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　计算机科学与技术

学　　号　1170301024

班　　级　1703010

学　　生　宋赟祖

指 导 教 师　史先俊

计算机科学与技术学院

2018 年 12 月

摘　要

计算机系统是由硬件和软件组成，他们共同来运行应用程序，hello 的一生就经历了硬件和软件，我们通过软件来编写软件，通过预处理器，编译器，汇编器和链接器等软件将 hello 编译成可执行文件，然后在软甲你 shell 中键入命令，shell 通过系统函数将 hello 加载在内存中。这样软件大部分操作就结束了，接下来的操作交给硬件，CPU，cache， memory，MMU， TLB，等实现了 hello 的硬件过程。然后 hello 可能会遇到有硬件和软件协同完成的异常等。因为硬件和软件的协同操作，才使得我们的 hello 程序能够在计算机跑出来，hello 的一生坎坷多难，接下来我们陪 hello 一起看看。

关键词：hello， 预处理，编译，汇编，链接，虚拟内存，进程，可重定位文件，可执行文件，unixI/O

（摘要 0 分，缺失-1 分，根据内容精彩称都酌情加分 0-1 分）

目 录

第 1 章 概述

1.1 Hello 简介

根据 Hello 的自白，利用计算机系统的术语，简述 Hello 的 P2P，020 的整个过程。

P2P 过程：hello 程序的生命周期是从一个高级 C 语言程序开始的，因为这种形式能够被人读懂，为了在系统上运行 hello.c 程序，每条 C 语句都必须通过被预处理器，编译器，汇编器、链接器转化为一系列的低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打好包，并以二进制磁盘的形式存放起来，当我们在 shell 上键入运行 hello 程序时，shell 进程会通过专门的系统调用来执行我们的请求即创建一个 hello 程序的子进程及其上下文。每个进程都有独立的虚拟内存空间，每个进程的虚拟地址空间由大量准确定义的区构，。每个区都有专门的功能。然后内核将控制交给 hello 进程，在 hello 运行的过程中，会出现很多的异常，控制在用户模式和内核模式不停地切换，并且控制也在不同的进程中来回切换。当 hello 结束终止时，shell 会回收僵死的 hello 进程，删除 hello 在内核中的所有信息，hello 程序生命周期结束。

020 过程：hello 终止之后，shell 会对它进行回收，shell 删除 hello 在内核中的进程信息，并且删除 hello 在 shell 中的任何信息。Hello 程序之后仅存在与硬盘之中，内存没有它的任何信息。

1.2 环境与工具

X64 CPU；2GHz；2G RAM；256GHD Disk
Windows10 64 位；Vmware 14；Ubuntu 18.04 LTS
Ubuntu 18.04 LTS、Vim

1.3 中间结果

Hello.c（c 程序）， hello.i（预处理文件），hello.s（汇编文件）， hello.o（可重定位文件），hello（可执行文件）

1.4 本章小结

本章讲述了 hello 的一生，对 hello 的整个生命周期有个总体的理解，并且介绍了

探索 hello 进程我们所必须的工具。

（第 1 章 0.5 分）

第 2 章 预处理

2.1 预处理的概念与作用

预处理的概念：C 预处理器在程序执行之前查看程序，故称为预处理器在 C 语言的程序中可包括各种以符号#开头的编译指令，这些指令称为预处理命令。预处理命令属于 C 语言编译器，而不是 C 语言的组成部分。

预处理的作用：根据程序中的预处理指令，预处理器把符号缩写替换成其表示的内容。预处理器可以包含程序所需要的其他文件，可以选择的让编译器查看那些代码。预处理器并不知道 C。基本上它的工作是把一些文本转换为另一些文本。通过预处理命令可扩展 C 语言程序设计的环境。

2.2 在 Ubuntu 下预处理的命令

cpp -v hello.c hello.i

```
syz170301024@ubuntu:/mnt/hgfs/hitics/work$ cpp -v hello.c hello.i
Using built-in specs.
COLLECT_GCC=cpp
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.3.0-27ubuntu1~18.04' --with-bugurl=f
ile:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --
prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --en
able-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-thread
s=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug
 --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vt
able-verify --enable-libmpx --enable-plugin --enable-default-pie --with-system-zlib --with-target-sys
tem-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64
 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvp
tx-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-
gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)
COLLECT_GCC_OPTIONS='-E' '-v' '-o' 'hello.i' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/7/cc1 -E -quiet -v -imultiarch x86_64-linux-gnu hello.c -o hello.i -mt
une=generic -march=x86-64 -fstack-protector-strong -Wformat -Wformat-security
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/7/include
 /usr/local/include
 /usr/lib/gcc/x86_64-linux-gnu/7/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-l
inux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-linux-g
nu/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86
_64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-E' '-v' '-o' 'hello.i' '-mtune=generic' '-march=x86-64'
```

2.3 Hello 的预处理结果解析

部分 hello.i 文件信息：

```
# 2 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello.c"




# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4




# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 34 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
```

hello.i 中 stdio.h 文件中插入的 FILE 文件的别名操作：

```
typedef struct _IO_FILE FILE;
# 38 "/usr/include/stdio.h" 2 3 4
```

hello.i 中从 stdio.h 文件中插入的文件结构体的结构：

```
# 245 "/usr/include/x86_64-linux-gnu/bits/libio.h" 3 4
struct _IO_FILE {
  int _flags;



  char* _IO_read_ptr;
  char* _IO_read_end;
  char* _IO_read_base;
  char* _IO_write_base;
  char* _IO_write_ptr;
  char* _IO_write_end;
  char* _IO_buf_base;
  char* _IO_buf_end;

  char *_IO_save_base;
  char *_IO_backup_base;
  char *_IO_save_end;

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;



  int _flags2;

  __off_t _old_offset;


  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];
```

分析：预处理器根据源码中字符#开头的命令，修改原始的 C 程序，比如说 hello 中第一行的#include <stdio.h>命令告诉预处理器读取系统头文件 stdio.h 中的内容并把它直接插入到源码程序文本中，上面三图是将 stdio.h 插入到 hello 中部分信息的截图。还有#define 的预处理命令，预处理器在程序中查找宏的实例

后，就会用替换体代替该宏。除此之外预处理中还处理了其他的一些预处理指令比如#undef、#ifdef、#else、#endif。

2.4 本章小结

C 预处理器是 C 语言很重要的附件，C 预处理器遵循预处理指令，在编译源代码之前调整代码，通过预处理器可以控制编译过程、列出要替换的内容、指明要编译的代码行和影响编译器其他方面的行为。

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

编译的概念编译阶段是对预编译后的.i 文件编译，生成汇编代码的.s 文件的过程，生成的汇编语言的每条语句都以一种标准的文本格式确切的描述了一条低级机器语言指令。

编译的作用：这个阶段编译器主要做词法分析、语法分析、语义分析等，以确定代码的实际要做的工作，在检查无误后，编译器把代码翻译成汇编语言，将高级语言程序转化为低级语言程序，它为不同高级语言的不同编译器提供了通用的输出语言，因为编译操作也是从高级语言到机器语言的过渡操作，汇编语言相对于高级语言来说更加接近于机器。

3.2 在 Ubuntu 下编译的命令

gcc -m64 -no-pie -fno-PIC -v -S hello.i -o hello.s

截图如下：

```
syz170301024@ubuntu:~/hitics/work$ gcc -m64 -no-pie -fno-PIC  -v -S hello.i -o hello.s
Using built-in specs.
COLLECT_GCC=gcc
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.3.0-27ubuntu1~18.04' --with-bugu
rl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,ob
j-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-lin
ux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext
 --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --en
able-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-un
ique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-s
ystem-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --
with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=
generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --bui
ld=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)
COLLECT_GCC_OPTIONS='-m64' '-no-pie' '-fno-PIC' '-v' '-S' '-o' 'hello.s' '-mtune=generic' '-march
=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/7/cc1 -fpreprocessed hello.i -quiet -dumpbase hello.i -m64 -mtune=
generic -march=x86-64 -auxbase-strip hello.s -version -fno-PIC -o hello.s -fstack-protector-stron
g -Wformat -Wformat-security
GNU C11 (Ubuntu 7.3.0-27ubuntu1~18.04) version 7.3.0 (x86_64-linux-gnu)
        compiled by GNU C version 7.3.0, GMP version 6.1.2, MPFR version 4.0.1, MPC version 1.1.0
, isl version isl-0.19-GMP

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
GNU C11 (Ubuntu 7.3.0-27ubuntu1~18.04) version 7.3.0 (x86_64-linux-gnu)
        compiled by GNU C version 7.3.0, GMP version 6.1.2, MPFR version 4.0.1, MPC version 1.1.0
, isl version isl-0.19-GMP

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: c8081a99abb72bbfd9129549110a350c
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_
64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-lin
ux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr
/lib/x86_64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-m64' '-no-pie' '-fno-PIC' '-v' '-S' '-o' 'hello.s' '-mtune=generic' '-march
=x86-64'
```

3.3 Hello 的编译结果解析

```
        .file   "hello.c"
        .text
        .globl  sleepsecs
        .data
        .align 4
        .type   sleepsecs, @object
        .size   sleepsecs, 4
sleepsecs:
        .long   2
        .section        .rodata
        .align 8
.LC0:
        .string "Usage: Hello 1170301024 \345\256\213\350\265\237\347\245\226\357\274\201"
.LC1:
        .string "Hello %s %s\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB5:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        cmpl    $3, -20(%rbp)
        je      .L2
        movl    $.LC0, %edi
        call    puts
        movl    $1, %edi
        call    exit
.L2:
        movl    $0, -4(%rbp)
        jmp     .L3
.L4:
        movq    -32(%rbp), %rax
        addq    $16, %rax
        movq    (%rax), %rdx
        movq    -32(%rbp), %rax
        addq    $8, %rax
        movq    (%rax), %rax
        movq    %rax, %rsi


        movq    (%rax), %rax
        movq    %rax, %rsi
        movl    $.LC1, %edi
        movl    $0, %eax
        call    printf
        movl    sleepsecs(%rip), %eax
        movl    %eax, %edi
        call    sleep
        addl    $1, -4(%rbp)
.L3:
        cmpl    $9, -4(%rbp)
        jle     .L4
        call    getchar
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE5:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
        .section        .note.GNU-stack,"",@progbits
```

数据：对于 int sleepsecs=2.5 这个全局变量，因为他将成为目标程序中的一个符号，所以编译器会在汇编代码中注明它的符号类型，位置（.data），以及 size，并且编译器还会将 2.5 自己向 0 舍入到 2，如图所示

```
sleepsecs:
        .long   2
```

，整个对于 sleepsecs 这个全局变量的解析如下图所示

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

。对于局部变量 i，汇编中并没有 i 的相关信息，i 在这个 hello 程序中可以说被存放在了`-4(%rbp)`中。

```
movl    $0, -4(%rbp)
```
和
```
addl    $1, -4(%rbp)
cmpl    $9, -4(%rbp)
```
这三个语句就是
```
for(i=0;i<10;i++)
```
这一个 for 循环中 i 的效果。对于程序中出现的字符串，汇编会将他们放在.rodata 只读数据区，本程序中出现的有 printf 函数中所用到的一条打印字符串和一个格式化字符串，在汇编文件中如图所示

```
        .section        .rodata
        .align 8
.LC0:
        .string "Usage: Hello 1170301024 \345\256\213\350\265\237\347\245\226\357\274\201"
.LC1:
        .string "Hello %s %s\n"
```

，可以看到他们被放在了只读存储区。

常量的操作：常量在汇编中是以立即数的形式操作的。如图是 argv != 3 的汇编中用到常量 3
的部分的汇编指令
```
cmpl    $3, -20(%rbp)
```
。

赋值操作操作：汇编中赋值操作大多数是通过 mov 类数据转移指令和算术逻辑指令实现的比如 i++这条语句是
```
addl    $1, -4(%rbp)
```
这条汇编实现的。

算术逻辑操作：算术逻辑操作就是通过 add、sub mult、div、and or xor 等指令来实现的

2) 算术运算指令(20条)
ADD、ADC、AAA、DAA：加法；
INC：加“1”；
SUB、SBB、AAS、DAS：减法；
DEC：减“1”；
CMP：比较；
NEG：求补；
MUL、IMUL、AAM：乘法；
DIV、IDIV、AAD：除法；
CBW，CWD：符号扩展。

。

关系操作：关系操作是通过 test，com， jxx 等指令来实现的。

数组：数组的实现是通过在在数组地址的基础上进行偏移实现的，比如以 char *argv[]为例，我们用-32(%rsp)来存储数组首地址，然后用下图来读取第二个元素
```
movq    -32(%rbp), %rax
addq    $8, %rax
```
下图读取第一个元素
```
movq    -32(%rbp), %rax
addq    $16, %rax
```

控制转移操作：汇编中控制转移是通过一系列的控制转移指令来操作的，下图表示了一条 if 语句的汇编代码，通过 com 设置标志位，通过 jxx 指令来进行相应的跳转

```
cmpl    $3, -20(%rbp)
je      .L2
```

