|  |
| --- |
| 深圳大学 |
| GNU/Linux C程序观察  (V 0.9) |
| 罗秋明 |

目录

[第1章 实验环境构建 1](#_Toc2446674)

[1.1. 安装Linux 1](#_Toc2446675)

[1.1.1. 下载CentOS7 1](#_Toc2446676)

[1.1.2. CentOS7安装 2](#_Toc2446677)

[1.2. 虚拟机安装Linux 7](#_Toc2446678)

[1.2.1. VirtualBox安装 7](#_Toc2446679)

[1.2.2. 虚拟机配置 10](#_Toc2446680)

[1.2.3. 虚拟机安装Linux 16](#_Toc2446681)

[1.3. ssh远程终端访问 17](#_Toc2446682)

[1.3.1. Putty客户端 17](#_Toc2446683)

[1.3.2. 无密码登陆 19](#_Toc2446684)

[1.3.3. xming图形终端 21](#_Toc2446685)

[1.4. 初次接触Linux 23](#_Toc2446686)

[1.4.1. 简单操作 23](#_Toc2446687)

[1.4.2. 运行HelloWorld程序 27](#_Toc2446688)

[1.5. 小结 30](#_Toc2446689)

[第2章 程序编译与运行 31](#_Toc2446690)

[2.1. 编译的各阶段 32](#_Toc2446691)

[2.1.1. 源代码 32](#_Toc2446692)

[2.1.2. 预处理 33](#_Toc2446693)

[2.1.3. 编译 35](#_Toc2446694)

[2.1.4. 汇编 36](#_Toc2446695)

[2.1.5. 链接 38](#_Toc2446696)

[2.1.6. GCC编译驱动 39](#_Toc2446697)

[2.2. GCC基本用法 40](#_Toc2446698)

[2.2.1. C语言标准 41](#_Toc2446699)

[2.2.2. 库的使用 42](#_Toc2446700)

[2.2.3. 搜索路径 46](#_Toc2446701)

[2.2.4. 编译警告 50](#_Toc2446702)

[2.3. GDB调试 53](#_Toc2446703)

[2.3.1. 代码准备 53](#_Toc2446704)

[2.3.2. 运行代码 55](#_Toc2446705)

[2.3.3. 查看变量和内存 58](#_Toc2446706)

[2.3.4. 图形前端TUI 62](#_Toc2446707)

[2.4. 小结 62](#_Toc2446708)

[练习 62](#_Toc2446709)

[第3章 数据、运算与控制 64](#_Toc2446710)

[3.1. X86-64 ISA 64](#_Toc2446711)

[3.1.1. 寄存器 64](#_Toc2446712)

[3.1.2. 内存空间与IO空间 68](#_Toc2446713)

[3.2. 数据 69](#_Toc2446714)

[3.2.1. 数据大小、字节序 69](#_Toc2446715)

[3.2.2. 数组、结构体和联合体 71](#_Toc2446716)

[3.2.3. 数据布局 73](#_Toc2446717)

[3.3. 运算 76](#_Toc2446718)

[3.3.1. 数据传送 76](#_Toc2446719)

[3.3.2. 算术/逻辑运算 77](#_Toc2446720)

[3.3.3. 加载有效地址 78](#_Toc2446721)

[3.4. 控制 79](#_Toc2446722)

[3.4.1. 条件跳转 79](#_Toc2446723)

[3.4.2. C语言控制语句 81](#_Toc2446724)

[3.4.3. 函数调用 91](#_Toc2446725)

[3.5. 小结 113](#_Toc2446726)

[练习 113](#_Toc2446727)

[第4章 链接与可执行文件 114](#_Toc2446728)

[4.1. 生成可执行文件 114](#_Toc2446729)

[4.1.1. 样例代码 115](#_Toc2446730)

[4.1.2. 进程影像 116](#_Toc2446731)

[4.1.3. ELF文件与装入 119](#_Toc2446732)

[4.2. 可重定位目标文件 129](#_Toc2446733)

[4.2.1. 目标文件的节（section） 130](#_Toc2446734)

[4.2.2. 符号及重定位 133](#_Toc2446735)

[4.2.3. 符号表 141](#_Toc2446736)

[4.3. 静态链接 144](#_Toc2446737)

[4.3.1. 布局 145](#_Toc2446738)

[4.3.2. 符号解析 148](#_Toc2446739)

[4.3.3. 静态重定位 149](#_Toc2446740)

[4.4. 动态链接 154](#_Toc2446741)

[4.4.1. 样例代码 155](#_Toc2446742)

[4.4.2. 动态链接库 157](#_Toc2446743)

[4.4.3. 动态链接步骤 166](#_Toc2446744)

[4.5. 小结 180](#_Toc2446745)

[练习 181](#_Toc2446746)

[第5章 链接脚本与makefile 182](#_Toc2446747)

[5.1. 二进制工具和链接脚本 182](#_Toc2446748)

[5.1.1. binutils 182](#_Toc2446749)

[5.1.2. 链接器脚本 184](#_Toc2446750)

[5.2. makefile 192](#_Toc2446751)

[5.2.1. makefile基本格式 192](#_Toc2446752)

[5.2.2. makefile规则 194](#_Toc2446753)

[5.2.3. makefile变量 197](#_Toc2446754)

[5.2.4. 文件指示 200](#_Toc2446755)

[5.2.5. 函数 201](#_Toc2446756)

[5.2.6. make 203](#_Toc2446757)

[5.3. 小结 203](#_Toc2446758)

[练习 204](#_Toc2446759)

[第6章 程序运行 205](#_Toc2446760)

[6.1. 装入与运行 205](#_Toc2446761)

[6.1.1. ELF装载器 205](#_Toc2446762)

[6.1.2. 内核代码 206](#_Toc2446763)

[6.1.3. 进程与线程 206](#_Toc2446764)

[6.1.4. 工作环境 207](#_Toc2446765)

[6.2. 基本行为观察 208](#_Toc2446766)

[6.2.1. ptrace 209](#_Toc2446767)

[6.2.2. strace 213](#_Toc2446768)

[6.2.3. gdb断点原理 216](#_Toc2446769)

[6.2.4. ltrace 217](#_Toc2446770)

[6.3. 异常行为 219](#_Toc2446771)

[6.3.1. 非法操作 219](#_Toc2446772)

[6.3.2. 响应信号 222](#_Toc2446773)

[6.3.3. core文件 227](#_Toc2446774)

[6.4. 小结 234](#_Toc2446775)

[练习 234](#_Toc2446776)

[第7章 性能剖析 235](#_Toc2446777)

[7.1. 打桩方法 235](#_Toc2446778)

[7.1.1. 源代码预处理时 236](#_Toc2446779)

[7.1.2. 静态链接时 237](#_Toc2446780)

[7.1.3. 运行加载时 238](#_Toc2446781)

[7.2. gprof 240](#_Toc2446782)

[7.2.1. 工作原理 240](#_Toc2446783)

[7.2.2. gprof示例 240](#_Toc2446784)

[7.2.3. 性能数据解读 245](#_Toc2446785)

[7.2.4. 图形化显示（gprof2dot.py+graphviz） 249](#_Toc2446786)

[7.3. gcov 250](#_Toc2446787)

[7.3.1. 基于函数分析的缺点 250](#_Toc2446788)

[7.3.2. gcov逐行分析 251](#_Toc2446789)

[7.4. 其他分析工具 253](#_Toc2446790)

[7.4.1. Valgrind 253](#_Toc2446791)

[7.4.2. perf 262](#_Toc2446792)

[7.5. 小结 266](#_Toc2446793)

[练习 267](#_Toc2446794)

[第8章 综合实例: HDFS中实现zlib库的旁路 268](#_Toc2446795)

[8.1. 项目需求 268](#_Toc2446796)

[8.2. 系统分析 268](#_Toc2446797)

[8.2.1. 整体方案 269](#_Toc2446798)

[8.2.2. Haddop的gzip JNI 269](#_Toc2446799)

[8.2.3. zlib分析 271](#_Toc2446800)

[8.2.4. 测定z\_stream成员大小 273](#_Toc2446801)

[8.3. 编码实现 275](#_Toc2446802)

[8.3.1. zlib日志 275](#_Toc2446803)

[8.3.2. 使用libcprss.so库 281](#_Toc2446804)

[8.4. 功能验证 282](#_Toc2446805)

[8.4.1. 准备输入文件 282](#_Toc2446806)

[8.4.2. zlib原生库的压缩 283](#_Toc2446807)

[8.4.3. libcprss.so库的压缩 283](#_Toc2446808)

[8.5. 小结 284](#_Toc2446809)

[第9章 附录 285](#_Toc2446810)

# 实验环境构建

由于我们在Linux环境上观察C程序的行为细节，因此需要先获得一个可访问的Linux系统。读者可以自己安装配置一个Linux系统（真实机器或虚拟机都可以），也可以让Linux系统管理员给你开设一个账号。如果读者已经使用过或安装配置过Linux系统，可以直接跳过本章内容，从第二章开始阅读。

* 1. 安装Linux
     1. 下载CentOS7
     2. CentOS7安装
  2. 虚拟机安装Linux
     1. VirtualBox安装
     2. 虚拟机配置
     3. 虚拟机安装Linux
  3. ssh远程终端访问
     1. Putty客户端
     2. 无密码登陆
     3. xming图形终端
  4. 初次接触Linux
     1. 简单操作
     2. 运行HelloWorld程序

