哈爾濱工業大學

计算机系统

大作业

题　　目　程序人生-Hello’s P2P

专　　业　计算机类

学　　号　1180300303

班　　级　1836101/1803003

学　　生　宿梓航

指 导 教 师　刘宏伟

计算机科学与技术学院

2019 年 12 月

摘　要

本文以在作为 Windows 子系统的 Linux 环境(WSL)下，分四步（预处理，编译，汇编，链接）编译了简单程序 hello，如此通过探索 Hello 的 Program 到 Process(Hello’s P2P)过程，研究了程序的诞生过程（编译）。之后又研究 Hello 的从装载、执行到信号、退出等的整个 0to0 的进程生命周期，进而探索了操作系统对于内存、IO、进程的管理过程。

关键词：编译；操作系统；WSL；

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目　录

目录

第 1 章 概述

1.1 Hello 简介

编写 hello.c（Program）后，运行编译器进行预处理、编译、汇编、链接等过程，生成可执行文件 hello。

Shell 通过 fork 函数调用 OS（进程管理）产生一个子进程，然后利用 execve 函数对 hello 可执行文件进行内存映射，并通过进程管理分配给 hello 的代码一定的 CPU 时间，然后可以使得 hello 在 Hardware(CPU/RAM/IO)上运行（取指译码执行/流水线等）；这时的 Hello 是一个进程(Progress)。这便是 P2P。

OS（存储管理）与 MMU 为需要将代码中描述的 VA 转换为 PA 来进行内存访问；TLB、4 级页表、3 级 Cache，Pagefile 等等为程序运行时的储存访问加速；IO 管理与信号处理，以软硬件相结合的方式，才使 Hello 能在键盘、主板、显卡、屏幕间流畅的运行。在运行结束后，被 hello 占用的资源又会被操作系统回收，这就是 020

1.2 环境与工具

列出你为编写本论文，折腾 Hello 的整个过程中，使用的软硬件环境，以及开发与调试工具。

1.2.1 硬件环境

Intel Core i7-8750H, 2.2GHz
16G RAM
Samsung SSD 970 PRO 1TB

1.2.2 软件环境

Windows 10 1903 x64
Ubuntu 18.04.2 LTS(4.4.0-18362-Microsoft)
Windows Subsystem for Linux
zsh 5.4.2
VcXsrv X Server 1.20.5.1

1.2.3 开发工具

Visual Studio Code 1.41.1
clang version 6.0.0-1ubuntu2
GNU nano 2.9.3
Edb 1.1.0

1.3 中间结果

列出你为编写本论文，生成的中间结果文件的名字，文件的作用等。

hello.c	hello 的 c 源文件
hello.i	预处理后的源代码
hello.s	编译生成的汇编程序
hello.o	汇编生成的可重定位目标文件
hello.objdump	hello.o 用 objdump 处理的输出
hello.readelf	hello.o 用 readelf 处理的输出
hello	链接生成的可执行文件
hello.exe.objdump	hello 用 objdump 处理的输出
hello.exe.readelf	hello 用 readelf 处理的输出

1.4 本章小结

本章简单地介绍了 hello 及其 P2P 和 020 的过程，然后列出了完成本文的环境和工具，阐明了编写本论文时出现的中间文件及其作用。

（第 1 章 0.5 分）

第 2 章 预处理

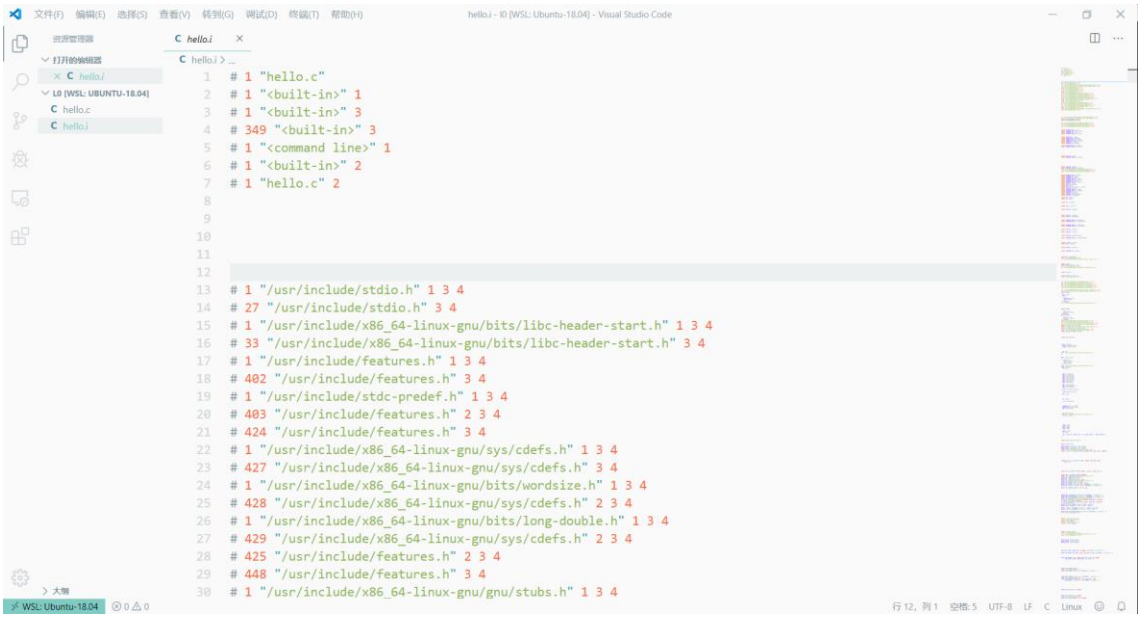
2.1 预处理的概念与作用

预处理器（cpp）根据以字符#开头的命令，修改原始的 C 程序。这些命令通常是.c 文件开头的一些以#include 命令。这些命令告诉预处理器读取哪些相应的头文件，之后把这些文件直接插入程序文本中，就得到了另一个 C 程序，通常是以.i 作为文件扩展名。

2.2 在 Ubuntu 下预处理的命令

```
ferdinand@Ferdinand-IS ~/l0> clang -E hello.c -o hello.i
ferdinand@Ferdinand-IS ~/l0>
```

应截图，展示预处理过程！



2.3 Hello 的预处理结果解析

预处理后，hello.c 变为 hello.i 文件。定义的宏已经被展开，头文件中的内容被包含进该文件中，代码行数由 29 行变为 3096 行。内容增加，且仍是 C 语言程序文本文件。

2.4 本章小结

本章里，我们对 hello.c 进行了预处理。利用指令 clang -E hello.c -o hello.i 可以将其

转换为.i 文件。之后我们可以进行汇编。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译器（ccl）将文本文件 hello.i 翻译成 hello.s，包含一个汇编语言程序，其中包含一个汇编语言程序。

本质上，编译就是把高级语言的源程序翻译成汇编程序。

注意：这儿的编译是指从 .i 到 .s 即预处理后的文件到生成汇编语言程序

3.2 在 Ubuntu 下编译的命令

```
hello.c:10:15: warning: implicit conversion from 'double' to 'int' changes value from 2.5 to 2 [-Wliteral-conversion]
int sleepsecs=2.5;
    ~~~~~~~~~~ ^~~
1 warning generated.
```

应截图，展示编译过程！

3.3 Hello 的编译结果解析

3.3.1 数据

3.3.1.1 全局变量

Hello.c 只包含一个全局变量。

```
int sleepsecs=2.5;
```

```
sleepsecs:
    .long   2                       # 0x2
    .size   sleepsecs, 4

    .type   .L.str,@object          # @.str
    .section    .rodata.str1.1,"aMS",@progbits,1
```

编译后，被放在.data 节中；同时，因为被声明为 int，它的值不是 2.5，而是 2（这也是前面编译时 Warning 的来历）

3.3.1.2 局部变量

在 main 中存在局部变量 i，argc，argv

其中 int i 存放在栈上 -20(%rbp)

另两者作为参数,

int argc 存放在寄存器 rdi 中，

char* argv[] 存放在寄存器 rsi 中

3.3.1.3 常量

Main 函数中的两个字符串常量

"Usage: Hello 学号 姓名！\n"

"Hello %s %s\n"

均被放置在.rodata 节中.

```
    .section    .rodata.str1.1,"aMS",@progbits,1
.L.str:
    .asciz  "Usage: Hello \345\255\246\345\217\267 \345\247\223\34
    .size   .L.str, 31

    .type   .L.str.1,@object        # @.str.1
.L.str.1:
    .asciz  "Hello %s %s\n"
    .size   .L.str.1, 13


    .ident  "clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
    .section    ".note.GNU-stack","",@progbits
```

3.3.1.4 表达式

共有两个比较表达式 argc!=3 和 i<10

它们分别由如下的汇编语句计算:

```
  cmpl    $3, -8(%rbp)
 cmpl    $10, -20(%rbp)
```

3.3.1.5 类型

共涉及到三个类型:char*,char**和 int 型

前两者均属于指针型,所以长度都为 8,在汇编语句中的操作以 q 为后缀;

Int 长度为 4,操作以 l 为后缀

3.3.1.6 宏

Hello.c 中没有发现可扩展的宏.

3.3.2 赋值

3.3.2.1 =

全局变量的赋值是直接写在可执行文件里的;

Int i 的赋值是通过 movl 一个立即数实现的.

i 的声明(不赋初始值)由

```
subq    $48, %rsp
```

直接完成.