循环是通过多次跳转来实现的

```
.L2:
        movl    $0, -4(%rbp)
        jmp     .L3
.L4:
        movq    -32(%rbp), %rax
        addq    $16, %rax
        movq    (%rax), %rdx
        movq    -32(%rbp), %rax
        addq    $8, %rax
        movq    (%rax), %rax
        movq    %rax, %rsi
        movl    $.LC1, %edi
        movl    $0, %eax
        call    printf
        movl    sleepsecs(%rip), %eax
        movl    %eax, %edi
        call    sleep
        addl    $1, -4(%rbp)
.L3:
        cmpl    $9, -4(%rbp)
        jle     .L4
```

上图表示的是 hello 中的 for 循环其中.L2:符号和.L4 之间的部分是 for 的初始化 i=0；.L4 和.L3 之间的是 for 循环的主体部分;.L3 之后就是 for 循环的条件判断部分，我们可以发现通过多次跳转我们就可以实现循环。

函数操作：首先 64 位函数传参是通过寄存器和堆栈来进行的，当参数小于等于 6 的时候通过寄存器，当大于 6 的时候大于部分通过栈来传递，

```
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
```

通过 64 位栈传参数的特点，我们就可以知道%edi 为传递给 main 函数的第一个参数即 argc，%esi 为第二个参数 argv。当我们调用函数的时候我们首先将参数传递，然后用 call 指令来将控制改变为被调函数，下图为 sleep 函数的调用过程

```
movl    sleepsecs(%rip), %eax
movl    %eax, %edi
call    sleep
```

。Hello 中用到其他的函数 printf、exit、getchar 都有相同的特点。

3.4 本章小结

汇编代码表示和 C 程序的差别很大，各种数据类型之间的差别很小，程序是以指令序列来表示的，每条指令都完成一个单独的操作，部分程序状态，如寄存器和运行时栈，对于程序员来说都是可见的。在这一阶段我们的操作接近机器实现，深层的了解了那些被机器屏蔽的细节，理解了编译器的优化能力，并分析了代码中隐含的低效率，同时在这一节中我们还了解了漏洞是如何实现的，以及如何防御他们的知识和技巧。

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

汇编的概念：汇编器（as）将 hello.s 翻译成机器语言指令，并把这些指令打包成一种叫做可重定位目标程序的格式，并将结果保留在目标文件 hello.o 中。

汇编的作用：汇编操作将汇编语言程序翻译成可重定位的目标程序，该文件虽然不能运行，但是它已经进行了机器代码的阶段，它拥有变为可执行文件所必须的信息。

4.2 在 Ubuntu 下汇编的命令

gcc -m64 -no-pie -fno-PIC -v -c hello.s -o hello.o

截图如下：

```
syz170301024@ubuntu:~/hitics/work$ gcc -m64 -no-pie -fno-PIC  -v -c hello.s -o hello.o
Using built-in specs.
COLLECT_GCC=gcc
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.3.0-27ubuntu1~18.04' --with-bugu
rl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,ob
j-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-lin
ux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext
 --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --en
able-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-un
ique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-s
ystem-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --
with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=
generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --bui
ld=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)
COLLECT_GCC_OPTIONS='-m64' '-no-pie' '-fno-PIC' '-v' '-c' '-o' 'hello.o' '-mtune=generic' '-march
=x86-64'
 as -v --64 -o hello.o hello.s
GNU assembler version 2.30 (x86_64-linux-gnu) using BFD version (GNU Binutils for Ubuntu) 2.30
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_
64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/7/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-lin
ux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr
/lib/x86_64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/7/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-m64' '-no-pie' '-fno-PIC' '-v' '-c' '-o' 'hello.o' '-mtune=generic' '-march
=x86-64'
```

4.3 可重定位目标 elf 格式

```
syz170301024@ubuntu:~/hitics/work$ readelf -a hello.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          1112 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 12

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       000000000000007d  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000318
       00000000000000c0  0000000000000018   I       10     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000c0
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  000000c4
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  000000c8
       0000000000000032  0000000000000000   A       0     0     8
  [ 6] .comment          PROGBITS         0000000000000000  000000fa
       000000000000002b  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000125
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  00000128
       0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  000003d8
       0000000000000018  0000000000000018   I       10     8     8
  [10] .symtab           SYMTAB           0000000000000000  00000160
       0000000000000180  0000000000000018          11     9     8
       0000000000000061  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)

There are no section groups in this file.

There are no program headers in this file.

There is no dynamic section in this file.

Relocation section '.rela.text' at offset 0x318 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000016  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000000001b  000b00000002 R_X86_64_PC32     0000000000000000 puts - 4
000000000025  000c00000002 R_X86_64_PC32     0000000000000000 exit - 4
00000000004c  00050000000a R_X86_64_32       0000000000000000 .rodata + 25
000000000056  000d00000002 R_X86_64_PC32     0000000000000000 printf - 4
00000000005c  000900000002 R_X86_64_PC32     0000000000000000 sleepsecs - 4
000000000063  000e00000002 R_X86_64_PC32     0000000000000000 sleep - 4
000000000072  000f00000002 R_X86_64_PC32     0000000000000000 getchar - 4

Relocation section '.rela.eh_frame' at offset 0x3d8 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is no
upported.

Symbol table '.symtab' contains 16 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 sleepsecs
    10: 0000000000000000   125 FUNC    GLOBAL DEFAULT    1 main
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

ELF 格式分析：

Elf 头：ELF 头以一个 16 字节的序列开始，包括 ELF 头的大小，字节顺序，文件类型，机器类型，节头部表的偏移以及节头部表中的条目的大小和数量。

然后是目标文件中的每个节，每个节都有一个固定大小的条目 hello.o 文件中包括了以下节

.text: 以编译程序的机器代码，其二进制信息为

```
Hex dump of section '.text':
 NOTE: This section has relocations against it, but these have NOT been applied to this dump
  0x00000000 554889e5 4883ec20 897dec48 8975e083 UH..H.. .}.H.u..
  0x00000010 7dec0374 14bf0000 0000e800 000000bf }..t............
  0x00000020 01000000 e8000000 00c745fc 00000000 ..........E.....
  0x00000030 eb39488b 45e04883 c010488b 10488b45 .9H.E.H...H..H.E
  0x00000040 e04883c0 08488b00 4889c6bf 00000000 .H...H..H.......
  0x00000050 b8000000 00e80000 00008b05 00000000 ................
  0x00000060 89c7e800 00000083 45fc0183 7dfc097e ........E...}..~
  0x00000070 c1e80000 0000b800 000000c9 c3       ............
```

.rela.text：.text 节的可重定位信息，在可执行文件中需要修改的指令地址需修改的指令.

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .rela.text hello.o

Hex dump of section '.rela.text':
  0x00000000 16000000 00000000 0a000000 05000000 ................
  0x00000010 00000000 00000000 1b000000 00000000 ................
  0x00000020 02000000 0b000000 fcffffff ffffffff ................
  0x00000030 25000000 00000000 02000000 0c000000 %...............
  0x00000040 fcffffff ffffffff 4c000000 00000000 ........L.......
  0x00000050 0a000000 05000000 25000000 00000000 ........%.......
  0x00000060 56000000 00000000 02000000 0d000000 V...............
  0x00000070 fcffffff ffffffff 5c000000 00000000 ........\.......
  0x00000080 02000000 09000000 fcffffff ffffffff ................
  0x00000090 63000000 00000000 02000000 0e000000 c...............
  0x000000a0 fcffffff ffffffff 72000000 00000000 ........r.......
  0x000000b0 02000000 0f000000 fcffffff ffffffff ................
```

.data: 已初始化全局变量

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .data hello.o

Hex dump of section '.data':
  0x00000000 02000000                             ....
```

.bss: 未初始化的全局变量和静态变量。

.rodata: 只读数据： printf 的格式化字符串，跳转表等；

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .rodata hello.o

Hex dump of section '.rodata':
  0x00000000 55736167 653a2048 656c6c6f 20313137 Usage: Hello 117
  0x00000010 30333031 30323420 e5ae8be8 b59fe7a5 0301024 ........
  0x00000020 96efbc81 0048656c 6c6f2025 73202573 .....Hello %s %s
  0x00000030 0a00                                 ..
```

.comment: 未初始化的全局变量

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .comment hello.o

Hex dump of section '.comment':
  0x00000000 00474343 3a202855 62756e74 7520372e .GCC: (Ubuntu 7.
  0x00000010 332e302d 32377562 756e7475 317e3138 3.0-27ubuntu1~18
  0x00000020 2e303429 20372e33 2e3000             .04) 7.3.0.
```

.note.GNU-stack

.eh_frame

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .eh_frame hello.o

Hex dump of section '.eh_frame':
 NOTE: This section has relocations against it, but these have NOT been applied to this dump
  0x00000000 14000000 00000000 017a5200 01781001 .........zR..x..
  0x00000010 1b0c0708 90010000 1c000000 1c000000 ................
  0x00000020 00000000 7d000000 00410e10 8602430d ....}....A....C.
  0x00000030 0602780c 07080000                   ..x.....
```

.rela.eh_frame

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .rela.eh_frame hello.o

Hex dump of section '.rela.eh_frame':
  0x00000000 20000000 00000000 02000000 02000000  ................
  0x00000010 00000000 00000000                    ........
```

.symtab: 一个符号表，它存放在程序中定义和引用的函数和全局变量的信息

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .symtab hello.o

Hex dump of section '.symtab':
  0x00000000 00000000 00000000 00000000 00000000 ................
  0x00000010 00000000 00000000 01000000 0400f1ff ................
  0x00000020 00000000 00000000 00000000 00000000 ................
  0x00000030 00000000 03000100 00000000 00000000 ................
  0x00000040 00000000 00000000 00000000 03000300 ................
  0x00000050 00000000 00000000 00000000 00000000 ................
  0x00000060 00000000 03000400 00000000 00000000 ................
  0x00000070 00000000 00000000 00000000 03000500 ................
  0x00000080 00000000 00000000 00000000 00000000 ................
  0x00000090 00000000 03000700 00000000 00000000 ................
  0x000000a0 00000000 00000000 00000000 03000800 ................
  0x000000b0 00000000 00000000 00000000 00000000 ................
  0x000000c0 00000000 03000600 00000000 00000000 ................
  0x000000d0 00000000 00000000 09000000 11000300 ................
  0x000000e0 00000000 00000000 04000000 00000000 ................
  0x000000f0 13000000 12000100 00000000 00000000 ................
  0x00000100 7d000000 00000000 18000000 10000000 }...............
  0x00000110 00000000 00000000 00000000 00000000 ................
  0x00000120 1d000000 10000000 00000000 00000000 ................
  0x00000130 00000000 00000000 22000000 10000000 ........".......
  0x00000140 00000000 00000000 00000000 00000000 ................
  0x00000150 29000000 10000000 00000000 00000000 )...............
  0x00000160 00000000 00000000 2f000000 10000000 ......../.......
  0x00000170 00000000 00000000 00000000 00000000 ................
```

.strtab: 一个字符串表，其内容包括.symtab 和.debug 节中的符号表，以及节头部表中的节名字，字符串表就是以 null 结尾的字符串序列。

```
syz170301024@ubuntu:~/hitics/work$ readelf -x .strtab hello.o

Hex dump of section '.strtab':
  0x00000000 0068656c 6c6f2e63 00736c65 65707365 .hello.c.sleepse
  0x00000010 6373006d 61696e00 70757473 00657869 cs.main.puts.exi
  0x00000020 74007072 696e7466 00736c65 65700067 t.printf.sleep.g
  0x00000030 65746368 617200                     etchar.
```

接下来就是节头部表（如下图所示）

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       000000000000007d  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000318
       00000000000000c0  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000c0
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  000000c4
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  000000c8
       0000000000000032  0000000000000000   A       0     0     8
  [ 6] .comment          PROGBITS         0000000000000000  000000fa
       000000000000002b  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000125
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  00000128
       0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  000003d8
       0000000000000018  0000000000000018   I      10     8     8
  [10] .symtab           SYMTAB           0000000000000000  00000160
       0000000000000180  0000000000000018          11     9     8
  [11] .strtab           STRTAB           0000000000000000  000002e0
       0000000000000037  0000000000000000           0     0     1
  [12] .shstrtab         STRTAB           0000000000000000  000003f0
       0000000000000061  0000000000000000           0     0     1
```

节头部表中保存着每个节的大小和偏移。

对重定位节的分析：

我们主要分析.rela.text 节的内容，.rela.text 节信息如下图：

```
Relocation section '.rela.text' at offset 0x318 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000016  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000000001b  000b00000002 R_X86_64_PC32     0000000000000000 puts - 4
000000000025  000c00000002 R_X86_64_PC32     0000000000000000 exit - 4
00000000004c  00050000000a R_X86_64_32       0000000000000000 .rodata + 25
000000000056  000d00000002 R_X86_64_PC32     0000000000000000 printf - 4
00000000005c  000900000002 R_X86_64_PC32     0000000000000000 sleepsecs - 4
000000000063  000e00000002 R_X86_64_PC32     0000000000000000 sleep - 4
000000000072  000f00000002 R_X86_64_PC32     0000000000000000 getchar - 4
```

ELF 重定位条目的格式如下：

