 0        |            |
| gdt          | 00000998 ( ...) |         | 000001a2 | (2) 0800         | or byte ptr ds[edx], al      | 00000E8C | 00000000 | 0        |            |
| idt          | 00000198 ( ...) |         | 000001a4 | (6) 008E00001401 | add byte ptr ds[esi+180879]  | 00000E90 | 00000000 | 0        |            |
| ldtr         | 0be0            |         | 000001aa | (2) 0800         | or byte ptr ds[edx], al      | 00000E94 | 00000000 | 0        |            |
| tr           | 0bf8            |         | 000001ac | (6) 008E00001401 | add byte ptr ds[esi+180879]  | 00000E98 | 000010f4 | 4340     |            |
| cr0          | 60000011        |         | 000001b2 | (2) 0800         | or byte ptr ds[edx], al      | 00000E9C | 00000200 | 512      |            |
| cr2          | 00000000        |         | 000001b4 | (6) 008E00001401 | add byte ptr ds[esi+180879]  | 00000EA0 | 00000000 | 0        |            |
| cr3          | 00000000        |         | 000001ba | (2) 0800         | or byte ptr ds[edx], al      | 00000EA4 | 00000000 | 0        |            |
| cr4          | 00000000        |         | 000001bc | (6) 008E00001401 | add byte ptr ds[esi+180879]  | 00000EA8 | 00000000 | 0        |            |
| efer         | 00000000        |         | 000001c2 | (2) 0800         | or byte ptr ds[edx], al      | 00000EAC | 00000000 | 0        |            |

### ➤ system 函数执行 iret 返回后的栈

- 栈为任务 0 的用户栈：CS = 0x0f, 特权级为 11, 在用户模式。
- 栈底为 init\_stack=0xbd8, 栈顶为 SS:ESP=0x17:0x0BD8, 栈为空。
- 此时程序状态正与刚刚内核栈中保存的五个数值相符，SS=0x17, ESP=0x0BD8, EFLAGS, CS=0xF, EIP=0x10EB。说明 iret 将内核中值弹出。中断返回到 task0 被中断的下一条代码。

与前面 task0 执行 int 0x80 前的相关数值相符，说明此时栈中保存的是 task0 在执行系统中断前压栈的内容。系统中断时发生了栈切换，从用户栈切换到了内核栈。



## 8.3 模式切换

### (1) 特权级的改变

- task0 进入系统中断前，CS=0x0f, 特权级为 11, 说明在用户模式
- task0 进入系统中断后，CS=0x08, 特权级为 00, 说明在内核模式
- task0 退出系统中断后，CS=0x0f, 特权级为 11, 说明在用户模式
- 从 11 到 00: 通过 int 0x80 指令。查找 IDT 表中的 80 号中断描述符为 0x08:0x166

| Interrupt | L.Address         |
|-----------|-------------------|
| 80        | 0x0008:0x00000166 |

该中断描述符的段选择子 0x08 代表特权级为 00, 将它加载到 CS 寄存器中, 就实现了特权级从 11 到 00 的改变。

- 从 00 到 11: 特权级的降低只有一种方式, 通过 iret 指令。当执行该指令时, 即从中断中退出。执行弹栈操作 CS 变回 0x0f, 特权级变回 11。

## (2) 栈的切换

- task0 进入系统中断前, SS=0x17, 说明在用户栈

Bochs Enhanced Debugger

| Continue [c] | Step [s]       | Step N [s ###] | Refresh  | Break [*C]    |                          |          |          |             |            |
|--------------|----------------|----------------|----------|---------------|--------------------------|----------|----------|-------------|------------|
| Reg N...     | Hex Value      | Decimal        | LAddress | Bytes         | Mnemonic                 | LAddress | Value    | (dec.)      | Break [*C] |
| eax          | 00000041       | 65             | 000010e0 | (5) B81700000 | mov eax, 0x00000017      | 00000BD8 | 00000bd8 | 3032        |            |
| ebx          | 00000000       | 0              | 000010e5 | (2) 8ED8      | mov ds, ax               | 00000BDC | 90660010 | -1872363504 |            |
| ecx          | 00000080       | 128            | 000010e7 | (2) B041      | mov al, 0x41             | 00000BE0 | 00000000 | 0           |            |
| edx          | 0000ef00       | 61184          | 000010e9 | (2) CD80      | int 0x80                 | 00000BE4 | 00000000 | 0           |            |
| esi          | 00000598       | 1432           | 000010eb | (5) B9FF0F000 | mov ecx, 0x00000fff      | 00000BE8 | 000003ff | 1023        |            |
| edi          | 00000998       | 2456           | 000010fd | (2) E2FE      | loop -2 (0x000010fd)     | 00000BEC | 00c0fb00 | 12647168    |            |
| ebp          | 00000000       | 0              | 000010f2 | (2) EBEC      | jmp -20 (0x000010e0)     | 00000BF0 | 000003ff | 1023        |            |
| esp          | 00000bd8       | 3032           | 000010f4 | (5) B81700000 | mov eax, 0x00000017      | 00000BF4 | 00c0f300 | 12645120    |            |
| eip          | 000010e9       | 4329           | 000010f5 | (2) 8ED8      | mov ds, ax               | 00000BF8 | 00000000 | 0           |            |
| eflags       | 00000246       |                | 000010fb | (2) B042      | mov al, 0x42             | 00000BFC | 00000e60 | 3680        |            |
| cs           | 000f           |                | 000010fd | (2) CD80      | int 0x80                 | 00000C00 | 00000010 | 16          |            |
| ds           | 0017           |                | 000010ff | (5) B9FF0F000 | mov ecx, 0x00000fff      | 00000C04 | 00000000 | 0           |            |
| es           | 0000           |                | 00001104 | (2) E2FE      | loop -2 (0x00001104)     | 00000C08 | 00000000 | 0           |            |
| ss           | 0017           |                | 00001106 | (2) EBEC      | jmp -20 (0x000010f4)     | 00000C0C | 00000000 | 0           |            |
| fs           | 0000           |                | 00001108 | (2) 0000      | add byte ptr ds[edx], al | 00000C10 | 00000000 | 0           |            |
| gs           | 0000           |                | 0000110a | (2) 0000      | add byte ptr ds[edx], al | 00000C14 | 00000000 | 0           |            |
| gdt          | 00000998 ( ... |                | 0000110c | (2) 0000      | add byte ptr ds[edx], al | 00000C18 | 00000000 | 0           |            |
| idt          | 00000198 ( ... |                | 0000110e | (2) 0000      | add byte ptr ds[edx], al | 00000C1C | 00000000 | 0           |            |
| ldtr         | 0be0           |                | 00001110 | (2) 0000      | add byte ptr ds[edx], al | 00000C20 | 00000000 | 0           |            |
| tr           | 0bf8           |                | 00001112 | (2) 0000      | add byte ptr ds[edx], al | 00000C24 | 00000000 | 0           |            |
| cr0          | 60000011       |                | 00001114 | (2) 0000      | add byte ptr ds[edx], al | 00000C28 | 00000000 | 0           |            |

bx\_dbg\_read\_linear: physical memory read error <phy=0x000100000000, lin=0x0000000100000000>  
<0> Breakpoint 1, 0x0000000000000000 in ?? <>  
bx\_dbg\_read\_pm\_descriptor: selector 0x0030 points to a system descriptor and is not supported!  
bx\_dbg\_read\_pm\_descriptor: selector 0x0020 points to a system descriptor and is not supported!  
<0> Breakpoint 2, 0x00000000000000ac in ?? <>  
<0> Breakpoint 3, 0x00000000000000e9 in ?? <>

Break CPU: Protected Mode 32= 14577640 IOPL=0 id vip vif ac vm rf nt of df IF tf sf ZF af PF cf

