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　1180800916

班　　级　1803004

学　　生　陶子康

指 导 教 师　史先俊

计算机科学与技术学院

2019 年 *12* 月

摘　要

计算机系统是一个程序员的基本功，学好计算机系统是写出快速、可靠、移植性强的程序必不可少的前提条件。本文简要地描述了一个程序 hello 经过预处理、编译、汇编、链接，被加载到内存中作为一个进程运行，最后终止被父进程回收的简洁却不简单的一生，贯穿整个课本，旨在为其他同学提供参考，也为自己的漫漫复习长路理理思路。

关键词：计算机系统、汇编、链接、进程、虚拟内存

（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

P2P(Program to Process)：由程序员编写出 C 程序(Program)，将她预处理、编译、汇编、链接后生成可执行目标文件，再将利用加载器将其加载到内存，这时她就成为一个运行中的程序实例——进程(Process)

020(Zero-0 to Zero-0)：在程序员编写 hello.c 程序前，一切为空，即'0'；在编写程序，由 gcc 编译链接后，操作系统将其加载到内存中作为一个进程运行一段时间，而后终止，操作系统再将她回收，从此，她在内存中存在的一切痕迹都被消除，仿佛没有来过，回归于'0'。

1.2 环境与工具

硬件环境：X64CPU，1.6GHz，8GB RAM，500 GB HD

软件环境：Ubuntu 18.04 LTS 64 位 / Windows 10 64 位

开发工具：gcc、objdump、readelf、edb 等

1.3 中间结果

hello.i——预处理文件，预处理器(cpp)处理以'#'开头的命令，修改原始的 C 程序，得到.i 文件。它作为编译器的输入

hello.s——汇编文件，编译器(cc1)将.i 文件翻译成汇编语言，得到.s 文件。它包含汇编代码，作为汇编器的输入

hello1.s——汇编文件，是将 hello.o 通过 objdump 命令得到的

hello2.s——汇编文件，是将 hello.o 通过 objdump 命令得到的

hello.o——可重定位目标文件，汇编器(as)将.s 文件翻译为机器语言指令，得到.o 文件。它包含机器代码，作为链接器的输入

hello.out——可执行目标文件，链接器(ld)将各种.o 文件进行链接，得到可执行目标文件，它可以被加载到内存中，由系统执行

hello——可执行目标文件，由 gcc -m64 -no-pie -fno-PIC hello.c -o hello 命令直接生成

1.4 本章小结

本章简要介绍了 P2P，020 的过程，介绍了生成可执行目标文件的中间过程——预处理、编译、汇编、链接等四个阶段及其产生的中间结果文件，介绍了笔者的实验环境与使用的开发工具等

（第 1 章 0.5 分）

第 2 章 预处理

2.1 预处理的概念与作用

概念：在编译之前进行的处理

作用：解析以'#'开头的命令。预处理将程序包含的头文件直接加入到程序中；它还会解析宏定义，完成替换；它还能完成条件编译

2.2 在 Ubuntu 下预处理的命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ cpp -m64 -no-pie -fno-PIC hello.c -o hello.i
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 1 Ubuntu 下预处理命令截图

2.3 Hello 的预处理结果解析

```
extern int rpmatch (const char *__response) __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1))) ;
# 954 "/usr/include/stdlib.h" 3 4
extern int getsubopt (char **__restrict __optionp,
        char *const *__restrict __tokens,
        char **__restrict __valuep)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1, 2, 3))) ;
# 1006 "/usr/include/stdlib.h" 3 4
extern int getloadavg (double __loadavg[], int __nelem)
     __attribute__ ((__nothrow__ , __leaf__)) __attribute__ ((__nonnull__ (1)));
# 1016 "/usr/include/stdlib.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/stdlib-float.h" 1 3 4
# 1017 "/usr/include/stdlib.h" 2 3 4
# 1026 "/usr/include/stdlib.h" 3 4

# 9 "hello.c" 2


# 10 "hello.c"
int main(int argc,char *argv[]){
 int i;

 if(argc!=4){
  printf("用法: Hello 学号 姓名 秒数! \n");
  exit(1);
 }
 for(i=0;i<8;i++){
  printf("Hello %s %s\n",argv[1],argv[2]);
  sleep(atoi(argv[3]));
 }
 getchar();
 return 0;
}
```

图 2 生成的预处理文件 hello.i 的部分代码

如上图，仅截取了一小部分。hello.c 程序经过预处理后，由短短的不过二十多行变成了超过三千行的代码，预处理将 hello.c 用到的头文件直接插入到程序文本中，得到预处理文件

2.4 本章小结

预处理是源程序到可执行目标文件的第一步，它解释以'#'开头的代码，生成.i 文件，为下一步编译作充足的准备。本章介绍了与处理的概念与作用、ubuntu 下预处理的命令以及 hello 的预处理效果解析

（第 2 章 0.5 分）

第 3 章 编译

3.1 编译的概念与作用

概念：利用编译程序将源语言编写的源程序产生目标程序的过程

作用：将 C 语言程序翻译成汇编语言代码，为汇编器提供输入

3.2 在 Ubuntu 下编译的命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ gcc -m64 -no-pie -fno-PIC -S hello.c -o
 hello.s
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 3 ubuntu 下编译命令

3.3 Hello 的编译结果解析

3.3.1 数据

int 型局部变量，如 hello.c 中的 i 和 argc，在通用寄存器可用的情况下，编译器将其存储在通用寄存器中，如果通用寄存器不可用，则将它们存储在栈中

数组：编译器根据数组的大小和每个数据所占的字节数为它们在栈上分配空间，在访问某一元素时，编译器通过数组下标常量，计算出该元素在数组中的偏移量从而访问到该元素

指针：*，编译器根据编译环境(32 位/64 位)，为其在栈上分配 4 字节或 8 字节的空间，当取其内容时则按如(%rdi)的方式访问指针指向的地址存储的内容

字符串常量，如 hello.c 中的”用法: Hello 学号 姓名 秒数！\n”以及 "Hello %s %s\n"，编译器将它们看作为只读数据，存放在只读数据段

3.3.2 各类操作

++运算符：编译器通过 add 指令实现++

赋值=：编译器通过 mov 指令实现赋值，还有 leaq 来完成加载有效地址，也可用于计算赋值

+、-、×、/：编译器使用 mul、div、add、sub、shl、shr 等指令来完成算数运算，leaq 也能完成算数运算

关系操作!=、<等：编译器通过 cmp 指令，将关系运算符左右两边的表达式的值作差在与一个常数(如 0、1 等)比较，同时设置标志位

if 条件判断：编译器通过条件跳转来实现 if 语句，编译器根据表达式来设置标

志位，再利用条件跳转指令，如 ja，jne 等实现跳转，若不满足跳转条件则执行下一条指令

for 循环语句：通常情况下，编译器将循环变量存储在一个通用寄存器中，并 jmp 到循环边界判断指令的位置进行判断，设置标志位，再根据标志位决定是跳转到循环体对应的指令，还是执行下一条指令，结束循环

函数：

参数传递：在 32 位环境下，函数参数通过栈来传递；在 64 位环境下，当函数的参数不超过 6 个时，函数通过寄存器传参，当参数个数超过 6 个时，超过部分通过栈传参。前 6 个参数分别存放在%rdi、%rsi、%rdx、%rcx、%r8、%r9 中

函数调用：编译器利用 call 指令来实现对函数的调用，call 指令将当前 PC 保存下来，并将其值加上 call 指令紧接着的立即数所得到的值作为当前的 PC，完成对函数的调用

函数返回 return：ret 指令，编译器将函数的返回值存储在通用寄存器%rax 中，函数返回时，会将其他寄存器恢复到调用以前的状态，将 PC 还原到调用函数前的值，继续执行下一条指令

栈：popq、pushq 等，pop 弹出栈顶元素并赋给紧跟的操作数，push 将一个元素压栈

3.4 本章小结

编译是从源文件到可执行目标文件的一个重要的中间过程，它将源程序文件翻译为汇编语言代码，输出汇编语言.s 文件中的符号。本章介绍了编译的概念和作用、Ubuntu 下编译的命令以及编译的结果解析

（第 3 章 2 分）

第 4 章 汇编

4.1 汇编的概念与作用

概念：将汇编语言翻译为机器语言的过程

作用：将汇编语言翻译为机器语言，生成可重定位目标文件，为链接器提供输入

4.2 在 Ubuntu 下汇编的命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ as hello.s -o hello.o
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ls
hello.c  hello.o  HIT-ICS2019大作业报告模板.doc
hello.i  hello.s  HIT-ICS2019大作业要求.ppt
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 4 ubuntu 下汇编命令