```
typedef struct {
        long offset;
        long type: 32,
             symbol: 32;
        long addend;
}Elf64_Rela;
```

Offset 是需要被修改引用的节偏移，symbol 表示被修改引用应该指向的符号即符号表的一个索引，type 告诉链接器如何修改新的引用，addend 是有一个有符号常数，一些类型的重定位要使用它对修改引用的值做偏移调整。

ELF 定义了 32 中不同的重定位类型，在 hello.o 程序中我们只用到其中的两个：1、R_X86_64_PC32：重定位一个使用 32 位 PC 相对地址的引用。2、R_X86_64_32：重定位一个使用 32 位绝对地址的引用，通过绝对地址，CPU 直接使用在指令中编码的 32 位置作为有效地址，不需要进一步修改。

在 hello.o 中我们可以看出只有两个 printf 函数中用到的字符串用到了 32 位绝对寻址，其他所有的函数调用用到的都是 PC 相对寻址。现在我们仔细的观察.rela.text 中的每行信息。比如这一行

```
000000000016  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
```

首先 offset 为 0x16，说明该重定位符号在.text 中的偏移为 0x16 个字节，info 应该分为两部分，
分析 hello.o 的 ELF 格式，用 readelf 等列出其各节的基本
信息，特别是重定位项目分析。0x0000000a 表示的是重定位的类型信息，它告诉编译器应该怎么修改新的应用而在 hello.o 中 0x0000 000a 就是命令编译器使用 R_X86_64_32 绝对寻址的方式处理新的引用。0x0005 代表了改符号在符号中的索引，因为这一行重定位符号是.rodata，我们结合符号表和节头部表可以知道.rodata 在符号表中的索引是 5 如下两图：

```
 [ 5] .rodata           PROGBITS         0000000000000000  000000c8
      0000000000000032  0000000000000000   A       0     0     8
```

```
  5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
```

，
这就是 info 为 00050000000a 的原因，后面的 Type 就对应了 Info 中的第四字节，然后我们来看 Addend，它表示的是重定位的时候要对被修改应用的值做偏移调整，因为当我们在重定位的时候要在该符号位置放 prinf 字符串的地址，而我们知道.rodata 有两个字符串，这个字符串的地址就是.rodata 的地址，我们比较一下另一个字符串的重定位信息

```
00000000004c  00050000000a R_X86_64_32       0000000000000000 .rodata + 25
```

这两个字符串重定位的时候他们的符号是相同的，但是他们的地址确是不相同的，第二个字符串紧接在第一个字符串的后面，所以我们必须要设置偏移，在重定位的时候对符号的地址进行偏移调整。
我们在来看一个函数符号重定位信息，如图

```
00000000001b  000b00000002 R_X86_64_PC32     0000000000000000 puts - 4
```

首先还是来看 offset，它表示的就是这个函数符号重定位的是在.text 中的偏移，然后 Info 分为两部分，第一部分 0x000b,表示改符号在符号表中的索引为 11，我们找到符号表中 puts 符号如图：

```
 11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

puts 符号的索引就是 11（0xb）后面的四字节就是重定位类型（0x0000 0002 代表的就是 R_X86_64_PC32）然后我们再来看 Addend，我们知道当 CPU 执行一条使用 PC 相对寻址的指令时，它就将在指令中编码的 32 位值加上 PC 的当前地址，得到有效地址（如 puts 指令的目标），但是我们知道 PC 的值通常是指下一条指令在内存中的地址，如果我们仅仅将 puts 符号的地址的偏移放进去，这时候我们在求 puts 函数的地址的时候其实 PC 与计算相对寻址是的 PC 不一致，已经指向下一条指令，所以我们必须在重定位的时候进行偏移处理，又因为 hello.o 运行时用 32 位 PC 相对地址来访问，32 位占 4 个字节，所以我们得将重定位后的引用-4 来进行调整，这样我们就可以知道偏移为-4，即 Addend = -4.
其他的函数符号重定位信息是一样的。

4.4 Hello.o 的结果解析

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 20             sub    $0x20,%rsp
   8:   89 7d ec                mov    %edi,-0x14(%rbp)
   b:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
   f:   83 7d ec 03             cmpl   $0x3,-0x14(%rbp)
  13:   74 14                   je     29 <main+0x29>
  15:   bf 00 00 00 00          mov    $0x0,%edi
                        16: R_X86_64_32 .rodata
  1a:   e8 00 00 00 00          callq  1f <main+0x1f>
                        1b: R_X86_64_PC32       puts-0x4
  1f:   bf 01 00 00 00          mov    $0x1,%edi
  24:   e8 00 00 00 00          callq  29 <main+0x29>
                        25: R_X86_64_PC32       exit-0x4
  29:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  30:   eb 39                   jmp    6b <main+0x6b>
  32:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  36:   48 83 c0 10             add    $0x10,%rax
  3a:   48 8b 10                mov    (%rax),%rdx
  3d:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  41:   48 83 c0 08             add    $0x8,%rax
  45:   48 8b 00                mov    (%rax),%rax
  48:   48 89 c6                mov    %rax,%rsi
  4b:   bf 00 00 00 00          mov    $0x0,%edi
                        4c: R_X86_64_32 .rodata+0x25
  50:   b8 00 00 00 00          mov    $0x0,%eax
  55:   e8 00 00 00 00          callq  5a <main+0x5a>
                        56: R_X86_64_PC32       printf-0x4
  5a:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 60 <main+0x60>
                        5c: R_X86_64_PC32       sleepsecs-0x4
  60:   89 c7                   mov    %eax,%edi
  62:   e8 00 00 00 00          callq  67 <main+0x67>
                        63: R_X86_64_PC32       sleep-0x4
  67:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  6b:   83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
  6f:   7e c1                   jle    32 <main+0x32>
  71:   e8 00 00 00 00          callq  76 <main+0x76>
                        72: R_X86_64_PC32       getchar-0x4
  76:   b8 00 00 00 00          mov    $0x0,%eax
  7b:   c9                      leaveq
  7c:   c3                      retq
```

机器语言的构成，与汇编语言的映射关系：机器语言由操作码、功能码寄存器文件指示符编码，和其他常数字节构成。一条汇编语句对于一条机器指令，机器指令的长度从 1 到 15 不等，常用的指令以及操作较少的指令所需的字节数较少，而那些不太常用或操作数较多的指令所需的字节数较多，机器代码中的常数字是小端表示的，即高位字节在后，低位字节在前。机器代码中分支转移不在用符号表示标记表示，而是直接映射成标记所在位置的偏移，如图所示

```
83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
7e c1                   jle    32 <main+0x32>
```

函数调用的反汇编指令中不在是函数名表示，而是直接指向了下一条指令的地址，在函数调用的下

面会有关于这个符号的重定位信息，如图所示

```
b8 00 00 00 00          mov    $0x0,%eax
e8 00 00 00 00          callq  5a <main+0x5a>
                56: R_X86_64_PC32       printf-0x4
8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 60 <main+0x60>
```

。

objdump -d -r hello.o　分析 hello.o 的反汇编，并请与第 3 章的 hello.s 进行对照分析。

说明机器语言的构成，与汇编语言的映射关系。特别是机器语言中的操作数与汇编语言不一致，特别是分支转移函数调用等。

4.5 本章小结

一个目标文件就是一个字节序列，它所包含的信息对于我们进一步的链接至关重要。至此，我们已经从汇编代码来到了机器指令。我们细致的分析了可重定位文件的相关信息。看到了汇编代码和目标文件之间的区别。

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

链接的概念：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，这个文件可以被加载到内存执行。链接可以执行于编译时，也就是源代码被翻译成机器代码时，也可以执行于加载时，也就是程序被加载器加载到内存并执行时
链接的作用：链接在软件开发中扮演着一个关键的角色，链接是的分离编译成为可能。我们不用讲一个大型的应用程序组织为一个巨大的源代码，而是可以把他分为更小，更好管理的模块，可以独立的修改和编译这些模块。当我们改变这些模块中的一个时，我们只需简单的重新编译它，并开始链接应用，而不必要重新编译所有的其他文件。

5.2 在 Ubuntu 下链接的命令

（以下格式自行编排，编辑时删除）
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o hello.o -lc /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -z relro -o a.out

```
syz170301024@ubuntu:/mnt/hgfs/hitics/work$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2  /usr/
ib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/7/crtb
gin.o -v hello.o -lc /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -
 relro -o a.out
GNU ld (GNU Binutils for Ubuntu) 2.30
```

5.3 可执行目标文件 hello 的格式

ELF 格式分析：

Elf 头：ELF 头以一个 16 字节的序列开始，包括 ELF 头的大小，字节顺序，文件类型，机器类型，节头部表的偏移以及节头部表中的条目的大小和数量。

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endi
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-
  Version:                           0x1
  Entry point address:               0x400500
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6512 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

程序头部表：我们用 readelf 来查看如下图

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000830 0x0000000000000830  R E    0x200000
  LOAD           0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000254 0x0000000000000258  RW     0x200000
  DYNAMIC        0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000200 0x0000000000000200  R      0x1
```

程序头部表中我们可以看到 hello 可执行目标文件的内容初始化为两个段

```
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x0000000000000830 0x0000000000000830  R E    0x200000
```

这个部分告诉我们第一个段（代码段）有读/运行的访问权限，开始于地址 0x400000 处，总内存大小是 0x830 字节。

```
LOAD           0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
               0x0000000000000254 0x0000000000000258  RW     0x200000
```

这一部分告诉我们第二个段（数据段）有读/写的访问权限，开始于内存地址为 0x600e00 处，总的内存戴小伟 0x254 字节。

夹在程序头部表和节头部表中间的节：

每个节都有一个固定大小的条目。hello 文件中包括了以下节

.interp.note.ABI-tag
.hash
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init

.plt
.text　　已编译的代码
.fini
.rodata　　只读数据，比如 printf 的格式化字符串和跳转表
.en_frame
.init_array
.fini_array
.dynamic
.got
.got.plt
.data　　初始化的静态变量和全局变量
.bss　　未初始化的静态变量和未初始化全局变量
.comment　　未初始化的全局变量
.symtab　　符号表
.strtab　　字符串表
.shstrtab

节头部表：

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400200  00000200
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             000000000040021c  0000021c
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .hash             HASH             0000000000400240  00000240
       0000000000000034  0000000000000004   A       5     0     8
  [ 4] .gnu.hash         GNU_HASH         0000000000400278  00000278
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000400298  00000298
       00000000000000c0  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400358  00000358
       0000000000000057  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           00000000004003b0  000003b0
       0000000000000010  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          00000000004003c0  000003c0
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             00000000004003e0  000003e0
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000400410  00000410
       0000000000000078  0000000000000018  AI       5    21     8
  [11] .init             PROGBITS         0000000000400488  00000488
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004004a0  000004a0
       0000000000000060  0000000000000010  AX       0     0     16
  [13] .text             PROGBITS         0000000000400500  00000500
       00000000000001e2  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         00000000004006e4  000006e4
       0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         00000000004006f0  000006f0
       000000000000003a  0000000000000000   A       0     0     8
  [16] .eh_frame         PROGBITS         0000000000400730  00000730
       0000000000000100  0000000000000000   A       0     0     8
  [17] .init_array       INIT_ARRAY       0000000000600e00  00000e00
       0000000000000008  0000000000000008  WA       0     0     8
  [18] .fini_array       FINI_ARRAY       0000000000600e08  00000e08
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .dynamic          DYNAMIC          0000000000600e10  00000e10
       00000000000001e0  0000000000000010  WA       6     0     8
  [20] .got              PROGBITS         0000000000600ff0  00000ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [21] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000040  0000000000000008  WA       0     0     8
```

```
[22] .data             PROGBITS         0000000000601040  00001040
     0000000000000014  0000000000000000  WA       0     0     8
[23] .bss              NOBITS           0000000000601054  00001054
     0000000000000004  0000000000000000  WA       0     0     1
[24] .comment          PROGBITS         0000000000000000  00001054
     000000000000002a  0000000000000001  MS       0     0     1
[25] .symtab           SYMTAB           0000000000000000  00001080
     0000000000000600  0000000000000018          26    41     8
[26] .strtab           STRTAB           0000000000000000  00001680
     000000000000020e  0000000000000000           0     0     1
[27] .shstrtab         STRTAB           0000000000000000  0000188e
     00000000000000e2  0000000000000000           0     0     1
```

各段的基本信息如下图

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000830 0x0000000000000830  R E    0x200000
  LOAD           0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000254 0x0000000000000258  RW     0x200000
  DYNAMIC        0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000200 0x0000000000000200  R      0x1
```

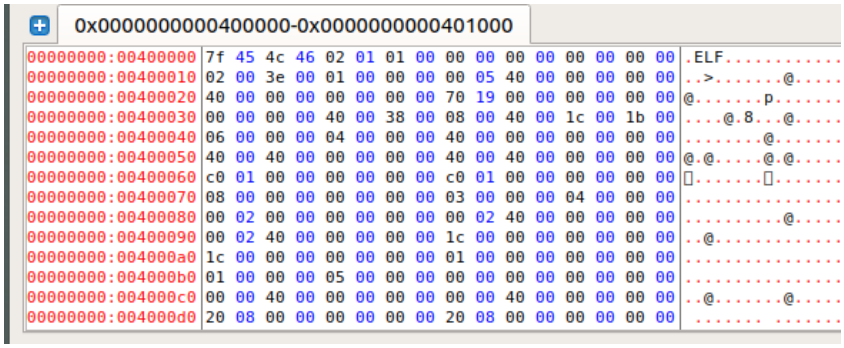
从截图可以看出代码段有读/运行的访问权限，开始于地址 0x400000 处，总内存大小是 0x830 字节。数据段有读/写的访问权限，开始于内存地址为 0x600e00 处，总的内存戴小伟 0x254 字节。