赋值则是

```
movl    $0, -20(%rbp)
```

3.3.2.2 逗号操作符

Hello.c 中的逗号只用于参数的分隔.

3.3.3 类型转换

```
int sleepsecs=2.5;
```

只有此处,double 2.5 被隐式转换为 int;没有使用显示转换.

3.3.4 sizeof

没有使用此操作符.

3.3.5 算术操作

i++ 循环中的算术操作;

被这样完成

```
movl    -20(%rbp), %eax
addl    $1, %eax
movl    %eax, -20(%rbp)
```

3.3.6 逻辑/位操作

没有出现.

3.3.7 关系操作

已在 3.3.1.4 中说明,不再赘述.

3.3.8 数组/指针/结构操作

数组操作是相邻的两处:

```
argv[1],argv[2]
```

被转化为相对寻址过程.

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

3.3.9 控制转移

3.3.9.1 if 块

```
if(argc!=3)
{
    printf("Usage: Hello 学号 姓名! \n");
    exit(1);
}
```

被编译器转化为 cmpl+je+标签的形式

```
    cmpl    $3, -8(%rbp)
    je  .LBB0_2
# %bb.1:
    movabsq $.L.str, %rdi
    movb    $0, %al
    callq   printf
    movl    $1, %edi
    movl    %eax, -24(%rbp)         # 4-byte Spill
    callq   exit
.LBB0_2:
    movl    $0, -20(%rbp)
```

3.3.9.2 for 循环

```
for(i=0;i<10;i++)
{
    printf("Hello %s %s\n",argv[1],argv[2]);
    sleep(sleepsecs);
}
```

被转化成 cmpl+jmp 的形式.

```
.LBB0_2:
    movl    $0, -20(%rbp)
.LBB0_3:                                # =>This Inner Loop Header: Depth=1
    cmpl    $10, -20(%rbp)
    jge .LBB0_6
# %bb.4:                                #   in Loop: Header=BB0_3 Depth=1
    movabsq $.L.str.1, %rdi
    movq    -16(%rbp), %rax
    movq    8(%rax), %rsi
    movq    -16(%rbp), %rax
    movq    16(%rax), %rdx
    movb    $0, %al
    callq   printf
    movl    sleepsecs, %edi
    movl    %eax, -28(%rbp)         # 4-byte Spill
    callq   sleep
    movl    %eax, -32(%rbp)         # 4-byte Spill
# %bb.5:                                #   in Loop: Header=BB0_3 Depth=1
    movl    -20(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -20(%rbp)
    jmp .LBB0_3
.LBB0_6:
```

3.3.10 函数操作

共五次函数调用

```
printf("Usage: Hello 学号 姓名! \n");
exit(1);
printf("Hello %s %s\n",argv[1],argv[2]);
sleep(sleepsecs);
getchar();
```

3.3.10.1 参数传递

前四次调用有参数传递,实际上是用 mov 把实参拷贝到形参对应位置的寄存器上.

第一次 printf

```
movabsq $.L.str, %rdi
```

第二次 exit

```
movl    $1, %edi
```

第三次 printf

```
movabsq $.L.str.1, %rdi
movq    -16(%rbp), %rax
movq    8(%rax), %rsi
movq    -16(%rbp), %rax
movq    16(%rax), %rdx
```

第四次 sleep

```
movl    sleepsecs, %edi
```

3.3.10.2 函数调用

利用 callq 完成,但是 call 之前可能要保存寄存器.

第一次

```
callq   printf
```

第二次

```
movl    %eax, -24(%rbp)
callq   exit
```

第三次

```
callq   printf
```

第四次

```
movl    %eax, -28(%rbp)
callq   sleep
```

第五次

```
callq   getchar
```

3.3.10.3 函数返回

恢复栈指针和帧指针,然后 retq 即可.

```
addq    $48, %rsp
popq    %rbp
retq
```

3.4 本章小结

此外，完成该阶段转换后，可以进行下一阶段的汇编处理。
本章里，我们对 hello.i 进行了编译。利用指令 clang -S hello.i -o hello.s 可以将其转换为汇编程序.s 文件。本章还通过与原高级语言程序文件与汇编程序文件的比较进行比较，完成了对编译器任务的解析。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

由汇编器（as）将.s 文件翻译成为机器语言指令，把这些指令打包成一种叫做可重定位目标程序的格式，并把结果保存在目标文件.o 中的过程,叫做汇编。
Hello.o 是一个二进制文件.
汇编的左右在于把对人类友好的文本文件程序,转化为机器友好的机器代码.

4.2 在 Ubuntu 下汇编的命令

```
ferdinand@Ferdinand-IS ~/l0> clang -c hello.s -o hello.o
ferdinand@Ferdinand-IS ~/l0>
```

4.3 可重定位目标 elf 格式

4.3.1 ELF 头

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          976 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         11
  Section header string table index: 1
```

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,包括 ELF 头的大小、目标文件的类型、机器类型、字节头部表的文件偏移，以及节头部表中条目的大小和数量等信息。
根据文件头的信息，可以知道该文件是可重定位目标文件，含有 11 个节.
4.3.2 节头表

```
Section Headers:
  [Nr] Name              Type            Address          Offset      Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000  0000000000000000  0000000000000000           0     0     0
  [ 1] .strtab           STRTAB          0000000000000000  00000340  000000000000008a  0000000000000000           0     0     1
  [ 2] .text             PROGBITS        0000000000000000  00000040  0000000000000097  0000000000000000  AX       0     0     16
  [ 3] .rela.text        RELA            0000000000000000  00000268  00000000000000c0  0000000000000018          10     2     8
  [ 4] .data             PROGBITS        0000000000000000  000000d8  0000000000000004  0000000000000000  WA       0     0     4
  [ 5] .rodata.str1.1    PROGBITS        0000000000000000  000000dc  000000000000002c  0000000000000001  AMS      0     0     1
  [ 6] .comment          PROGBITS        0000000000000000  00000108  0000000000000037  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000  0000013f  0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         X86_64_UNWIND   0000000000000000  00000140  0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA            0000000000000000  00000328  0000000000000018  0000000000000018          10     8     8
  [10] .symtab           SYMTAB          0000000000000000  00000178  00000000000000f0  0000000000000018           1     4     8
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)
```

节头表中描述了不同节的位置和大小，目标文件中的每个节都有一个固定大小的条目。

由于是可重定位目标文件，所以每个节都从 0 开始，方便进行重定位。

在文件头中得到节头表的信息，然后再使用节头表中的字节偏移信息,可以得到各节在文件中的起始位置，以及各节所占空间的大小。

同时可知，.text 是可执行的，但是不能写；.data 和.rodata 都不可执行，.rodata 也不可写。

4.3.3 .rela.text

```
Relocation section '.rela.text' at offset 0x268 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000001e  000300000001 R_X86_64_64       0000000000000000 .rodata.str1.1 + 0
000000000029  000700000002 R_X86_64_PC32     0000000000000000 printf - 4
000000000036  000400000002 R_X86_64_PC32     0000000000000000 exit - 4
000000000049  000300000001 R_X86_64_64       0000000000000000 .rodata.str1.1 + 1f
000000000064  000700000002 R_X86_64_PC32     0000000000000000 printf - 4
00000000006b  00090000000b R_X86_64_32S      0000000000000000 sleepsecs + 0
000000000073  000800000002 R_X86_64_PC32     0000000000000000 sleep - 4
000000000086  000500000002 R_X86_64_PC32     0000000000000000 getchar - 4
```

重定位节.rela.text 是包含.text 节中位置的列表，列出了.text 节中需要进行重定位的信息，当链接器把这个目标文件和其他文件组合时，需要修改这些位置。图中 8 条重定位信息是 printf 中的两个常量字符串,printf 函数,exit 函数,sleepsecs,sleep 函数,getchar 函数进行的重定位声明。

4.3.4 .symtab

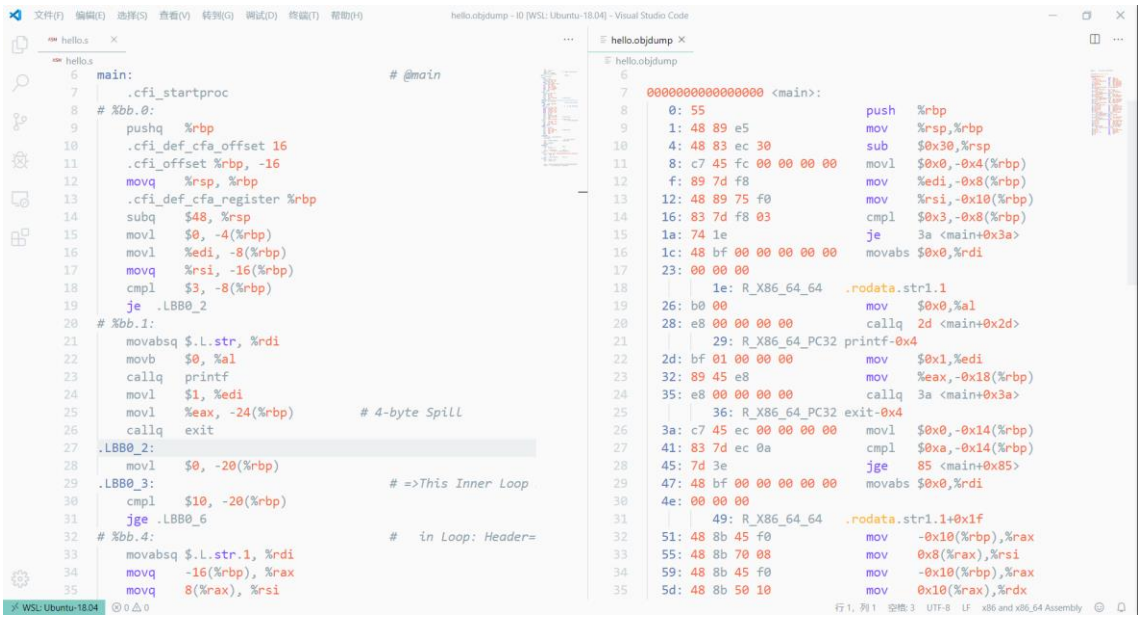
```
Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     4: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
     5: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND getchar
     6: 0000000000000000   151 FUNC    GLOBAL DEFAULT    2 main
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
     8: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sleep
     9: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    4 sleepsecs
```

符号表记录了使用编译器输出到汇编语言.s 文件中的符号.

4.4 Hello.o 的结果解析