4.3 可重定位目标 elf 格式

hello.o 是可重定位目标文件，其文件格式如下图所示

ELF 头
.text
.rodata
.data
.bss
.rel.text
.rel.data
.debug
.line
.strtab
节头部表

图 5 典型的 ELF 可重定位目标文件

ELF 头： 描述文件的总体格式，包括程序的入口点、ELF 头的大小、目标文件类型、机器类型、节头部表的文件偏移、节头部表中条目的大小和数量；

.text： 是程序的机器代码；

.rodata： 是只读数据，如 printf 中的格式串和 switch 语句中的跳转表；

.data： 存放已初始化的全局和静态 C 变量；
.bss： 存放未初始化的和初始化为 0 的全局和静态变量（在文件中、磁盘中不占据实际空间、运行时在内存中分配空间、其大小由节头部表指出）；
.debug： 调试符号表
.symtab： 符号表，存放在程序中定义和引用的函数和全局变量的信息……；
节头部表： 描述不同的节的位置和大小。
.rel：重定位信息

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ readelf -S -W hello.o
There are 13 section headers, starting at offset 0x460:

节头：
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00008a 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 000320 0000c0 18   I 10   1  8
  [ 3] .data             PROGBITS        0000000000000000 0000ca 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 0000ca 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        0000000000000000 0000d0 000033 00   A  0   0  8
  [ 6] .comment          PROGBITS        0000000000000000 000103 00002a 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000 00012d 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        0000000000000000 000130 000038 00   A  0   0  8
  [ 9] .rela.eh_frame    RELA            0000000000000000 0003e0 000018 18   I 10   8  8
  [10] .symtab           SYMTAB          0000000000000000 000168 000180 18     11   9  8
  [11] .strtab           STRTAB          0000000000000000 0002e8 000032 00      0   0  1
  [12] .shstrtab         STRTAB          0000000000000000 0003f8 000061 00      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)
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 6 hello.o 的各节基本信息

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ readelf -r hello.o

重定位节 '.rela.text' at offset 0x320 contains 8 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000016  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000000001b  000a00000002 R_X86_64_PC32     0000000000000000 puts - 4
000000000025  000b00000002 R_X86_64_PC32     0000000000000000 exit - 4
00000000004c  00050000000a R_X86_64_32       0000000000000000 .rodata + 26
000000000056  000c00000002 R_X86_64_PC32     0000000000000000 printf - 4
000000000069  000d00000002 R_X86_64_PC32     0000000000000000 atoi - 4
000000000070  000e00000002 R_X86_64_PC32     0000000000000000 sleep - 4
00000000007f  000f00000002 R_X86_64_PC32     0000000000000000 getchar - 4

重定位节 '.rela.eh_frame' at offset 0x3e0 contains 1 entry:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 7 hello.o 的重定位信息

4.4 Hello.o 的结果解析

```
        .file   "hello.c"
        .text
        .section        .rodata
        .align 8
.LC0:
        .string "\347\224\250\346\26
\345\247\223\345\220\215 \347\247\22
.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    $4, -20(%rbp)
        je      .L2
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $1, %edi
        call    exit@PLT
.L2:
        movl    $0, -4(%rbp)
        jmp     .L3
.L4:
        movq    -32(%rbp), %rax
        addq    $16, %rax
        movq    (%rax), %rdx
        movq    -32(%rbp), %rax
        addq    $8, %rax
        movq    (%rax), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movq    -32(%rbp), %rax
```

图 8 第三章的 hello.s

```
hello.o:     文件格式 elf64-x86-64


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 04             cmpl   $0x4,-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 46                   jmp    78 <main+0x78>
  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+0x26
  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:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  5e:   48 83 c0 18             add    $0x18,%rax
  62:   48 8b 00                mov    (%rax),%rax
  65:   48 89 c7                mov    %rax,%rdi
  68:   e8 00 00 00 00          callq  6d <main+0x6d>
                        69: R_X86_64_PC32       atoi-0x4
  6d:   89 c7                   mov    %eax,%edi
  6f:   e8 00 00 00 00          callq  74 <main+0x74>
                        70: R_X86_64_PC32       sleep-0x4
  74:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  78:   83 7d fc 07             cmpl   $0x7,-0x4(%rbp)
  7c:   7e b4                   jle    32 <main+0x32>
  7e:   e8 00 00 00 00          callq  83 <main+0x83>
```

图 9 objdump 得到的 hello1.s

对照分析：两个文件最大的区别是有无机器语言代码，总的来说由 objdump 得到的 hello1.s 的可读性更好，hello.s 去除汇编语言伪指令与 hello1.s 基本一致，hello.s 中没有重定位信息，且只读数据直接用一个标签表示出来，如.LC1 代表”Hello %s %s\n“

机器语言构成：机器语言由指令和操作数构成。

映射关系：一条汇编代码对应一串机器代码，机器代码的长度由汇编代码的种类决定，一般而言，机器代码的操作数对应汇编代码中的操作数，如 movl $0x0,-0x4(%rbp)对应的机器代码为 c7 45 fc 00 00 00 00，末尾 4 个 00 代表了立即数 0；然而，机器代码的操作数也可能与汇编代码中的操作数不一致，如 jmp 指令和 call 指令，在机器代码中，它们的操作数是跳转目的地指令的地址和它们的下一条指令的地址差的 16 进制补码表示

4.5 本章小结

汇编是将汇编语言代码输出为可重定位目标文件的过程。本章介绍了汇编的概念、ubuntu 下汇编的命令、可重定位目标文件 elf 格式、hello.o 的结果解析

（第 4 章 1 分）

第 5 章 链接

5.1 链接的概念与作用

概念：链接是将各种代码和数据片段收集并合并成为一个单一文件的过程

作用：完成符号解析和重定位任务，使得代码文件成为可执行目标文件，并能加载到内存中运行

5.2 在 Ubuntu 下链接的命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ 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/7/crtbegin.o hello.o -lc /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -z relro -o hello.out
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 10 ubuntu 下 ld 链接命令