5.4 hello 的虚拟地址空间

使用 readelf 查看程序头部表所得信息截图如下：

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000830 0x0000000000000830  R E    0x200000
  LOAD           0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000254 0x0000000000000258  RW     0x200000
  DYNAMIC        0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000020 0x0000000000000020  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e00 0x0000000000600e00 0x0000000000600e00
                 0x0000000000000200 0x0000000000000200  R      0x1
```

使用 edb 查看的结果如下：



5.5 链接的重定位过程分析

```
00000000004005e7 <main>:
  4005e7:       55                      push   %rbp
  4005e8:       48 89 e5                mov    %rsp,%rbp
  4005eb:       48 83 ec 20             sub    $0x20,%rsp
  4005ef:       89 7d ec                mov    %edi,-0x14(%rbp)
  4005f2:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  4005f6:       83 7d ec 03             cmpl   $0x3,-0x14(%rbp)
  4005fa:       74 14                   je     400610 <main+0x29>
  4005fc:       bf f8 06 40 00          mov    $0x4006f8,%edi
  400601:       e8 aa fe ff ff          callq  4004b0 <puts@plt>
  400606:       bf 01 00 00 00          mov    $0x1,%edi
  40060b:       e8 d0 fe ff ff          callq  4004e0 <exit@plt>
  400610:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400617:       eb 39                   jmp    400652 <main+0x6b>
  400619:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  40061d:       48 83 c0 10             add    $0x10,%rax
  400621:       48 8b 10                mov    (%rax),%rdx
  400624:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  400628:       48 83 c0 08             add    $0x8,%rax
  40062c:       48 8b 00                mov    (%rax),%rax
  40062f:       48 89 c6                mov    %rax,%rsi
  400632:       bf 1d 07 40 00          mov    $0x40071d,%edi
  400637:       b8 00 00 00 00          mov    $0x0,%eax
  40063c:       e8 7f fe ff ff          callq  4004c0 <printf@plt>
  400641:       8b 05 09 0a 20 00       mov    0x200a09(%rip),%eax        # 601050 <sleepsecs>
  400647:       89 c7                   mov    %eax,%edi
  400649:       e8 a2 fe ff ff          callq  4004f0 <sleep@plt>
  40064e:       83 45 fc 01             addl   $0x1,-0x4(%rbp)
  400652:       83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
  400656:       7e c1                   jle    400619 <main+0x32>
  400658:       e8 73 fe ff ff          callq  4004d0 <getchar@plt>
  40065d:       b8 00 00 00 00          mov    $0x0,%eax
  400662:       c9                      leaveq
  400663:       c3                      retq
  400664:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40066b:       00 00 00
  40066e:       66 90                   xchg   %ax,%ax
```

（以下格式自行编排，编辑时删除）

将 hello 反汇编之后你会看到很多其他的函数代码比如：

```
00000000004004a8 <_init>:
  4004a8:       48 83 ec 08             sub    $0x8,%rsp
  4004ac:       48 8b 05 45 0b 20 00    mov    0x200b45(%rip),%rax        # 600ff8 <__gmon_start_
>
  4004b3:       48 85 c0                test   %rax,%rax
  4004b6:       74 02                   je     4004ba <_init+0x12>
  4004b8:       ff d0                   callq  *%rax
  4004ba:       48 83 c4 08             add    $0x8,%rsp
  4004be:       c3                      retq
```

等等其他的需要动态链接共享库代码

```
00000000004004d0 <puts@plt>:
  4004d0:       ff 25 42 0b 20 00       jmpq   *0x200b42(%rip)        # 601018 <puts@GLIBC_2.2.5>
  4004d6:       68 00 00 00 00          pushq  $0x0
  4004db:       e9 e0 ff ff ff          jmpq   4004c0 <.plt>
```

。并且这时候 main 函数中有符号引用的地方也发生了改变，在 hello.o 的反汇编代码中调用 printf 函数的时候我们还没有将此时的符号重定位只是在 hello.o 文件中记录了符号的位置和其他信息

```
55:   e8 00 00 00 00          callq  5a <main+0x5a>
                      56: R_X86_64_PC32     printf-0x4
```

，而在 hello 的反汇编中调用 printf 的地方已经进行了重定位

```
e8 7f fe ff ff          callq  4004e0 <printf@plt>
8b 05 e9 09 20 00       mov    0x2009e9(%rip),%eax        # 601050 <sleepsecs>
```

。

Hello.o 重定位过程：首先链接器将所有.o 文件中相同类型的节合并成一个类

型的节，然后链接器将运行是内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。当这一步完成时，程序中的每条指令和全局变量都有唯一的运行时内存地址了，接下来，链接器通过一定的算法修改代码节和数据节中对每个符号的引用，使得他们只想正确的运行时地址，上文中已经详细简述了如何根据重定位项目来进行计算该引用的正确值。再次不在赘述。

objdump -d -r hello 分析 hello 与 hello.o 的不同，说明链接的过程。

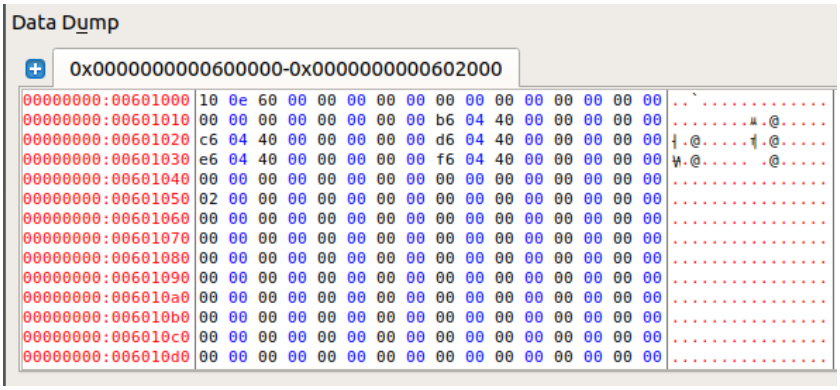
结合 hello.o 的重定位项目，分析 hello 中对其怎么重定位的。

5.6 hello 的执行流程

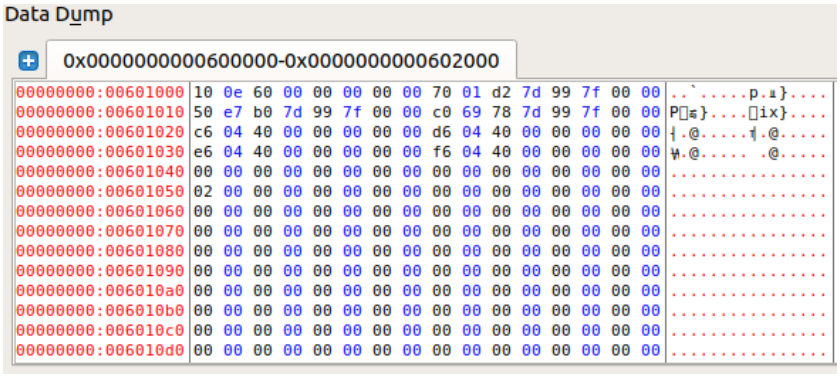
ld-2.27.so!_dl_start	0x7f8972d8dea0
ld-2.27,so!_dl_init	0x7f182474f630
libc-2.27.so!__libc_start_main	0x7f0b20a36ae0
Libc-2.27.so!_cxa_atexit	0x00007fbd71729430
Libc-2.27.so!_setjmp	0x00007fbd71724c10
Hello!puts@plt	0x00000000004004b0
Hello!exit@plt	0x00000000004004e0

5.7 Hello 的动态链接分析

在一个 X86-64 系统中，对同一个目标模块符号的引用是不需要特殊处理使之成为 PIC，可以通过 PC 相对寻址来编译这些引用，构造目标文件是有静态链接器重定位即可。编译器在数据段开始的地方创建了一个表，叫做全局偏移量表，在 GOT 中，每个别这个目标木块引用的全局目标都有一个 8 字节的条目，编译器会为 GOT 中的每个条目生成一个重定位记录。在加载时，动态链接器会重定位 GOT 中的每个条目，使得它包含目标的正确的绝对地址，每个引用全局目标的目标模块都有自己的 GOT。下图说明了 GOT 的变化：



调用_dl_init 之前



调用_dl_init 之后

5.8 本章小结

链接的两个主要任务就是符号解析和重定位，符号解析将目标文件中的每个全局符号都绑定到一个唯一的定义，而重定位确定每个符号的最终内存地址，并修改对那些目标的引用。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

进程的概念：进程的经典定义就是一个执行中程序的实例，系统中的么米格程序都运行在某个进程的上下文中，上下文是由程序正确运行所需的状态组成的，这个状态包括存放在内存中的程序的代码和数据，它的栈，通用目的寄存器的内容，程序计数器，环境变量以及打开文件描述符的集合。

进程的作用：在现代系统中运行一个程序时，我们得到一个假象，就好像我们的程序时系统中当前运行的唯一的程序一样，我们的程序好像是独占的使用处理器和内存。处理器就好像无间断的一条一条的执行我们程序中的指令。最后我们的程序中的代码和数据好像是系统内存中唯一的对象。

6.2 简述壳 Shell-bash 的作用与处理流程

Shell 的作用是进行用户和操作系统的直接通讯，shell 尽管并不是操作系统的一部分，但是他使用了大量的操作系统特性，并且很重要的方面是它会在子进程结束之后对其进行回收，防止僵死进程太多而浪费大量内存。当用户登录进入系统是，同时将启动一个 shell，它以终端作为标准的输入和输出，一般来说，它会首先显示一个系统提示符，比如$，然后它会等待用户输入命令，当用户输入命令之后 shell 对命令进行解析，然后判断此命令是不是内嵌命令，如果是那么 shell 直接调用执行，如果该命令是执行可执行文件，那么 shell 会创建一个子进程并调用 exceve 系统调用来运行可执行文件，在子进程运行期间，shell 将等待它结束，如果用户在一条命令后加上一个“&”，那么 shell 将不再等待期结束，直接显示系统提示符。当子进程停止或者终止的时候，shell 都会对每个子进程的状态做出反应，如果子进程终止了，那么 shell 将对其进行回收，删除该进程在内核和 shell 中的所有信息。

6.3 Hello 的 fork 进程创建过程

当我们在终端键入./hello 的时候，因为 hello 是可执行文件，所以 shell 进行系统调用 fork 来创建一个子进程，创建的子进程与父进程一模一样但不完全相同，子进程得到与父进程用户及虚拟地址空间相同的的一份副本，包括代码和数据段、堆、共享库、以及用户栈。子进程还获得与父进程打开文件描述符相同的副本，这意味着当 shell 调用 fork 的时候，hello 进程可以都写 shell 进程中打开的任何文件，

之所以是这样的是因为，当 fork 函数被 shell 调用的时候，内核为新进程创建各种数据结构，并分配一个为以的 PID，为了给这个新进程创建虚拟内存，他穿见了 shell 进程的 mm_struct、区域结构和页表的原样副本。这两个进程中的每个页面都标记为只读，并将这两个进程中的每个区域结构都标记为私有的写时复制。

6.4 Hello 的 execve 过程

Execve 函数加载并运行可执行目标文件 hello，且带参数列表 argv 和环境变量列表 envp，只有当错误的时候，execve 函数才会返回到调用程序。Execve 函数在当前进程的上下文中加载并运行一个 hello，它会覆盖当前进程的地址空间。首先 execve 函数会删除当前进程的虚拟地址空间的用户部分中的已存在的数据结构，然后为 hello 程序的代码、数据、bss 和栈区域创建新的区域结构。当 hello 与共享对象链接的时候，这些对象都是通过动态链接到 hello 的，然后在映射到用户虚拟地址空间中的共享区域内，最后，execve 做的就是设置当前进程上下文的程序计数器，是指指向代码区域的入口点。

6.5 Hello 的进程执行

内核为每一个进程维持一个上下文，上下文就是内核重新启动一个被抢占的进程所需的状态，它有一些对象值构成，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，内核栈和各种数据结构。每隔一定的周期，操作系统会暂停当前进程的执行，转而启动另一个进程，这样做的原因是，比如说：在过去的一秒钟内，第一个进程已经运行完了分配给它的时间片，这是操作系统就要暂停这个进程的运行，这是内核会抢占当前进程，然后重新开始先前一个被抢占得进程。这种决策称之为调度，有内核中成为调度器的代码处理的，从内核中选择一个新进程运行时，我们说内核调用了这个进程，在内核调度一个进程运行后，它就抢占了当前进程，并使用一种上下文切换的机制将控制转移到新的进程运行，这时候控制从用户态转移到内核态。内核会保护当前进程的上下文，并且恢复某个先前被抢占的进程被保存的上下文，然后控制再次从内核态转移到用户态开始该进程的执行。当然进程的调度并不仅仅在这一种情况下发生，进程执行某个系统调用的时候，控制会从用户态转为内核态，在内核代表用户执行系统调用的时候，也可能发生进程调度：如果系统调用因为等待某个时间发生而阻塞，那么内核可以让当前进程休眠，进行进程调度，上下文切换来执行另一个进程，控制从内核转移到用户态，进程调度还会发生在其他的情况下。

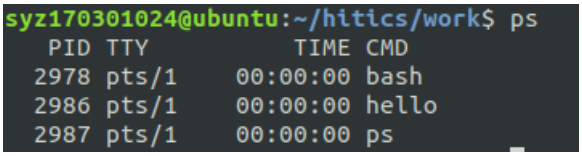
6.6 hello 的异常与信号处理

Hello 的执行过程会发生陷阱（因为有 sleep 函数）故障（缺页故障），可能会发生中断异常和终止异常。

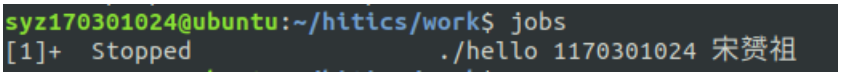
在 hello 的运行过程中可能产生的信号有 SIGINT、SINGTSTP SIGCHLD

SIGQUIT；当遇到 SIGINT 和 SIGQUIT 信号的时候 shell 会接受 SIGINT 和 SIGQUIT 信号，并且将它们发送给 hello 进程，hello 进程处理 SIGINT 和 SIGQUIT 信号是默认行为是终止 hello 进程；当遇到 SIGTSTP 信号时，同样，shell 会接受 SIGTSTP 信号，并且将它发送给 hello 进程，hello 进程处理 SIGTSTP 信号的默认行为是停止 hello 进程知道下一个 SIGCONT 信号。当 hello 终止或者挺值得时候会产生 SIGCHLD 信号发送给 shell 进程，SIGCHLD 的默认行为是忽略，当 shell 接收到此信号的时候，它会调用 shell 自己的 SIGCHLD 信号处理子程序，这这个子程序中 shell 会将 hello 进程回收，删除它在内核以及 shell 中的所有信息。

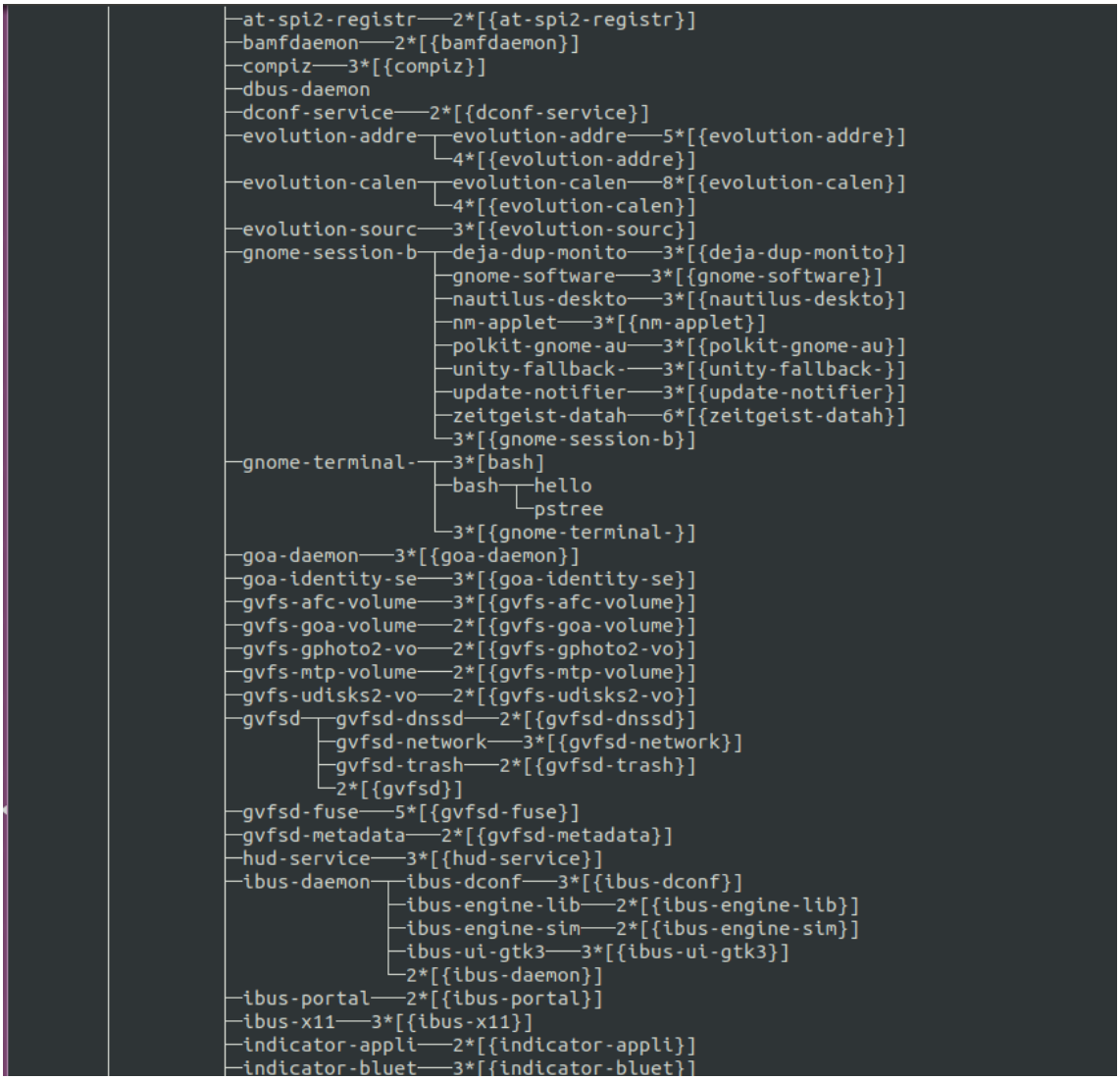
ps：



Jobs



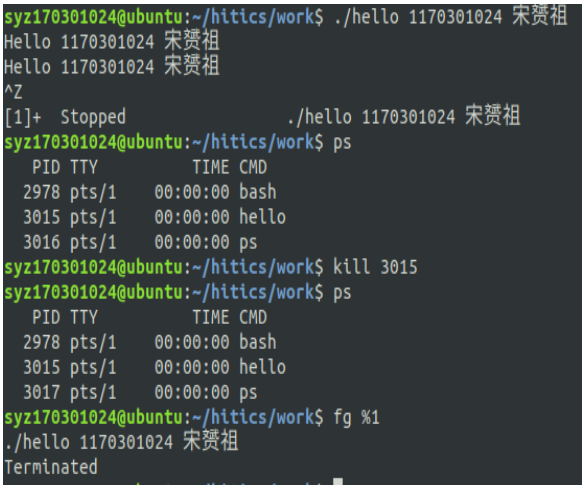
Pstree



fg %1