```
ferdinand@Ferdinand-IS ~/l0> objdump -d -r hello.o > hello.objdump
ferdinand@Ferdinand-IS ~/l0>
```

然后将该反汇编结果与 hello.s 的进行对照:



通过比较, 两份程序的差异之处主要是:

1. 反汇编代码不再有伪指令和位置标签, 取而代之的是一些绝对或者相对位置引用
2. 反汇编代码中,十进制立即数全都换成了 16 进制
3. 反汇编中的部分指令(mov)和原汇编有所不同.

4.5 本章小结

在本章, 我们利用汇编器把汇编代码程序汇编成可重定位的目标文件. 然后通过 readelf 读取了可重定位目标文件的相关信息, 再用 objdump 反汇编,并与原汇编文件进行了对比.

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

链接器将不同的目标文件（包括本项目的，各种裤库的）合并为一个完整的，可以直接装入内存执行的可执行程序。链接分为两个步骤，一是符号解析，二是重定位。在符号解析时，链接器将每个符号引用与一个确定的符号定义关联起来。然后将多个目标文件中单独的代码节和数据节合并为单文件的各种节。重定位时，将符号从它们的在.o 文件的相对位置重新定位到可执行文件的最终绝对内存位置。更新所有对这些符号的引用来更新它们的新位置。

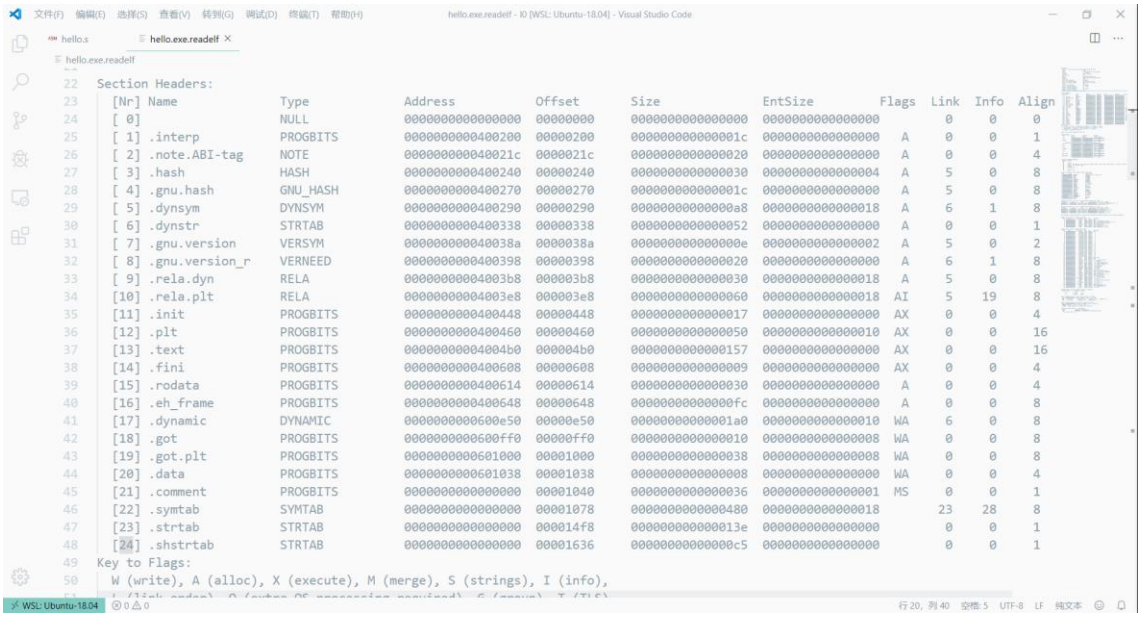
5.2 在 Ubuntu 下链接的命令