5.3 可执行目标文件 hello 的格式

ELF 头
段头部表
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
节头部表

图 11 典型的 ELF 可执行目标文件格式

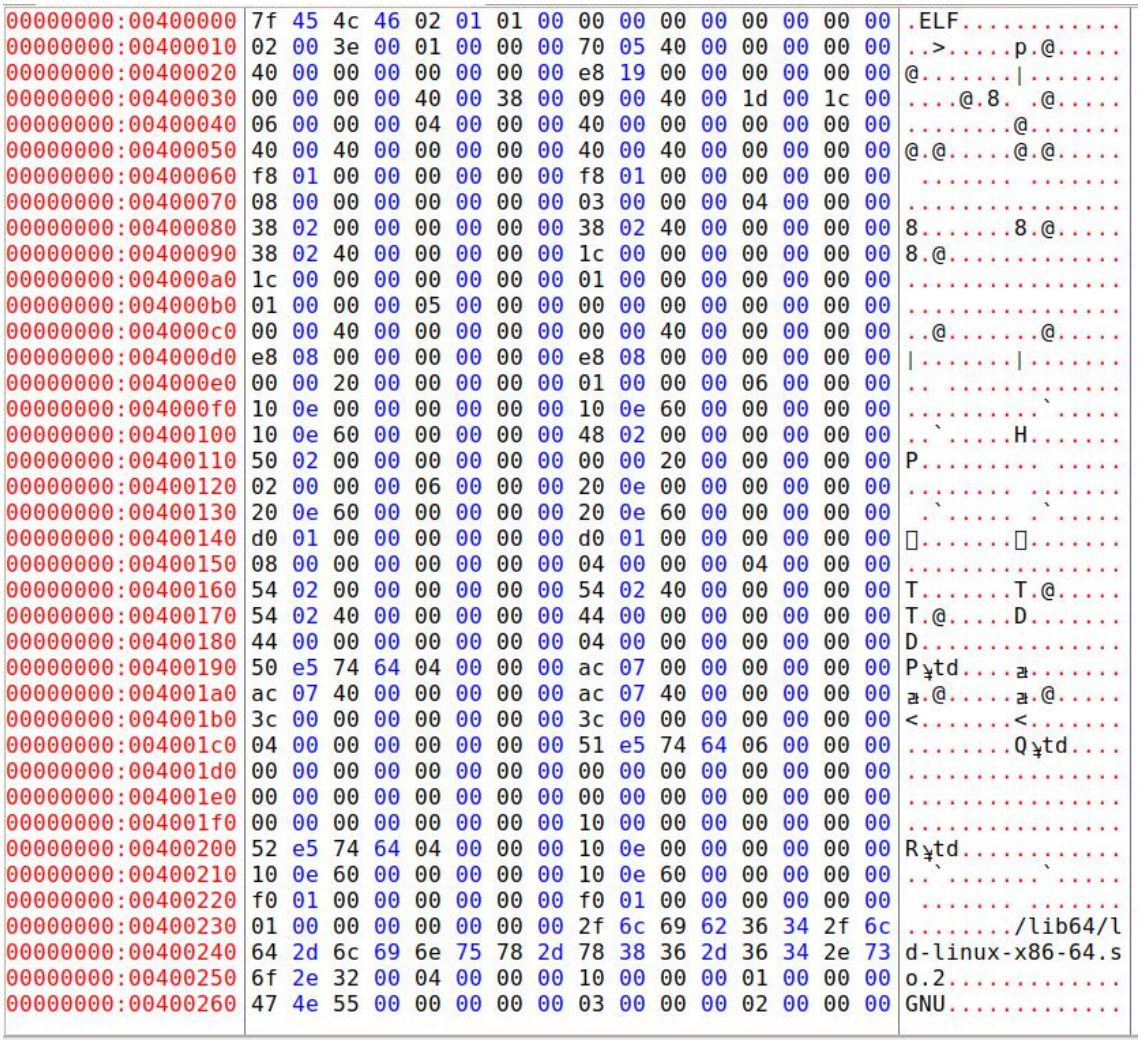
程序头部表： 描述了可执行文件各片、节与虚拟内存中各段之间的映射关系；

.init： 定义了_init 函数，用于程序初始化；

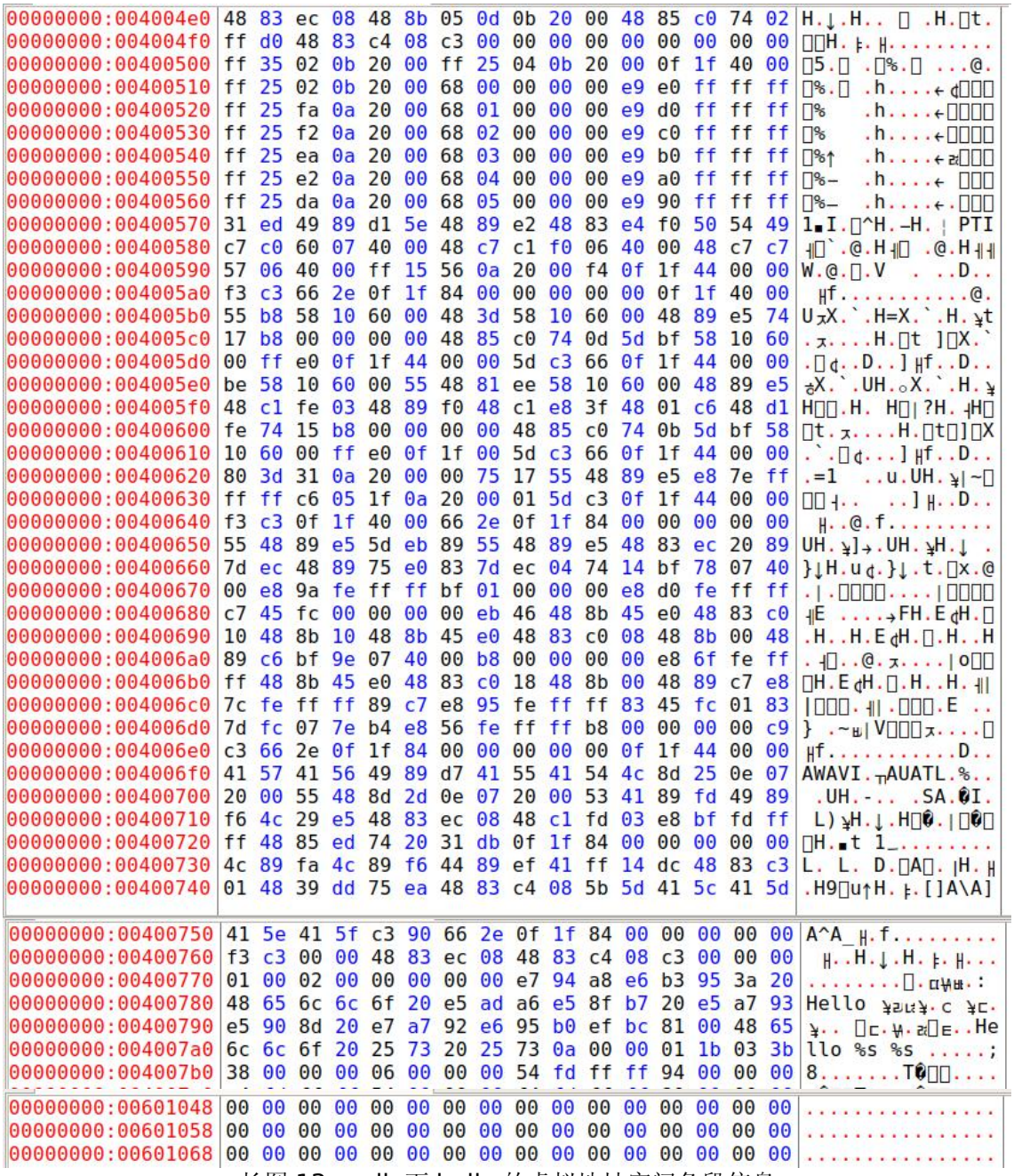
```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ readelf -S -W hello
There are 29 section headers, starting at offset 0x19e8:

节头：
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000400238 000238 00001c 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            0000000000400254 000254 000020 00   A  0   0  4
  [ 3] .note.gnu.build-id NOTE            0000000000400274 000274 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0000000000400298 000298 00001c 00   A  5   0  8
  [ 5] .dynsym           DYNSYM          00000000004002b8 0002b8 0000d8 18   A  6   1  8
  [ 6] .dynstr           STRTAB          0000000000400390 000390 00005c 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          00000000004003ec 0003ec 000012 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         0000000000400400 000400 000020 00   A  6   1  8
  [ 9] .rela.dyn         RELA            0000000000400420 000420 000030 18   A  5   0  8
  [10] .rela.plt         RELA            0000000000400450 000450 000090 18  AI  5  22  8
  [11] .init             PROGBITS        00000000004004e0 0004e0 000017 00  AX  0   0  4
  [12] .plt              PROGBITS        0000000000400500 000500 000070 10  AX  0   0 16
  [13] .text             PROGBITS        0000000000400570 000570 0001f2 00  AX  0   0 16
  [14] .fini             PROGBITS        0000000000400764 000764 000009 00  AX  0   0  4
  [15] .rodata           PROGBITS        0000000000400770 000770 00003b 00   A  0   0  8
  [16] .eh_frame_hdr     PROGBITS        00000000004007ac 0007ac 00003c 00   A  0   0  4
  [17] .eh_frame         PROGBITS        00000000004007e8 0007e8 000100 00   A  0   0  8
  [18] .init_array       INIT_ARRAY      0000000000600e10 000e10 000008 08  WA  0   0  8
  [19] .fini_array       FINI_ARRAY      0000000000600e18 000e18 000008 08  WA  0   0  8
  [20] .dynamic          DYNAMIC         0000000000600e20 000e20 0001d0 10  WA  6   0  8
  [21] .got              PROGBITS        0000000000600ff0 000ff0 000010 08  WA  0   0  8
  [22] .got.plt          PROGBITS        0000000000601000 001000 000048 08  WA  0   0  8
  [23] .data             PROGBITS        0000000000601048 001048 000010 00  WA  0   0  8
  [24] .bss              NOBITS          0000000000601058 001058 000008 00  WA  0   0  1
  [25] .comment          PROGBITS        0000000000000000 001058 000029 01  MS  0   0  1
  [26] .symtab           SYMTAB          0000000000000000 001088 000630 18     27  43  8
  [27] .strtab           STRTAB          0000000000000000 0016b8 000229 00      0   0  1
  [28] .shstrtab         STRTAB          0000000000000000 0018e1 000103 00      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)
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 12 hello 各段的基本信息

5.4 hello 的虚拟地址空间


00000000:00400000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000000:00400010 02 00 3e 00 01 00 00 00 70 05 40 00 00 00 00 00 ..>.....p.@.....
00000000:00400020 40 00 00 00 00 00 00 00 e8 19 00 00 00 00 00 00 @.......|.......
00000000:00400030 00 00 00 00 40 00 38 00 09 00 40 00 1d 00 1c 00 ....@.8. .@.....
00000000:00400040 06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00 ....@.......
00000000:00400050 40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00 @.@.....@.@.....
00000000:00400060 f8 01 00 00 00 00 00 00 f8 01 00 00 00 00 00 00 ....
00000000:00400070 08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00 ....
00000000:00400080 38 02 00 00 00 00 00 00 38 02 40 00 00 00 00 00 8.......8.@.....
00000000:00400090 38 02 40 00 00 00 00 00 1c 00 00 00 00 00 00 00 8.@.............
00000000:004000a0 1c 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ....
00000000:004000b0 01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00 ....
00000000:004000c0 00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 ..@.......@.....
00000000:004000d0 e8 08 00 00 00 00 00 00 e8 08 00 00 00 00 00 00 |.......|.......
00000000:004000e0 00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00 .. ..
00000000:004000f0 10 0e 00 00 00 00 00 00 10 0e 60 00 00 00 00 00 ....`.....
00000000:00400100 10 0e 60 00 00 00 00 00 48 02 00 00 00 00 00 00 ..`.....H.......
00000000:00400110 50 02 00 00 00 00 00 00 00 00 20 00 00 00 00 00 P.......... .....
00000000:00400120 02 00 00 00 06 00 00 00 20 0e 00 00 00 00 00 00 ....
00000000:00400130 20 0e 60 00 00 00 00 00 20 0e 60 00 00 00 00 00 .`..... .`.....
00000000:00400140 d0 01 00 00 00 00 00 00 d0 01 00 00 00 00 00 00 □.......□.......
00000000:00400150 08 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00 ....
00000000:00400160 54 02 00 00 00 00 00 00 54 02 40 00 00 00 00 00 T.......T.@.....
00000000:00400170 54 02 40 00 00 00 00 00 44 00 00 00 00 00 00 00 T.@.....D.......
00000000:00400180 44 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 D...............
00000000:00400190 50 e5 74 64 04 00 00 00 ac 07 00 00 00 00 00 00 P¥td....¬.......
00000000:004001a0 ac 07 40 00 00 00 00 00 ac 07 40 00 00 00 00 00 ¬.@.....¬.@.....
00000000:004001b0 3c 00 00 00 00 00 00 00 3c 00 00 00 00 00 00 00 <.......<.......
00000000:004001c0 04 00 00 00 00 00 00 00 51 e5 74 64 06 00 00 00 ....Q¥td....
00000000:004001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00000000:004001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00000000:004001f0 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 ....
00000000:00400200 52 e5 74 64 04 00 00 00 10 0e 00 00 00 00 00 00 R¥td............
00000000:00400210 10 0e 60 00 00 00 00 00 10 0e 60 00 00 00 00 00 ..`.......`.....
00000000:00400220 f0 01 00 00 00 00 00 00 f0 01 00 00 00 00 00 00 ....
00000000:00400230 01 00 00 00 00 00 00 00 2f 6c 69 62 36 34 2f 6c ..../lib64/l
00000000:00400240 64 2d 6c 69 6e 75 78 2d 78 38 36 2d 36 34 2e 73 d-linux-x86-64.s
00000000:00400250 6f 2e 32 00 04 00 00 00 10 00 00 00 01 00 00 00 o.2.............
00000000:00400260 47 4e 55 00 00 00 00 00 03 00 00 00 02 00 00 00 GNU.............

这里是.interp节



长图 13　edb 下 hello 的虚拟地址空间各段信息

代码段从 0x400000 开始，第一部分是只读内存段(代码段)，0x238 以前的部分是 ELF 头以及程序头部表；偏移 0x238 处开始的 0x1c 字节内容是.interp 段，它存储的是动态连接器的目录；偏移 0x4e0 处开始的 0x17 字节内容是.init 段，该节中定义了一个小函数叫做_init，程序初始化会用到它；偏移 0x570 处开始的 0x1f2 字节内容是.text 段，它存放了程序的机器级代码；偏移 0x770 处开始的 0x3b 字节内容是.rodata 段，它存放程序的只读数据；接下来，进入读/写内存段(数据段)，相对 0x600000 处偏移 0x1048 开始的 0x10 字节内容是.data 段，它存储了已初始化(非初始化为 0)的全局变量和静态变量；相对 0x600000 处偏移 0x1058 开始的 0x8 字节内容是.bss 段，它存储未初始化的全局和静态变

量以及初始化为 0 的全局和静态变量，该段不占实际空间；.bss 段后面有 4 个节没有虚拟地址，因为它们不会被加载到虚拟内存中运行

5.5 链接的重定位过程分析

hello2.s 是 hello 经过 objdump 得到的，如下图 15

hello1.s 是 hello.o 经过 objdump 得到的，如下图 16

不同：hello 是可执行目标文件，hello.o 是可重定位目标文件。hello 已经完全链接，所以在 hello2.s 中没有重定位信息，程序中的每一条指令和全局变量都有唯一的运行时内存地址了，所有的符号引用也已经定位完成；而 hello.o 需要重新定位，所以在 hello1.s 中，存在未定位的符号引用，除此之外，指令和全局变量的运行时地址还未确定。

链接过程：链接器完成两个主要任务——符号解析和重定位；在链接器完成符号解析后，就开始重定位工作，重定位由两步组成：

1)重定位节和符号定义，在这一过程中，链接器将所有相同类型的节合并为同一类型的聚合节，然后链接器将运行时地址赋给新的聚合节、输入目标模块的每个节以及输入模块定义的每个符号。至此，程序中的每一条指令和全局变量都有唯一的运行时内存地址了

2)重定位节中的符号引用，在这一过程中，链接器依赖于重定位条目修改代码节和数据节中对每个符号的引用，使它们指向正确的运行时地址

如何重定位的：图 14 是 hello.o 的重定位信息，可以看到在.text 节偏移 0x16、0x1b、0x25、0x4c、0x56、0x69、0x70、0x7f 的位置处需要重定位，其中 0x16 和 0x4c 位置的重定位类型是 R_X86_64_32，使用 32 位绝对地址引用，其他位置处都为 R_X86_64_PC32，使用 32 位 PC 相对地址引用；重定位时，先跟据节偏移定位需要重定位的引用，再根据重定位类型按照如下的算法计算引用的运行时地址，并修改符号的引用使其指向正确的地址