**代码 1‑1 HelloWorld.c**

1. #include <stdio.h>
2. int main()
3. {
4. printf(“HelloWorld!\n”);
5. return 0;
6. }

**代码 1‑2 HelloWorld-getchar.c**

1. #include <stdio.h>
2. int main()
3. {
4. printf(“HelloWorld!\n”);
5. getchar();
6. return 0;
7. }
   1. 小结

# 程序编译与运行

* 1. 编译的各阶段
     1. 源代码

代码 2‑1 Hello\_World0.c

1. #include <stdio.h>
2. int f\_sum(int a, int b);
3. int main()
4. {
5. int var\_sum=0;
6. printf("HelloWorld!\n");
7. var\_sum=f\_sum(2,3);
8. return var\_sum;
9. }

代码 2‑2 f\_sum.c

1. int f\_sum(int a, int b)
2. {
3. return a+b;
4. }
   * 1. 预处理

代码 2‑3 macro.c

1. #include <stdio.h>
2. #define NUM 5
3. int main (void)
4. {
5. printf("Value of NUM is %d\n", NUM);
6. return 0;
7. }

代码 2‑4 ifdef-else.c

1. #include <stdio.h>
2. int main (void)
3. {
4. #ifdef TEST
5. printf ("Test mode\n");
6. #endif
7. printf ("Running...\n");
8. return 0;
9. }
   * 1. 编译
     2. 汇编
     3. 链接
     4. GCC编译驱动
   1. GCC基本用法
      1. C语言标准
      2. 库的使用

代码 2‑5 main-lib.c

1. #include <stdio.h>
2. #include "vector.h"
3. int x[2] = {1, 2};
4. int y[2] = {3, 4};
5. int z[2];
6. int main()
7. {
8. addvec(x, y, z, 2);
9. printf("z = [%d %d]\n", z[0], z[1]);
10. return 0;
11. }

代码 2‑6 addvec.c

1. void addvec(int \*x, int \*y,int \*z, int n)
2. {
3. int i;
4. for (i = 0; i < n; i++)
5. z[i] = x[i] + y[i];
6. }

代码 2‑7 multvec.c

1. void multvec(int \*x, int \*y,int \*z, int n)
2. {
3. int i;
4. for (i = 0; i < n; i++)
5. z[i] = x[i] \* y[i];
6. }

代码 2‑8 vector.h

1. void addvec(int \*x, int \*y,int \*z, int n);
2. void multvec(int \*x, int \*y,int \*z, int n);
   * 1. 搜索路径
     2. 编译警告

代码 2‑9 GCC-Wall.c

1. #include <stdio.h>
2. int main (void)
3. {
4. printf ("Two plus two is %f\n", 4);
5. return 0;
6. }

代码 2‑10 嵌套的注释

1. /\* commented out
2. double x = 1.23 ; /\* x-position \*/
3. \*/

代码 2‑11 由编译制导符完成注释功能

1. /\* commented out \*/
2. #if 0
3. double x = 1.23 ; /\* x-position \*/
4. #endif

代码 2‑12 无符号数是否<0的w.c

1. int
2. foo (unsigned int x)
3. {
4. if (x < 0)
5. return 0; /\* cannot occur \*/
6. else
7. return 1;
8. }

代码 2‑13 var-shadown.c不同作用域中声明同名变量

1. double
2. test (double x)
3. {
4. double y = 1.0;
5. {
6. double y;
7. y = 2\*x;
8. }
9. return y;
10. }

代码 2‑14 cast-qual.c

1. void
2. f (const char \* str)
3. {
4. char \* s = (char \*)str;
5. s[0] = ’\0’;
6. }
   1. GDB调试
      1. 代码准备

代码 2‑15 demo-gdb.c

1. #include <stdio.h>
2. int func(int n)
3. {
4. int sum=0,i;
5. for(i=0;i<n;i++)
6. {
7. sum+=i;
8. }
9. return sum;
10. }
11. int main()
12. {
13. int i;
14. long result = 0;
15. for (i=1;i<=100;i++)
16. {
17. result += i;
18. }
19. printf("result[1-100]= %d \n",result);
20. printf("result[1-200]= %d \n",func(200) );
21. return 0;
22. }
    * 1. 运行代码
      2. 查看变量和内存
      3. 图形前端TUI
    1. 小结

练习

代码 2‑16 sum.c（gdb练习）

1. #include <stdio.h>
2. #include <string.h>
3. int sum(int a[],unsigned len)
4. {
5. int i,sum=0;
6. for(i=0; i<=len -1;i++)
7. sum+=a[i];
8. return sum;
9. }
10. int main()
11. {
12. int a[]={};
13. int len;
15. len=sizeof(a)/sizeof(a[0]);
16. printf("len= %d ; sum= %d\n",len,sum(a,len));
17. return 0;
18. }

# 数据、运算与控制

* 1. X86-64 ISA
     1. 寄存器
     2. 内存空间与IO空间
  2. 数据
     1. 数据大小、字节序

代码 3‑1 data\_size.c

1. char var\_char1=0x11;
2. int var\_int1=0x12345678;
3. short var\_short1=0x2323;
4. long long var\_64int1=0xF1AAAAAAAAAAAAF2;
5. long long main()
6. {
7. long long var\_64int2;
8. var\_64int2=var\_64int1+var\_short1+var\_int1+var\_char1;
9. return var\_64int2;
10. }
    * 1. 数组、结构体和联合体

代码 3‑2 array-struct-union.c

1. #include <stdio.h>
2. int a[8]={0,1,2,3,4,5,6,7};
3. struct mystr {
4. char c1;
5. double d1;
6. short s1;
7. int i1;
8. }mystr1;
9. union myuni{
10. char c1;
11. double d1;
12. short s1;
13. int i1;
14. }myuni1;
15. int main()
16. {
17. printf("Array/struct/union demo1\n");
18. return 0;
19. }
    * 1. 数据布局

代码 3‑3 data-place.c

1. #include <stdlib.h>
2. int a;
3. int b=100;
4. int func(int c, int d);
5. int main()
6. {
7. int \*buf;
8. a=func(b,5);
9. buf=(int \*)malloc(1024);
10. return a;
11. }

代码 3‑4 data-place-func.c

1. int c;
2. int d=200;
3. int func(int x,int y)
4. {
5. int k;
6. c=x+d;
7. k=c\*y;
8. return k;
9. }
   1. 运算
      1. 数据传送

代码 3‑5 swap.c

1. void swap (long \*xp, long \*yp)
2. {
3. long t0 = \*xp;
4. long t1 = \*yp;
5. \*xp = t1;
6. \*yp = t0;
7. }

代码 3‑6 swap.s

1. .file "swap.c"
2. .text
3. .globl swap
4. .type swap, @function
5. swap:
6. .LFB0:
7. .cfi\_startproc
8. movq (%rdi), %rax
9. movq (%rsi), %rdx
10. movq %rdx, (%rdi)
11. movq %rax, (%rsi)
12. ret
13. .cfi\_endproc
14. .LFE0:
15. .size swap, .-swap
16. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
17. .section .note.GNU-stack,"",@progbits
    * 1. 算术/逻辑运算

代码 3‑7 arith.c

1. long arith(long x,long y,long z)
2. {
3. long t1=x+y;
4. long t2=z\*48;
5. long t3=t1&0xFFFF;
6. long t4=t2\*t3;
7. return t4;
8. }

代码 3‑8 arith.s

1. .file "arithm.c"
2. .text
3. .globl arith
4. .type arith, @function
5. arith:
6. .LFB0:
7. .cfi\_startproc
8. addq %rdi, %rsi
9. leaq (%rdx,%rdx,2), %rax
10. salq $4, %rax
11. movzwl %si, %esi
12. imulq %rsi, %rax
13. ret
14. .cfi\_endproc
15. .LFE0:
16. .size arith, .-arith
17. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
18. .section .note.GNU-stack,"",@progbits
    * 1. 加载有效地址

代码 3‑9 leaq.c

1. long scale(long x,long y, long z)
2. {
3. long t=x+4\*y+12\*z;
4. return t;
5. }

代码 3‑10 leaq.s

1. .file "leaq.c"
2. .text
3. .globl scale
4. .type scale, @function
5. scale:
6. .LFB0:
7. .cfi\_startproc
8. leaq (%rdi,%rsi,4), %rcx
9. leaq (%rdx,%rdx,2), %rax
10. salq $2, %rax
11. addq %rcx, %rax
12. ret
13. .cfi\_endproc
14. .LFE0:
15. .size scale, .-scale
16. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
17. .section .note.GNU-stack,"",@progbits
    1. 控制
       1. 条件跳转

代码 3‑11 testq.s

1. .file "testq.s"
2. .text
3. .globl testq
4. .type testq, @function
5. testq:
6. .LFB0:
7. .cfi\_startproc
8. testq %rdi, %rsi
9. movq %rsi,%rax
10. ret
11. .cfi\_endproc
12. .LFE0:
13. .size testq, .-testq
14. .globl main
15. .type main, @function
16. main:
17. .LFB1:
18. .cfi\_startproc
19. movabsq $281474976710503, %rsi
20. movl $174, %edi
21. call testq
22. rep ret
23. .cfi\_endproc
24. .LFE1:
25. .size main, .-main
26. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
27. .section .note.GNU-stack,"",@progbits
    * 1. C语言控制语句