```
ferdinand@Ferdinand-IS ~/l0> g=/usr/lib/x86_64-linux-gnu
ld -dynamic-linker \
/lib64/ld-linux-x86-64.so.2 \
$g/crt1.o \
$g/crti.o \
$g/libc.so \
$g/crtn.o \
hello.o \
-o hello
ferdinand@Ferdinand-IS ~/l0> ./hello
Usage: Hello 学号 姓名!
```

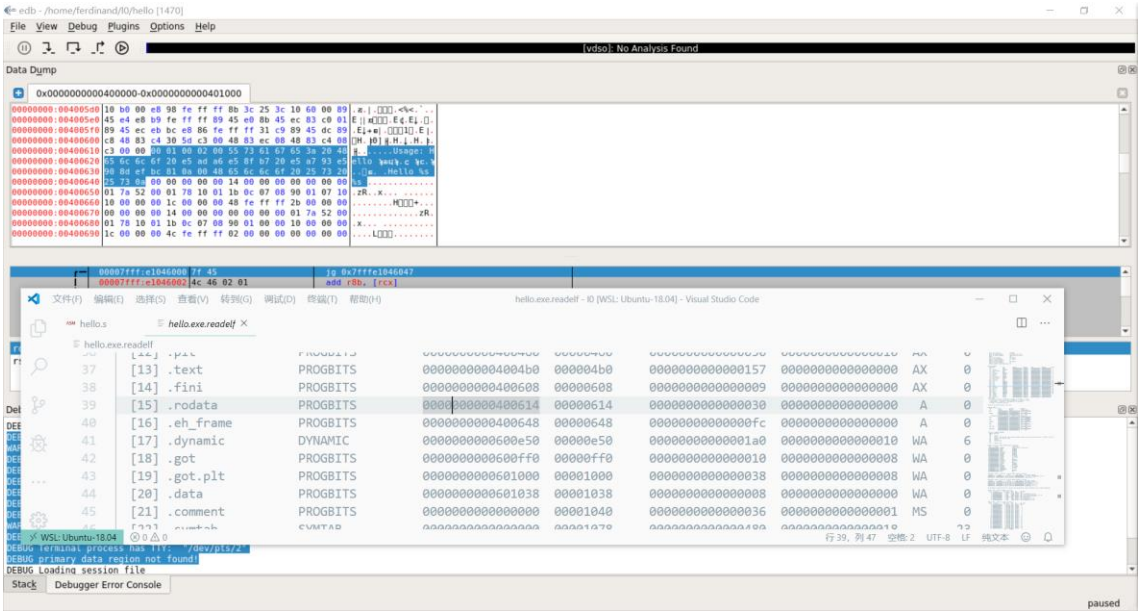
使用 ld 的链接命令，应截图，展示汇编过程！ 注意不只连接 hello.o 文件

5.3 可执行目标文件 hello 的格式

阅读节头表即可获得各段的起始地址(Address)，大小(Size)等信息：



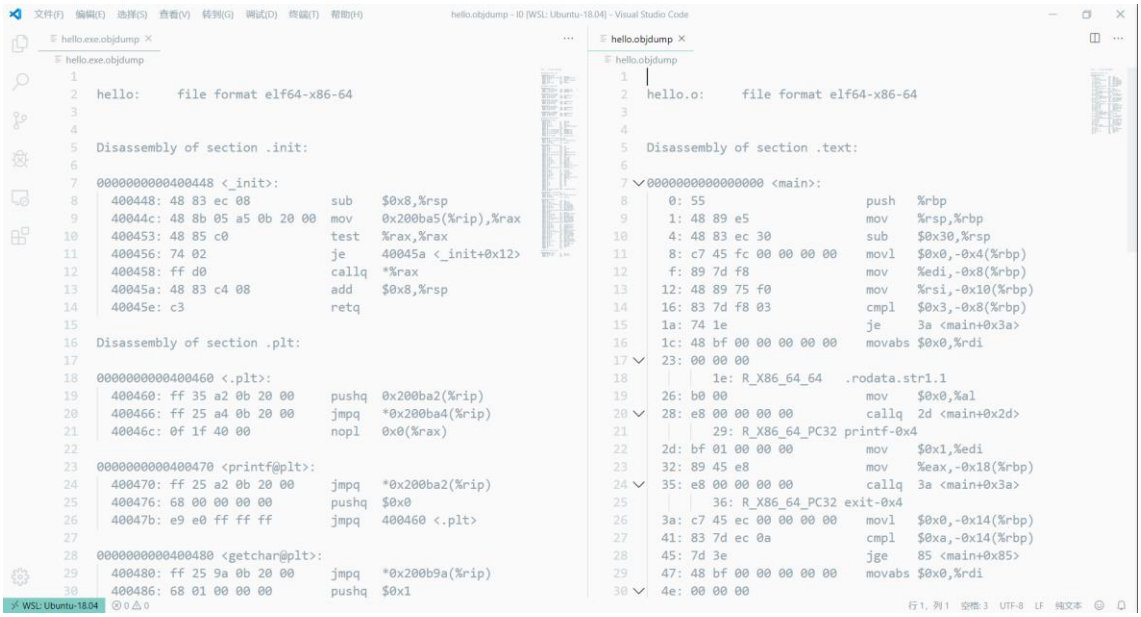
5.4 hello 的虚拟地址空间

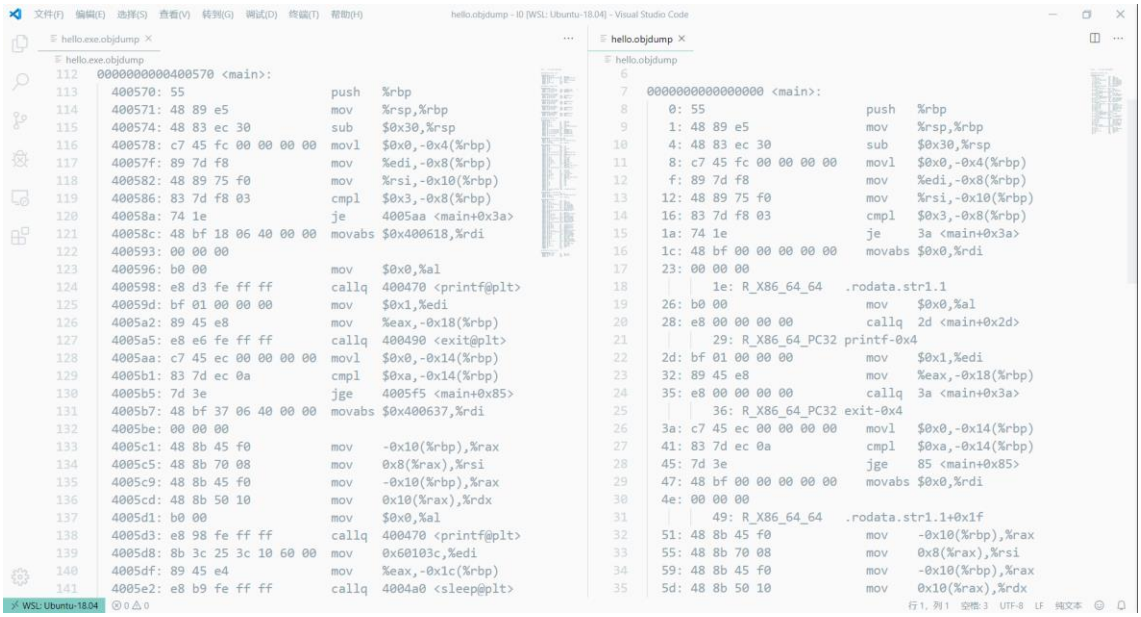


使用 edb 加载 hello，即可查看本进程的虚拟地址空间各段信息；
以最容易分辨的.rodata 段为例，如图，.rodata 段的数据映射到了期待的位置和大小。其它的段依此类推。

5.5 链接的重定位过程分析

5.5.1 反汇编结果分析





如图对比 hello(左)和 hello.o 的反汇编结果。主要区别体现在：

1. hello 中，文件以.init 开始，而非.text 开始
2. 各种地址有所变化
3. 加入了链接进的函数
4. 彻底没有了‘.’开头的伪指令

5.5.2 链接的过程

5.5.2.1 符号解析

在这个阶段，链接器将每个符号引用与各个可重定位目标文件的符号表中的一个确定的符号定义关联起来。

对于那些本地符号引用，符号解析十分容易。然而对于外部引用，链接器需要按照命令行给出的顺序扫描给出的.o 文件和.a 文件。然后如果扫描到最后，还有未被解析的符号，则发生链接错误。

5.5.2.2 重定位

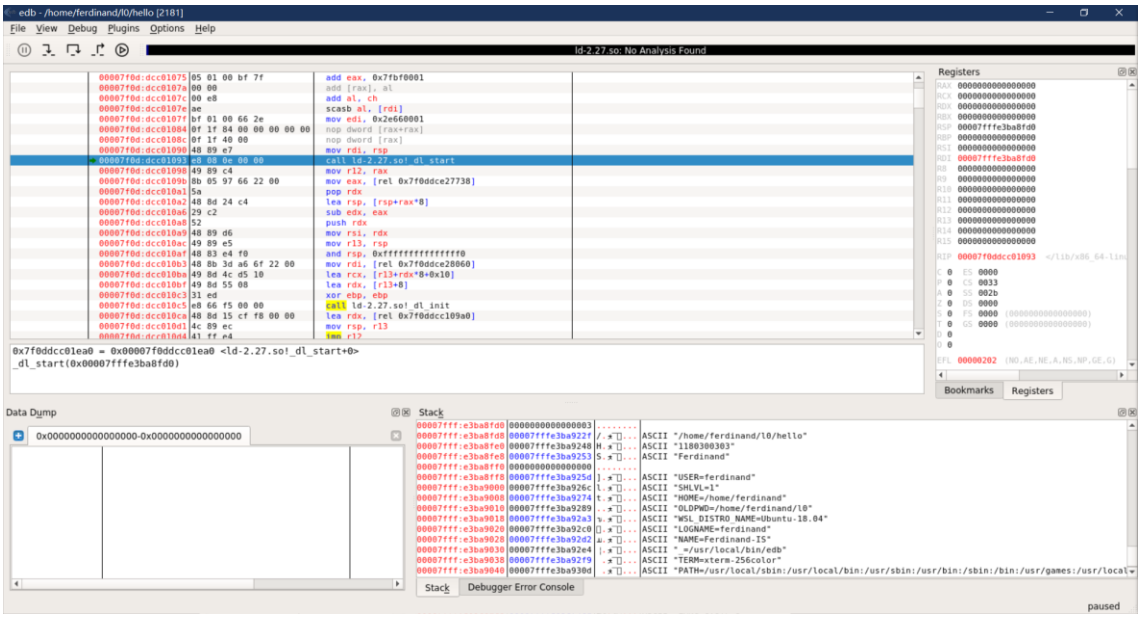
首先将待链接的所有目标文件中相同的节合并成新节，然后，链接器将虚拟内存地址赋予合成的节和每个符号，再对引用符号的位置进行更新(重定位)。

例如：符号.rodata.str1.1 被重定位在$0x400618 的位置。

```
40058c: 48 bf 18 06 40 00 00  movabs $0x400618,%rdi
400593: 00 00 00
400596: b0 00                 mov    $0x0,%al
400598: e8 d3 fe ff ff        callq  400470 <printf@plt>
```

```
1a: 74 1e                     je     3a <main+0x3a>
1c: 48 bf 00 00 00 00 00 00   movabs $0x0,%rdi
23: 00 00 00
    1e: R_X86_64_64   .rodata.str1.1
```

5.6 hello 的执行流程



单步执行 hello，获得如下执行顺序（因为 MSWORD 的拼写检查问题所以截图呈现）