- task0 进入系统中断后, SS=0x10, 说明在内核栈

Bochs Enhanced Debugger

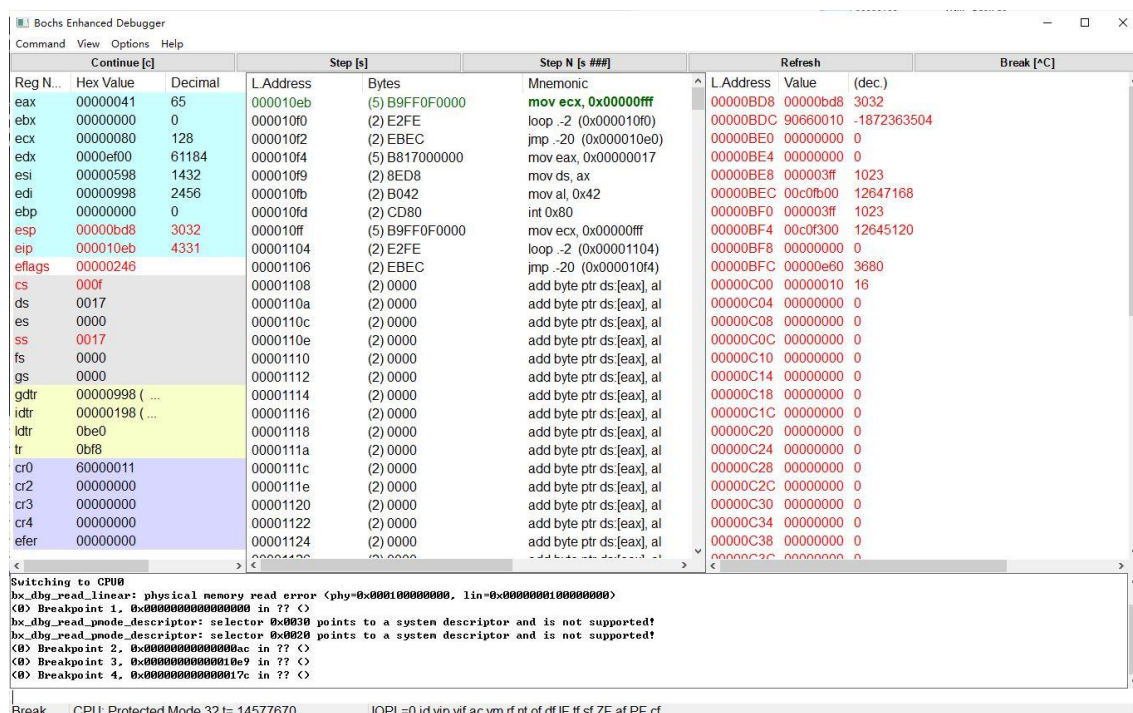
| Continue [c] | Step [s]       | Step N [s ###] | Refresh  | Break [*C]     |                          |          |          |          |            |
|--------------|----------------|----------------|----------|----------------|--------------------------|----------|----------|----------|------------|
| Reg N...     | Hex Value      | Decimal        | LAddress | Bytes          | Mnemonic                 | LAddress | Value    | (dec.)   | Break [*C] |
| eax          | 00000041       | 65             | 00000166 | (1) 1E         | push ds                  | 00000E4C | 000010eb | 4331     |            |
| ebx          | 00000000       | 0              | 00000167 | (1) 52         | push ebx                 | 00000E50 | 0000000f | 15       |            |
| ecx          | 00000080       | 128            | 00000168 | (1) 51         | push ecx                 | 00000E54 | 00000246 | 582      |            |
| edx          | 0000ef00       | 61184          | 00000169 | (1) 53         | push edx                 | 00000E58 | 00000bd8 | 3032     |            |
| esi          | 00000598       | 1432           | 0000016a | (1) 50         | push esi                 | 00000E5C | 00000017 | 23       |            |
| edi          | 00000998       | 2456           | 0000016b | (5) BA10000000 | mov edi, 0x00000010      | 00000E60 | 00000000 | 0        |            |
| ebp          | 00000000       | 0              | 00000170 | (2) 8EDA       | mov ds, dx               | 00000E64 | 00000000 | 0        |            |
| esp          | 00000e4c       | 3660           | 00000172 | (5) E8EFFFFF   | call -146 (0x000000e5)   | 00000E68 | 000003ff | 1023     |            |
| eip          | 00000166       | 358            | 00000177 | (1) 58         | pop eax                  | 00000E6C | 00c0fa00 | 12646912 |            |
| eflags       | 00000246       |                | 00000178 | (1) 5B         | pop ebx                  | 00000E70 | 000003ff | 1023     |            |
| cs           | 0008           |                | 00000179 | (1) 59         | pop ecx                  | 00000E74 | 00c0f200 | 12644864 |            |
| ds           | 0017           |                | 0000017a | (1) 5A         | pop edx                  | 00000E78 | 00000000 | 0        |            |
| es           | 0000           |                | 0000017b | (1) 1F         | pop ds                   | 00000E7C | 000010e0 | 4320     |            |
| ss           | 0010           |                | 0000017c | (1) CF         | iret                     | 00000E80 | 00000010 | 16       |            |
| fs           | 0000           |                | 0000017d | (2) 0000       | add byte ptr ds[edx], al | 00000E84 | 00000000 | 0        |            |
| gs           | 0000           |                | 0000017f | (2) 0000       | add byte ptr ds[edx], al | 00000E88 | 00000000 | 0        |            |
| gdt          | 00000998 ( ... |                | 00000181 | (2) 0000       | add byte ptr ds[edx], al | 00000E8C | 00000000 | 0        |            |
| idt          | 00000198 ( ... |                | 00000183 | (2) 0000       | add byte ptr ds[edx], al | 00000E90 | 00000000 | 0        |            |
| ldtr         | 0be0           |                | 00000185 | (1) 90         | nop                      | 00000E94 | 00000000 | 0        |            |
| tr           | 0bf8           |                | 00000186 | (2) FF07       | inc dword ptr ds[edi]    | 00000E98 | 000010f4 | 4340     |            |
| cr0          | 60000011       |                | 00000188 | (1) 98         | cwde                     | 00000E9C | 00000200 | 512      |            |

bx\_dbg\_read\_linear: physical memory read error <phy=0x000100000000, lin=0x0000000100000000>  
<0> Breakpoint 1, 0x0000000000000000 in ?? <>  
bx\_dbg\_read\_pm\_descriptor: selector 0x0030 points to a system descriptor and is not supported!  
bx\_dbg\_read\_pm\_descriptor: selector 0x0020 points to a system descriptor and is not supported!  
<0> Breakpoint 2, 0x00000000000000ac in ?? <>  
<0> Breakpoint 3, 0x00000000000000e9 in ?? <>

Break CPU: Protected Mode 32= 14577641 IOPL=0 id vip vif ac vm rf nt of df IF tf sf ZF af PF cf

- task0 退出系统中断后, SS=0x17, 说明在用户栈





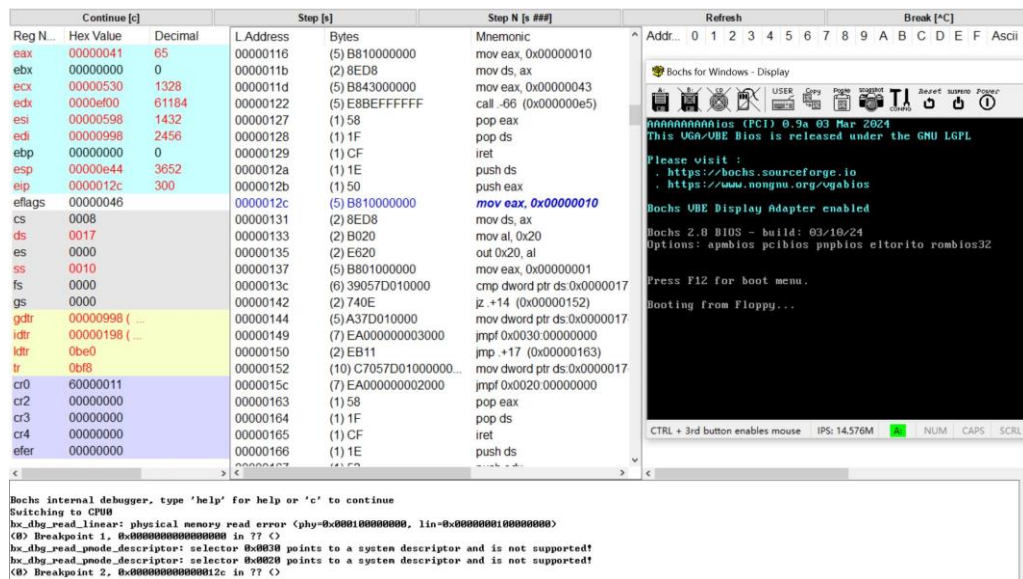
### (3) 总结模式切换

- **从用户模式到内核模式：**int 0x80 操作系统通过查询中断向量表获取中断向量，得到 CS 等值，再通过查询 TSS0 获取到内核栈的 SS ESP 等值。将这些值赋值给相应的寄存器。同时操作系统将当前用户模式下的 SS,ESP,EFLAGS,CS,EIP 等值压入内核栈。
- **从内核模式到用户模式：**iret 操作系统通过弹栈操作，将 SS,ESP,EFLAGS,CS,EIP 等值弹出栈，赋值给相应的寄存器，可以改变特权级，进行栈切换，实现到用户模式的切换。