代码 3‑12 if-else的语法结构

1. if (test-expr)
2. then-statement
3. else
4. else-statement

代码 3‑13 if-else编译时采用的模板

1. t = test-expr;
2. if (!t)
3. goto false;
4. then-statement
5. goto done;
6. false:
7. else-statement
8. done:

代码 3‑14 if-else.c

1. long ifelse(long x, long y)
2. {
3. if (x<y)
4. x=y+3;
5. else
6. y=10\*x;
7. return x+y;
8. }

代码 3‑15 if-else.s

1. .file "if-else.c"
2. .text
3. .globl ifelse
4. .type ifelse, @function
5. ifelse:
6. .LFB0:
7. .cfi\_startproc
8. cmpq %rsi, %rdi
9. jge .L2
10. leaq 3(%rsi), %rdi
11. jmp .L3
12. .L2:
13. leaq (%rdi,%rdi,4), %rsi
14. addq %rsi, %rsi
15. .L3:
16. leaq (%rdi,%rsi), %rax
17. ret
18. .cfi\_endproc
19. .LFE0:
20. .size ifelse, .-ifelse
21. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
22. .section .note.GNU-stack,"",@progbits

代码 3‑16 if-else-main.c

1. long ifelse(long x, long y)
2. {
3. if (x<y)
4. x=y+3;
5. else
6. y=10\*x;
7. return x+y;
8. }
9. long main()
10. {
11. long k;
12. k=ifelse(3,5);
13. return k;
14. }

代码 3‑17 if-else-main.s

1. .file "if-else.c"
2. .text
3. .globl ifelse
4. .type ifelse, @function
5. ifelse:
6. .LFB0:
7. .cfi\_startproc
8. cmpq %rsi, %rdi
9. jge .L2
10. leaq 3(%rsi), %rdi
11. jmp .L3
12. .L2:
13. leaq (%rdi,%rdi,4), %rsi
14. addq %rsi, %rsi
15. .L3:
16. leaq (%rdi,%rsi), %rax
17. ret
18. .cfi\_endproc
19. .LFE0:
20. .size ifelse, .-ifelse
21. .globl main
22. .type main, @function
23. main:
24. .LFB1:
25. .cfi\_startproc
26. movl $5, %esi
27. movl $3, %edi
28. call ifelse
29. rep ret
30. .cfi\_endproc
31. .LFE1:
32. .size main, .-main
33. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
34. .section .note.GNU-stack,"",@progbits

代码 3‑18 do-while的语法结构

1. do
2. body-statement
3. while (test-expr);

代码 3‑19 do-while编译时采用的模板

1. loop:
2. body-statement
3. t = test-expr;
4. if (t)
5. goto loop;

代码 3‑20 do-while.c

1. long do\_while(long n,long k)
2. {
3. do{
4. k=3\*k;
5. n--;
6. }while (n>1);
7. return n+k;
8. }

代码 3‑21 do-while.s

1. .file "do-while.c"
2. .text
3. .globl do\_while
4. .type do\_while, @function
5. do\_while:
6. .LFB0:
7. .cfi\_startproc
8. .L2:
9. leaq (%rsi,%rsi,2), %rsi
10. subq $1, %rdi
11. cmpq $1, %rdi
12. jg .L2
13. leaq (%rdi,%rsi), %rax
14. ret
15. .cfi\_endproc
16. .LFE0:
17. .size do\_while, .-do\_while
18. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
19. .section .note.GNU-stack,"",@progbits

代码 3‑22 while语法结构

1. while (test-expr)
2. body-statement

代码 3‑23 while等效的语法结构

1. if (!test-expr)
2. goto done;
3. do
4. body-statement
5. while (test-expr);
6. done:

代码 3‑24 while编译时可使用的模板

1. t = test-expr;
2. if (!t)
3. goto done;
4. loop:
5. body-statement
6. t = test-expr;
7. if (t)
8. goto loop;
9. done:

代码 3‑25 while编译时可使用的模板（跳转到中间）

1. goto test;
2. loop:
3. body-statement
4. test:
5. t=test-expr
6. if(t)
7. goto loop;

代码 3‑26 while.c

1. long my\_while(long n)
2. {
3. long a=1;
4. while (n>1){
5. a+=n;
6. n=n-1;
7. }
8. return a;
9. }

代码 3‑27 while.s

1. .file "while.c"
2. .text
3. .globl my\_while
4. .type my\_while, @function
5. my\_while:
6. .LFB0:
7. .cfi\_startproc
8. movl $1, %eax
9. jmp .L2
10. .L3:
11. addq %rdi, %rax
12. subq $1, %rdi
13. .L2:
14. cmpq $1, %rdi
15. jg .L3
16. rep ret
17. .cfi\_endproc
18. .LFE0:
19. .size my\_while, .-my\_while
20. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
21. .section .note.GNU-stack,"",@progbits

代码 3‑28 for语法结构

1. for (init-expr; test-expr; update-expr)
2. body-statement

代码 3‑29 for语法结构用while表述

1. init-expr;
2. while (test-expr) {
3. body-statement
4. update-expr;
5. }

代码 3‑30 for编译所用的模板

1. init-expr;
2. goto test;
3. loop:
4. body-statement
5. update-expr;
6. test:
7. t=test-expr
8. if(t)
9. goto loop;

代码 3‑31 for.c

1. long my\_for(long n)
2. {
3. long i=10;
4. long result=1;
5. for(i=2;i<=n;i++)
6. result\*=i;
7. return result;
8. }

代码 3‑32 for.s

1. .file "for.c"
2. .text
3. .globl my\_for
4. .type my\_for, @function
5. my\_for:
6. .LFB0:
7. .cfi\_startproc
8. movl $1, %eax
9. movl $2, %edx
10. jmp .L2
11. .L3:
12. imulq %rdx, %rax
13. addq $1, %rdx
14. .L2:
15. cmpq %rdi, %rdx
16. jle .L3
17. rep ret
18. .cfi\_endproc
19. .LFE0:
20. .size my\_for, .-my\_for
21. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
22. .section .note.GNU-stack,"",@progbits

代码 3‑33 switch.c

1. long my\_switch(long x,long n)
2. {
3. long val=x;
4. switch(n){
5. case 10:
6. val+=12;
7. break;
8. case 12:
9. val\*=3;
10. case 13:
11. val+=1;
12. break;
13. case 14:
14. case 15:
15. val=8;
16. break;
17. default:
18. val=100;
19. }
20. val=val+x\*20;
21. return val;
22. }

代码 3‑34 switch.s

1. .file "switch.c"
2. .text
3. .globl my\_switch
4. .type my\_switch, @function
5. my\_switch:
6. .LFB0:
7. .cfi\_startproc
8. subq $10, %rsi
9. cmpq $5, %rsi
10. ja .L8
11. jmp \*.L4(,%rsi,8)
12. .section .rodata
13. .align 8
14. .align 4
15. .L4:
16. .quad .L3
17. .quad .L8
18. .quad .L5
19. .quad .L9
20. .quad .L7
21. .quad .L7
22. .text
23. .L7:
24. movl $8, %edx
25. jmp .L2
26. .L3:
27. leaq 12(%rdi), %rdx
28. .p2align 4,,2
29. jmp .L2
30. .L5:
31. leaq (%rdi,%rdi,2), %rdx
32. jmp .L6
33. .L9:
34. movq %rdi, %rdx
35. .L6:
36. addq $1, %rdx
37. jmp .L2
38. .L8:
39. movl $100, %edx
40. .L2:
41. leaq (%rdi,%rdi,4), %rax
42. salq $2, %rax
43. addq %rdx, %rax
44. ret
45. .cfi\_endproc
46. .LFE0:
47. .size my\_switch, .-my\_switch
48. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
49. .section .note.GNU-stack,"",@progbits

代码 3‑35 switch-simp.c

1. long my\_switch(long x,long n)
2. {
3. long val=x;
4. switch(n){
5. case 10:
6. val+=12;
7. break;
8. case 19:
9. val\*=3;
10. break;
11. default:
12. val=100;
13. }
14. val=val+x\*20;
15. return val;
16. }

代码 3‑36 switch-simp.s

1. .file "switch-simp.c"
2. .text
3. .globl my\_switch
4. .type my\_switch, @function
5. my\_switch:
6. .LFB0:
7. .cfi\_startproc
8. cmpq $10, %rsi
9. je .L3
10. cmpq $19, %rsi
11. je .L4
12. jmp .L6
13. .L3:
14. leaq 12(%rdi), %rdx
15. .p2align 4,,3
16. jmp .L2
17. .L4:
18. leaq (%rdi,%rdi,2), %rdx
19. .p2align 4,,3
20. jmp .L2
21. .L6:
22. movl $100, %edx
23. .L2:
24. leaq (%rdi,%rdi,4), %rax
25. salq $2, %rax
26. addq %rdx, %rax
27. ret
28. .cfi\_endproc
29. .LFE0:
30. .size my\_switch, .-my\_switch
31. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
32. .section .note.GNU-stack,"",@progbits
    * 1. 函数调用

代码 3‑37 call-params.c

1. int func(int a,int b, char c, long d,long k,char \* j,long p\_s\_1,short p\_s\_2)
2. {
3. int local\_A;
4. long local\_B;
6. local\_A=a+b;
7. if(k<0)
8. local\_A=local\_A+c+d+\*j;
9. local\_B=p\_s\_1-p\_s\_2;
10. return local\_A+local\_B;
11. }
12. long main()
13. {
14. long mainA;
15. char mainB;
16. mainA=func(1,2,3,4,5,&mainB,0xaa,0xcc);
17. return mainA;
18. }