```
19  ld-2.27.so!_dl_start
20  ld-2.27.so!_dl_init
21  hello!_start
22  libc-2.27.so!__libc_start_main
23  libc-2.27.so!__cxa_atexit
24  libc-2.27.so!__libc_csu_init
25  hello!_init
26  libc-2.27.so!_setjmp
27  libc-2.27.so!_sigsetjmp
28  libc-2.27.so!__sigjmp_save
29
30  hello!main
31
32  hello!puts@plt
33  hello!exit@plt
34  hello!printf@plt
35  hello!sleep@plt
36  hello!getchar@plt
37  ld-2.27.so!_dl_runtime_resolve_xsave
38  ld-2.27.so!_dl_fixup
39  ld-2.27.so!_dl_lookup_symbol_x
40  libc-2.27.so!exit
```

5.7 Hello 的动态链接分析

_dl_init 前：

_dl_init 后：

可以看到.got 节(00000000:00600ff0 起始)的数据发生了变化

5.8 本章小结

在本章，我们利用链接器把可重定位的目标文件链接成了一个完整的可执行文件，然后通过 readelf 读取了 ELF 可执行文件的相关信息，再用 objdump 反汇编，并与原可执行目标文件的反汇编文件进行了对比；通过 edb 工具查看虚拟内存映射、执行流程、动态链接过程。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程是一个正在运行的程序实例。是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

进程方便了操作系统对资源的管理。通过上下文切换和虚拟内存，当前的程序程序好像是系统中当前运行的唯一程序一样，流畅地执行着。。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用：

操作系统外壳提供用户与内核的交互接口。它接收用户输入的命令并把它送入内核去执行

Shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。，Shell 也有自己的编程语言用于对命令的编辑，它允许用户编写由 shell 命令组成的程序。

Shell 的处理流程

1. shell 首先检查命令是否是内部命令，
2. 若不是再检查是否是一个应用程序。然后 shell 在 Path 里寻找程序。
3. 如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件，将会显示一条错误信息。
4. 如果能够成功找到命令，该内部命令或应用程序将被分解为系统调用并传给 Linux 内核。

6.3 Hello 的 fork 进程创建过程

当用户通过 shell 启动 hello 时，shell 调用 fork 函数创建一个子进程，该子进程除了 PID 以外，几乎与父进程完全一致，有着完全相同结构但是实际上独立的虚拟内存空间。在 fork 返回的瞬间，这个虚拟空间的内容也完全相同。

在输入./hello 命令后，shell 调用 fork 函数创建一个子进程，之后内核为 hello 进程创建各种数据结构，并分配给它们一个唯一的 PID。同时，内核还将把两个进程的内存各个区域标记为私有的写时复制。

Fork 在父进程与新产生的子进程中都会返回一次，在父进程中返回子进程 PID，在子进程中返回 0.

6.4 Hello 的 execve 过程

Fork 后的子进程会调用 execve 函数，加载并运行 hello；并且将参数列表 argv 和环境变量列表 envp 传递给 hello 的 main 函数。仅当出现错误时，如文件未找到时，execve 才会返回到子进程中。在 execve 加载了 Hello 之后，它调用初始化代码来设置栈，然后将控制传递给 main 函数。

当 main 开始执行时，用户栈的组织结构如图 8-22 所示。让我们从栈底(高地址)往栈顶(低地址)依次看一看。首先是参数和环境字符串。栈往上紧随其后的是以 null 结尾的指针数组，其中每个指针都指向栈中的一个环境变量字符串。全局变量 environ 指向这些指针中的第一个 envp[0]。紧随环境变量数组之后的是以 null 结尾的 argv[ ]数组，其中每个元素都指向栈中的一个参数字符串。在栈的顶部是系统启动函数 libc_start_main(见 7.9 节)的栈帧。

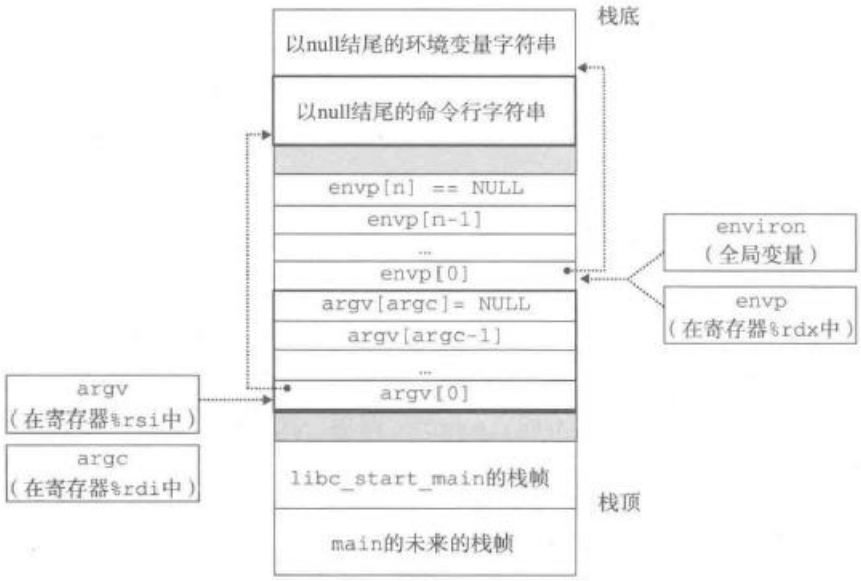


图 8-22 一个新程序开始时，用户栈的典型组织结构

6.5 Hello 的进程执行

上下文信息：

上下文就是内核重新启动一个被抢占的进程所需的状态。它由一些对象的值构成，这些对象包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程已经打开文件信息的文件表。

对于 hello 程序的执行，当用户由其它程序切入 hello 时，先进入内核态，hello 的上下文被恢复，之后才进入用户态运行；切出 hello 时，会先由用户态变为内核态，保存上下文。

调度：在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程，这种决策就叫调度。

进程时间片：

是分时操作系统分配给每个正在运行的进程的一段 CPU 时间。

对于 hello 程序的执行，当用户由其它程序切入 hello 时，操作系统就会分给它时间片。

Hello 运行时：

Hello 在内核态被装入内存，之后运行在用户模式中，当调用 sleep 函数时，内核进行调度执行上下文切换，hello 进入内核模式，内核将上下文控制权交给其他进程，计时器开始计时。当时间到两秒后，定时器发送中断信号到 CPU 引脚，CPU 调用内核程序执行中断处理，又将上下文的控制权交回 hello，hello 继续运行。

6.6 hello 的异常与信号处理

hello 在执行的过程中，共会遇到四种异常：外部设备引起的中断，执行指令导致的陷阱、故障和终止。中断属于异步异常，最常见的是外部设备的 I/O 中断。其它的属于同步异常。陷阱是有意产生的异常，最常见的是系统调用。故障是非有意产生，但是可能被修复的异常，如缺页故障；而终止是非故意产生的不可修复的致命错误。

在发生异常时，内核会发送信号给相应进程。例如程序进行非法内存访问，内核就发送 SIGSEGV 信号给它。常见信号种类和处理方式如下：

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

不停乱按:

第一个回车符以前的输入，被通过中断的方式输入，最终发送给程序。程序执行 getchar()读取这些输入的第一个字符之后退出，然后内核发送 SIGCHLD 给 zsh。此后的输入，则由 zsh 接收，并被视为无效命令。

```
ferdinand@Ferdinand-IS ~/l0> ./hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
adefiguwlsfghosajfihaifpghoeaHello 1180300303 宿梓航
sdafaadafaf
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
adafdaefdaicanfa
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
cfafhopawdmasfpaaf
afnaHello 1180300303 宿梓航
ofnafaja
afHello 1180300303 宿梓航
iaf
afasHello 1180300303 宿梓航
das
ferdinand@Ferdinand-IS ~/l0> adafdaefdaicanfa
zsh: command not found: adafdaefdaicanfa
ferdinand@Ferdinand-IS ~/l0> cfafhopawdmasfpaaf
zsh: command not found: cfafhopawdmasfpaaf
ferdinand@Ferdinand-IS ~/l0> afnaofnafaja
zsh: command not found: afnaofnafaja
ferdinand@Ferdinand-IS ~/l0> afiaf
zsh: command not found: afiaf
ferdinand@Ferdinand-IS ~/l0> afasdas
zsh: command not found: afasdas
```

回车：

回车符通过中断的方式输入，最终发送给程序。程序执行 getchar()读取这回车符之后退出，然后内核发送 SIGCHLD 给 zsh。

```
ferdinand@Ferdinand-IS ~/10> ./hello 1180300303 宿梓航
Hello 1180300303 宿梓航

Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
```

^C：

键盘中断信号 SIGINT 被内核发送给 hello，hello 收到后立即终止。之后 zsh 受到 SIGCHLD 信号。

```
ferdinand@Ferdinand-IS ~/10> ./hello 1180300303 宿梓航
Hello 1180300303 宿梓航
^C
```

^Z:

键盘发送 SIGTSTP 信号给 hello，hello 收到后立即停止，之后 zsh 受到 SIGCHLD 信号，更新 hello 状态为挂起并显示。

```
ferdinand@Ferdinand-IS ~/10> ./hello 1180300303 宿梓航
Hello 1180300303 宿梓航
^Z
[1]  + 3253 suspended  ./hello 1180300303 宿梓航
```

ps:

ps 查看当前执行的进程和开始时间。

```
ferdinand@Ferdinand-IS ~/10> ps
  PID TTY          TIME CMD
    7 tty1     00:00:00 zsh
24442 tty1     00:00:01 zsh
25422 tty1     00:00:00 dbus-launch
 2256 tty1     00:00:00 zsh
 2514 tty1     00:00:00 zsh
 3253 tty1     00:00:00 hello
 3313 tty1     00:00:00 ps
```

jobs:

jobs 查看当前挂起的进程。

```
ferdinand@Ferdinand-IS ~/10> jobs
[1]  + suspended  ./hello 1180300303 宿梓航
```

pstree:

查看进程树

```
ferdinand@Ferdinand-IS ~/10> pstree
init─┬─dbus-daemon
     ├─dbus-launch
     ├─hud-service───{hud-service}
     ├─init───zsh───zsh───zsh───zsh─┬─hello
     │                              └─pstree
     ├─init───sh───sh───sh───node─┬─node───10*[{node}]
     │                            ├─node─┬─Microsoft.VSCod───21*[{Microsoft.VSCod}]
     │                            │      ├─2*[zsh]
     │                            │      └─16*[{node}]
     │                            └─11*[{node}]
     └─{init}
```

Fg,^Z,kill:

Fg 指令发送 SIGCONT 信号给 hello 使得 hello 继续在前台执行

^Z 则发送 SIGTSTP 将 hello 再次挂起

最后 kill 发送 SIGKILL 终止 hello