```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset;
        if(r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset;
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend -
        refaddr);
        }
        if(r.type == R_X86_64_32) {
```

```
                *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
            }
        }
    }
```

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ readelf -r hello.o

重定位节 '.rela.text' at offset 0x320 contains 8 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000016  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000000001b  000a00000002 R_X86_64_PC32     0000000000000000 puts - 4
000000000025  000b00000002 R_X86_64_PC32     0000000000000000 exit - 4
00000000004c  00050000000a R_X86_64_32       0000000000000000 .rodata + 26
000000000056  000c00000002 R_X86_64_PC32     0000000000000000 printf - 4
000000000069  000d00000002 R_X86_64_PC32     0000000000000000 atoi - 4
000000000070  000e00000002 R_X86_64_PC32     0000000000000000 sleep - 4
00000000007f  000f00000002 R_X86_64_PC32     0000000000000000 getchar - 4

重定位节 '.rela.eh_frame' at offset 0x3e0 contains 1 entry:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 14 hello.o 的重定位信息

```
0000000000400657 <main>:
  400657:       55                      push   %rbp
  400658:       48 89 e5                mov    %rsp,%rbp
  40065b:       48 83 ec 20             sub    $0x20,%rsp
  40065f:       89 7d ec                mov    %edi,-0x14(%rbp)
  400662:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  400666:       83 7d ec 04             cmpl   $0x4,-0x14(%rbp)
  40066a:       74 14                   je     400680 <main+0x29>
  40066c:       bf 78 07 40 00          mov    $0x400778,%edi
  400671:       e8 9a fe ff ff          callq  400510 <puts@plt>
  400676:       bf 01 00 00 00          mov    $0x1,%edi
  40067b:       e8 d0 fe ff ff          callq  400550 <exit@plt>
  400680:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400687:       eb 46                   jmp    4006cf <main+0x78>
  400689:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  40068d:       48 83 c0 10             add    $0x10,%rax
  400691:       48 8b 10                mov    (%rax),%rdx
  400694:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  400698:       48 83 c0 08             add    $0x8,%rax
  40069c:       48 8b 00                mov    (%rax),%rax
  40069f:       48 89 c6                mov    %rax,%rsi
  4006a2:       bf 9e 07 40 00          mov    $0x40079e,%edi
```

图 15 hello 可执行目标文件 objdump 后的结果 hello2.s 部分代码

红线框标注部分为重定位后得到的

```
hello.o:     文件格式 elf64-x86-64


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 04             cmpl   $0x4,-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 46                   jmp    78 <main+0x78>
  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+0x26
```

图 16 hello.o 可重定位目标文件 objdump 后结果 hello1.s 的部分代码

5.6 hello 的执行流程

_dl_start ——> _dl_init ——> _start——> __libc_start_main ——> _cxa_atexit ——> _setjmp ——> main ——> printf ——> getchar ——> exit

5.7 Hello 的动态链接分析

.got 段在_dl_init 前后变化，红线框中为变化的部分