## 8.4 任务切换

### (1) 任务 0 被时钟中断前

- 进入 head.s 程序，观察汇编代码，找到时钟中断程序。
- 由于断点的设置是依据 CP:IP 寄存器的值进行，而 push 操作仅仅是对栈内存的修改，本身并没有明确的、稳定的内存访问地址，没有影响 CS:IP 寄存器的更新，无法设置精确的断点。
- 将断点设置在 0x12c 的 mov 指令处
- 观察输出终端，可以看到输出了十个 A，而没有 B 的输出，这说明在执行该时钟中断前，task0 执行了十次，而 task1 尚未执行。



### ➤ info tss 查看此时 task0 的 tss:

task0 的段选择符为 0x20, ldt 为 0x28; 由于此时 task0 是首次执行, 尚未被中断过, 因此 tss 值为初始化的值。

```
.long task1, 0x200 /* eip, eflags */
.long 0, 0, 0, 0 /* eax, ecx, edx, ebx */
.long usr_stk1, 0, 0, 0 /* esp, ebp, esi, edi */
.long 0x17, 0x0f, 0x17, 0x17, 0x17, 0x17 /* es, cs, ss, ds, fs, gs */

tr:s=0x20, base=0x000000000000bf8, valid=1
ss:esp(0): 0x0010:0x00000e60
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x00000000
eflags: 0x00000000
cs: 0x0000 ds: 0x0000 ss: 0x0000
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00000000
ldt: 0x0028
i/o map: 0x0800
```

### ➤ 查看分析 time\_interrupt 代码

跳转逻辑为, 如果 current 为 1, 则跳转到 0x152 处执行, 将 current 值变为 0, 跳转到 0x20:0 段选择符为 0x20 的 task0 执行。如果 current 为 0, 则顺序执行, 将 current 值变为 1, 跳转到跳转到 0x30:0 段选择符为 0x30 的 task1 执行。

|          |                        |                                         |
|----------|------------------------|-----------------------------------------|
| 00000137 | (5) B801000000         | mov eax, 0x00000001                     |
| 0000013c | (6) 39057D010000       | cmp dword ptr ds:0x0000017d, eax        |
| 00000142 | (2) 740E               | jz .+14 (0x00000152)                    |
| 00000144 | (5) A37D010000         | mov dword ptr ds:0x0000017d, eax        |
| 00000149 | (7) EA000000003000     | jmpf 0x0030:00000000                    |
| 00000150 | (2) EB11               | jmp .+17 (0x00000163)                   |
| 00000152 | (10) C7057D01000000... | mov dword ptr ds:0x0000017d, 0x00000000 |
| 0000015c | (7) EA000000002000     | jmpf 0x0020:00000000                    |

➤ 验证当前 task0 跳转到 task1。

current 的存放地址为 0x17d。查看该地址，为当前执行的任务 0 的标志 0。

```
[bochs ]:
0x00000000000000017d <bogus+ 0>: 0x00000000
```

代码将会执行 jmpf 0x0030:0 指令，其中 0x30 应为 task1 的段选择符，并把 1 赋给 current，标志切换到任务 1。

(2) 任务 0 被时钟中断后

```
mov dword ptr ds:0x0000017d, eax
jmpf 0x0030:00000000
```

➤ 单步执行指令，程序在执行到 jmpf 0x30:0 后跳转到 task1,可以观察到只打印了 A,task1 尚未开始执行。

| Continue [c] |                |         | Step [s]  |                | Step N [s ###]            | Refresh       | Break [^C] |
|--------------|----------------|---------|-----------|----------------|---------------------------|---------------|------------|
| Reg N...     | Hex Value      | Decimal | L.Address | Bytes          | Mnemonic                  | Addr... 0 1 2 |            |
| eax          | 00000000       | 0       | 000010f4  | (5) B817000000 | mov eax, 0x00000017       |               |            |
| ebx          | 00000000       | 0       | 000010f9  | (2) 8ED8       | mov ds, ax                |               |            |
| ecx          | 00000000       | 0       | 000010fb  | (2) B042       | mov al, 0x42              |               |            |
| edx          | 00000000       | 0       | 000010fd  | (2) CD80       | int 0x80                  |               |            |
| esi          | 00000000       | 0       | 000010ff  | (5) B9FF0F0000 | mov ecx, 0x00000fff       |               |            |
| edi          | 00000000       | 0       | 00001104  | (2) E2FE       | loop -2 (0x00001104)      |               |            |
| ebp          | 00000000       | 0       | 00001106  | (2) EBEC       | jmp -20 (0x000010f4)      |               |            |
| esp          | 00001308       | 4872    | 00001108  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| eip          | 000010f4       | 4340    | 0000110a  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| eflags       | 00000202       |         | 0000110c  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| cs           | 000f           |         | 0000110e  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| ds           | 0017           |         | 00001110  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| es           | 0017           |         | 00001112  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| ss           | 0017           |         | 00001114  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| fs           | 0017           |         | 00001116  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| gs           | 0017           |         | 00001118  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| gdtr         | 00000998 ( ... |         | 0000111a  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| idtr         | 00000198 ( ... |         | 0000111c  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| ldtr         | 0e60           |         | 0000111e  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| tr           | 0e78           |         | 00001120  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| cr0          | 60000019       |         | 00001122  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| cr2          | 00000000       |         | 00001124  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| cr3          | 00000000       |         | 00001126  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| cr4          | 00000000       |         | 00001128  | (2) 0000       | add byte ptr ds:[eax], al |               |            |
| efer         | 00000000       |         | 0000112a  | (2) 0000       | add byte ptr ds:[eax], al |               |            |

Bochs for Windows - Display

AAAAAAAAAAAA BIOS (PCI) 0.9a 03 Mar 2024  
This UGA/UE BIOS is released under the  
Please visit :  
https://bochs.sourceforge.io  
https://www.nongnu.org/vgabios  
Bochs UBE Display Adapter enabled  
Bochs 2.8 BIOS - build: 03/10/24  
Options: apmbios pcibios pnpbios eltorito  
Press F12 for boot menu.  
Booting from Floppy...

CTRL + 3rd button enables mouse IPS: 14.576M

➤ 使用 info tss 查看此时的 tss

可以看出 task1 的段选择符为 0x30,ldt 为 0x38，说明已经由 task0 跳转至 task1



```

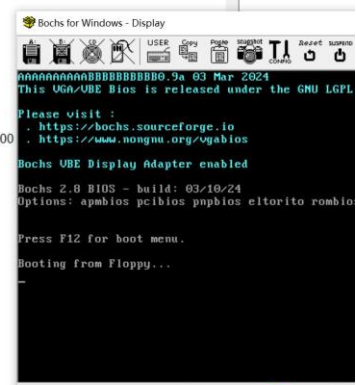
tr:s=0x30, base=0x00000000000000e78, valid=1
ss:esp(0): 0x0010:0x000010e0
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x000010f4
eflags: 0x00000200
cs: 0x000f ds: 0x0017 ss: 0x0017
es: 0x0017 fs: 0x0017 gs: 0x0017
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000
ldt: 0x0038
i/o map: 0x0800