```
ferdinand@Ferdinand-IS ~/10> fg
[1]  + 3253 continued  ./hello 1180300303 宿梓航
Hello 1180300303 宿梓航
Hello 1180300303 宿梓航
^Z
[1]  + 3253 suspended  ./hello 1180300303 宿梓航
ferdinand@Ferdinand-IS ~/10> kill
kill: not enough arguments
ferdinand@Ferdinand-IS ~/10> kill 3253
[1]  + 3253 terminated  ./hello 1180300303 宿梓航
```

6.7 本章小结

本阶段通过在 shell 中运行 hello，然后在运行 hello 的过程中执行了各种操作，回顾了与操作系统相关的如信号、异常等知识，计算机硬件、操作系统（外壳和内核）以及应用程序之间的配合和协作的形式。

（第 6 章 1 分）

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

a) 逻辑地址是机器语言中指向一个操作数或一条指令的地址。每一个逻辑地址都由一个段和偏移量组成，偏移量指明了从段开始的地方到实际地址之间的距离。在 hello 中，就是 hello 里各个元素的相对偏移地址。

b) 线性地址是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，再加上相应段的基地址就生成了一个线性地址。对于 hello 而言，就是 hello 中的虚拟内存地址。

c) 虚拟地址是包含虚拟内存的操作系统中，对应一个虚拟内存单元的地址。在 hello 中，虚拟地址就是线性地址。

d) 物理地址是对应在物理储存器中储存位置的地址。hello 运行时的地址翻译会将 hello 的一个虚拟地址转化为物理地址。

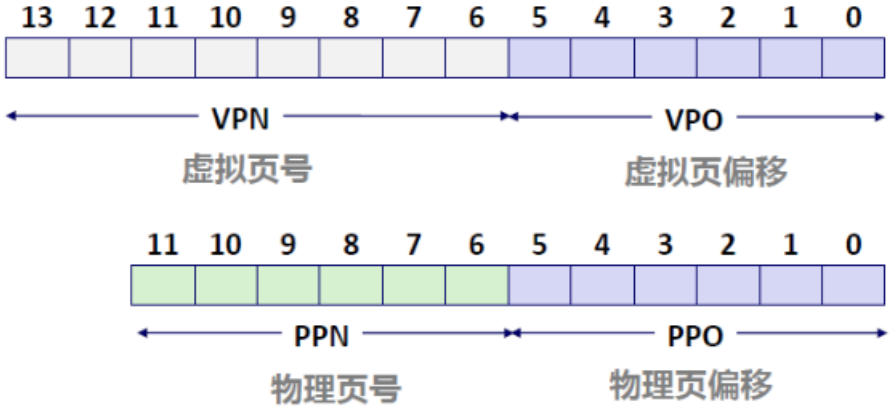
7.2 Intel 逻辑地址到线性地址的变换-段式管理

一个逻辑地址是由一个段标识符加上一个指定段内相对地址的偏移量构成，表示为[段标识符：段内偏移量]。先根据段标识符最低位，判断当前要转换的时是全局段描述符表中的段，还是局部段描述符表中的段；再根据相应寄存器得到其地址和大小。之后再用段选择符中剩余前 13 位的索引在相应的数组中查找到对应的段描述符，这样就找到了它的基地址。基地址 + 段内偏移量就是要转换成的线性地址。

7.3 Hello 的线性地址到物理地址的变换-页式管理

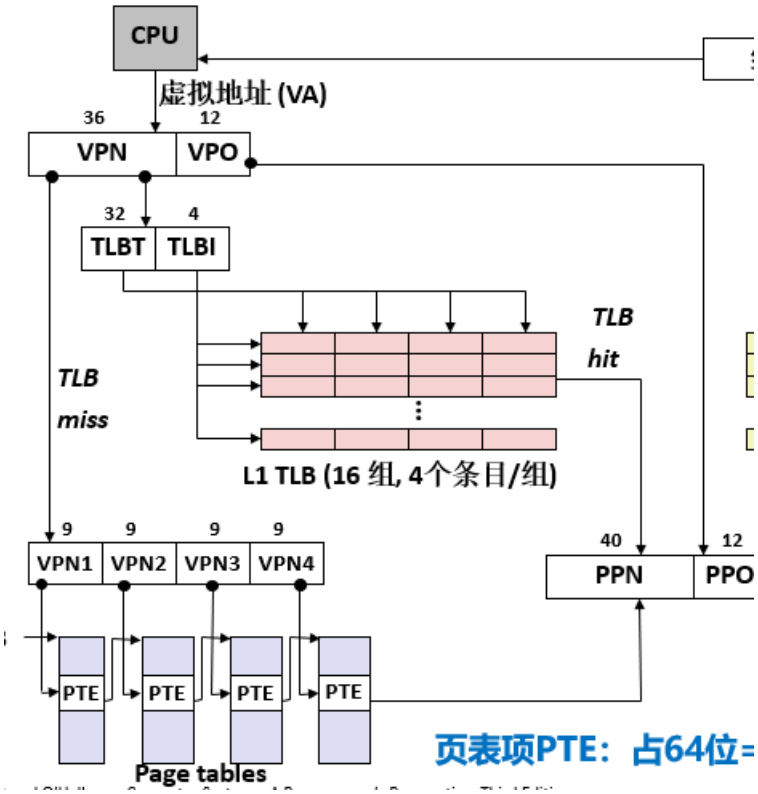
无论是虚拟地址空间，还是物理地址空间，地址们都被组织成为具有相同大小的一些页。每个虚拟页具有虚拟页码 VPN，物理页具有物理页码 PPN，通过储存 VPN 到 PPN 的映射，而页内偏移 VPO 和 PPO 相同，即可完成线性地址到物理地址的变换。

- 14位虚拟地址（n=14）
- 12位物理地址（m = 12）
- 页面大小64字节（P = 64）



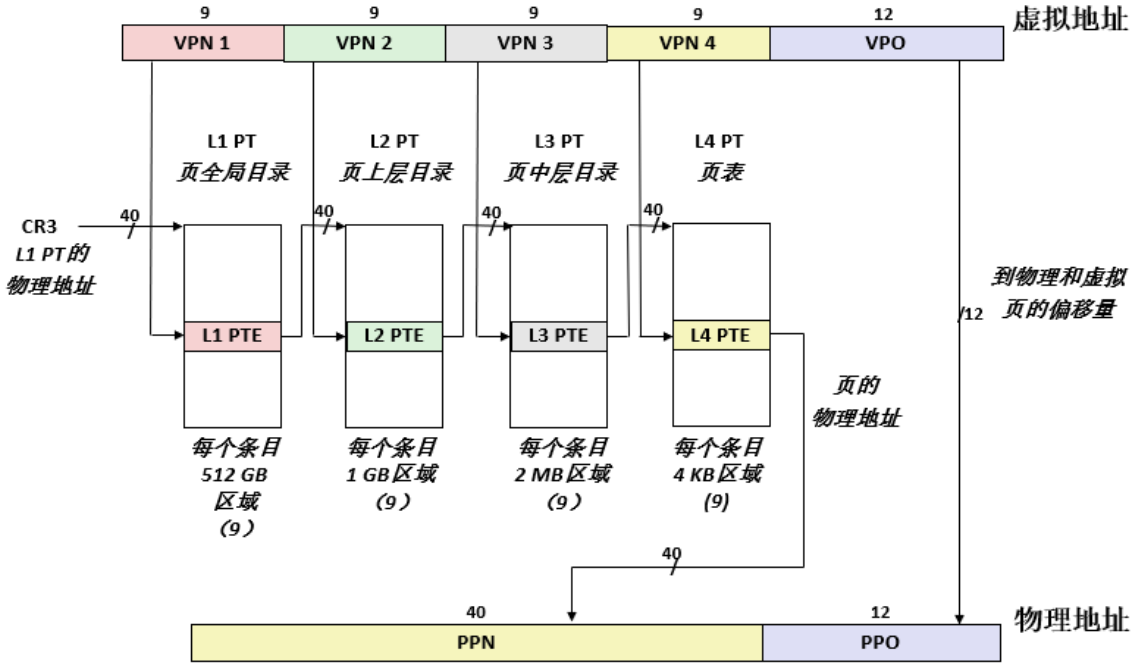
7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

由于 VPO=PPO，VPO 部分立即被送往 PPO 部分；
之后 36 位的 PPN 分别被送往 TLB 和页表；
在 TLB 处，后四位 TLBI 选定组索引，然后再匹配 32 位的 TLBT 标记。如果 TLB 命中，则直接传送 PPN 过去。



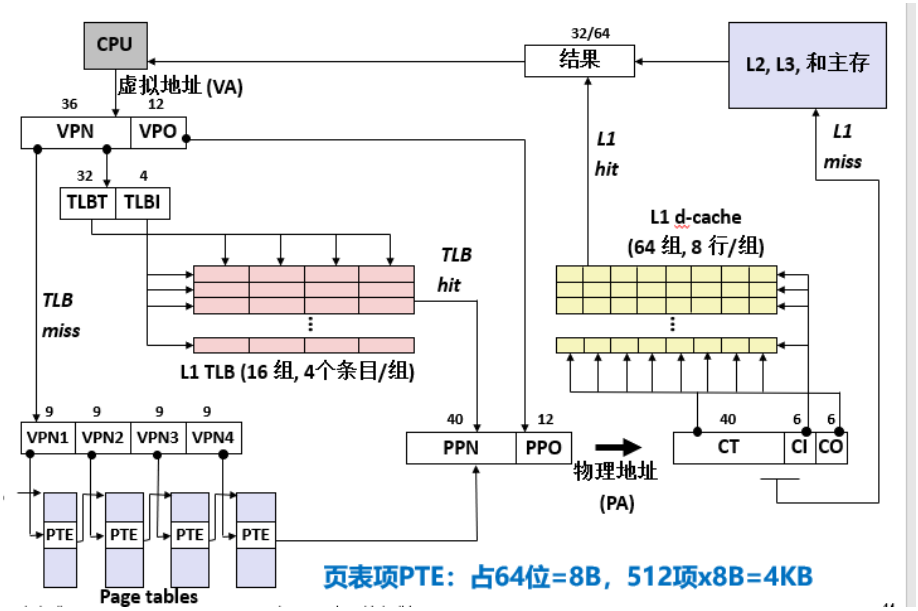
否则，在四级页表中，VPN 被分为四组各九位；利用 VPN1 在页全局目录中找到

对应页上层目录的位置，然后 VPN2 在页上层目录中找到页中层目录位置，VPN3 在页中层目录中找到页表的位置，最后再用 VPN4 匹配页表得到 PPN。
PPN 和 PPO 合起来即为 PA，由此完成了 VA->PA 的转换。



7.5 三级 Cache 支持下的物理内存访问

程序需要访存时，CPU 会先生成一个虚拟地址，然后通过 TLB 和页表等经过地址翻译后，转化为物理地址；再根据物理地址访问 L1 Cache 请求数据，若发生不命中，则向下一级缓存中请求数据(L2 ->L3->主存)。



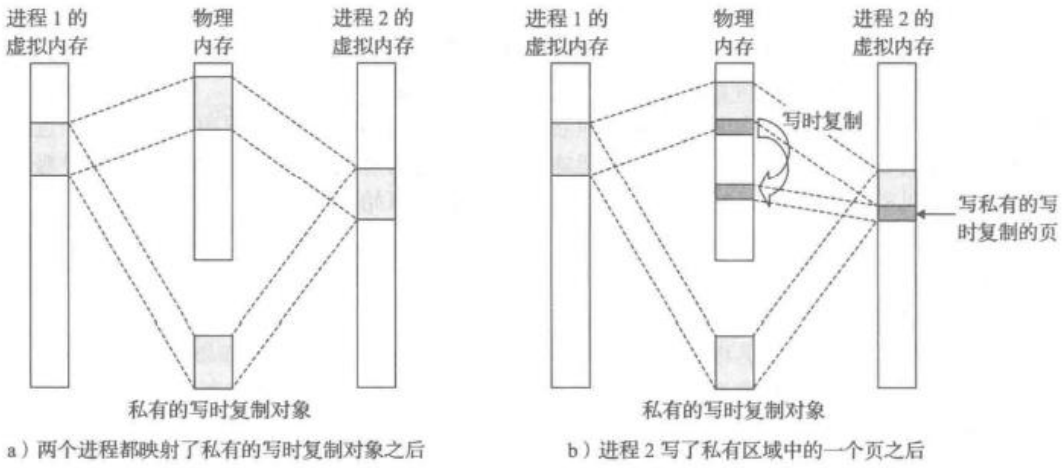
对于某一级特定的 Cache，其被分为若干的组；每一组内有若干行，每行储存有效位、标记和数据块。

物理地址被分为 CI 组索引，CT 标记和 CO 块偏移三部分。向 Cache 请求时，先用 CI 找到对应组，然后使用 CT 在组内比对；如果比对成功且有效位为有效，则缓存命中，用块偏移取出数据；否则前往下一级缓存/内存进行请求。

7.6 hello 进程 fork 时的内存映射

当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的 PID。为了给这个新进程创建虚拟内存，它创建了当前进程的 mm_struct.区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将两个进程中的每个区域结构都标记为私有的写时复制。

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



对于 hello 而言，被 fork 的是当前系统的外壳。

7.7 hello 进程 execve 时的内存映射

虚拟内存和内存映射在将程序加载到内存的过程中也扮演着关键的角色。既然已经理解了这些概念，我们就能够理解 execve 函数实际上是如何加载和执行程序的。hello 进程 execve 时实际上是被 fork 的 shell 程序执行了如下的 execve 调用:execve("hello", NULL, NULL);

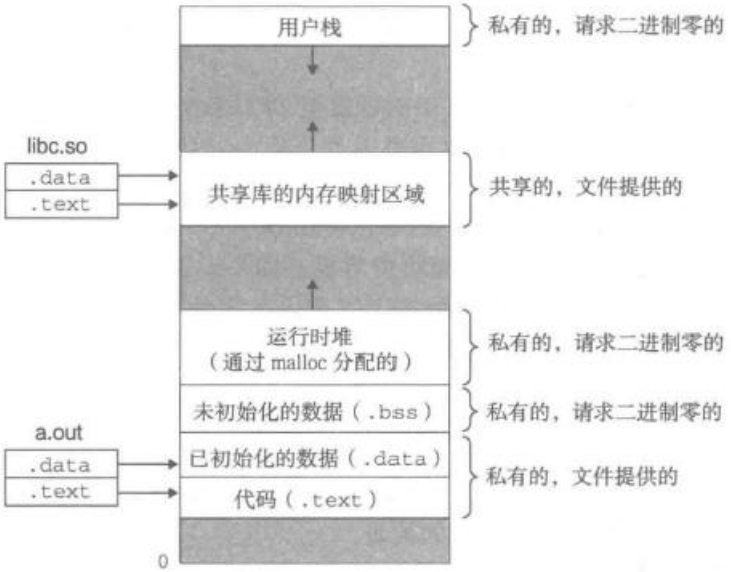
而 execve 函数在当前进程中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序。加载并运行 hello 需要以下几个步骤:

1. 删除已存在的用户区域。删除当前进程虚拟地址的用户部分中的已存在的区域结构。

2. 映射私有区域。为新程序的代码、数据、bss 和栈区域创建新的区域结构。所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 hello 文件中的. text 和.data 区。bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中。栈和堆区域也是请求二进制零的，初始长度为零。图 9-31 概括了私有区域的不同映射。

3. 映射共享区域。如果 hello 程序与共享对象(或目标)链接，比如标准 C 库 libc.so，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