代码 3‑38 call-params.s

1. .file "call-params.c"
2. .text
3. .globl func
4. .type func, @function
5. func:
6. .LFB0:
7. .cfi\_startproc
8. movzwl 16(%rsp), %eax //short p\_s\_2=204，参见图 3‑4的中部
9. addl %edi, %esi //a+b, local\_A映射到esi
10. testq %r8, %r8 // 判定k<0
11. jns .L2
12. movsbl %dl, %edx //c->edx
13. addl %edx, %esi //local\_A+=c
14. addl %ecx, %esi //local\_A+=d
15. movsbl (%r9), %edi //取\*j
16. addl %edi, %esi //local\_A+=\*j
17. .L2:
18. movswq %ax, %rax //p\_s\_2，前面已经转入eax
19. movq 8(%rsp), %rcx //p\_s\_1，参见图 3‑4的中部
20. subq %rax, %rcx //p\_s\_1-p\_s\_2，local\_B映射到%rcx
21. movq %rcx, %rax // local\_B保存到%rax
22. addl %esi, %eax //local\_A+local\_B保存到%eax
23. ret
24. .cfi\_endproc
25. .LFE0:
26. .size func, .-func
27. .globl main
28. .type main, @function
29. main:
30. .LFB1:
31. .cfi\_startproc
32. subq $32, %rsp
33. .cfi\_def\_cfa\_offset 40
34. movl $204, 8(%rsp)
35. movq $170, (%rsp)
36. leaq 31(%rsp), %r9
37. movl $5, %r8d
38. movl $4, %ecx
39. movl $3, %edx
40. movl $2, %esi
41. movl $1, %edi
42. call func
43. cltq
44. addq $32, %rsp
45. .cfi\_def\_cfa\_offset 8
46. ret
47. .cfi\_endproc
48. .LFE1:
49. .size main, .-main
50. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
51. .section .note.GNU-stack,"",@progbits

代码 3‑39 call-localvar1.c

1. int leaf\_fun(long lf1,long lf2)
2. { int k;
3. k=lf1+lf2;
4. return k;
5. }
6. int func(long f1,long f2)
7. {
8. int i;
9. int a1[20];
10. for(i=0;i<19;i++)
11. a1[i]=leaf\_fun(i,i);
12. return a1[2];
13. }
14. long main()
15. {
16. int mf1=20;
17. int mf2=11;
18. mf2=func(mf1,mf2);
19. return mf2;
20. }

代码 3‑40 call-localvar1.s

1. .file "call-localvar1.c"
2. .text
3. .globl leaf\_fun
4. .type leaf\_fun, @function
5. leaf\_fun:
6. .LFB0:
7. .cfi\_startproc
8. leal (%rdi,%rsi), %eax // lf1+lf2
9. ret
10. .cfi\_endproc
11. .LFE0:
12. .size leaf\_fun, .-leaf\_fun
13. .globl func
14. .type func, @function
15. func:
16. .LFB1:
17. .cfi\_startproc
18. pushq %rbp //保存rbp
19. .cfi\_def\_cfa\_offset 16
20. .cfi\_offset 6, -16
21. pushq %rbx //保存rbx
22. .cfi\_def\_cfa\_offset 24
23. .cfi\_offset 3, -24
24. subq $80, %rsp //建立栈帧
25. .cfi\_def\_cfa\_offset 104
26. movl $0, %ebx
27. jmp .L3
28. .L4:
29. movslq %ebx, %rbp
30. movq %rbp, %rsi
31. movq %rbp, %rdi
32. call leaf\_fun
33. movl %eax, (%rsp,%rbp,4) //%rsp+%rbp\*4指向a1[i]
34. addl $1, %ebx
35. .L3:
36. cmpl $18, %ebx
37. jle .L4
38. movl 8(%rsp), %eax
39. addq $80, %rsp //撤销栈帧
40. .cfi\_def\_cfa\_offset 24
41. popq %rbx //恢复rbx
42. .cfi\_def\_cfa\_offset 16
43. popq %rbp //恢复rbp
44. .cfi\_def\_cfa\_offset 8
45. ret
46. .cfi\_endproc
47. .LFE1:
48. .size func, .-func
49. .globl main
50. .type main, @function
51. main:
52. .LFB2:
53. .cfi\_startproc
54. movl $11, %esi
55. movl $20, %edi
56. call func
57. cltq
58. ret
59. .cfi\_endproc
60. .LFE2:
61. .size main, .-main
62. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
63. .section .note.GNU-stack,"",@progbits

代码 3‑41 call-localvar2.c

1. int func(long f1,long f2)
2. {
3. int i;
4. int a1[20];
5. for(i=0;i<19;i++)
6. a1[i]=f1\*i+f2;
7. return a1[2];
8. }
9. long main()
10. {
11. int mf1=20;
12. int mf2=11;
13. mf2=func(mf1,mf2);
14. return mf2;
15. }

代码 3‑42 call-localvar2.s

1. .file "call-localvar2.c"
2. .text
3. .globl func
4. .type func, @function
5. func:
6. .LFB0:
7. .cfi\_startproc
8. movl $0, %eax
9. jmp .L2
10. .L3:
11. movslq %eax, %rdx
12. movl %eax, %ecx
13. imull %edi, %ecx
14. addl %esi, %ecx
15. movl %ecx, -80(%rsp,%rdx,4) //%rsp-80+rdx\*4指向a1[i]
16. addl $1, %eax
17. .L2:
18. cmpl $18, %eax
19. jle .L3
20. movl -72(%rsp), %eax
21. ret
22. .cfi\_endproc
23. .LFE0:
24. .size func, .-func
25. .globl main
26. .type main, @function
27. main:
28. .LFB1:
29. .cfi\_startproc
30. movl $11, %esi
31. movl $20, %edi
32. call func
33. cltq
34. ret
35. .cfi\_endproc
36. .LFE1:
37. .size main, .-main
38. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
39. .section .note.GNU-stack,"",@progbits

代码 3‑43 fibonacci.c

1. #include<stdio.h>
2. #define N 8
3. int Fibonacci(int n)
4. {
5. int f;
7. if(n<=2)
8. f=1;
9. else
10. f = Fibonacci(n-1) + Fibonacci(n-2);
11. return f;
12. }
13. int main()
14. {
15. int c=0;
16. c = Fibonacci(N);
17. return c;
18. }

代码 3‑44 fibonacci.s

1. .file "fibonacci.c"
2. .text
3. .globl Fibonacci
4. .type Fibonacci, @function
5. Fibonacci:
6. .LFB11:
7. .cfi\_startproc
8. pushq %rbp //保存rbp
9. .cfi\_def\_cfa\_offset 16
10. .cfi\_offset 6, -16
11. pushq %rbx //保存rbx
12. .cfi\_def\_cfa\_offset 24
13. .cfi\_offset 3, -24
14. subq $8, %rsp //堆栈指针下移8字节
15. .cfi\_def\_cfa\_offset 32
16. movl %edi, %ebx //将参数n拷贝一份到ebx
17. cmpl $2, %edi //检查n<=2
18. jle .L3 //n<=2则跳转（返回数值1）
19. leal -1(%rdi), %edi //n-1作为Fibonacci（）参数
20. call Fibonacci //eax=Fibonacci(n-1)
21. movl %eax, %ebp //Fibonacci(n-1)保存到ebp
22. leal -2(%rbx), %edi //n-2作为Fibonacci()参数
23. call Fibonacci //eax=Fibonacci(n-2)
24. addl %ebp, %eax //eax=Fibonacci(n-2)+Fibonacci(n-1)
25. jmp .L2
26. .L3:
27. movl $1, %eax
28. .L2:
29. addq $8, %rsp //堆栈指针上移8字节
30. .cfi\_def\_cfa\_offset 24
31. popq %rbx //恢复rbx
32. .cfi\_def\_cfa\_offset 16
33. popq %rbp //恢复rbp
34. .cfi\_def\_cfa\_offset 8
35. ret
36. .cfi\_endproc
37. .LFE11:
38. .size Fibonacci, .-Fibonacci
39. .globl main
40. .type main, @function
41. main:
42. .LFB12:
43. .cfi\_startproc
44. subq $8, %rsp
45. .cfi\_def\_cfa\_offset 16
46. movl $8, %edi //Fibonacci()参数为8
47. call Fibonacci
48. addq $8, %rsp
49. .cfi\_def\_cfa\_offset 8
50. ret
51. .cfi\_endproc
52. .LFE12:
53. .size main, .-main
54. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
55. .section .note.GNU-stack,"",@progbits

代码 3‑45 argc-argv.c

1. #include <stdio.h>
2. int main(int argc,char \*argv[])
3. {
4. int i;
5. printf("argc = %d \n",argc);
6. for(i=0;i<argc; i++)
7. printf("arg%d:%s\n",i,argv[i]);
8. return 0;
9. }

代码 3‑46 env.c

1. #include <stdlib.h>
2. #include <stdio.h>
3. int main(void)
4. {
5. char \*pathvar;
6. pathvar = getenv("PATH");
7. printf("pathvar=%s",pathvar);
8. return 0;
9. }
   1. 小结