```
syz170301024@ubuntu:~/hitics/work$ fg %1
./hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
Hello 1170301024 宋赟祖
```

Kill （下面截图说明 kill 成功终止进程）



hello 执行过程中会出现哪几类异常，会产生哪些信号，又怎么处理的。

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

6.7 本章小结

在操作系统层，内核用 ECF 提供进程的基本概念。进程根据提供给应用两个重要的抽象：1、逻辑控制流，它提供给每个程序一个假象，好像它是在独占的使用处理器。2、私有地址空间，它提供给每个程序一个假象，好像它是在独占的使用内存。在系统和应用程序之间的接口处，应用程序可以创建子进程，等待它们的子进程停止或者终止，运行新的程序，以及捕获来自其他进程的信号，信号处理的语义有事微妙的，并且随系统不同而不同。

（第 6 章 1 分）

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

逻辑地址：逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址，其实是指由程序产生的与段相关的偏移地址部分（段内偏移量）。映射到 hello.o 里面的相对偏移地址。

线性地址：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，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。此间映射到 hello 里面的虚拟内存地址。

虚拟地址：CPU 通过生成一个虚拟地址。映射到 hello 里面的虚拟内存地址。

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

7.2 Intel 逻辑地址到线性地址的变换-段式管理

段式管理： 逻辑地址->线性地址==虚拟地址

1、逻辑地址=段选择符+偏移量
2、每个段选择符大小为 16 位，段描述符为 8 字节（注意单位）。
3、GDT 为全局描述符表，LDT 为局部描述符表。
4、段描述符存放在描述符表中，也就是 GDT 或 LDT 中。
5、段首地址存放在段描述符中。

每个段的首地址都存放在自己的段描述符中，而所有的段描述符都存放在一个描述符表中（描述符表分为全局描述符表 GDT 和局部描述符表 LDT）。而要想找到某个段的描述符必须通过段选择符才能找到。

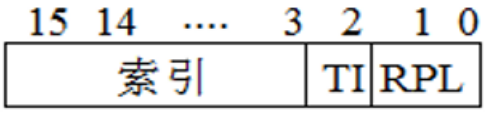


图 7-1-段选择符格式

由图 7-1 可以看出，段选择符由三部分组成，从左到右依次是 index【索引】，TI，RPL。

Index 处，我们可以将描述符表看成是一个数组，每个元素都存放一个段描述符，那么 index 就表示数组下标，亦即某个段描述符在数组中的索引。

再者，当 TI 为 0 时，表示段描述符在 GDT 中，当 TI 为 1 的时候，表示段描述符在 LDT 中。

RPL 代表请求特权级，RPL=00，为第 0 级，位于最高级的内核态，RPL=11，为第 3 级，位于最低级的用户态，第 0 级高于第 3 级。

现在假设我们有一个段的段选择符，他的 TI 是 0，Index 是 8，那么我们可以知道这个段的段描述符实在 GDT 数组中索引为 8 的位置。从而由我们知道的 GDT 的起始地址，每个段描述符的大小，就可以精确地找到我们想要的段描述符，从而获取某个段的首地址，然后再将从段描述符中获取到的段首地址与逻辑地址的偏移量相加就得到了线性地址。

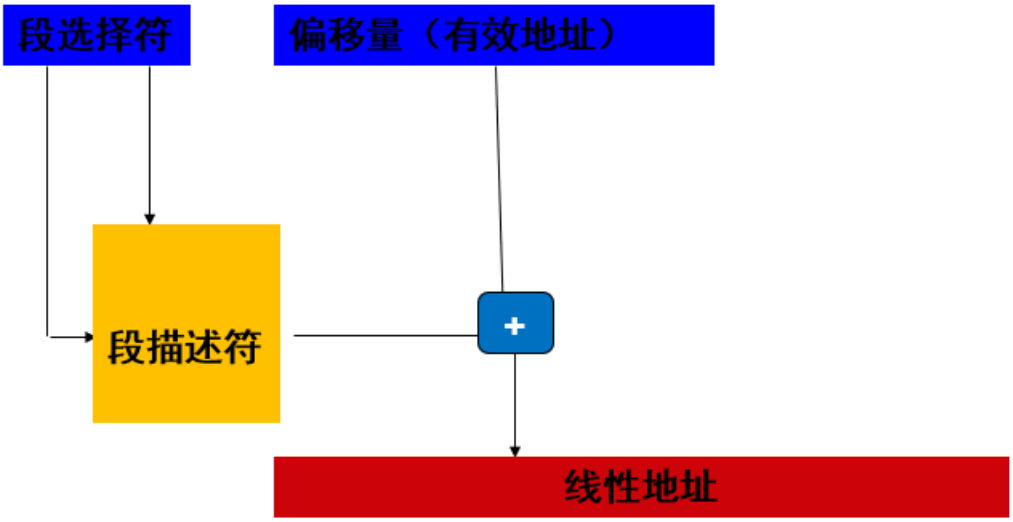


图 7-2-逻辑地址转换简化版

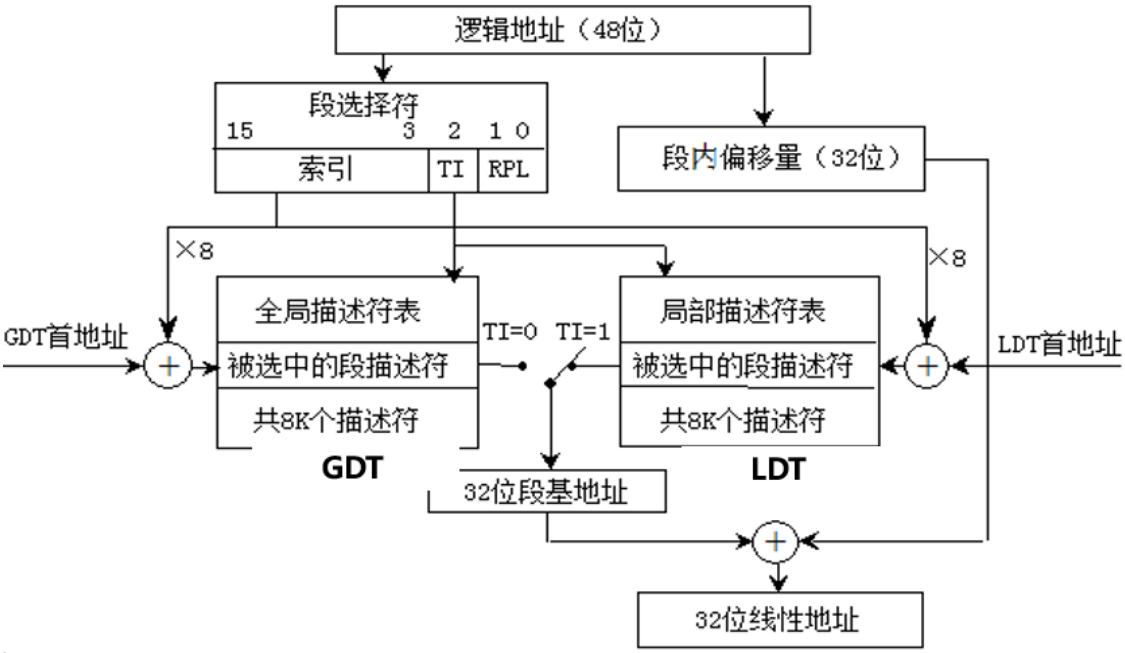


图 7-3-逻辑地址转化真实版

7.3 Hello 的线性地址到物理地址的变换-页式管理

Linux 系统有自己的虚拟内存系统。 图 7-4 强调了记录一个进程中虚拟内存区域的内核数据结构。内核为系统中每个进程维护一个单独的任务结构。任务结构中的元素包含或者指向内核运行该进程所需要的的所有信息。任务机构中的一个条目指向 mm_struct，它描述了虚拟内存的当前状态。我们感兴趣的两个字段分别是 pgd 和 mmap，其中 pgd 指向第一级页表的基址，而 mmap 指向一个 vm_area_structs 的链表，其中每个 vm_area_structs 都描述了当前虚拟地址空间的一个区域。当内核运行这个进程的时候，就将 pgd 存放在 CR3 控制寄存器内。

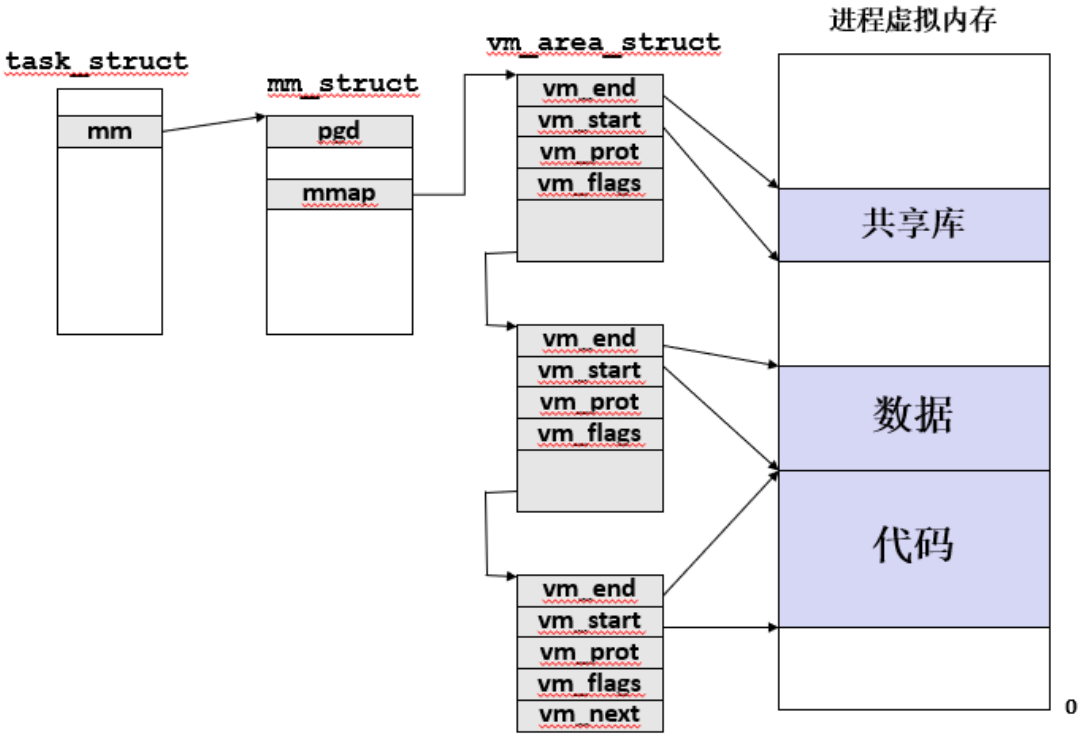


图 7-4-linux 下的虚拟内存系统

概念上而言，虚拟内存被组织为一个由存放在磁盘上 N 个连续的字节大小的单元组成的数组，每子节都有唯一一个虚拟地址作为到数组的索引。磁盘上数组的内容被缓存在主存中。和存储器结构中其他的缓存一样，磁盘（较低层）的数据被分割成块，此间虚拟内存系统将虚拟内存分割成拟页 VP 大小固定的块来处理这个问题，linux 下通常每个虚拟页的大小为 4KB，与之相类似，物理内存也被分割成物理页 PP，大小和虚拟页大小一致。

此间虚拟内存系统中的 MMU 内存管理单元对地址的翻译，就形象为物理内存中叫做页表的数据结构从虚拟页映射到物理页的过程。图 7.5 详细的展示了地址翻译的全过程。

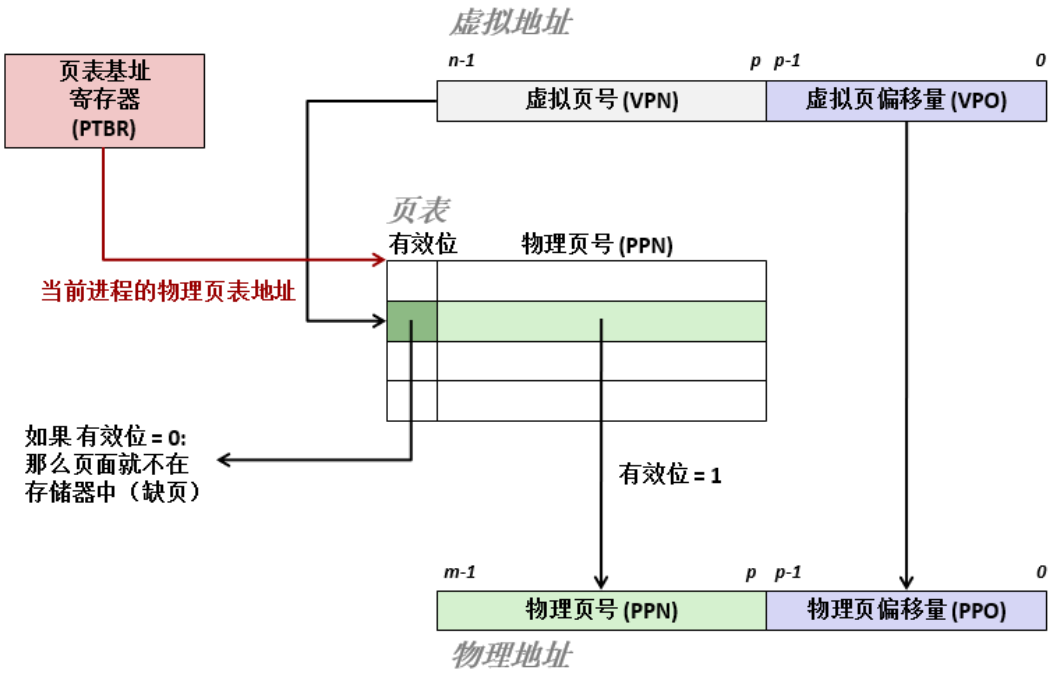


图 7.5-地址翻译

页表基址寄存器指向当前页表，n 位的虚拟地址包含虚拟页号和虚拟页偏移量两部分，同样物理地址也由物理页号和物理页偏移量组成。MMU 通过 VPN 来选择适当的 PTE，由此，将索引到的页表条目中的 PPN 和 VPO 串联起来就是虚拟地址