4. 设置程序计数器(PC)。execve 做的最后一件事情就是设置当前进程上下文中的程序计数器，使之指向代码区域的入口点。
下一次调度这个进程时，它将从这个人口点开始执行。Linux 将根据需要换入代码和数据页面。



7.8 缺页故障与缺页中断处理

缺页即 DRAM 缓存不命中。当程序读/写虚拟内存中的一个字，但是字所属的虚拟页并未缓存在物理内存中，就会触发一个缺页故障。

尽管页面命中完全是由硬件来处理的，处理缺页却要求硬件和操作系统内核协作完成。

当 MMU 在页表中找到 PTE 时，却 PTE 中的有效位是零，所以 MMU 触发了

一次异常，传递 CPU 中的控制到操作系统内核中的缺页异常处理程序。缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘。之后，缺页处理程序页面调人新的页面，并更新内存中的 PTE。缺页处理程序返回到原来的进程，再次执行导致缺页的指令。CPU 将引起缺页的虚拟地址重新发送给 MMU。因为虚拟页面现在缓存在物理内存中，所以就会命中，在 MMU 执行了访存步骤之后，主存就会将所请求字返回给处理器。

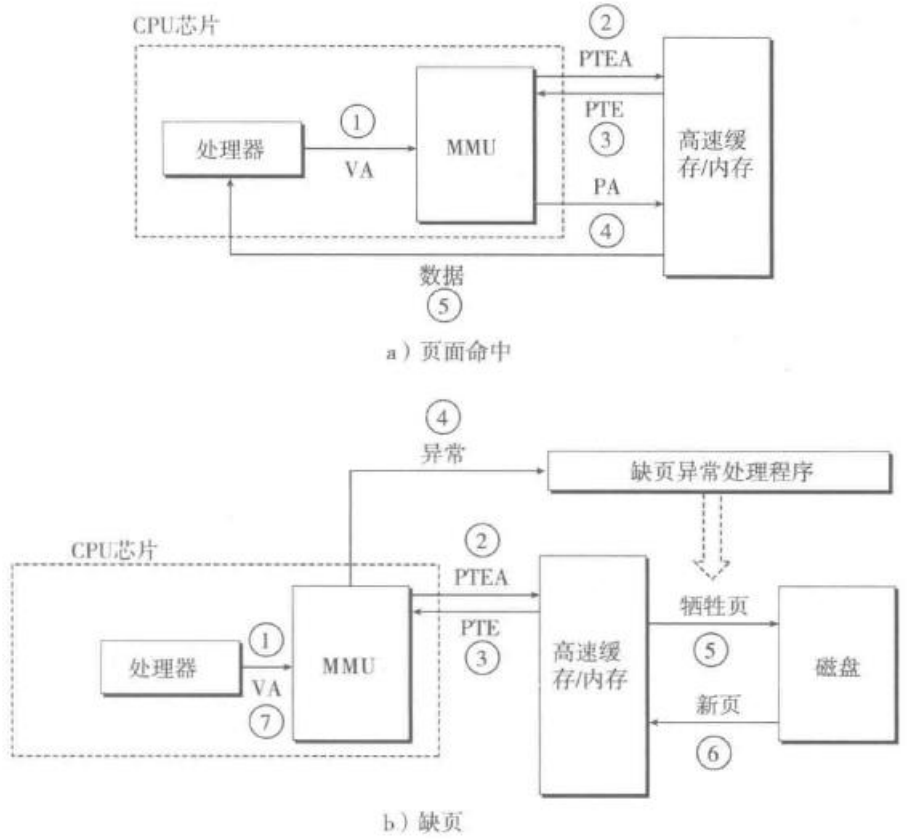


图 9-13 页面命中和缺页的操作图（VA：虚拟地址。PTEA：页表条目地址。PTE：页表条目。PA：物理地址）

7.9 动态存储分配管理

***基本方法*:**

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长(向更高的地址)。对于每个进程，内核维护着一个变量 brk，它指向堆的顶部。

分配器将堆视为一组不同大小的块(block)的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用

程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

***策略*:**

带边界标签的隐式空闲链表分配器中，一个块是由一个字的头部(包含大小和是否被占用的信息)、有效载荷、可能的一些额外的填充，以及一个字大小、和头部一样的尾部。

因为采用八字节对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内高位来储存块大小，最低位来记录占用状态。

头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，用于满足对齐要求。

因为空闲块是通过头部中的大小字段隐含地连接着的，可以视为一个隐式空闲链表。分配器可以通过遍历堆中所有的块，间接地遍历整个空闲块的集合。

通过遍历，我们可以对块进行回收、合并、分配。适配策略还细分为首次适配、第二次适配和最佳适配。

7.10 本章小结

本章我们从 hello 的储存器地址空间起，探究了 intel 的段式管理方式、hello 的页式管理方式，综合后分析了虚拟地址到物理地址的转换过程以及物理内存的访问方式，由此分析出 hello 进程 fork 时的内存映射、execve 时的内存映射，并了解了缺页故障及其处理和动态存储分配的基本方法与策略。

（第 7 章 2 分）

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

Linux 将所有的 IO 设备都模型化为文件，设备的读写被映射为文件的读写，这提供了一种称为 UNIX IO 接口的设备管理办法。

8.2 简述 Unix IO 接口及其函数

设备被 Unix IO 接口被映射为文件，因此所有的 IO 操作都能以一种统一方式被执行。

打开文件

一个应用程序通过要求内核打开相应的文件，来宣告它想要访问一个 IO 设备，内核返回一个小的非负整数，叫做文件描述符，它被用于在所有此后的操作中引用这个文件，有关这个打开文件的所有信息则由内核记录。应用程序只需记录这个描述符即可。

被 Shell 创建的每个进程开始时都有三个打开的文件，即 stdin=0，stdout=1 和 stderr=2。另外，头文件<unistd.h>定义了常量 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO，它们可以被用来代替显式的描述符值。

在 Unix IO 接口中，进程可以调用 open 函数来打开一个存在的文件或者创建一个新文件：

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

filename 是要打开的文件名，

flags 参数指明了进程打算如何访问这个文件，

mode 参数指定了新文件的访问权限位。

open 函数将 filename 打开，然后返回它的描述符。返回的描述符总是在进程中当前没有打开的最小描述符。

作为上下文的一部分，每个进程都有一个 umask，它是通过调用 umask 函数来设置的。当进程通过带某个 mode 参数的 open 函数调用来创建一个新文件时，文件的访问权限位被设置成 mode&~umask。

改变当前的文件流位置

对于每个打开的文件流，内核维护着一个初始为 0 的文件位置，这个文件位置就是从文件起始的字节偏移量，应用程序能够通过调用 seek 函数，显式地将改变当前文件流位置。

读文件

一个读操作就是从文件复制 n>0 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 k+n。如果文件大小只有 m 字节，但是 k>=m 时，读操作会得到一个 EOF，应用程序能检测到 EOF。但是在文件结尾处并没有明确的“EOF 符号”。

程序通过调用 read 函数来执行输入的。函数声明如下：

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

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

写文件

一个写操作就是从内存中复制 n>0 个字节到一个文件，从当前文件位置 k 开始，然后更新 k。

程序通过调用 write 函数来执行输出。函数声明如下：

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

write 函数从内存位置 buf 复制至多 n 个字节到描述符为 fd 的当前文件位置

关闭文件

当程序完成了对文件(IO 设备)的访问之后，它就通知内核关闭这个文件，关闭文件时，内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。当一个进程终止时，内核总会关闭所有打开的文件并释放它们的内存资源。

程序可以通过调用 close 函数关闭一个打开的文件。函数声明如下：

int close(int fd);

fd 是要关闭的文件的描述符。

关闭一个已关闭的描述符会出错。文件关闭成功返回 0，若出错则返回-1.

8.3 printf 的实现分析

首先观察 printf 函数的函数体