```
00000000:00600ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000000:00601000 20 0e 60 00 00 00 00 00 00 00 00 00 00 00 00 00  .`.............
00000000:00601010 00 00 00 00 00 00 00 00 16 05 40 00 00 00 00 00 ..........@.....
00000000:00601020 26 05 40 00 00 00 00 00 36 05 40 00 00 00 00 00 &.@.....6.@.....
00000000:00601030 46 05 40 00 00 00 00 00 56 05 40 00 00 00 00 00 F.@.....V.@.....
00000000:00601040 66 05 40 00 00 00 00 00 00 00 00 00 00 00 00 00 f.@.............
```

图 17 dl_init 前.got 段的内容

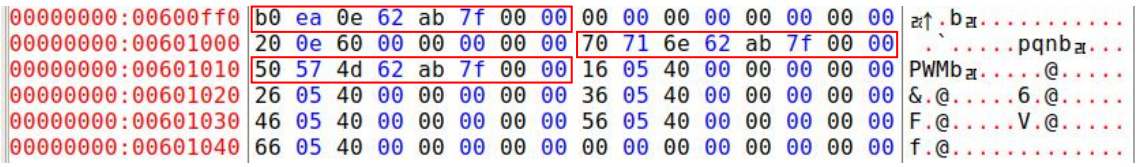


图 18 dl_init 后.got 段的内容

5.8 本章小结

链接是将各种代码和数据片段收集并合并成为一个单一文件的过程，是生成可执行目标文件的最后一步。本章介绍了链接的概念与作用、在 Ubuntu 下链接的命令、可执行目标文件 hello 的格式、hello 的虚拟地址空间链接的重定位过程分析、hello 的执行流程以及 hello 的动态链接分析。

（第 5 章 1 分）

第 6 章 hello 进程管理

6.1 进程的概念与作用

概念：进程是一个执行中的程序实例

作用：进程是对处理器、主存和 I/O 设备的抽象表示

6.2 简述壳 Shell-bash 的作用与处理流程

作用：shell 是交互级应用程序，代表用户运行其他程序

流程：

1．终端进程读取用户从键盘输入的命令

2．分析命令行字符串，获取命令行参数，并构造传递给 execve 的 argv 向量

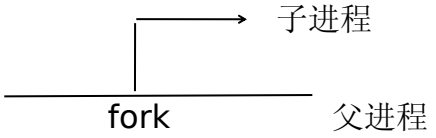
3．检查首个命令行参数是否为内置的 shell 命令，如果是则解释它；如果不是，shell 将它解释成执行某个可执行目标文件的命令并调用 fork 生成一个子进程，在这个子进程的上下文中加载并运行这个可执行目标文件

4．如果命令行参数的最后有'&'，说明用户要求后台运行，则 shell 返回；否则，shell 使用 waitpid 等待前台作业终止后返回

5．重复上述流程，直到用户发出终止信号

6.3 Hello 的 fork 进程创建过程

shell 为 hello 创建一个新的子进程，子进程得到与父进程用户级虚拟地址空间相同但独立的一个副本，并在新进程的上下文中运行



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，在 execve 加载了 hello 后，它调用加载器，加载器将可执行目

标文件中的代码和数据从磁盘复制到内存中，然后通过跳转到程序的入口点来运行该程序

6.5 Hello 的进程执行

上下文信息：内核重新启动一个被抢占的进程所需的状态，它由一些对象的值组成，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等

进程时间片：一个进程执行它的控制流的一部分的每一时间段

调度过程：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，这种决策就叫做调度，是由内核中称为调度器的代码处理的。在内核调度了一个新的进程运行后，它就抢断当前进程，并使用一种称为上下文切换的机制来将控制转移到新的进程，上下文切换的过程：

1．保存当前进程的上下文

2．恢复某个先前被抢断的进程被保存的上下文

3．将控制权传递给这个新恢复的进程

用户态与核心态转换：进程从用户态进入核心态的唯一方法是通过中断、故障或陷入系统调用这样的异常，当异常发生时，控制传递到异常处理程序，处理器将模式从用户模式变为内核模式，处理程序运行在内核模式中，当它返回到应用程序代码时，处理器把模式从内核模式改回用户模式

6.6 hello 的异常与信号处理

出现的异常：

1． 故障，会出现缺页，因为一开始物理页面中没有缓存 hello，系统调用异常处理程序，将虚拟页缓存到物理页面中，在重新执行引起故障的指令，此时由于物理页面中已经缓存了 hello 的各节信息，程序会正常运行

2． 陷阱，调用了系统调用 sleep

hello 自身产生的信号：

SIGCHLD，hello 终止时会向父进程发送 SIGCHLD 信号

1.按回车，不发送信号

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ./hello 1180800916 陶子康 1
Hello 1180800916 陶子康

Hello 1180800916 陶子康

Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 19 运行过程中按回车

2.Ctrl-C 向进程发送 SIGINT 信号，使进程终止

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ./hello 1180800916 陶子康 1
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
^C
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 20 运行过程中按 Ctrl-C

3.Ctrl-Z 向进程发送 SIGTSTP 信号，停止进程

(1)Ctrl-Z 后运行 ps 命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ./hello 1180800916 陶子康 1
Hello 1180800916 陶子康
Hello 1180800916 陶子康
^Z
[1]+  已停止               ./hello 1180800916 陶子康 1
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ps
  PID TTY          TIME CMD
14786 pts/0    00:00:00 bash
14796 pts/0    00:00:00 hello
14797 pts/0    00:00:00 ps
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 21 Ctrl-Z 后运行 ps 命令

(2)Ctrl-Z 后运行 jobs 命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ./hello 1180800916 陶子康 1
Hello 1180800916 陶子康
Hello 1180800916 陶子康
^Z
[1]+  已停止               ./hello 1180800916 陶子康 1
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ jobs
[1]+  已停止               ./hello 1180800916 陶子康 1
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 22 Ctrl-Z 后运行 jobs 命令

(3)Ctrl-Z 后运行 pstree 命令

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ./hello 1180800916 陶子康 1
Hello 1180800916 陶子康
Hello 1180800916 陶子康
^Z
[2]+  已停止               ./hello 1180800916 陶子康 1
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager─┬─dhclient
        │                └─2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─avahi-daemon───avahi-daemon
        ├─bluetoothd
        ├─boltd───2*[{boltd}]
        ├─canonical-livep───11*[{canonical-livep}]
        ├─colord───2*[{colord}]
        ├─cron
        ├─cups-browsed───2*[{cups-browsed}]
        ├─cupsd
        ├─2*[dbus-daemon]
        ├─deepin-terminal─┬─bash───sudo───netease-cloud-m─┬─netease-cloud-m───netease-cloud-m───14+
        │                 │                               └─48*[{netease-cloud-m}]
        │                 └─2*[{deepin-terminal}]
        ├─fcitx───2*[{fcitx}]
        ├─fcitx-dbus-watc
        ├─fwupd───4*[{fwupd}]
        ├─gdm3─┬─gdm-session-wor─┬─gdm-wayland-ses─┬─gnome-session-b─┬─gnome-shell─┬─Xwayland───9*+
        │      │                 │                 │                 │             └─18*[{gnome-sh+
```

图 23 Ctrl-Z 后运行 pstree 命令

(4)Ctrl-Z 后运行 fg 命令，发送 SIGCONT 信号，继续被停止的 hello 进程

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ./hello 1180800916 陶子康 1
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
^Z
[1]+  已停止               ./hello 1180800916 陶子康 1
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ fg
./hello 1180800916 陶子康 1
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
Hello 1180800916 陶子康
```

图 24 Ctrl-Z 后运行 fg 命令

(5)Ctrl-Z 后运行 kill 命令，发送 SIGKILL 信号给 hello 进程，使其终止