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

TLB 是一个小的、虚拟寻址的内存，其中的每一行都保存着一个有 PTE 组成的的块，PTE 通常有高度的相连度，用于组选择和行匹配的索引和标记字段师兄虚拟地址中的虚拟页号中提取出来的如图 7-6。这里的关键点在于，所有的地址翻译的步骤都是在 MMU 中执行的，因此非常快

翻译步骤：

1、CPU 生成一个虚拟地址。

2 和 3、 MMU 从 TLB 中取出相应的 PTE

4、 MMU 将这个虚拟地址翻译成物理地址传送给高速缓存/主存

5、高速缓存/主存返回所请求的数据字给处理器 CPU

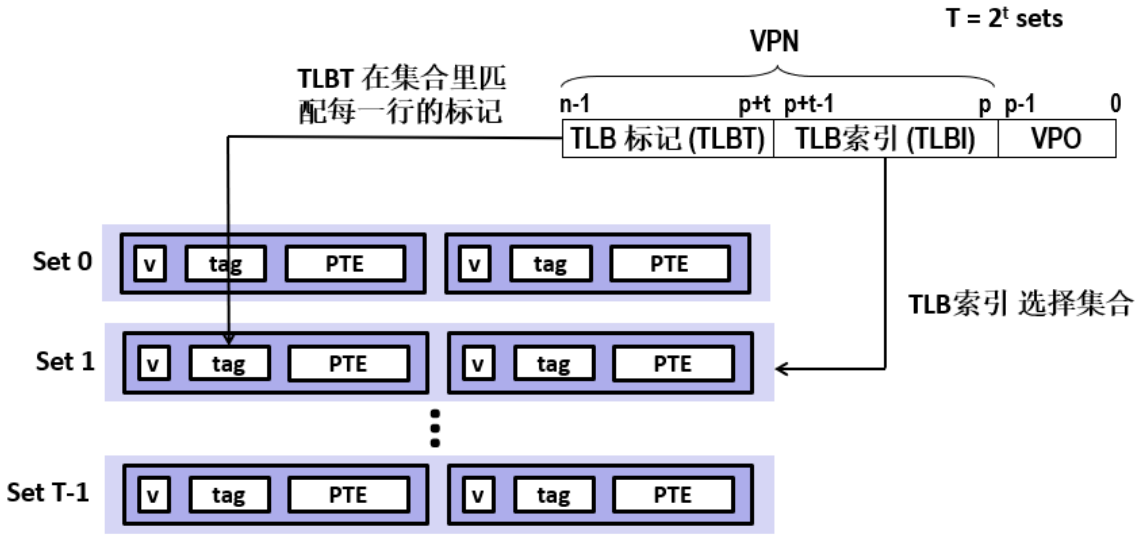


图 7.6-TLB 支持下的地址翻译

多级页表：

图 7-7 展示了我们构造的一个谅解的页表层次结构：

一级页表中的每个 PTE 负责映射虚拟地址空间中的一个 4MB 的片，这里每一个片都是有 1024 个连续的页面组成的。

二级页表中的每个 PTE 都负责一个 4KB 的虚拟内存页面，就像我们产看的只有一级的页表一样，注意，使用四字节的 PTE，每一个一级和二级页表都是 4KB 的字节，这刚好和一个页面的大小是一样的。

这种方法从两个方面减少了内存要求。第一，如果一个一级页表是空的，那么其对应的二级页表将不存在。这代表着一种巨大的潜在节约，移位对于一个典型的程序,四 GB 的虚拟地址空间的大部分都将会是未分配的。第二，只有一级页表才需要总是在主存里，虚拟系统可以再需要时创建，页面调入或者调出二级页表这就减少了主存的压力，只有经常使用的二级页表才会在主存里。

图 7-8 表示多级页表结构。

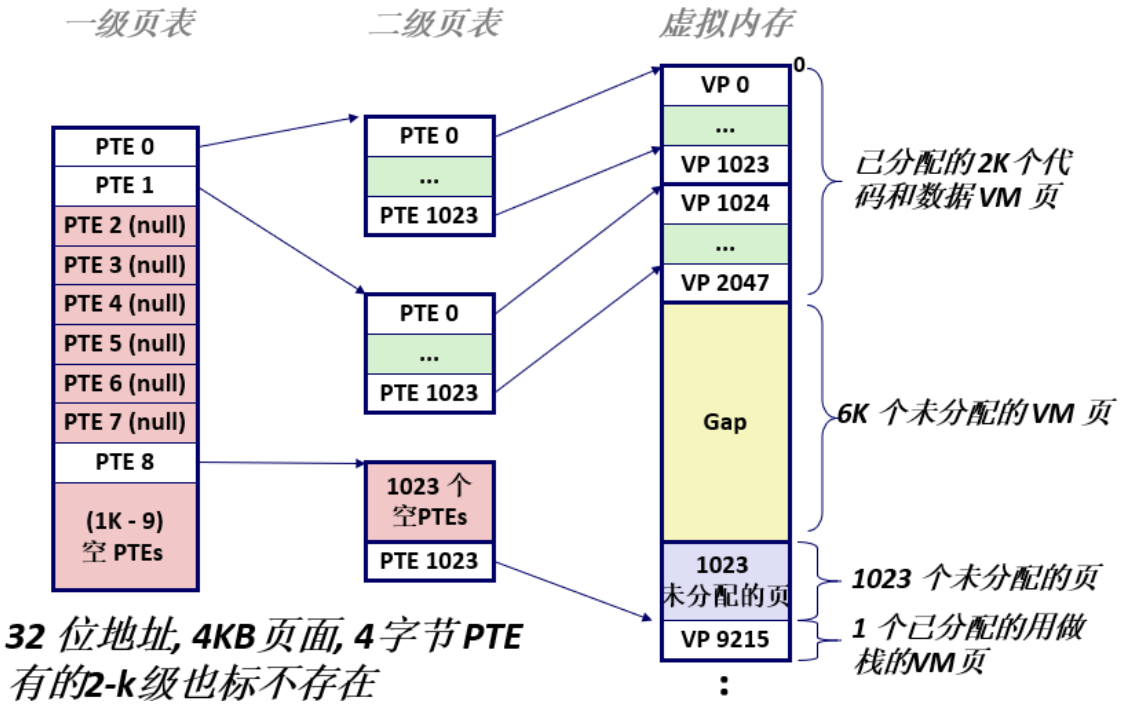


图 7-7-二级页表示例

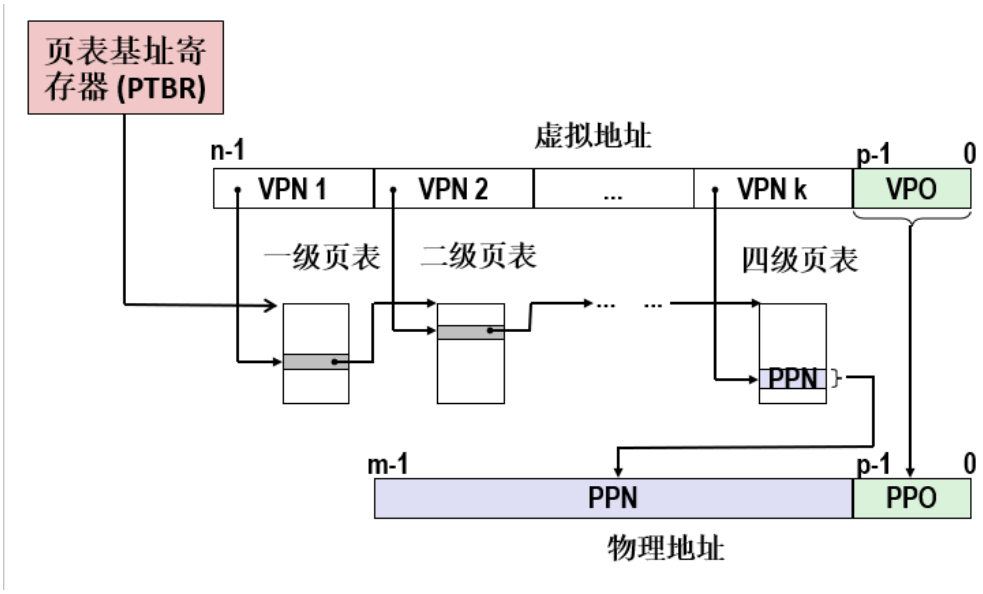


图 7-8-四级页表下的翻译

图 7-9 是 Core i7 MMU 如何使用四级页表来讲虚拟地址翻译成物理地址的全过程。36 位 VPN 被划分成了四个 9 位的片，每个片被用作到一个也表的偏移量。CR3 寄存器包含 L1 也表的物理地址。VPN1 提供一个到 L1PTE（页表条目）的偏

移量，这个 PTE 包含 L2 也表的基地址。VPN2 提供一个到 L2PTE 的偏移量，以此类推...。

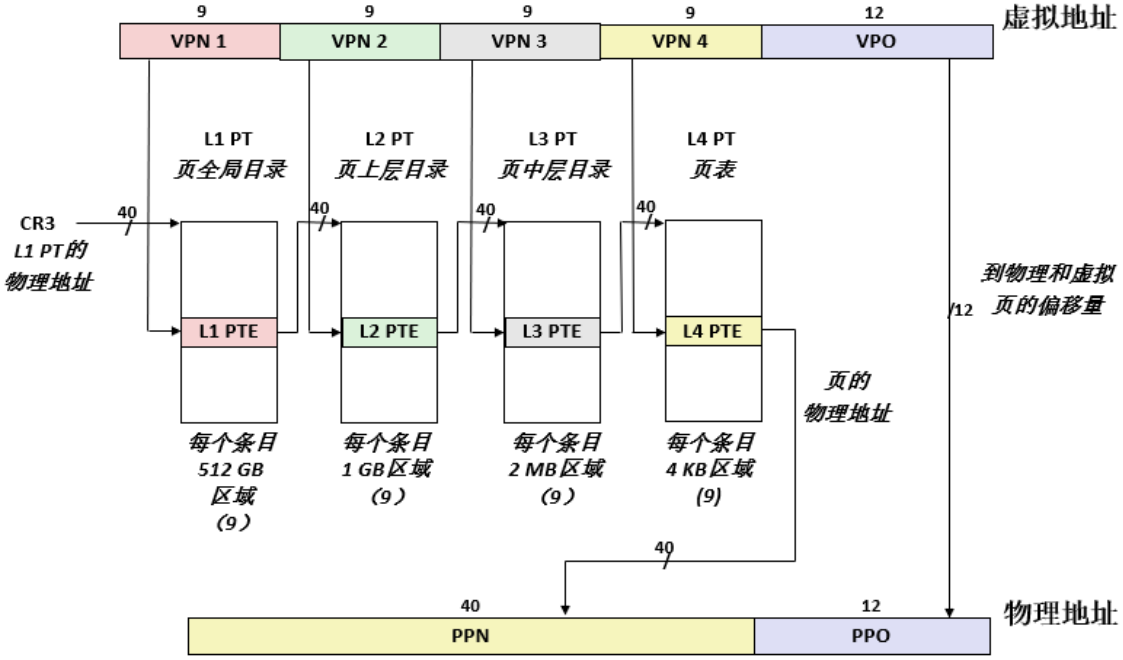
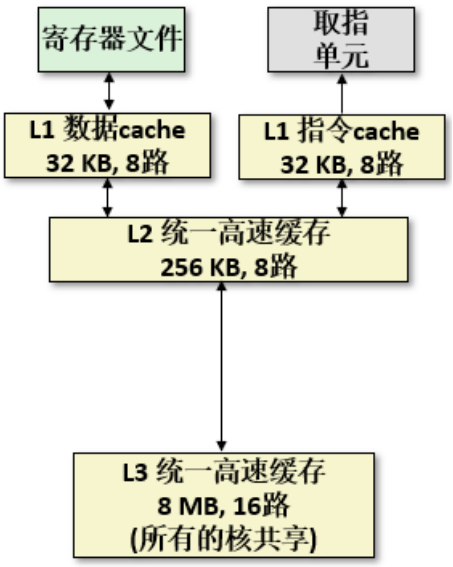


图 7-9-Core i7 下的四级页表翻译

7.5 三级 Cache 支持下的物理内存访问



三级 cache

运行的 Linux 的 Core i7 内存系统中对于处理器的封装很有讲究，我们可知的是，Linux 使用的是 4KB 的页，并且通过 TLB 虚拟寻址能得到我们想要的物理地址，然而有了这个物理地址之后，处理器可以通过对物理地址的处理而得到相应的在依赖物理地址寻址的 L1L2L3 三级高速缓存快速判断命中与否。

具体实现如下图（图 7-10）

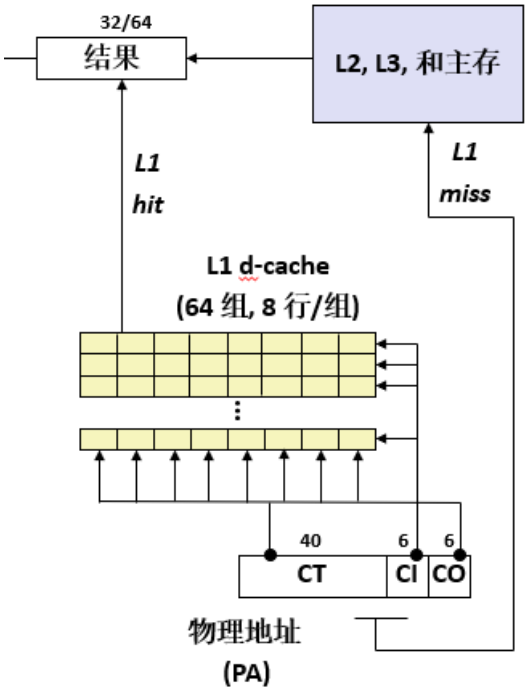


图 7.10-高速缓存对物理地址的处理阶段

事实上实际系统在运行的时候当在需要翻译虚拟地址的时，CPU 就已经将 VPN 发送到了高速缓存中。也就是说，理解翻译过程之后我们知道由于物理地址的 PPO 就是就是虚拟地址的 VPO，所以，在 MMU 忙着向 TLB 请求一个 PTE 页表条目的时候，L1 高速缓存实际上已经开始在分离组索引并查找相应的组了。这极大地情况上加快了翻译效率。

7.6 hello 进程 fork 时的内存映射

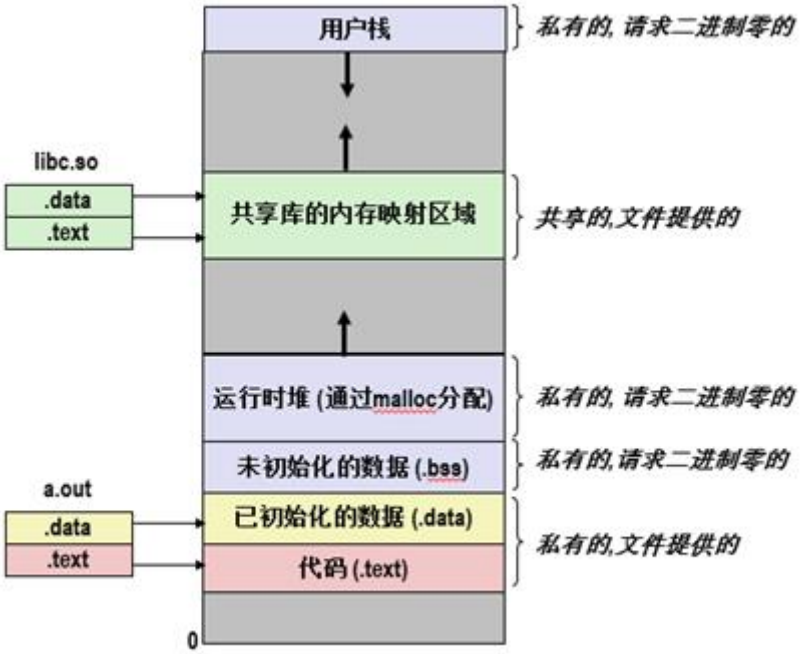
当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的 pid。为了给这个新进程创建虚拟内存，它创建了当前进程的 mm_struct、区域结构和页表的原样副本。它将两个进程中的每个页面都标记为只读，并将这两个进程中的每一个区域结构都标记为私有的写时复制。

当 fork 函数在新的进程中返回时，新进程现在的虚拟内存刚好和调用 fork 函数时存在的虚拟内存相同。当这两个进程中的人一个后来在读写操作时，写实复制机制

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。

7.7 hello 进程 execve 时的内存映射

exceve 执行步骤：
（1）删除已存在的用户区域
（2）创建新的区域结构（私有的、写时复制）
--代码和初始化数据映射到.text 和.data 区（目标文件提供）
--.bss 和栈堆映射到匿名文件 ，栈堆的初始长度 0
（3）共享对象由动态链接映射到本进程共享区域
（4）设置 PC，指向代码区域的入口点。Linux 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

缺页故障是一种常见的故障，当指令引用一个虚拟地址，在 MMU 中查找页表时发现与该地址相对应的物理地址不在内存中，就会触发一个缺页，这个异常导致控制转移到内核的缺页处理程序，处理程序随后会执行下面的步骤：

1、判断该虚拟地址是不是合法的，为了进行这个判断，缺页处理程序搜索区域结构的链表，如果这个指令是不合法的，那么缺页处理程序就会触发一个段错误，从而终止这个进程。

2、判断此时进行的内存访问是否合法，也就是说，判断进程是否有读、写或

者执行这个区域的权限，如果试图进行的访问是不合法的，那么缺页处理程序就会触发一个保护异常，从而终止这个进程。

3、经过上述处理判断，内核知道这个缺页是由于对合法的虚拟地址进行合法的操作造成的，它将会选择一个牺牲页面，如果这个牺牲页面被修改过，那么就将它交换出去，换入新的页面并更新页表，当缺页处理程序返回时，CPU 重新启动引用缺页的指令，这条指令将再次发送到这个虚拟地址到 MMU，这次 MMU 就会正常的翻译这个虚拟地址，而不会在产生缺页中断了。