代码 3‑47 缓冲区溢出攻击的目标函数

1. int getbuf()
2. {
3. char buf[12];
4. gets(buf);
5. return 1;
6. }

# 链接与可执行文件

* 1. 生成可执行文件
     1. 样例代码

代码 4‑1 vector.h

1. void addvec(int \*x, int \*y,int \*z, int n);
2. void multvec(int \*x, int \*y,int \*z, int n);

代码 4‑2 main-lib.c

1. #include <stdio.h>
2. #include "vector.h"
3. int x[2] = {1, 2};
4. int y[2] = {3, 4};
5. int z[2];
6. int main()
7. {
8. addvec(x, y, z, 2);
9. printf("z = [%d %d]\n", z[0], z[1]);
10. return 0;
11. }

代码 4‑3 addvec.c

1. void addvec(int \*x, int \*y,int \*z, int n)
2. {
3. int i;
4. for (i = 0; i < n; i++)
5. z[i] = x[i] + y[i];
6. }

代码 4‑4 multvec.c

1. void multvec(int \*x, int \*y,int \*z, int n)
2. {
3. int i;
4. for (i = 0; i < n; i++)
5. z[i] = x[i] \* y[i];
6. }
   * 1. 进程影像
     2. ELF文件与装入

代码 4‑5 文件头数据结构Elf64\_hdr

1. typedef struct elf64\_hdr {
2. unsigned char e\_ident[EI\_NIDENT]; /\* ELF "magic number" \*/
3. Elf64\_Half e\_type;
4. Elf64\_Half e\_machine;
5. Elf64\_Word e\_version;
6. Elf64\_Addr e\_entry; /\* Entry point virtual address \*/
7. Elf64\_Off e\_phoff; /\* Program header table file offset \*/
8. Elf64\_Off e\_shoff; /\* Section header table file offset \*/
9. Elf64\_Word e\_flags;
10. Elf64\_Half e\_ehsize;
11. Elf64\_Half e\_phentsize;
12. Elf64\_Half e\_phnum;
13. Elf64\_Half e\_shentsize;
14. Elf64\_Half e\_shnum;
15. Elf64\_Half e\_shstrndx;
16. }Elf64\_Ehdr;

代码 4‑6 程序头表Elf64\_phdr结构体

1. typedef struct elf64\_phdr {
2. Elf64\_Word p\_type;
3. Elf64\_Word p\_flags;
4. Elf64\_Off p\_offset; /\* Segment file offset \*/
5. Elf64\_Addr p\_vaddr; /\* Segment virtual address \*/
6. Elf64\_Addr p\_paddr; /\* Segment physical address \*/
7. Elf64\_Xword p\_filesz; /\* Segment size in file \*/
8. Elf64\_Xword p\_memsz; /\* Segment size in memory \*/
9. Elf64\_Xword p\_align; /\* Segment alignment, file & memory \*/
10. }Elf64\_Phdr;

代码 4‑7 Elf64\_shdr结构体

1. typedef struct elf64\_shdr {
2. Elf64\_Word sh\_name; /\* Section name, index in string tbl \*/
3. Elf64\_Word sh\_type; /\* Type of section \*/
4. Elf64\_Xword sh\_flags; /\* Miscellaneous section attributes \*/
5. Elf64\_Addr sh\_addr; /\* Section virtual addr at execution \*/
6. Elf64\_Off sh\_offset; /\* Section file offset \*/
7. Elf64\_Xword sh\_size; /\* Size of section in bytes \*/
8. Elf64\_Word sh\_link; /\* Index of another section \*/
9. Elf64\_Word sh\_info; /\* Additional section information \*/
10. Elf64\_Xword sh\_addralign; /\* Section alignment \*/
11. Elf64\_Xword sh\_entsize; /\* Entry size if section holds table \*/
12. }Elf64\_Shdr;
    1. 可重定位目标文件
       1. 目标文件的节（section）

* + 1. 符号及重定位

代码 4‑8 static-local.c静态局部变量示例代码

1. int f()
2. {
3. static int x=0;
4. x=rand()\*10;
5. if (x>5)
6. { static int y;
7. y=y+1;
8. return y;
9. }else
10. {
11. static unsigned y;
12. y=y+100;
13. return y;
14. }
15. }

代码 4‑9 Elf64\_Rela

1. typedef struct elf64\_rela {
2. Elf64\_Addr r\_offset; /\* Location at which to apply the action \*/
3. Elf64\_Xword r\_info; /\* index and type of relocation \*/
4. Elf64\_Sxword r\_addend; /\* Constant addend used to compute value \*/
5. }Elf64\_Rela;
   * 1. 符号表

代码 4‑10 Elf64\_sym结构体

1. typedef struct elf64\_sym {
2. Elf64\_Word st\_name; /\* Symbol name, index in string tbl \*/
3. unsigned char st\_info; /\* Type and binding attributes \*/
4. unsigned char st\_other; /\* No defined meaning, 0 \*/
5. Elf64\_Half st\_shndx; /\* Associated section index \*/
6. Elf64\_Addr st\_value; /\* Value of the symbol \*/
7. Elf64\_Xword st\_size; /\* Associated symbol size \*/
8. } Elf64\_Sym;
   1. 静态链接
      1. 布局

代码 4‑11 addvec.s

1. .file "addvec.c"
2. .text
3. .globl addvec
4. .type addvec, @function
5. addvec:
6. .LFB0:
7. .cfi\_startproc
8. movl $0, %eax
9. jmp .L2
10. .L3:
11. movslq %eax, %r8
12. movl (%rsi,%r8,4), %r9d
13. addl (%rdi,%r8,4), %r9d
14. movl %r9d, (%rdx,%r8,4)
15. addl $1, %eax
16. .L2:
17. cmpl %ecx, %eax
18. jl .L3
19. rep ret
20. .cfi\_endproc
21. .LFE0:
22. .size addvec, .-addvec
23. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
24. .section .note.GNU-stack,"",@progbits
    * 1. 符号解析
      2. 静态重定位

代码 4‑12 main-lib2.c

1. #include <stdio.h>
2. #include "vector.h"
3. int x[2] = {1, 2};
4. int y[2] = {3, 4};
5. int z[2];
6. extern int gval=200;
7. extern int gval2;
8. int main()
9. {
10. addvec(x, y, z, 2);
11. printf("z = [%d %d]\n", z[0], z[1]);
12. return gval+gval2;
13. }

代码 4‑13 addvec2.c

1. int gval=200;
2. int gval2;
3. int addvec(int \*x, int \*y,int \*z, int n)
4. {
5. int i;
6. for (i = 0; i < n; i++)
7. z[i] = x[i] + y[i];
8. gval+=n;
9. return gval+gval2;
10. }
    1. 动态链接
       1. 样例代码
       2. 动态链接库
       3. 动态链接步骤小结

练习

代码 4‑14 multvec2.c

1. void multvec(int \*x, int \*y,int \*z, int n)
2. {
3. int i;
5. addvec(x,y,z,n);
6. for (i = 0; i < n; i++)
7. z[i] = x[i] \* y[i];
8. }

# 链接脚本与makefile

* 1. 二进制工具和链接脚本
     1. binutils
     2. 链接器脚本

代码 5‑1 简单的SECTIONS命令示例

1. SECTIONS
2. {
3. . = 0x10000;
4. .text : { \*(.text) }
5. . = 0x8000000;
6. .data : { \*(.data) }
7. .bss : { \*(.bss) }
8. }

代码 5‑2 SECTIONS命令格式

1. SECTIONS
2. {
3. sections-command
4. sections-command
5. ...
6. }

代码 5‑3 SECTIONS内的输出节描述命令

1. section [address] [(type)] :
2. [AT(lma)]
3. [ALIGN(section\_align) | ALIGN\_WITH\_INPUT]
4. [SUBALIGN(subsection\_align)]
5. [constraint]
6. {
7. output-section-command
8. output-section-command
9. ...
10. } [>region] [AT>lma\_region] [:phdr :phdr ...] [=fillexp] [,]

代码 5‑4 内存区域/内存块的定义

1. MEMORY
2. {
3. name [(attr)] : ORIGIN = origin, LENGTH = len
4. ...
5. }

代码 5‑5 定义rom、ram两个内存区域

1. MEMORY
2. {
3. rom (rx) : ORIGIN = 0, LENGTH = 256K
4. ram (!rx) : org = 0x40000000, l = 4M
5. }

代码 5‑6 定义内存块并将节映射到该内存块

1. MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
2. SECTIONS { ROM : { \*(.text) } >rom }

代码 5‑7 平台无关的输出节示例代码

1. INCLUDE linkcmds.memory
2. SECTIONS
3. {
4. .text :
5. {
6. \*(.text)
7. } > REGION\_TEXT
8. .rodata :
9. {
10. \*(.rodata)
11. rodata\_end = .;
12. } > REGION\_RODATA
13. .data : AT (rodata\_end)
14. {
15. data\_start = .;
16. \*(.data)
17. } > REGION\_DATA
18. data\_size = SIZEOF(.data);
19. data\_load\_start = LOADADDR(.data);
20. .bss :
21. {
22. \*(.bss)
23. } > REGION\_BSS
24. }