```
int printf(const char *fmt, ...)
{
    int i;
    char buf[256];

    va_list arg = (va_list)((char *)(&fmt) + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
```

```
    return i;
}
```

在形参列表里含有一个‘...’，是可变形参的一种写法。当传递参数的个数不确定时，就可以用这种方式来表示。那么显然，我们需要一种方法，使得函数体可以知道具体调用时参数的个数。

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

其中

```
typedef char *va_list;
```

而`((char *)(&fmt) + 4)`表示的是...中的第一个参数。

再看 `i = vsprintf(buf, fmt, arg);`

其中 vsprintf 接受一个格式化的命令，并把指定的匹配的参数格式化输出到 buf，然后返回字符串的长度。之后再调用

```
write(buf, i);
```

把长度为 i 的 buf 字符串输出到 stdout

对于 write(buf, i);的实现，

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

可知，write 将栈中参数放入寄存器，ecx 是字符个数，ebx 存放第一个字符地址，int INT_VECTOR_SYS_CALLA 代表通过系统调用 syscall。

下面查看 syscall 的实现

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

syscall 将字符串中的的每一个字节从内存中通过总线复制到显存中(以 ASCII 方式

储存)，之后调用字符显示驱动子程序将通过 ASCII 码在字模库中找到点阵信息将点阵信息存储到 vram 中。最后，显卡会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

8.4 getchar 的实现分析

当键盘某一个键被按下时，就会产生一个中断信号到 CPU 的中断引脚，之后 CPU 调用中断处理子程序。接受按键扫描码转成 ascii 码，保存到系统的键盘缓冲区中。

当程序调用 getchar 时，它会调用 read 系统函数，之后通过 systemcall 读取按键 ascii 码，但是直到接受到回车键才返回。

8.5 本章小结

本章通过研究 hello.c 中调用的 IO 函数，研究对应的 UNIX IO，了解了 UNIX IO 接口的工作方式。

（第 8 章 1 分）

结论

1. hello.c 的代码被通过 IO 设备输入计算机，以文件的方式被储存。
2. 使用预处理器，将 hello.c 转化为 hello.i;
3. 使用编译器，将 hello.i 转化为 hello.s
4. 使用汇编器，将 hello.s 转化为可重定位的目标文件 hello.o
5. 最后利用链接器 ld 将 hello.o 和各种库链接成为可执行文件 hello
6. 我们在 shell 中输入命令后，shell 先 fork 出一个子进程，然后通过 exceve 加载并运行 hello
7. 在一个时间片中，hello 有自己的 CPU 资源，顺序执行逻辑控制流以及虚拟内存空间。在切出或者切回 hello 时，内核利用上下文切换完成调度。
8. hello 运行时需要访存，那么 hello 程序生成的虚拟地址通过 TLB 和四级页表翻译为物理地址，之后再通过三级 Cache 来获得内存中的信息。
9. hello 在运行时，也会有异常和信号，如^C,^Z 等
10. hello 调用的 printf 函数会调用 malloc 通过动态内存分配器申请堆中的内存。Printf 还会调用 write，实现输出。
11. 当 hello 结束执行时，Shell 父进程会回收 hello 子进程，然后内核删除为 hello 创建的所有数据结构，并关闭打开的文件。

通过对计算机系统的设计与实现的学习，了解了前人的智慧，我期待计算机系统能够在未来进一步完善，如融入现代的大数据、人工智能等思想，完成

系统管理的进一步优化；为系统的各种功能提供更高级的封装，使之更易被使用等等。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

列出所有的中间产物的文件名，并予以说明起作用。

hello.c	hello 的 c 源文件
hello.i	预处理后的源代码
hello.s	编译生成的汇编程序
hello.o	汇编生成的可重定位目标文件
hello.objdump	hello.o 用 objdump 处理的输出
hello.readelf	hello.o 用 readelf 处理的输出
hello	链接生成的可执行文件
hello.exe.objdump	hello 用 objdump 处理的输出
hello.exe.readelf	hello 用 readelf 处理的输出

（附件 0 分，缺失 -1 分）

参考文献

为完成本次大作业你翻阅的书籍与网站等

[1] Randal E. Bryant / David O' Halloran. Computer Systems: A Programmer's Perspective (3rd Edition) [M]. 北京：机械工业出版社，2016-11 ISBN: 9787111544937.

[2] eteran. edb-debugger[CP] https://github.com/eteran/edb-debugger

[3] ThinkOver33. Win10 安装 Ubuntu 子系统及图形化界面详细教程[OL]. https://blog.csdn.net/daybreak222/article/details/87968078

[4] pianist. printf 函数实现的深入剖析[OL]. https://www.cnblogs.com/pianist/p/3315801.html

[5] Kernighan, Brian; Pike, Rob (1984). The UNIX Programming Environment. Englewood Cliffs: Prentice Hall. p. 200.

[6] cppreference. Cstdio reference[OL] http://www.cplusplus.com/reference/cstdio/

（参考文献 0 分，缺失 -1 分）