处理过程如下：

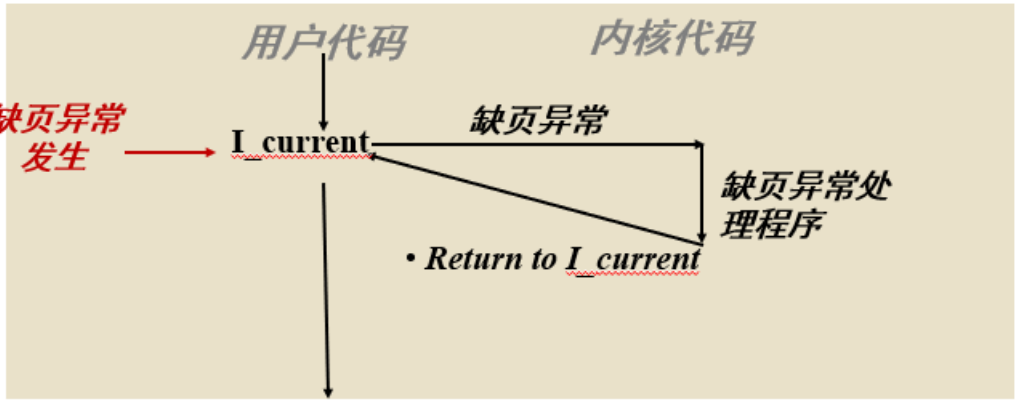


图 7-11-缺页异常处理

7.9 动态存储分配管理

动态内存分配器维护着一个进程的虚拟内存区域，称为堆（heap）（图 7-12），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面开始，并向上生长（更高的地址）。对于每一个进程，内核维护着一个变量 brk，用它来指向堆的顶部。

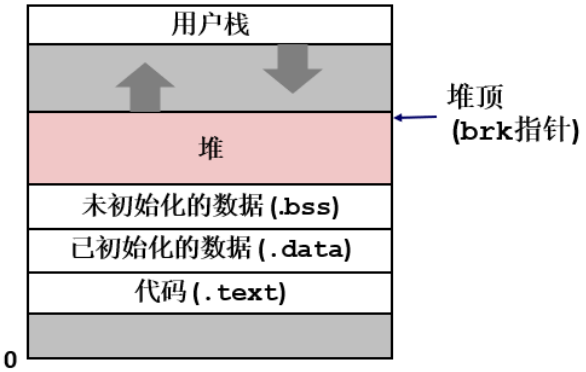


图 7-12-简化版虚拟内存空间

分配器将堆视为一组不同大小的 块(blocks)的集合来维护，每个块要么是已分配的，

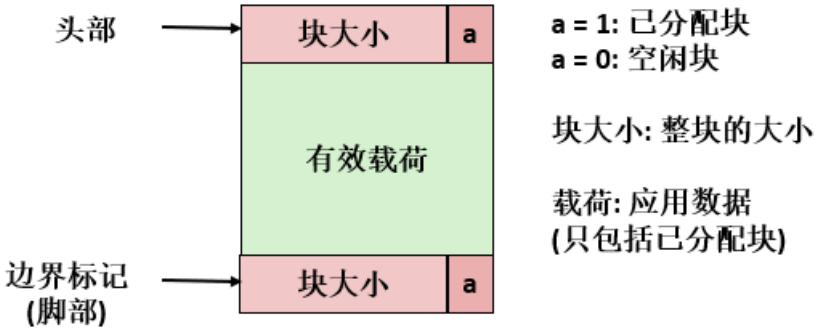
要么是空闲的。分配器有两种基本的风格。两种风格都要求应用显示分配快，他们的不同之处在于由哪个实体来负责释放已分配的块，分为显示分配器和隐式分配器

分配器必须在一些相当严格的约束条件下工作，比如必须能处理任何请求序列、立即响应请求、只是用堆、要满足对齐要求。

两种动态分配器的实现有：1）隐式空闲链表法 2）显示空闲链表法；

隐式空闲链表的实现：

1. 堆及堆中内存块的组织结构：



2.适配方法（放置策略）：

首次适配 (First fit)、下一次适配 (Next fit)和最佳适配 (Best fit):

3.分割空闲块：

在分配块小于空闲块的时候我们可以把空闲块分割成两部分；

4. 释放已分配块：

在程序中不没有用的块或者已经用完了的块需要释放回收；

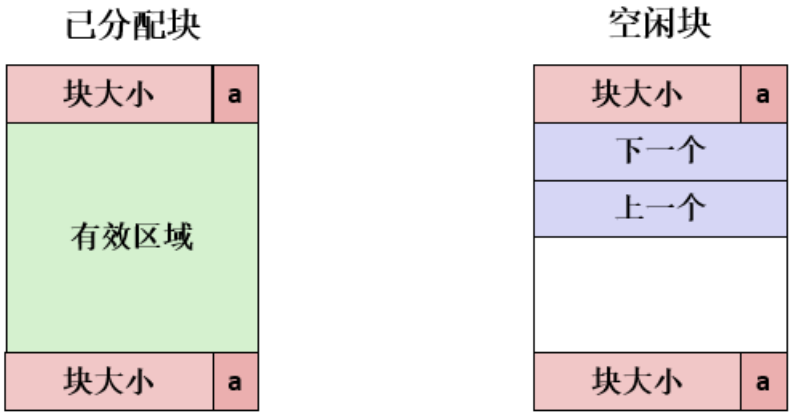
5. 合并相邻的空闲块：

立即合并 (Immediate coalescing): 每次释放都合并

延迟合并 (Deferred coalescing): 尝试通过延迟合并，即直到需要才合并来提高释放的性能.例如:为 malloc 扫描空闲链表时可以合并；外部碎片达到阈值时可以合并

显示空闲链表的实现：

堆中块的结构：



插入原则：1、后进先出（LIFO）的顺序维护链表 2、按照地址顺序来维护链表
一般而言，显示链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部，这就导致了更大的最小块大小，也潜在的提高了内部碎片的程度。

7.10 本章小结

虚拟内存是对主存的一个抽象，支持虚拟内存的处理器通过一种叫做虚拟寻址的间接形式来引用主存，处理器产生一个虚拟地址，在被发送到主存之前，这个地址被翻译成一个物理地址。从虚拟地址空间到物理地址空间的地址翻译要求硬件与软件的紧密结合，专门的硬件通过使用页表来翻译虚拟地址，而页表的内容是由操作系统提供的。

（第 7 章 2 分）

第 8 章 hello 的 I0 管理

8.1 Linux 的 I0 设备管理方法

一个 Linux 文件就是一个每个字节的序列：

B0 , B1 , ... , Bk , ... , Bm-1

所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入文和输出都被当做对相应文件的读和写来执行。这种设备优雅的映射为文件的方式，允许 Linux 内核引出一个简单的、低级的应用接口，成为 UnixI/O，这使得所有的输入和输出独能以一种统一且一致的方式来执行：

1、打开文件。一个应用程序通过要求内核打开相应的文件，来宣告他想要访问一个 I/O 设备。

2、Linux shell 创建的每个进程开始时都有三个打开的文件：标准输入（描述符为 0）、标准输出（描述符为 1）和标准错误（描述符为 2）。

3、改变当前的文件位置。对于每个打开的文件，内核保持着一个文件位置 k，初始为 0，这个文件位置是从文件开头起始的字节偏移。

4、读写文件。一个读操作就是从文件复制 n>0 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 k+n。

5、关闭文件。当应用完成了对文件的访问后，它就会通知内核关闭这个文件。

8.2 简述 Unix I0 接口及其函数

Unix IO 接口基本操作：

1.打开和关闭文件

open()and close()

2.读写文件

read() and write()

3.改变当前的文件位置 (seek)

指示文件要读写位置的偏移量

lseek()

函数的具体声明：

1.int open(char* filename,int flags,mode_t mode) ，进程通过调用 open 函数来打开一个存在的文件或是创建一个新文件的。open 函数将 filename 转换为一个文件描述符，并且返回描述符数字，返回的描述符总是在进程中当前没有打开的最小描述符，否则返回值为-1 表示一个错误。flags 参数指明了进程访问这个文件的形式。mode 参数指定了新文件的访问权限位。

2.int close(fd)，fd 是需要关闭的文件的描述符，close 返回操作结果（成功为 0 出错为-1）。关闭一个已关闭的描述符会出错！

3.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表示成功。

4.size_t wirte(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。返回值-1 表示一个错误，否则返回值表示实际传送的字节数量表示成功。

5.lseek 函数，应用程序调用该函数能够显示地修改当前文件的位置。

8.3 printf 的实现分析

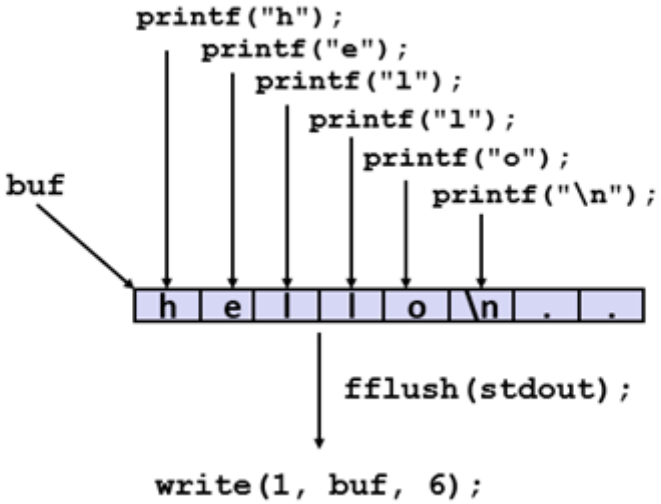


图 8.2-printf 函数的一个简单的实现过程

C 语言标准输入输出库中对于 printf 函数的定义是这样的：