代码 5‑8 A型硬件平台用的linkcmds.memory文件

1. MEMORY
2. {
3. RAM : ORIGIN = 0, LENGTH = 4M
4. }
5. REGION\_ALIAS("REGION\_TEXT", RAM);
6. REGION\_ALIAS("REGION\_RODATA", RAM);
7. REGION\_ALIAS("REGION\_DATA", RAM);
8. REGION\_ALIAS("REGION\_BSS", RAM);

代码 5‑9 B型硬件平台用的linkcmds.memory文件

1. MEMORY
2. {
3. ROM : ORIGIN = 0, LENGTH = 3M
4. RAM : ORIGIN = 0x10000000, LENGTH = 1M
5. }
6. REGION\_ALIAS("REGION\_TEXT", ROM);
7. REGION\_ALIAS("REGION\_RODATA", ROM);
8. REGION\_ALIAS("REGION\_DATA", RAM);
9. REGION\_ALIAS("REGION\_BSS", RAM);

代码 5‑10 C型硬件平台用的linkcmds.memory文件

1. MEMORY
2. {
3. ROM : ORIGIN = 0, LENGTH = 2M
4. ROM2 : ORIGIN = 0x10000000, LENGTH = 1M
5. RAM : ORIGIN = 0x20000000, LENGTH = 1M
6. }
7. REGION\_ALIAS("REGION\_TEXT", ROM);
8. REGION\_ALIAS("REGION\_RODATA", ROM2);
9. REGION\_ALIAS("REGION\_DATA", RAM);
10. REGION\_ALIAS("REGION\_BSS", RAM);

代码 5‑11 bithd-mcu/vendor/libopenc3/linker.ld.S

1. /\* Generic linker script for all targets using libopencm3. \*/
2. /\* Enforce emmition of the vector table. \*/
3. EXTERN(vector\_table)
4. /\* Define the entry point of the output file. \*/
5. ENTRY(reset\_handler)
6. /\* Define memory regions. \*/
7. MEMORY
8. {
9. /\* RAM is always used \*/
10. ram (rwx) : ORIGIN = \_RAM\_OFF, LENGTH = \_RAM
11. #if defined(\_ROM)
12. rom (rx) : ORIGIN = \_ROM\_OFF, LENGTH = \_ROM
13. #endif
14. #if defined(\_ROM1)
15. rom1 (rx) : ORIGIN = \_ROM1\_OFF, LENGTH = \_ROM1
16. #endif
17. #if defined(\_ROM2)
18. rom2 (rx) : ORIGIN = \_ROM2\_OFF, LENGTH = \_ROM2
19. #endif
20. #if defined(\_RAM1)
21. ram1 (rwx) : ORIGIN = \_RAM1\_OFF, LENGTH = \_RAM1
22. #endif
23. #if defined(\_RAM2)
24. ram2 (rwx) : ORIGIN = \_RAM2\_OFF, LENGTH = \_RAM2
25. #endif

… 省略到其内存他配置，以节约版面 …

1. #if defined(\_NFCRAM)
2. nfcram (rw) : ORIGIN \_NFCRAM\_OFF, LENGTH = \_NFCRAM
3. #endif
4. }
5. /\* Define sections. \*/
6. SECTIONS
7. {
8. .text : {
9. \*(.vectors) /\* Vector table \*/
10. \*(.text\*) /\* Program code \*/
11. . = ALIGN(4);
12. \*(.rodata\*) /\* Read-only data \*/
13. . = ALIGN(4);
14. } >rom
15. /\* C++ Static constructors/destructors, also used for
16. \* \_\_attribute\_\_((constructor)) and the likes.
17. \*/
18. .preinit\_array : {
19. . = ALIGN(4);
20. \_\_preinit\_array\_start = .;
21. KEEP (\*(.preinit\_array))
22. \_\_preinit\_array\_end = .;
23. } >rom
24. .init\_array : {
25. . = ALIGN(4);
26. \_\_init\_array\_start = .;
27. KEEP (\*(SORT(.init\_array.\*)))
28. KEEP (\*(.init\_array))
29. \_\_init\_array\_end = .;
30. } >rom
31. .fini\_array : {
32. . = ALIGN(4);
33. \_\_fini\_array\_start = .;
34. KEEP (\*(.fini\_array))
35. KEEP (\*(SORT(.fini\_array.\*)))
36. \_\_fini\_array\_end = .;
37. } >rom
38. /\*
39. \* Another section used by C++ stuff, appears when using newlib with
40. \* 64bit (long long) printf support
41. \*/
42. .ARM.extab : {
43. \*(.ARM.extab\*)
44. } >rom
45. .ARM.exidx : {
46. \_\_exidx\_start = .;
47. \*(.ARM.exidx\*)
48. \_\_exidx\_end = .;
49. } >rom
50. . = ALIGN(4);
51. \_etext = .;
52. .data : {
53. \_data = .;
54. \*(.data\*) /\* Read-write initialized data \*/
55. . = ALIGN(4);
56. \_edata = .;
57. } >ram AT >rom
58. \_data\_loadaddr = LOADADDR(.data);
59. .bss : {
60. \*(.bss\*) /\* Read-write zero initialized data \*/
61. \*(COMMON)
62. . = ALIGN(4);
63. \_ebss = .;
64. } >ram
65. #if defined(\_CCM)
66. .ccm : {
67. \*(.ccmram\*)
68. . = ALIGN(4);
69. } >ccm
70. #endif
71. #if defined(\_RAM1)
72. .ram1 : {
73. \*(.ram1\*)
74. . = ALIGN(4);
75. } >ram1
76. #endif
77. #if defined(\_RAM2)
78. .ram2 : {
79. \*(.ram2\*)
80. . = ALIGN(4);
81. } >ram2
82. #endif

… 省略到其内存他配置，以节约版面 …

1. #if defined(\_NFCRAM)
2. .nfcram : {
3. \*(.nfcram\*)
4. . = ALIGN(4);
5. } >nfcram
6. #endif
7. /\*
8. \* The .eh\_frame section appears to be used for C++ exception handling.
9. \* You may need to fix this if you're using C++.
10. \*/
11. /DISCARD/ : { \*(.eh\_frame) }
12. . = ALIGN(4);
13. end = .;
14. }
15. PROVIDE(\_stack = ORIGIN(ram) + LENGTH(ram));
    1. makefile
       1. makefile基本格式

代码 5‑12 Makefile示例

1. EXE = main-lib main-lib-shared
2. all: $(EXE)
3. libvector.so : addvec.c multvec.c
4. gcc -shared -fPIC -o libvector.so addvec.c multvec.c
5. libvector.a: addvec.o multvec.o
6. $(AR) rcs libvector.a addvec.o multvec.o
7. $(RM) addvec.o multvec.o
8. main-lib-shared: main-lib.c libvector.so
9. $(CC) -o $@ $^
10. main-lib: main-lib.c libvector.a
11. $(CC) -o $@ $^
12. .PHONY : clean
13. clean:
14. $(RM) \*.o
    * 1. makefile规则

代码 5‑13 makefile中的规则

1. …
2. target ... : prerequisites ...
3. command
4. ...

代码 5‑14 隐式规则示例

1. main : main.o
2. gcc -o main main.o

代码 5‑15 隐式规则的等效代码

1. #等效于
2. main : main.o
3. gcc -o main main.o
4. main.o: main.c
5. gcc -c main.c

代码 5‑16 （动态）模式规则示例

1. #sample Makefile
2. %.o : %.c
3. $(CC) $(CFLAGS) $< -o $@

如果有f1.c、f2.c和f3.c三个C代码，那么上面的的第3、4行将实例化为规则：

1. f1.o :f1.c
2. $(CC) $(CFLAGS) f1.c -o f1.o
3. f2.o :f2.c
4. $(CC) $(CFLAGS) f2.c -o f2.o
5. f3.o :f3.c
6. $(CC) $(CFLAGS) f3.c -o f3.o

代码 5‑17 （静态）模式规则示例

1. objects = f1.o f2.o
2. all: $(objects)
3. $(objects): %.o: %.c
4. $(CC) -c $(CFLAGS) $< -o $@
   * 1. makefile变量

代码 5‑18 wildcard.mk

1. #Makefile 文件
2. src=$(wildcard \*.c ./sub/\*.c)
3. dir=$(notdir $(src))
4. obj=$(patsubst %.c,%.o,$(dir) )
5. all:
6. @echo $(src)
7. @echo $(dir)
8. @echo $(obj)
   * 1. 文件指示
     2. 函数

代码 5‑19 call-mk

1. # Makefile 内容
2. log = "====debug====" $(1) "====end===="
3. all:
4. @echo $(call log,"正在 Make")
   * 1. make
   1. 小结

练习

# 程序运行

* 1. 装入与运行
     1. ELF装载器

代码 6‑1 binfmt\_elf.c中的linux\_binfmt结构体

1. static struct [linux\_binfmt](http://lxr.linux.no/linux+*/+code=linux_binfmt) [elf\_format](http://lxr.linux.no/linux+*/+code=elf_format) = {
2. .[module](http://lxr.linux.no/linux+*/+code=module) = [THIS\_MODULE](http://lxr.linux.no/linux+*/+code=THIS_MODULE),
3. .[load\_binary](http://lxr.linux.no/linux+*/+code=load_binary) = [load\_elf\_binary](http://lxr.linux.no/linux+*/+code=load_elf_binary), 可执行文件的装载
4. .[load\_shlib](http://lxr.linux.no/linux+*/+code=load_shlib) = [load\_elf\_library](http://lxr.linux.no/linux+*/+code=load_elf_library), 共享库的装载
5. .[core\_dump](http://lxr.linux.no/linux+*/+code=core_dump) = [elf\_core\_dump](http://lxr.linux.no/linux+*/+code=elf_core_dump), 内核转储（吐核）
6. .[min\_coredump](http://lxr.linux.no/linux+*/+code=min_coredump) = [ELF\_EXEC\_PAGESIZE](http://lxr.linux.no/linux+*/+code=ELF_EXEC_PAGESIZE),
7. };
   * 1. 内核代码
     2. 进程与线程
     3. 工作环境
   1. 基本行为观察
      1. ptrace