```
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ps
  PID TTY          TIME CMD
16936 pts/0    00:00:00 bash
17329 pts/0    00:00:00 hello
17331 pts/0    00:00:00 ps
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ kill -9 17329
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$ ps
  PID TTY          TIME CMD
16936 pts/0    00:00:00 bash
17333 pts/0    00:00:00 ps
[1]+  已杀死               ./hello 1180800916 陶子康 1
jr1preg@jr1preg:/media/jr1preg/hitics/homework/大作业$
```

图 25 Ctrl-Z 后运行 kill 命令

6.7 本章小结

异常是允许操作系统内核提供进程概念的基本构造块，进程是计算机科学中最深刻、最成功的概念之一。本章主要介绍了进程的概念与作用、壳 shell-bash 的作用与处理流程、hello 的 fork 进程创建过程、hello 的 execve 过程、hello 的进程执行、hello 的异常与信号处理。

（第 6 章 1 分）

第 7 章 hello 的存储管理

7.1 hello 的存储器地址空间

结合 hello 说明逻辑地址、线性地址、虚拟地址、物理地址的概念。

逻辑地址：由程序产生的和段相关的偏移地址部分[1]

线性地址：逻辑地址到物理地址变换的中间层[1]

虚拟地址：由 CPU 生成，等待被翻译的地址，如 hello 中的.text 段的地址 0x400570 就是虚拟地址

物理地址：CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址，如 hello 中的.text 段虚拟地址经过 MMU 翻译后得到的地址

7.2 Intel 逻辑地址到线性地址的变换-段式管理[1]

一个逻辑地址由两个部分组成：段标识符和段内偏移。段标识符是由一个 16 位长的字段组成，称为段选择符，其中前 13 位是索引号，后 3 位包含一些硬件信息。

这个 13 位索引号索引的信息存储在全局段描述符表(GDT)或局部段描述符表(LDT)中，这两张表的起始地址分别存储在 gdtr 和 ldtr 两个寄存器中，索引号中的 T1 字段决定选择 GDT 还是 LDT，T1==0 时用 GDT，T1==1 时 LDT。

执行从逻辑地址到线性地址的变换时，先根据 T1 获取 gdtr 或 ldtr 的起始地址，再根据 13 位索引号剩下部分索引到段描述符表中的一个条目，再利用这个条目中的内容和逻辑地址的段内偏移计算得到线性地址。如下图，是逻辑地址到线性地址变换的计算示意图

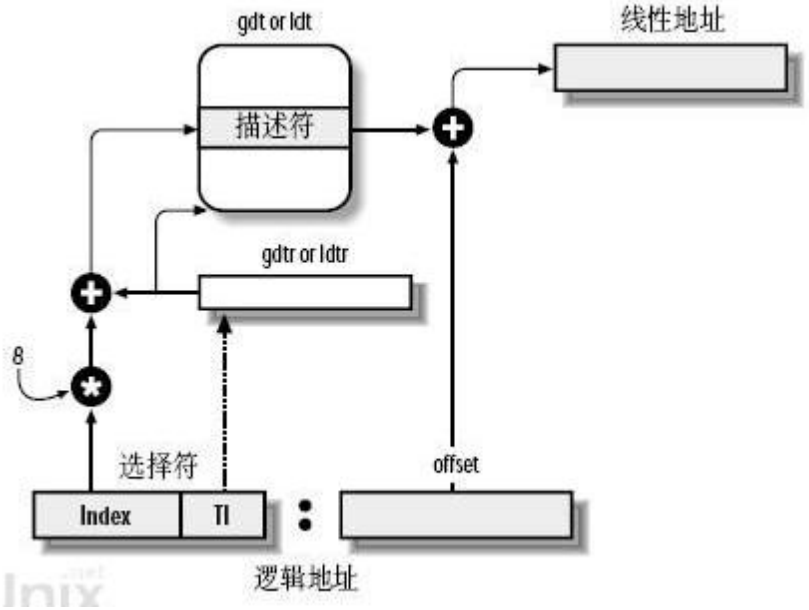


图 26 段式管理(图片来源网络[1])

7.3 Hello 的线性地址到物理地址的变换-页式管理

CPU 的页式内存管理单元，负责把一个线性地址，转换为物理地址。CR3 寄存器中存储页表的地址，通过线性地址的前若干位索引到页表中的内容，如果页面已缓存，则取出 PPN；否则，DRAM 缓存不命中，触发缺页故障，处理程序会选择一个牺牲页，如果它被修改过则将它写会，用新页覆盖并更新页表，此时，页面被缓存，MMU 能得到物理地址。虽然下图中是虚拟地址到物理地址的变换，但对线性地址同样适用，仅作为参考

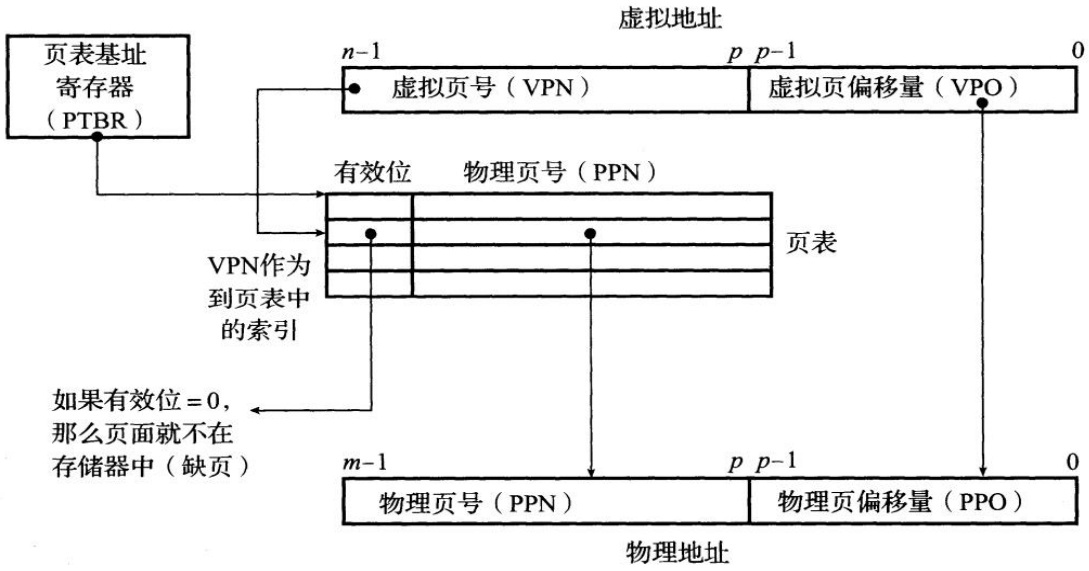


图 27 页式管理(图片来源于课本)

7.4 TLB 与四级页表支持下的 VA 到 PA 的变换

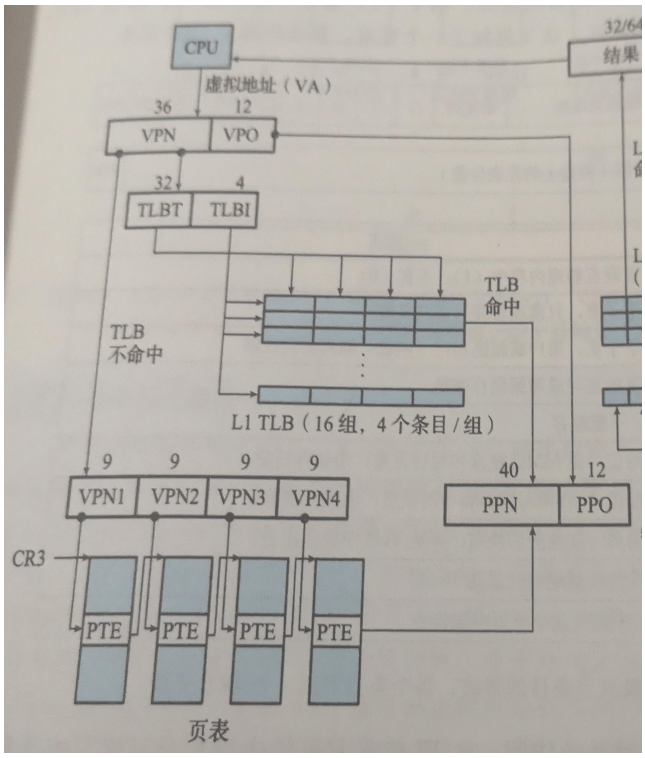


图 28 VA 到 PA 的变换(图片来源于课本)

如上图所示，虚拟地址高 36 位 VPN 部分的高 32 位作为 TLB 的标记，低 4 位作为 TLB 索引，如果 TLB 命中则直接从 TLB 中读出 PPN，否则 VPN 被分为 4 个部分，每个部分 9 位，分别指向四级页表中的内容,如果页表命中，从页表中读出 PPN，如果页表不命中，则会触发缺页异常，执行缺页异常处理程序，选择一个牺牲页，如果该页修改过，则将该页写回，然后用新页覆盖它，再次执行时，页表中缓存了该页，则读出 PPN，再将 VPO 直接传给 PPO，最终得到物理地址 PA

7.5 三级 Cache 支持下的物理内存访问

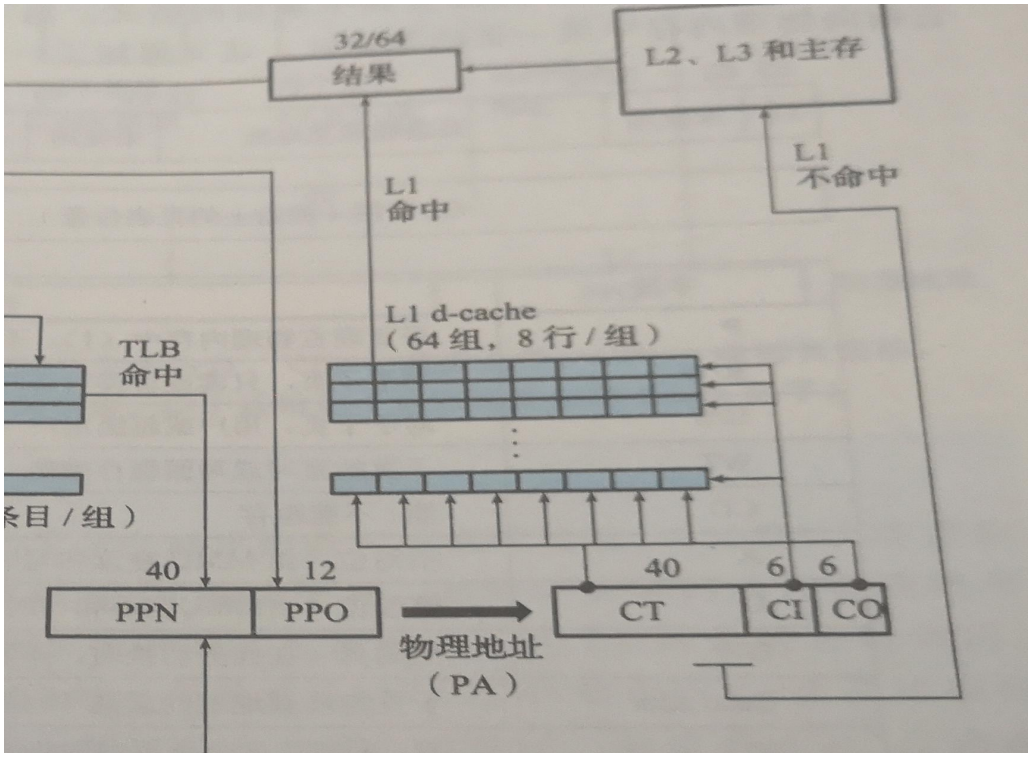


图 29 三级 Cache 支持下的物理内存访问(图片来源于课本)

52 位物理地址被分成 3 部分，分别是 CT 标记位，CI 索引位，CO 块偏移，先在 L1Cache 中找，如果 L1Cache 命中，则直接读取其中存储的内容，如果不命中，再到 L2 中找，如果 L2 还不命中，再到 L3 中找，如果 L3 也不命中，则直接从主存中读取该物理地址处的内容

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，并分配它一个唯一的 PID。为了给这个新进程创建虚拟内存，内核创建了当前进程的 mm_struct、区域结构和页表的原样副本。它将两个进程的所有页面都标记为只读，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。当这两个进程的任一个想要进行写操作时，写时复制机制就会创建新页面。下图，是写时复制的示意图

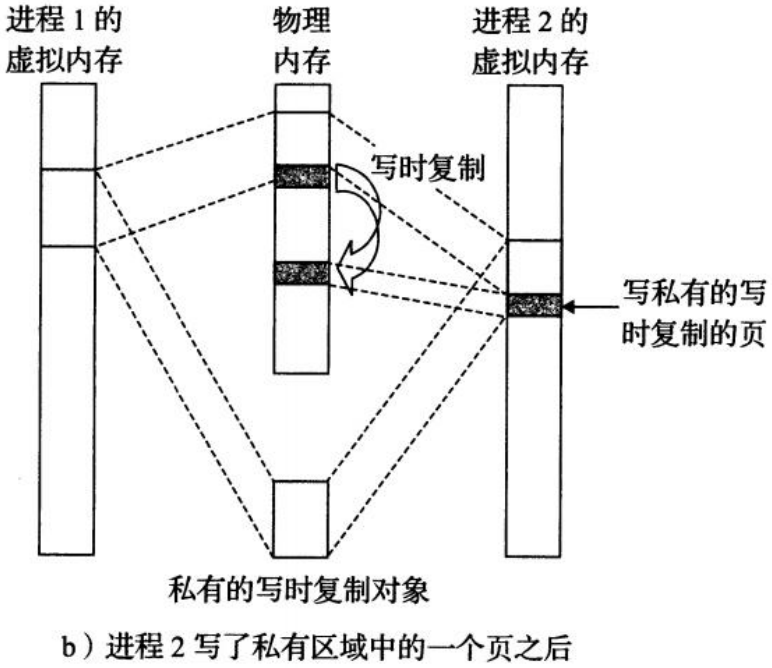


图 30 写时复制示意图(图片来源于课本)