```

### (3) 切换流程

执行十次 task1 程序后，屏幕中出现 10 个字符 B。由于定时器的设置，此时程序跳转到时间中断处理程序

| Continue [c] |                |         | Step [s]  |                        | Step N [s ###]                          |  | Refresh | Break [+C]      |  |  |  |
|--------------|----------------|---------|-----------|------------------------|-----------------------------------------|--|---------|-----------------|--|--|--|
| Reg N...     | Hex Value      | Decimal | L Address | Bytes                  | Mnemonic                                |  |         | Addr... 0 1 2 3 |  |  |  |
| eax          | 00000042       | 66      | 0000012c  | (5) B810000000         | mov eax, 0x00000010                     |  |         |                 |  |  |  |
| ebx          | 00000000       | 0       | 00000131  | (2) 8ED8               | mov ds, ax                              |  |         |                 |  |  |  |
| ecx          | 00000524       | 1316    | 00000133  | (2) B020               | mov al, 0x20                            |  |         |                 |  |  |  |
| edx          | 00000000       | 0       | 00000135  | (2) E620               | out 0x20, al                            |  |         |                 |  |  |  |
| esi          | 00000000       | 0       | 00000137  | (5) B801000000         | mov eax, 0x00000001                     |  |         |                 |  |  |  |
| edi          | 00000000       | 0       | 0000013c  | (6) 39057D010000       | cmp dword ptr ds:0x0000017d, eax        |  |         |                 |  |  |  |
| ebp          | 00000000       | 0       | 00000142  | (2) 740E               | jz +14 (0x00000152)                     |  |         |                 |  |  |  |
| esp          | 000010c4       | 4292    | 00000144  | (5) A37D010000         | mov dword ptr ds:0x0000017d, eax        |  |         |                 |  |  |  |
| eip          | 0000012c       | 300     | 00000149  | (7) EA000000003000     | jmpf 0x0030:00000000                    |  |         |                 |  |  |  |
| eflags       | 00000002       |         | 00000150  | (2) EB11               | jmp +17 (0x00000163)                    |  |         |                 |  |  |  |
| cs           | 0008           |         | 00000152  | (10) C7057D01000000... | mov dword ptr ds:0x0000017d, 0x00000000 |  |         |                 |  |  |  |
| ds           | 0017           |         | 0000015c  | (7) EA000000002000     | jmpf 0x0020:00000000                    |  |         |                 |  |  |  |
| es           | 0017           |         | 00000163  | (1) 58                 | pop eax                                 |  |         |                 |  |  |  |
| ss           | 0010           |         | 00000164  | (1) 1F                 | pop ds                                  |  |         |                 |  |  |  |
| fs           | 0017           |         | 00000165  | (1) CF                 | iret                                    |  |         |                 |  |  |  |
| gs           | 0017           |         | 00000166  | (1) 1E                 | push ds                                 |  |         |                 |  |  |  |
| gdtr         | 00000998 (...) |         | 00000167  | (1) 52                 | push edx                                |  |         |                 |  |  |  |
| idtr         | 00000198 (...) |         | 00000168  | (1) 51                 | push ecx                                |  |         |                 |  |  |  |
| ldtr         | 0e60           |         | 00000169  | (1) 53                 | push ebx                                |  |         |                 |  |  |  |
| tr           | 0e78           |         | 0000016a  | (1) 50                 | push eax                                |  |         |                 |  |  |  |
| cr0          | 80000019       |         | 0000016b  | (5) BA10000000         | mov edx, 0x00000010                     |  |         |                 |  |  |  |
| cr2          | 00000000       |         | 00000170  | (2) 8EDA               | mov ds, dx                              |  |         |                 |  |  |  |
| cr3          | 00000000       |         | 00000172  | (5) E86FFFFFFF         | call -146 (0x000000e5)                  |  |         |                 |  |  |  |
| cr4          | 00000000       |         | 00000177  | (1) 58                 | pop eax                                 |  |         |                 |  |  |  |
| efer         | 00000000       |         | 00000178  | (1) 5B                 | pop ebx                                 |  |         |                 |  |  |  |



- time\_interrupt 的执行原理与前面介绍的相同，下面查看此时的 current 值分析分支的判断，此时的 current 值为 1。

```

[bochs ]:
0x00000000000000017d <bogus+ 0>: 0x00000001

```

- 根据 time\_interrupt 的跳转逻辑，current 为 1 时，执行下列代码，将 current 值设置为 0，将段选择符设置为，task0 的段选择符为 0x20

```

mov dword ptr ds:0x0000017d, 0x00000000
jmpf 0x0020:00000000

```

### (4) TSS 的变化

- task1 的 TSS（执行 jmpf 0x20:0 指令前）

由于 task1 此时依然是第一次执行尚未被中断，因此此时的 TSS 中保存的依旧是初始化时的 tss1, task0 在用户模式, cs 为 0x0f, eip 值为 task1 代码段的起始地址为 0x10f4

```
.long task1, 0x200    /* eip, eflags */
.long 0, 0, 0, 0      /* eax, ecx, edx, ebx */
.long usr_stk1, 0, 0, 0 /* esp, ebp, esi, edi */
.long 0x17, 0x0f, 0x17, 0x17, 0x17, 0x17 /* es, cs, ss, ds, fs, gs */

tr:s=0x30, base=0x00000000000000e78, valid=1
ss:esp(0): 0x0010:0x000010e0
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x000010f4
eflags: 0x00000200
cs: 0x000f ds: 0x0017 ss: 0x0017
es: 0x0017 fs: 0x0017 gs: 0x0017
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x
ldt: 0x0038
i/o map: 0x0800
```

➤ task0 的 TSS（执行 jmpf 0x20:0 指令后）

由于 task0 此时已经是第二次执行，在执行过程中经过了一次中断，tss 值对比初始的 tss0 有所改变。task0 是在内核态运行

0x150 处的指令前，被中断，因此 tss0 保存的 CS:IP 的值就相应的为 0x08:0x150。

```
00000150      (2) EB11      jmp .+17 (0x00000163)

tr:s=0x20, base=0x00000000000000bf8, valid=1
ss:esp(0): 0x0010:0x00000e60
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x00000150
eflags: 0x00000097
cs: 0x0008 ds: 0x0010 ss: 0x0010
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000001 ebx: 0x00000000 ecx: 0x00000530
esi: 0x00000598 edi: 0x00000998 ebp: 0x00000000
ldt: 0x0028
i/o map: 0x0800
```

### (5) 寄存器值的变化

寄存器的值在切换到 task0 时，根据 tss0 中的值进行改变，与其保持一致，恢复到 task0 被中断前的寄存器值。CS=0X08 IP=0x150

| Reg N... | Hex Value      | Decimal | L.Address | Bytes                  | Mnemonic                                | L.Address | Value    | (dec.)   |
|----------|----------------|---------|-----------|------------------------|-----------------------------------------|-----------|----------|----------|
| eax      | 00000001       | 1       | 00000150  | (2) EB11               | jmp .+17 (0x00000163)                   | 00000E44  | 00000041 | 65       |
| ebx      | 00000000       | 0       | 00000152  | (10) C7057D01000000... | mov dword ptr ds:0x0000017d, 0x00000000 | 00000E48  | 00000017 | 23       |
| ecx      | 00000530       | 1328    | 0000015c  | (7) EA000000002000     | jmpf 0x0020:00000000                    | 00000E4C  | 00001000 | 4336     |
| edx      | 0000e000       | 61184   | 00000163  | (1) 58                 | pop eax                                 | 00000E50  | 0000000f | 15       |
| esi      | 00000598       | 1432    | 00000164  | (1) 1F                 | pop ds                                  | 00000E54  | 00000246 | 582      |
| edi      | 00000998       | 2456    | 00000165  | (1) CF                 | iret                                    | 00000E58  | 00000bd8 | 3032     |
| ebp      | 00000000       | 0       | 00000166  | (1) 1E                 | push ds                                 | 00000E5C  | 00000017 | 23       |
| esp      | 00000e44       | 3652    | 00000167  | (1) 52                 | push edx                                | 00000E60  | 00000000 | 0        |
| eip      | 00000150       | 336     | 00000168  | (1) 51                 | push ecx                                | 00000E64  | 00000000 | 0        |
| eflags   | 00000097       |         | 00000169  | (1) 53                 | push ebx                                | 00000E68  | 000003ff | 1023     |
| cs       | 0008           |         | 0000016a  | (1) 50                 | push eax                                | 00000E6C  | 00c0fb00 | 12647168 |
| ds       | 0010           |         | 0000016b  | (5) BA10000000         | mov edx, 0x00000010                     | 00000E70  | 000003ff | 1023     |
| es       | 0000           |         | 00000170  | (2) 8EDA               | mov ds, dx                              | 00000E74  | 00c0f300 | 12645120 |
| ss       | 0010           |         | 00000172  | (5) E86FFFFFFF         | call .-146 (0x000000e5)                 | 00000E78  | 00000000 | 0        |
| fs       | 0000           |         | 00000177  | (1) 58                 | pop eax                                 | 00000E7C  | 000010e0 | 4320     |
| gs       | 0000           |         | 00000178  | (1) 5B                 | pop ebx                                 | 00000E80  | 00000010 | 16       |
| gdt      | 00000998 ( ... |         | 00000179  | (1) 59                 | pop ecx                                 | 00000E84  | 00000000 | 0        |
| idt      | 00000198 ( ... |         | 0000017a  | (1) 5A                 | pop edx                                 | 00000E88  | 00000000 | 0        |
| ldtr     | 0be0           |         | 0000017b  | (1) 1F                 | pop ds                                  | 00000E8C  | 00000000 | 0        |
| tr       | 0bf8           |         | 0000017c  | (1) CF                 | iret                                    | 00000E90  | 00000000 | 0        |
| cr0      | 60000019       |         | 0000017d  | (2) 0000               | add byte ptr ds:[eax], al               | 00000E94  | 00000000 | 0        |
| cr2      | 00000000       |         | 0000017f  | (2) 0000               | add byte ptr ds:[eax], al               | 00000E98  | 00000163 | 355      |
| cr3      | 00000000       |         | 00000181  | (2) 1400               | adc al, 0x00                            | 00000E9C  | 00000046 | 70       |
| cr4      | 00000000       |         | 00000183  | (2) 0000               | add byte ptr ds:[eax], al               | 00000EA0  | 00000001 | 1        |
| efer     | 00000000       |         | 00000185  | (1) 90                 | nop                                     | 00000EA4  | 00000524 | 1316     |