```
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;
}
```

从上述代码中我们可以看到 它声明了一个缓冲区变量，大小是 256，又声明了一个类型为 va_list（定义为指针型变量）的变量，其中((char*)(&fmt) + 4)这部分代表 printf 参数中“...”的第一个参数，而参数中的 fmt 正好指向第一个参数。

好了了解了这些之后，我们再看看 vsprintf 的实现:

```
int vsprintf(char *buf, const char *fmt, va_list args)
    {
     char* p;
     char tmp[256];
     va_list p_next_arg = args;

     for (p=buf;*fmt;fmt++) { //从前向后扫描所有字符串中字符
        if (*fmt != '%') { //判断不是“%”就跳出继续循环
            *p++ = *fmt;
            continue;
        }

        fmt++; //判定是“%”，第一个%后面接的内容很重要，与参数有关例如：%d，那第一个参数就应该是整型变量

        switch (*fmt) { //分情况判断%后面规定的参数格式
            case 'x':
                itoa(tmp, *((int*)p_next_arg)); //参数写进缓冲区全过程
                strcpy(p, tmp);
                p_next_arg += 4;
                 p += strlen(tmp);
```

```
                break;
            case 's':
                break;
            default:
                 break;
        }
    }

    return (p - buf); //循环结束返回需要打印的字符串的长度。
}
```

详见注释说明， 已尽可能的详细。

再回到原 printf，此时 i 已经被设置为需要打印字符串的长度，接下来就是 write 的实现了，不用说也知道这句话无非是想告诉 OS，我需要打印出在缓冲区中的 i 个字符，下面追踪到系统 write 函数的反汇编语言实现；

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

其实简单来看，不过是放到六十四位系统，通过寄存器传了两个参数然后调用了一下系统函数就结束了，此间不做深究。

最后，纵观全局，printf 函数其实并不能确定其参数在什么地方结束，也不知道参数的个数，它只会根据 format 中打印格式的数目依次打印堆栈中参数 format 后面地址的内容直到结束，这一点其实在我们高级语言设计 C 语言代码实现过程中已经有所体会。

8.4 getchar 的实现分析

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

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

8.5 本章小结

Linux 提供了少量的基于 UnixI/O 模型的系统函数，他们允许应用程序打开、关闭、都和写文件。读取文件中的元数据，以及执行 I/O 重定向。

（第 8 章 1 分）

结论

1. C 语言实现--文本编辑器编写完毕保存时对扩展名的修改，诞生源程序
2. 预处理--将 hello.c 源程序所有调用的外部库扩展到该源程序中诞生 hello.i
3. 编译器处理 hello.i 文件编译成为 hello.s 汇编语言文件
4. 汇编器处理 hello.s 文件汇编成为 hello.o 可重定位文件
5. 链接器处理 hello.o 文件生成 hello 可执行文件
6. Shell 下键入“./hello 学号 姓名”，运行 hello
7. shell 通过 fork 函数创建子进程运行 hello 并且调用 execve，execve 调用启动加载器，加映射虚拟内存，虚拟地址映射到物理地址，运行到 main 函数
8. CPU 逐步执行 hello 中机器语言指令
9. 内存访问访问内存空间
10. 信号处理--遇到 shell 中的个别信号进入信号处理程序
11. 回收子进程--程序执行完毕后交由父进程回收子进程，结束以及执行

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

Hello.c（c 程序）， hello.i（预处理文件），hello.s（汇编文件）， hello.o（可重定位文件），hello（可执行文件）

（附件 0 分，缺失 -1 分）

参考文献

[1] 深入理解计算机系统

[2] . https://www.cnblogs.com/pianist/p/3315801.html

[3] 操作系统设计与实现

[4] 维基百科.

（参考文献 0 分，缺失 -1 分）