代码 6‑2 minimal\_strace.c

1. /\* C standard library \*/
2. #include <errno.h>
3. #include <stdio.h>
4. #include <stddef.h>
5. #include <stdlib.h>
6. #include <string.h>
7. /\* POSIX \*/
8. #include <unistd.h>
9. #include <sys/user.h>
10. #include <sys/wait.h>
11. /\* Linux \*/
12. #include <syscall.h>
13. #include <sys/ptrace.h>
14. #define FATAL(...) \
15. do { \
16. fprintf(stderr, "strace: " \_\_VA\_ARGS\_\_); \
17. fputc('\n', stderr); \
18. exit(EXIT\_FAILURE); \
19. } while (0)
20. int
21. main(int argc, char \*\*argv)
22. {
23. if (argc <= 1)
24. FATAL("too few arguments: %d", argc);
25. pid\_t pid = fork();
26. switch (pid) {
27. case -1: /\* error \*/
28. FATAL("%s", strerror(errno));
29. case 0: /\* child \*/
30. ptrace(PTRACE\_TRACEME, 0, 0, 0);
31. execvp(argv[1], argv + 1);
32. FATAL("%s", strerror(errno));
33. }
34. /\* parent \*/
35. waitpid(pid, 0, 0); // sync with PTRACE\_TRACEME
36. for (;;) {
37. /\* Enter next system call \*/
38. if (ptrace(PTRACE\_SYSCALL, pid, 0, 0) == -1)
39. FATAL("%s", strerror(errno));
40. if (waitpid(pid, 0, 0) == -1)
41. FATAL("%s", strerror(errno));
42. /\* Gather system call arguments \*/ //收到系统调用第一次信号（进入）
43. struct user\_regs\_struct regs;
44. if (ptrace(PTRACE\_GETREGS, pid, 0, &regs) == -1) //读取寄存器信息（调用参数）
45. FATAL("%s", strerror(errno));
46. long syscall = regs.orig\_rax; //系统调用号在RAX寄存器中
47. /\* Print a representation of the system call \*/
48. fprintf(stderr, "%ld(%ld, %ld, %ld, %ld, %ld, %ld)", //打印系统调用号、所有6个参数
49. syscall,
50. (long)regs.rdi, (long)regs.rsi, (long)regs.rdx,
51. (long)regs.r10, (long)regs.r8, (long)regs.r9);
52. /\* Run system call and stop on exit \*/ //收到系统调用第二次信号（返回）
53. if (ptrace(PTRACE\_SYSCALL, pid, 0, 0) == -1)
54. FATAL("%s", strerror(errno));
55. if (waitpid(pid, 0, 0) == -1)
56. FATAL("%s", strerror(errno));
57. /\* Get system call result \*/
58. if (ptrace(PTRACE\_GETREGS, pid, 0, &regs) == -1) { //读取系统调用返回值（RAX）
59. fputs(" = ?\n", stderr);
60. if (errno == ESRCH)
61. exit(regs.rdi); // system call was \_exit(2) or similar
62. FATAL("%s", strerror(errno));
63. }
64. /\* Print system call result \*/
65. fprintf(stderr, " = %ld\n", (long)regs.rax); //打印系统调用返回值
66. }
67. }
    * 1. strace

代码 6‑3 \_TIF\_WORK\_SYSCALL\_ENTRY标志（linux/arch/x86/include/asm/thread\_info.h）

123 #define \_TIF\_WORK\_SYSCALL\_ENTRY \

124 (\_TIF\_SYSCALL\_TRACE | \_TIF\_SYSCALL\_EMU | \_TIF\_SYSCALL\_AUDIT | \

124 \_TIF\_SECCOMP | \_TIF\_SINGLESTEP | \_TIF\_SYSCALL\_TRACEPOINT | \

126 \_TIF\_NOHZ)

代码 6‑4 arch/x86/kernel/entry\_64.S中的system\_call入口代码片段

…

618 testl $\_TIF\_WORK\_SYSCALL\_ENTRY,TI\_flags+THREAD\_INFO(%rsp,RIP-ARGOFFSET)

619 jnz tracesys

…

代码 6‑5 testlsof.c

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <sys/types.h>
4. #include <sys/stat.h>
5. #include <fcntl.h>
6. int main(void)
7. {
8. open("/tmp/foo", O\_CREAT|O\_RDONLY); /\* 打开文件/tmp/foo \*/
9. getchar(); /\* 等待按键，以便我们做lsof检查 \*/
10. return 0;
11. }
    * 1. gdb断点原理
      2. ltrace

代码 6‑6 pipe-demo.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <sys/types.h>
5. int main()
6. {
7. pid\_t pid = 0;
8. int fds[2];
9. char buf[128];
10. int nwr = 0;
11. pipe(fds);//should before fork()
12. pid = fork();
13. if(pid < 0)
14. {
15. printf("Fork error!\n");
16. return -1;
17. }else if(pid == 0)
18. {
19. printf("This is child process, pid = %d\n", getpid());
20. printf("Child:waiting for message...\n");
21. close(fds[1]);
22. nwr = read(fds[0], buf, sizeof(buf));
23. printf("Child:received\"%s\"\n", buf);
24. }else{
25. printf("This is parent process, pid = %d\n", getpid());
26. printf("Parent:sending message...\n");
27. close(fds[0]);
28. strcpy(buf, "Message from parent!");
29. nwr = write(fds[1], buf, sizeof(buf));
30. printf("Parent:send %d bytes to child.\n", nwr);
31. }
32. return 0;
33. }
    1. 异常行为
       1. 非法操作

代码 6‑7 io.S

1. .file "io.s"
2. .text
3. .globl main
4. .type main, @function
5. main:
6. .LFB0:
7. .cfi\_startproc
8. pushq %rbp
9. .cfi\_def\_cfa\_offset 16
10. .cfi\_offset 6, -16
11. movq %rsp, %rbp
12. .cfi\_def\_cfa\_register 6
13. addl $32, -4(%rbp)
14. in $0x70,%eax #执行对IO端口0x70的读入操作
15. movl $0, %eax
16. popq %rbp
17. .cfi\_def\_cfa 7, 8
18. ret
19. .cfi\_endproc
20. .LFE0:
21. .size main, .-main
22. .ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
23. .section .note.GNU-stack,"",@progbits
    * 1. 响应信号

代码 6‑8 kill()头文件和函数原型

1. **#include <sys/types.h>**
2. **#include <signal.h>**
3. **int kill(pid\_t pid, int sig);**

代码 6‑9 sigprocmask()的头文件和函数原型

1. #include <signal.h>
2. int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

代码 6‑10 HelloWorld-getchar-sig.c

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <signal.h>
4. void handler(){
5. printf("handling signal(alarm,14) in user code!\n");
6. }
7. int main()
8. {
9. signal(SIGALRM,handler);
10. printf("HelloWorld!\n");
11. getchar();
12. return 0;
13. }

代码 6‑11 rt\_sigframe (linux-3.13/arch/x86/include/asm/sigframe.h)

1. struct rt\_sigframe {
2. char \_\_user \*pretcode; 信号处理函数返回操作（将会执行sys\_sigreturn）
3. struct ucontext uc; 用户态的上下文环境
4. struct siginfo info; 信号相关的信息
5. /\* fp state follows here \*/
6. };
   * 1. core文件

代码 6‑12 coredump-demo.c

1. #include <stdio.h>
2. void core\_test1()
3. {
4. int i = 0;
5. // the next statement will call segmentfault
6. scanf("%d", i);
7. printf("%d\n", i);
9. }
10. int main()
11. {
12. core\_test1();
13. return 0;
14. }
    1. 小结

# 性能剖析

* 1. 打桩方法

代码 7‑1 interposit.c

1. #include <stdio.h>
2. #include <malloc.h>
3. int main()
4. {
5. int \*p= malloc(128);
6. free(p);
7. return (0);
8. }
   * 1. 源代码预处理时

代码 7‑2 malloc.h

1. #define malloc(size) mymalloc(size)
2. #define free (ptr) myfree(ptr);
3. void \*mymalloc(size\_t, size);
4. void myfree(void \*ptr);

代码 7‑3 mymalloc\_pp.c

1. #ifdef INTERPOSIT\_IN\_PREP
2. #include <stdio.h>
3. #include <malloc.h>
4. //wrapper function for malloc()
5. void \*mymalloc(sizt\_t size)
6. {
7. void \*ptr= malloc(size);
8. printf("malloc(%d) at addr %p\n",(int)size,ptr);
9. return ptr;
10. }
11. //wrapper functiion for free()
12. void myfree(void \*ptr)
13. {
14. free(ptr);
15. printf("free at (%p)\n",ptr);
16. }
17. #endif
    * 1. 静态链接时