继续执行 task0,输出 10 个 A

```
Bochs for Windows - Display
AAAAAAAAAAAAAAAAAAAAAAAAAAAA 2024
This UGA/VE Bios is released under the GNU LGPL

Please visit :
. https://bochs.sourceforge.io
. https://www.nongnu.org/vgabios

Bochs UVE Display Adapter enabled

Bochs 2.8 BIOS - build: 03/10/24
Options: apmbios pcibios pnpbios eltorito rombios32

Press F12 for boot menu.
Booting from Floppy...
```

### (6) 任务切换总结

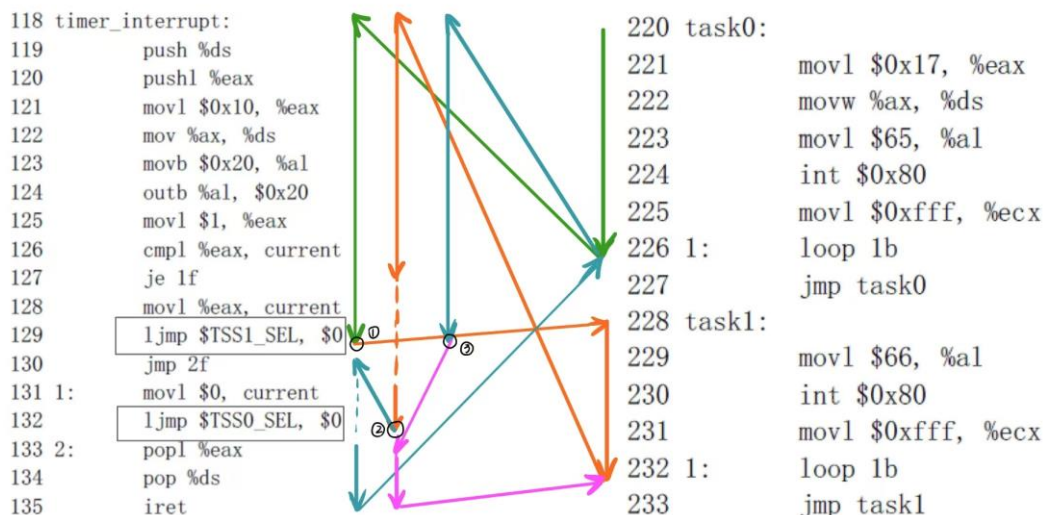
下面展示了从 task0 开始切换到 task1，从 task1 切换回 task0，又从 task0 切换到 task1 的三次切换的过程。

其中主要包括中断和任务切换两种改变控制流的方式。

- 中断的进入是通过定时的中断函数实现的，中断的返回是通过将返回地址入栈出栈来实现的。
- 任务的切换是通过改变 tr 寄存器的值，指向不同的 TSS 段，并对 TSS 段进行更新，保存 CS:IP 值来实现的。



# Interrupt, mode & tasking switching



| 任务切换        | 切换过程     |                                                                                                                                                          | TSS 变化                                                                                                                                                                                                                                                                                                                                       |
|-------------|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Task0→Task1 | 收到中断     | <ul style="list-style-type: none"> <li>task0 循环执行时被时钟中断</li> <li>将下一条指令地址（226 处）压入 task0 的内核栈</li> </ul>                                                 | <ul style="list-style-type: none"> <li>TSS0: 由于 task0 首次执行，此时为初始化的值，CS:IP=0:0。</li> <li>TSS1: 为初始化的值，CS:IP=0f:10f4，指向 task1 指令首地址。</li> <li>任务切换时 (1) tr 指向 tss1，利用 TSS1 保存的 CS: IP 的值，跳转到 task1。(2) 将下一条要运行指令的 CS: IP 值存入 TSS0。</li> <li>更新 TSS0: 保存的 CS:IP 为 <code>ljmp \$TSS1_SEL, \$0</code> 的下一条指令 (<code>jmp2f</code>) 的地址。</li> </ul> |
|             | 中断程序任务切换 | <ul style="list-style-type: none"> <li>current 初始值为 0</li> <li>在 127 处条件判断时选择顺序执行</li> <li>Current 赋值为 1</li> <li>Tr 赋值为 tss1 跳转到 task1 处执行</li> </ul>   |                                                                                                                                                                                                                                                                                                                                              |
| Task1→Task0 | 收到中断     | <ul style="list-style-type: none"> <li>Task1 循环执行时被时钟中断</li> <li>将下一条指令地址入栈（232 处）</li> </ul>                                                            | <ul style="list-style-type: none"> <li>任务切换时 (1) tr 指向 tss0，利用 TSS0 保存的 CS: IP 的值，跳转到 task0 的内核代码（图中 130 行指令）执行。(2) 将下一条要运行指令的 CS: IP 值存入 TSS1。即 133 行代码。</li> <li>更新 TSS1: 保存的 CS:IP 为 <code>ljmp \$TSS0_SEL, \$0</code> 的下一条指令 (<code>popl %eax</code>) 的地址。</li> </ul>                                                                    |
|             | 中断程序任务切换 | <ul style="list-style-type: none"> <li>current 初始值为 1</li> <li>在 127 处条件判断时到 1f 处执行</li> <li>Current 赋值为 0</li> <li>Tr 赋值为 tss0 跳转到 task0 处执行</li> </ul> |                                                                                                                                                                                                                                                                                                                                              |
| Task0→Task1 | 中断返回     | <ul style="list-style-type: none"> <li>Task0 执行到 135 行的中断返回 <code>iret</code></li> <li>对 Task0 的内核栈进行弹栈，返回 226 处继续执行</li> </ul>                          | <ul style="list-style-type: none"> <li>任务切换时 (1) tr 指向 tss1，利用 TSS1 保存的 CS: IP 的值，跳转到 task1 的内核代码（图中 133 行指令）执行。(2) 将下一条要运行指令的 CS: IP 值存入 TSS0。即 130 行指令。</li> <li>更新 TSS0: 保存的 CS:IP 为 <code>ljmp \$TSS1_SEL, \$0</code> 的下一条指令 (<code>jmp2f</code>) 的地址。</li> </ul>                                                                        |
|             | 收到中断     | <ul style="list-style-type: none"> <li>task0 循环执行时被时钟中断</li> <li>将下一条指令地址（226 处）压入 task0 的内核栈</li> </ul>                                                 |                                                                                                                                                                                                                                                                                                                                              |
|             | 中断程序任务切换 | <ul style="list-style-type: none"> <li>current 初始值为 0</li> <li>在 127 处条件判断时选择顺序执行</li> <li>Current 赋值为 1</li> <li>Tr 赋值为 tss1 跳转到 task1 处执行</li> </ul>   |                                                                                                                                                                                                                                                                                                                                              |

## 8.5 进程切换：调试分析 Linux0.11

(1) 编写 process.c 函数设置不同参数的进程，在实现进程切换的函数添加输出

### ➤ kernel/fork.c

- 在 copy\_process() 中添加输出新建进程 N、就绪态语句 J

```
141 | p->state = TASK_RUNNING; /* do this last, just in case */
142 | //进程就绪
143 | fprink(3,"%d\tJ\t%d\n",p->pid,jiffies);
144 | return last_pid;

97 | p->start_time = jiffies;
98 | // 新建进程
99 | fprink(3,"%d\tN\t%d\n",p->pid,jiffies);
```

### ➤ kernel/sched.c

- schedule() 函数：输出就绪态 J 和运行态 R

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;