7.7 hello 进程 execve 时的内存映射

execve 加载并运行可执行文件，它按照如下几个步骤执行：

1．删除已存在的用户区域

2．映射私有区域，为新程序的代码、数据、bss、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

3．映射共享区域

4．设置程序计数器

下图是 execve 时的内存映射示意图

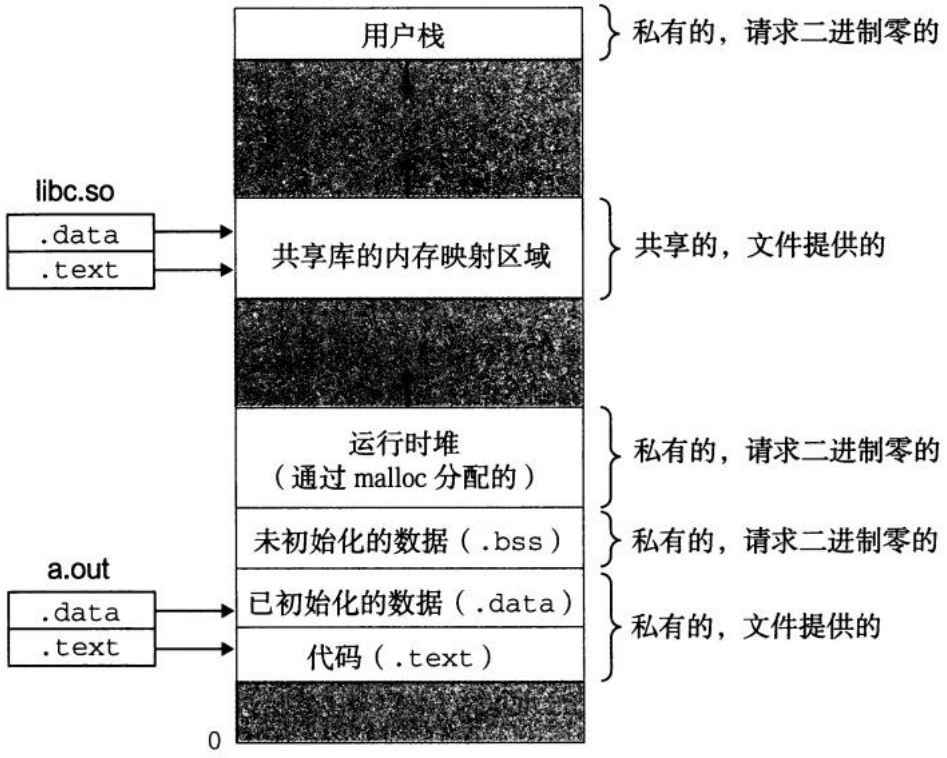


图 31 execve 时的内存映射示意图(图片来源于课本)

7.8 缺页故障与缺页中断处理

缺页故障发生在指令引用一个虚拟地址，而与该地址相对应的物理地址的物理页面不在内存中,必须从磁盘取出的情况。缺页故障会触发缺页处理程序

缺页中断处理：

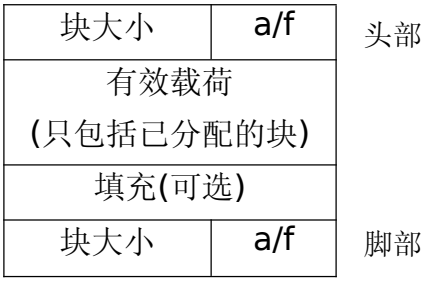
1. 处理程序首先判断触发缺页异常的虚拟地址是否合法，如果不合法，则触发段错误，终止进程
2. 如果虚拟地址合法，处理程序进一步判断内存的访问是否合法，即进程是否有读、写或执行这个区域内页面的权限。如果访问不合法，触发保护异常，终止进程
3. 到这一步，内核已经确定这个缺页是由于对合法的虚拟地址的合法操作造成的。内核会选择夜歌牺牲页面，如果它被修改过，则将他写回，并换入新的页面并更新页表，然后返回到触发缺页处理程序的那条指令并重新执行，此时 MMU 能够便正常翻译虚拟地址

7.9 动态存储分配管理

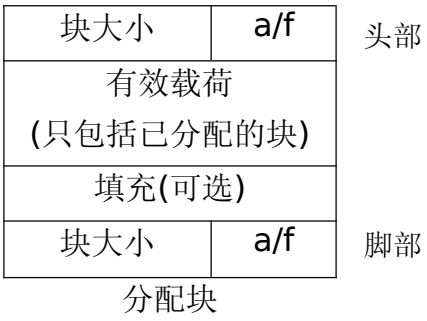
动态内存分配器维护着一个进程的虚拟内存区域，称为堆，它紧接在为初始化的数据区域后开始，并向高地址生长，当程序在运行时需要额外的虚拟内存时，动态内存分配器动态地在堆区域中分配块，即连续的虚拟内存片。每个块要么是空闲的，要么是已分配的，空闲块保持空闲直到它显示地被应用分配；已分配块保持已分配的状态直到它被释放，这种释放要么是应用程序显示地执行的，要么是内存分配器自身隐式执行的。

分配器分为两种基本风格：显式分配器——要求应用显式地释放已分配块，和隐式分配器——要求分配器检测一个已分配块何时不再被程序使用，就释放这个块。

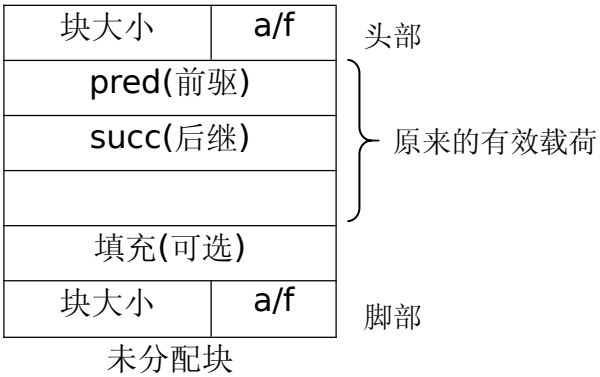
空闲链表常用的有带边界标签的隐式空闲链表分配器



和显示空闲链表



分配块



未分配块

策略有首次适配、最佳适配和下一次适配等等。

7.10 本章小结

本章主要介绍了 hello 的存储器地址空间、intel 逻辑地址到线性地址的变换——段式管理、hello 的线性地址到物理地址的变换——页式管理、TLB 与四级页表支持下的 VA 到 PA 的变换、利用三级 Cache 加速物理内存访问、hello 进程 fork 时的内存映射和 execve 时的内存映射、缺页故障与缺页中断处理以及动态内存分配管理

（第 7 章 2 分）

第 8 章 hello 的 IO 管理

8.1 Linux 的 IO 设备管理方法

设备的模型化：所有的 IO 设备都被模型化为文件，而所有的输入和输出都被当做对相应文件的读和写来执行，这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单低级的应用接口，称为 Unix I/O。

设备管理：unix I/O 接口

8.2 简述 Unix IO 接口及其函数

Unix I/O 接口：

一个 linux 文件就是一个 m 个字节的序列：B_0，B_1，……，B_{m-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，而所有的输入和输出都被当做对相应文件的读和写来执行。这种将设备优雅地映射为文件的方式，允许 Linux 内核引出一个简单、低级的应用接口，称为 Unix I/O，这使得所有的输入和输出都以一种统一且一致的方式来执行：

- 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 I/O 设备。内核返回一个小的非负整数，叫做描述符，它在后续对此文件的所有操作中标识这个文件，内核记录有关这个打开文件的所有信息。
- Linux shell 创建的每个进程都有三个打开的文件：标准输入，标准输出，标准错误。它们的描述符分别是 0、1、2
- 改变当前的文件位置。对于每个打开的文件，内核保持一个文件位置 k，初始为 0。这个文件位置表示的是从文件开头起始的字节偏移量。应用程序能够通过执行 seek 操作，显式地将改变当前文件位置 k。
- 读写文件。一个读操作及时从文件复制 n>0 个字节到内存，从当前文件位置 k 开始，然后将 k 增加到 k+n。给定一个大小为 m 字节的文件，当 k>=m 时执行读操作会触发一个称为 EOF 的条件，应用程序能检测到这个条件。在文件结尾处并没有明确的 EOF 符号，类似地，写操作就是从内存中复制 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=k+n。
- 关闭文件：内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中去

函数：

int open(char *filename, int flags, mode_t mode)：open 函数将 filename 转换为一个文件描述符，并且返回描述符数字。返回的描述符总是在进程中当前没有打开的最小描述符。flags 参数指明了进程打算如何访问这个文件，mode 参数指定了新文件的访问权限位。

int close(int fd)：关闭一个打开的文件

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

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

8.3 printf 的实现分析[2]

首先查看 printf 的实现