    /* check alarm, wake up any interruptible tasks that have got a signal */

    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                (*p)->signal |= (1<<(SIGALRM-1));
                (*p)->alarm = 0;
            }
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                (*p)->state==TASK_INTERRUPTIBLE)
            {
                (*p)->state=TASK_RUNNING;
                //就绪态
                fprink(3,"%d\tJ\t%d\n",(*p)->pid,jiffies);
            }
        }

    if(task[next] ->pid != current->pid) //切换到不同的进程时才输出
    {
        if(current->state == TASK_RUNNING) //变为就绪
        {
            //就绪
            fprink(3,"%d\tJ\t%d\n",current->pid,jiffies);
        }
        //运行
        fprink(3,"%d\tR\t%d\n",task[next]->pid,jiffies);
    }
    switch_to(next);
}
```

---

- **sys\_pause()**函数： 输出阻塞态 W

```
int sys_pause(void)
{
    if(current->state!=TASK_INTERRUPTIBLE)
    {
        //阻塞态
        fprintf(3,"%ld\t%c\t%ld",current->pid,'W',jiffies);
    }
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    return 0;
}
```

- **sleep\_on()**函数： 输出阻塞态 W 和就绪态 J

```
void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;
    *p = current;

    current->state = TASK_UNINTERRUPTIBLE;
    //阻塞态
    fprintf(3,"%ld\tW\t%ld\n",current->pid,jiffies);

    schedule();

    if (tmp)
        tmp->state=0;
    //就绪态
    fprintf(3,"%ld\tJ\t%ld\n",tmp->pid,jiffies);
}
```

- **interruptible\_sleep\_on()**函数： 输出阻塞态 W 和就绪态 J

```

void interruptible_sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp=*p;
    *p=current;
repeat: current->state = TASK_INTERRUPTIBLE;
    //阻塞态
    fprintf(3,"%ld\tw\t%ld\n",current->pid,jiffies);
    schedule();
    if (*p && *p != current) {
        (**p).state=0;
        //就绪态
        fprintf(3,"%ld\tj\t%ld\n",(**p).pid,jiffies);
        goto repeat;
    }
    *p=NULL;
    if (tmp)
        tmp->state=0;
        //就绪态
        fprintf(3,"%ld\tj\t%ld\n",tmp->pid,jiffies);
}

```

- wake\_up() 函数： 输出就绪态 J

```

void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;
        //就绪态
        fprintf(3,"%ld\t%c\t%ld\n",(**p).pid,'J',jiffies);
        *p=NULL;
    }
}

```

➤ kernel/exit.c:

- 修改 do\_exit()函数： 输出退出 E

```

current->state = TASK_ZOMBIE;
//退出进程
fprintf(3,"%ld\t%c\t%ld\n",current->pid,'E',jiffies);
current->exit_code = code;
tell_father(current->father);
schedule();
return (-1); /* just to suppress warnings */

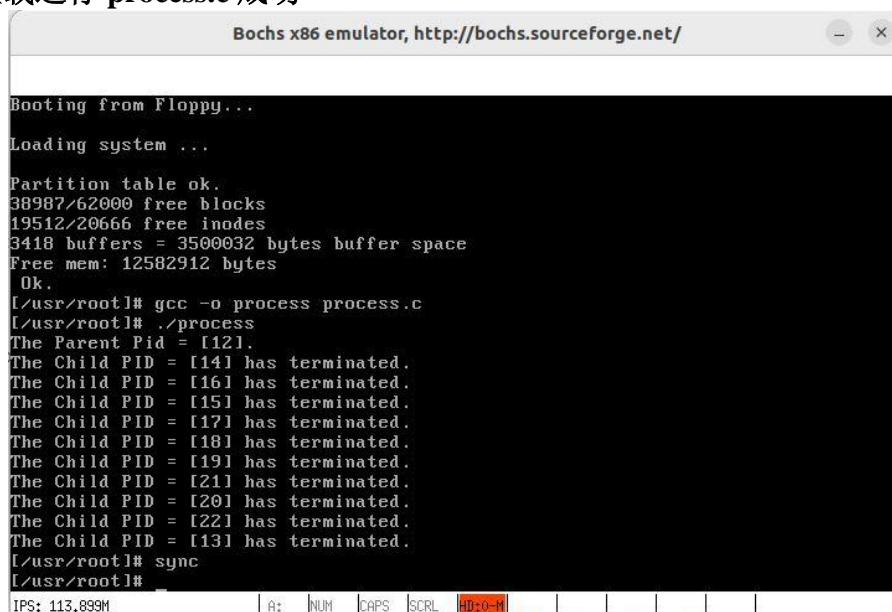
```



- sys\_waitpid()函数: 输出阻塞态 W

```
if (flag) {
    if (options & WNOHANG)
        return 0;
    current->state=TASK_INTERRUPTIBLE;
    //阻塞
    fprintf(3,"%ld\t%c\t%ld\n",current->pid,'W',jiffies);
    schedule();
    if (!(current->signal &= ~(1<<(SIGCHLD-1))))
        goto repeat;
    else
        return -EINTR;
}
```

(2) 挂载运行 process.c 成功



(3) 生成 log 文件

查看 log 文件: 顺利输出了 N、W、J、E、R 五种状态

|                |    |   |   |    |    |   |   |    |    |   |   |    |
|----------------|----|---|---|----|----|---|---|----|----|---|---|----|
| process1.log   | 9  | 0 | J | 50 | 9  | 0 | J | 51 | 9  | 0 | J | 51 |
| process2.log   | 10 | 2 | J | 50 | 10 | 2 | J | 51 | 10 | 2 | J | 51 |
| process3.log   | 11 | 0 | J | 50 | 11 | 0 | J | 51 | 11 | 0 | J | 51 |
| stat_log.py    | 12 | 2 | J | 50 | 12 | 2 | J | 51 | 12 | 2 | J | 51 |
| testlab2.c     | 13 | 0 | J | 51 | 13 | 0 | J | 51 | 13 | 0 | J | 51 |
| testlab2.sh    | 14 | 2 | J | 51 | 14 | 2 | J | 52 | 14 | 2 | J | 52 |
| unistd.h       | 15 | 0 | J | 51 | 15 | 0 | J | 52 | 15 | 0 | J | 52 |
| whoami         | 16 | 2 | J | 58 | 16 | 2 | J | 59 | 16 | 2 | J | 59 |
| whoami.c       | 17 | 0 | J | 58 | 17 | 0 | J | 59 | 17 | 0 | J | 59 |
| hdc            | 18 | 2 | J | 65 | 18 | 2 | J | 65 | 18 | 2 | J | 65 |
| linux-0.11     | 19 | 0 | J | 65 | 19 | 0 | J | 66 | 19 | 0 | J | 66 |
| bochsout.txt   | 20 | 3 | N | 68 | 20 | 3 | N | 69 | 20 | 3 | N | 69 |
| bx_enh_dbg.ini | 21 | 3 | J | 69 | 21 | 3 | J | 70 | 21 | 3 | J | 70 |
| dbg-bochs      | 22 | 2 | E | 74 | 22 | 2 | E | 74 | 22 | 2 | E | 74 |
| dbg-bochsgui   |    |   |   |    |    |   |   |    |    |   |   |    |
| dba-bochsui246 |    |   |   |    |    |   |   |    |    |   |   |    |

---

## 第 9 章 hello 的存储管理

### 9.1 hello 的存储器地址空间

**逻辑地址：**逻辑地址通常指的是程序代码中使用的地址，它是由一个段标识符和一个相对于该段起始地址的偏移量组成的。逻辑地址在编译后的汇编代码中出现。

**线性地址：**是逻辑地址到物理地址转换的中间层。在分段机制中，逻辑地址由段偏移量组成，加上段的基地址可以得到线性地址。线性地址是逻辑地址到物理地址转换的结果。

**虚拟地址：**是线性地址经过分页机制转换后的地址。在使用了内存分页的系统中，虚拟地址是程序使用的地址空间中的地址。虚拟地址空间通常比物理地址空间大得多，允许操作系统为每个进程提供独立的地址空间。

**物理地址：**物理地址是内存中实际物理位置的地址。它是通过内存管理单元（MMU）将虚拟地址或线性地址转换而来的。物理地址直接对应于内存芯片上的地址。

**编译：**当我们将 "Hello, World!" 程序编译成可执行文件时，编译器会为其生成逻辑地址。

**加载：**操作系统在加载可执行文件时，会将其逻辑地址转换为线性地址。这些线性地址是程序在内存中运行时的地址。

**执行：**当程序运行时，CPU 使用线性地址。如果操作系统使用了分页机制，那么线性地址会进一步转换为物理地址，以便 CPU 可以直接访问物理内存。

### 9.2 Intel 逻辑地址到线性地址的变换-段式管理

在 Intel 的架构中，逻辑地址到线性地址的变换是通过段式管理实现的。段式管理是一种内存管理技术，它允许程序使用比实际物理内存更大的地址空间。

在 Intel 架构中，逻辑地址通常由两部分组成：段选择符是一个 16 位的值，用于选择一个段描述符。段描述符包含了段的基地址和其他属性。段内偏移量是一个 32 位的值，表示段内的偏移量。

段描述符是一个数据结构，它包含了关于段的详细信息，包括段的基地址、段的界限、访问权限等。段描述符存储在全局描述符表（GDT）或局部描述符表（LDT）中。



---

变换过程如下：

(1) **获取段选择符**：CPU 从指令中获取段选择符。

(2) **查找段描述符**：根据段选择符，CPU 在 GDT 或 LDT 中查找对应的段描述符。

(3) **计算线性地址**：CPU 将段描述符中的基地址与指令中的段内偏移量相加，得到线性地址。

### 9.3 Hello 的线性地址到物理地址的变换-页式管理

(1) **地址分解**：

将 32 位线性地址分解为三部分：目录项、页表项和页内偏移量。

目录项通常是最高位的 10 位，页表项是中间的 10 位，页内偏移量是最低位的 12 位。

(2) **页目录和页表查找**：

操作系统为每个进程维护一个页目录，其中包含指向各个页表的指针。

当需要进行地址转换时，首先根据目录项查找页目录，得到对应的页表地址。在页表中根据页表项查找，获得物理页框地址。

(3) **页框和页内偏移量**：

物理页框地址与页内偏移量结合，形成物理地址。CPU 使用这个物理地址来访问实际的物理内存。

### 9.4 TLB 与四级页表支持下的 VA 到 PA 的变换

TLB 和四级页表是现代计算机系统中用于虚拟内存管理的重要组件。它们协同工作，将虚拟地址转换为物理地址，以便 CPU 可以访问物理内存中的数据。TLB 用于缓存最近使用的虚拟地址到物理地址的映射，以提高内存访问速度。四级页表用于支持更大的虚拟地址空间，并实现虚拟地址到物理地址的映射。

(1) **TLB 的作用**

TLB 是一个高速缓存，用于存储最近使用的虚拟地址到物理地址的映射。当 CPU 需要访问内存时，首先在 TLB 中查找对应的映射。如果找到了（称为 TLB 命中），则直接使用 TLB 中的物理地址进行内存访问，这样可以避免访问慢速的页表，从而提高内存访问速度。如果 TLB 中没有找到对应的映射（称为 TLB 缺失），则需要进行页表查找，并将新找到的映射添加到 TLB 中。

(2) **四级页表的作用**

现代操作系统为了支持更大的虚拟地址空间，通常使用多级页表结构。四级

---

页表是一种常见的多级页表结构，它将虚拟地址空间划分为多个层次，每一级页表都指向下一级页表或最终的物理内存地址。四级页表结构如下：

**第一级页表：**包含指向第二级页表的指针。

**第二级页表：**包含指向第三级页表的指针。

**第三级页表：**包含指向第四级页表的指针。

**第四级页表：**包含虚拟页号到物理页框号的映射。

当 CPU 产生一个虚拟地址时，MMU 会根据虚拟地址的各级页号，逐级查找页表，直到找到最终的物理页框号。然后，将虚拟地址的页内偏移量与物理页框号结合，得到最终的物理地址。

### **(3) VA 到 PA 的变换过程**

**TLB 查找：**CPU 首先在 TLB 中查找虚拟地址的映射。如果 TLB 命中，则直接使用 TLB 中的物理地址。

**页表查找：**如果 TLB 缺失，则从第一级页表开始，逐级查找页表，直到找到最终的物理页框号。

**地址组合：**将虚拟地址的页内偏移量与物理页框号结合，得到最终的物理地址。

**内存访问：**使用物理地址访问物理内存中的数据。

## **9.5 三级 Cache 支持下的物理内存访问**

- 1.查找 L1 Cache：首先在 L1 Cache 中查找数据。
- 2.查找 L2 Cache：如果 L1 Cache 未命中，则在 L2 Cache 中查找。
- 3.查找 L3 Cache：如果 L2 Cache 也未命中，则在 L3 Cache 中查找。
- 4.访问主存：如果三级 Cache 都未命中，则从主存中读取数据。
- 5.更新 Cache：将读取的数据更新到 Cache 中，以便下次快速访问。

## **9.6 hello 进程 fork 时的内存映射**

在调用 fork 函数时，操作系统会为新进程创建各种数据结构，并分配一个唯一的进程标识符（PID）。创建当前进程的 mm\_struct、区域结构和页表的副本，并将这些数据结构标记为只读。

将两个进程中的每个区域结构标记为私有的写时复制，这意味着两个进程共享相同的物理内存页面，但这些页面被标记为只读，直到其中一个进程尝试对其进行写操作。

当新进程中的 fork 函数返回时，新进程的虚拟内存状态与调用 fork 函数时的

---

虚拟内存状态相同。当两个进程中的任意一个尝试进行写操作时，写时复制机制会生效，创建新的页面副本，从而确保每个进程都保持私有的地址空间概念。这意味着在修改共享内存时，操作系统会为正在进行修改的进程创建一个新的独立副本，以防止对其他进程的影响。

写时复制机制使得 `fork` 函数能够高效地创建子进程，并避免了不必要的内存复制开销，从而提高了系统的性能和效率。这种机制使得父子进程在初始阶段能够共享内存空间，同时又保证了各自的数据操作不会相互干扰。

## 9.7 hello 进程 `execve` 时的内存映射

`execve` 函数通过清除现有用户空间、创建新的私有空间、映射共享库，以及更新程序计数器，实现了将新程序加载到当前进程中，并用新程序替代当前程序的过程。

1. 清除现有用户空间：在执行新程序之前，当前进程的用户空间（包括代码段、数据段、未初始化的全局变量段和栈）需要被清除。这包括释放原有程序的内存空间和清理其他资源。

2. 创建新的私有空间：`execve` 会为新程序创建新的私有空间，包括代码段、数据段（如初始化的全局变量）、未初始化的全局变量段和栈。这些新空间是私有的，意味着它们只能被当前进程访问，并且在需要时会进行写时复制，以确保进程间的内存隔离。

3. 映射共享库：如果新程序与共享库（如动态链接库）链接，那么这些库也会被映射到用户虚拟地址空间中的共享区域。共享区域允许多个进程共享相同的代码和数据，以节省内存和提高执行效率。