```
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;
}
```

va_list 实际是指向 char 类型的指针，arg 实际是指向第二个参数的指针，vasprintf 函数实际通过 fmt 格式和 arg 参数将输出以字符串形式存储到 buf 中，最后通过 write 系统调用将 buf 输出到屏幕。

从 vsprintf 生成显示信息，到 write 系统函数，到陷阱-系统调用 int 0x80 或 syscall。下面是 vsprintf 的部分代码，仅列出了 fmt 格式为”%x”的情况

```
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++) {
        if (*fmt != '%') {
            *p++ = *fmt;
```

```
            continue;
        }
        fmt++;
        switch (*fmt) {
        case 'x': //只处理%x 一种情况
            itoa(tmp, *((int*)p_next_arg));
            strcpy(p, tmp);
            p_next_arg += 4;
            p += strlen(tmp);
            break;
        case 's': break;
        default: break;
        }
    }
    return (p - buf); //返回最后生成的字符串的长度
}
```

字符显示驱动子程序：从 ASCII 到字模库到显示 vram（存储每一个点的 RGB 颜色信息），显示芯片按照刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB 分量）。

8.4 getchar 的实现分析[3]

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，中断抢占当前进程运行中断处理子程序。中断处理子程序接受按键扫描码转成 ascii 码，保存到系统的键盘缓冲区。

getchar 函数调用 read 系统函数，通过系统调用读取存储在键盘缓冲区的 ascii 码直到接受到回车键才返回整个字符串，getchar 的大体逻辑是读取字符串的第一个字符然后返回。

8.5 本章小结

本章主要介绍了 Linux 的 I/O 设备管理方法、Unix I/O 接口及其函数，分析了 printf 函数的实现以及 getchar 函数的实现

（第 8 章 1 分）

结论

hello 的精彩一生：

① 程序员编写正确的 hello.c 程序

② 预处理：解释以'#'开头的代码，生成 hello.i 文件，为下一步编译作充足的准备。

③ 编译：将预处理好的 hello.i 编译成汇编语言文件 hello.s

④ 汇编：将汇编语言文件 hello.s 变为可重定位目标文件 hello.o

⑤ 链接：将重定位目标文件 hello.o 和其他的重定位目标文件以及动态库进行链接，生成可执行目标文件 hello

⑥ 在 shell 中输入命令./hello 1180800916 陶子康

⑦ shell 调用 fork 为 hello 创建进程

⑧ 调用 execve，启动加载器加载 hello，创建内存映射，进入程序入口开始运行

⑨ 在 hello 的时间片中执行指令，可能触发缺页等异常，但都被完美解决

⑩ 向堆区域申请动态内存，动态地请求二进制零的页

⑪ 收到各种信号，被暂时停止甚至提前结束她的一生

⑫ 运行结束，shell 父进程将她回收，她存在的一切痕迹——内核为她创建的所有数据结构都被删除

在短暂到人类无法察觉的时间片段中，hello 已经完成了她的一生。虽然她没有绮丽复杂的数据结构、没有令人拍手称绝的算法妙想，但她同那些代码量和复杂程度远超她的程序一样，都完整地体验了二进制世界的精彩。屏幕上闪光的 Hello，似乎在诉说着她来过。

你的创新理念，如新的设计与实现方法。

（结论 0 分，缺失 -1 分，根据内容酌情加分）

附件

hello.i——预处理文件，预处理器(cpp)处理以’#’开头的命令，修改原始的 C 程序，得到.i 文件。它作为编译器的输入

hello.s——汇编文件，编译器(cc1)将.i 文件翻译成汇编语言，得到.s 文件。它包含汇编代码，作为汇编器的输入

hello1.s——汇编文件，是将 hello.o 通过 objdump 命令得到的

hello2.s——汇编文件，是将 hello.o 通过 objdump 命令得到的

hello.o——可重定位目标文件，汇编器(as)将.s 文件翻译为机器语言指令，得到.o 文件。它包含机器代码，作为链接器的输入

hello.out——可执行目标文件，链接器(ld)将各种.o 文件进行链接，得到可执行目标文件，它可以被加载到内存中，由系统执行

hello——可执行目标文件，由 gcc -m64 -no-pie -fno-PIC hello.c -o hello 命令直接生成

（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] https://blog.csdn.net/yuzaipiaofei/article/details/51219847

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

[3] 1170300821 罗瑞欣《程序人生-Hello’s P2P》

[4] 深入理解计算机系统第三版

（参考文献 0 分，缺失 -1 分）