4. 更新程序计数器：`execve` 最后一步是更新当前进程的程序计数器（PC），使其指向新程序代码段的入口点。这样，当 `execve` 返回时，新程序的入口点将被执行，从而开始新程序的执行。

## 9.8 缺页故障与缺页中断处理

1. 触发缺页中断：当 CPU 尝试访问一个不在内存中的虚拟地址时，会触发缺页中断。

2. 选择置换页面：根据页面置换算法选择一个置换页面。

3. 将置换页面写回磁盘：如果置换页面已被修改，则将其写回磁盘。

4. 加载新页面：从磁盘加载新页面到内存中。

5. 更新页表：更新页表项，指向新页面的物理页框。

---

6. 重新执行指令：重新执行导致缺页故障的指令。

## 9.9 本章小结

本章介绍了 `hello` 的存储器地址空间、Intel 逻辑地址到线性地址的变换-段式管理、Hello 的线性地址到物理地址的变换-页式管理、TLB 与四级页表支持下的 VA 到 PA 的变换、三级 Cache 支持下的物理内存访问、`hello` 进程 `fork` 时的内存映射、`hello` 进程 `execve` 时的内存映射、缺页故障与缺页中断处理。

---

## 第 10 章 hello 的 I/O 管理

### 10.1 Linux 的 I/O 设备管理方法

#### (1) 设备的模型化：文件

- **统一抽象：**Linux 将设备（如硬盘、显示器、键盘等）模型化为文件，以统一管理方式进行访问。

字符设备：如键盘、鼠标，通过字符流进行操作。

块设备：如硬盘、SSD，通过块操作进行读写。

网络设备：通过套接字接口操作。

- **文件系统接口：**用户程序通过文件系统接口（如 `open`、`read`、`write` 等）访问设备，屏蔽底层硬件差异。

#### (2) 设备管理：Unix I/O 接口

- **Unix 提供了标准的 I/O 接口，包括系统调用和库函数，通过这些接口实现对设备的管理和数据的输入输出。**

**系统调用层：**内核提供 `open`、`read`、`write`、`close` 等函数，与硬件设备直接交互。

**库函数层：**如标准输入输出库中的 `printf`、`scanf` 等，通过系统调用实现用户友好的接口。

### 10.2 简述 Unix I/O 接口及其函数

#### (1) Unix I/O 接口概述

Unix I/O 接口是操作系统与用户程序之间进行数据交互的桥梁，包括对文件、设备和管道的访问。

提供基本操作：打开文件、读取数据、写入数据、关闭文件等。

#### (2) 常用 Unix I/O 函数



| 函数    | 功能          | 描述                  |
|-------|-------------|---------------------|
| open  | 打开文件或设备     | 返回文件描述符，作为后续操作的标识符。 |
| read  | 从文件或设备中读取数据 | 从指定文件描述符中读取数据到缓冲区中。 |
| write | 向文件或设备中写入数据 | 将缓冲区中的数据写入到指定文件描述符。 |
| close | 关闭文件或设备     | 释放文件描述符及相关资源。       |
| lseek | 改变文件读写位置    | 用于随机访问文件中的某一部分数据。   |
| ioctl | 控制设备操作      | 用于执行设备特定的操作。        |

### (3) I/O 接口的特性

- 统一性：文件和设备操作接口一致。
- 高效性：通过缓冲区管理提升性能。
- 可靠性：支持错误检测与恢复。

## 10.3printf 的实现分析

### (1) printf 调用流程

- 格式化输出生成字符串：调用 `vsprintf` 等函数，将格式化数据写入内存缓冲区。
- 系统调用 `write`：将缓冲区中的数据通过 `write` 系统调用写入标准输出设备（文件描述符 1）。
- 陷阱-系统调用机制：系统调用通过触发软中断（如 `int 0x80` 或 `syscall`）将控制权转交给内核，由内核完成设备操作。

### (2) 字符显示驱动子程序：字符的显示涉及驱动程序和硬件的配合

- 从 ASCII 到字模库：根据 ASCII 码，从字模库中提取字符对应的点阵数据。
- 显示缓冲区 (VRAM)：将点阵数据写入显存 (VRAM)，每个点包含 RGB 三原色信息。
- 显示芯片刷新：显示芯片按刷新频率逐行读取 VRAM 数据，通过信号线向液晶屏传输点阵 RGB 信息。

---

## 10.4 getchar 的实现分析

### (1) 键盘中断处理

- 键盘中断触发：用户按下键盘时，键盘控制器产生中断请求（IRQ）。
- 中断服务程序：操作系统中断服务程序读取按键扫描码，并将其转换为 ASCII 码。
- 保存到缓冲区：ASCII 码存储在系统内核的键盘缓冲区中，供用户程序读取。

### (2) getchar 调用流程

- 调用 read 系统调用：getchar 调用 read 系统调用，从键盘缓冲区读取按键数据。
- 返回数据：用户程序等待按键输入，当检测到回车键（'\n'）后，将缓冲区内容返回给用户程序。

## 10.5 本章小结

- 本章主要探讨了 Linux 系统的 I/O 管理机制，包括设备的模型化和统一管理方法。
- Unix I/O 接口通过简单而高效的函数，为用户程序提供了对文件和设备的访问支持。
- 详细分析了 printf 和 getchar 的实现，揭示了 I/O 操作背后的系统调用和硬件协作机制。
- 通过了解字符显示和键盘中断处理的底层原理，可以更深入理解操作系统的 I/O 管理及其优化策略。

---

## 第 11 章 hello 的文件系统管理

### 11.1 文件路径解析

(1) 从根目录开始解析路径:

- 操作系统通过文件系统的根目录，读取路径中的每一级目录名称，逐步查找目标文件。例如，若路径为 `/home/user/hello`，系统会依次访问 `/home`、`/home/user`，最终定位到 `hello` 文件。

(2) 获取盘块地址:

- 每个文件或目录在文件系统中都有对应的盘块地址，操作系统通过文件分配表（如 FAT、i-node 表）获取目标文件的盘块地址。

### 11.2 检查文件权限

(1) **权限字段检查:** 文件元数据中通常包含权限字段（如 UNIX 文件权限位），定义了文件所有者、组用户及其他用户的访问权限（读、写、执行）。

(2) **用户身份验证:** 操作系统检查当前进程的用户 ID 和组 ID，确定是否有权限执行目标文件。

(3) **拒绝非法访问:** 如果权限检查失败，操作系统返回错误（如 `Permission Denied`），并终止后续操作。

### 11.3 获取文件元数据

**文件头信息:** `hello` 是 ELF 格式的可执行文件，操作系统需要读取其文件头以提取关键信息如：**文件大小**（分配内存空间时需要用到）、**入口地址**（程序执行的起始点）、**段偏移**（代码段、数据段的偏移地址和大小）、**文件系统结构**（解析文件系统的具体结构以正确读取文件内容）

### 11.4 分配磁盘缓存

(1) 分配缓存空间:

- 操作系统为 `test` 文件分配内存中的一块区域作为缓存，用于存储部分或全部文件内容。

- 
- 磁盘缓存的大小和策略（如 LRU 缓存）由操作系统决定。

## （2）预读取文件内容：

- 操作系统可能会将 test 文件的关键部分（如文件头、代码段）提前加载到缓存中，以减少磁盘 I/O 开销。

## （3）缓存一致性：

- 当文件被多次访问时，操作系统会优先从缓存中读取内容，从而提升文件系统的响应速度。

## 11.5 缺页中断时处理

在 hello 发生缺页中断时，操作系统会通知 CPU 从 I/O 中读取所缺的页，通过虚拟地址，可以获得该页在进程段中的偏移量，通过访问 hello 程序所在文件的路径，逐步访问文件系统，将所缺的页调入内存。

## 11.6 总结

文件系统的管理是操作系统执行程序的重要环节。在 hello 程序执行时，操作系统通过路径解析定位文件，检查权限确保安全，解析元数据获取执行信息，并利用磁盘缓存优化 I/O 性能。在发生缺页中断时，通过操作系统和文件系统找到该页所在磁盘中的位置，将该页调入内存。

---

## 结论

Hello 程序的生命周期涉及了从程序到进程（P2P）和从零到零（020）两个角度

1. 编写 `hello.c` 的源程序
2. 预处理阶段：进行预处理，`hello.c` 文件生成成为 `hello.i` 文件
3. 编译阶段：`hello.i` 文件编译生成汇编文件 `hello.s`
4. 汇编阶段：汇编器将汇编文件转化成为机器能识别的机器语言，得到可重定位目标文件 `hello.o`
5. 链接阶段：链接器将 `hello.o` 与其他库函数链接，生成可执行文件 `hello`，`hello` 被存入磁盘，可以被加载入内存运行。
6. 创建进程：`shell` 调用 `fork` 函数创建子进程
7. 加载程序：`shell` 调用 `execve` 函数启动加载器映射虚拟内存
8. 内存访问：将虚拟地址转化为物理地址，实现虚拟内存和物理内存的翻译
9. I/O：进程运行过程中与外界进行交互
10. 异常控制：常见的异常控制流的形式包括异常和信号两部分进行响应
11. 进程终止：`hello` 被父进程回收，回归 0 状态



---

## 附件

---

| 文件名            | 作用             |                                                                                                         |
|----------------|----------------|---------------------------------------------------------------------------------------------------------|
| hello.c        | 源程序            | <br>hello.c          |
| hello.i        | 预处理后的修改的 C 程序  | <br>hello.i          |
| hello.s        | 汇编程序           | <br>hello.s          |
| hello.o        | 可重定位目标文件       | <br>hello.o          |
| hello          | 可执行目标文件        | <br>hello          |
| objhello.o.txt | hello.o 的反汇编文件 | <br>objhello.o.txt |
| objhello.txt   | hello 的反汇编文件   | <br>objhello.txt   |
| head.s         | Linux00 中操作系统头 | <br>head.s         |
| process.c      | 进程状态切换函数       | <br>process.c      |
| process.log    | 进程切换 log 文件    | <br>process.log    |

---

---

## 参考文献

- [1] Computer Systems:A Programmer's Perspective
- [2] <https://www.cnblogs.com/pianist/p/3315801>