代码 7‑4 mymalloc\_sl.c

1. #ifdef INTERPOSIT\_IN\_STATIC\_LINK
2. #include <stdio.h>
3. void \*\_\_reall\_malloc(size\_t size);
4. void \_\_real\_free(void \*ptr);
5. //wrapper function for malloc()
6. void \*\_\_wrap\_malloc(size\_t size)
7. {
8. void \*ptr = \_\_real\_malloc(size);
9. printf("malloc(%d) at addr %p\n",(int)size,ptr);
10. return ptr;
11. }
12. //wrapper functiion for free()
13. void \_\_wrap\_free(void \*ptr)
14. {
15. \_\_real\_free(ptr);
16. printf("free at (%p)\n",ptr);
17. }
18. #endif
    * 1. 运行加载时

代码 7‑5 mymalloc\_rt.c

1. #ifdef INTERPOSIT\_IN\_RT
2. #define \_GNU\_SOURCE
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <dlfcn.h>
6. //wrapper function for malloc()
7. void \*malloc(size\_t size)
8. {
9. void \*(\*mallocp)(size\_t size)= NULL;
10. char \*error;
11. mallocp=dlsym(RTLD\_NEXT,"malloc"); //resolved malloc,get address
12. if((error = dlerror())!= NULL){
13. fputs(error, stderr);
14. exit(1);
15. }
16. char \*ptr = mallocp(size);
17. printf("malloc(%d) at addr %p\n",(int)size,ptr);
18. return ptr;
19. }
20. //wrapper functiion for free()
21. void free(void \*ptr)
22. {
23. void (\*freep)(void \*) =NULL;
24. char \*error;
25. if(!ptr)
26. return;
28. freep=dlsym(RTLD\_NEXT,"free"); //resolved free, get address
29. if((error =dlerror())!= NULL){
30. fputs(error, stderr);
31. exit(1);
32. }
33. freep(ptr);
34. printf("free at (%p)\n",ptr);
35. }
36. #endif
    1. gprof
       1. 工作原理
       2. gprof示例

代码 7‑6 gprof-exam1.c源代码

1. #define MILLION 1000000
2. void subroutine()
3. {
4. for (int i = 0; i < 100 \* MILLION; i++)
5. ;
6. return;
7. }
8. void loop\_100\_million()
9. {
10. for (int i = 0; i < 100 \* MILLION; i++)
11. ;
12. subroutine ();
13. return;
14. }
15. void loop\_1000\_million()
16. {
17. for (int i = 0; i < 1000 \* MILLION; i++)
18. ;
19. subroutine ();
20. return;
21. }
22. int main(int argc, char\*\* argv)
23. {
24. loop\_100\_million();
25. loop\_100\_million();
26. loop\_100\_million();
27. loop\_1000\_million();
28. loop\_1000\_million();
29. return 1;
30. }
    * 1. 性能数据解读
      2. 图形化显示（gprof2dot.py+graphviz）
    1. gcov
       1. 基于函数分析的缺点
       2. gcov逐行分析
    2. 其他分析工具
       1. Valgrind

代码 7‑7 mem-demo.c

1. #include <stdlib.h>
2. #include <stdio.h>
4. void test(int n) {
5. n = n + 1;
6. write(stdout, "xxx", n); //向系统函数传递未初始化参数
7. malloc(220); //内存泄露
8. }
10. int main() {
11. const int array\_count = 4;
12. int\* p = malloc(array\_count \* sizeof(int));
13. p[array\_count]=p[array\_count]+5; //读、写未分配空间
14. test(p[array\_count - 1]);
15. printf("%d\n",p[array\_count-1]); //向系统函数传递未初始化参数
16. memcpy(p+1,p,sizeof(char)\*array\_count); //内存拷贝源和目的区间重叠
17. free(&array\_count); //尝试释放堆栈上的空间
18. free(p);
19. free(p+1); //重复释放内存
20. return 0;
21. }

代码 7‑8 massif-demo.c

1. #include <stdlib.h>
2. #include <stdio.h>
3. int \*fa()
4. {
5. int \*p=(int\*)malloc(10000);
6. return p;
7. }
8. int \*fb(int \*p)
9. {
10. free(p);
11. }
12. int main(void)
13. {
14. printf("ok\n");
15. printf("really ok?\n");
16. int \*vec[10000];
17. for(int i=0;i<10000;i++)
18. {
19. vec[i]=fa();
20. }
21. for(int i=0;i<10000;i++)
22. {
23. fb(vec[i]);
24. }
25. return 0;
26. }
    * 1. perf
    1. 小结

# 综合实例: HDFS中实现zlib库的旁路

* 1. 项目需求
  2. 系统分析
     1. 整体方案
     2. Haddop的gzip JNI

代码 8‑1 libz.so.1中被HDFS JNI调用的函数列表

1. static int deflateInit2\_ (z\_streamp, int, int, int, int, int, const char \*, int);
2. static int deflate (z\_streamp, int);
3. static int deflateSetDictionary) (z\_streamp, const Bytef \*, uInt);
4. static int deflateReset (z\_streamp);
5. static int deflateEnd (z\_streamp);
6. static int inflateInit2\_ (z\_streamp, int, const char \*, int);
7. static int inflate (z\_streamp, int);
8. static int inflateSetDictionary (z\_streamp, const Bytef \*, uInt);
9. static int inflateReset (z\_streamp);
10. static int inflateEnd (z\_streamp);
    * 1. zlib分析

代码 8‑2 z\_stream结构体

1. typedef struct z\_stream\_s {
2. z\_const Bytef \*next\_in; /\* next input byte \*/
3. uInt avail\_in; /\* number of bytes available at next\_in \*/
4. uLong total\_in; /\* total number of input bytes read so far \*/
5. Bytef \*next\_out; /\* next output byte will go here \*/
6. uInt avail\_out; /\* remaining free space at next\_out \*/
7. uLong total\_out; /\* total number of bytes output so far \*/
8. z\_const char \*msg; /\* last error message, NULL if no error \*/
9. struct internal\_state FAR \*state; /\* not visible by applications \*/
10. alloc\_func zalloc; /\* used to allocate the internal state \*/
11. free\_func zfree; /\* used to free the internal state \*/
12. voidpf opaque; /\* private data object passed to zalloc and zfree \*/
13. int data\_type; /\* best guess about the data type: binary or text
14. for deflate, or the decoding state for inflate \*/
15. uLong adler; /\* Adler-32 or CRC-32 value of the uncompressed data \*/
16. uLong reserved; /\* reserved for future use \*/
17. } z\_stream;
18. typedef z\_stream FAR \*z\_streamp;
    * 1. 测定z\_stream成员大小
    1. 编码实现
       1. zlib日志

代码 8‑3 修改后的log4crc配置文件

1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE log4c SYSTEM "">
3. <log4c>
4. <config>
5. <bufsize>0</bufsize>
6. <debug level="2"/>
7. <nocleanup>0</nocleanup>
8. <reread>1</reread>
9. </config>
10. <!-- root category ========================================= -->
11. <category name="root" priority="notice"/>
12. <category name="zlib" priority="notice" appender="zlib"/>
13. <category name="log4c.examples.helloworld" priority="debug" appender="stdout"/>
14. <appender name="rollfile" type="rollingfile" logdir="." prefix="Testlog" layout="dated" rollingpolicy="RollingPolicy"/>
15. <appender name="zlib" type="rollingfile" logdir="/tmp" prefix="Zliblog" layout="dated" rollingpolicy="RollingPolicy2"/>
16. <!-- default appenders ===================================== -->
17. <appender name="stdout" type="stream" layout="basic"/>
18. <appender name="stderr" type="stream" layout="dated"/>
19. <appender name="syslog" type="syslog" layout="basic"/>
20. <!-- default layouts ======================================= -->
21. <layout name="basic" type="basic"/>
22. <layout name="dated" type="dated"/>
23. <rollingpolicy name="RollingPolicy" type="sizewin" maxsize="102400" maxnum="10"/>
24. <rollingpolicy name="RollingPolicy2" type="sizewin" maxsize="102400" maxnum="3"/>
25. </log4c>

代码 8‑4 log.h

1. #ifndef \_LOG\_H\_
2. #define \_LOG\_H\_
4. #include <string.h>
5. #include <stdlib.h>
6. #include <assert.h>
7. #ifdef \_\_cplusplus
8. extern "C"
9. {
10. #endif
12. #include <log4c.h>
14. #ifdef \_\_cplusplus
15. }
16. #endif
18. #define LOG\_PRI\_ERROR LOG4C\_PRIORITY\_ERROR
19. #define LOG\_PRI\_WARN LOG4C\_PRIORITY\_WARN
20. #define LOG\_PRI\_NOTICE LOG4C\_PRIORITY\_NOTICE
21. #define LOG\_PRI\_DEBUG LOG4C\_PRIORITY\_DEBUG
22. #define LOG\_PRI\_TRACE LOG4C\_PRIORITY\_TRACE
24. extern int log\_open(const char \*category);
25. extern void log\_message(int priority ,const char\* fmt, ...);
26. extern void log\_trace(const char \*file , int line , const char \*func, const char \*fmt ,...);
27. extern int log\_close();
29. #define LOG\_ERROR(fmt , args...) \
30. log\_message(LOG\_PRI\_ERROR, fmt, ##args)
31. #define LOG\_WARN(fmt, args...) \
32. log\_message(LOG\_PRI\_WARN, fmt , ##args)
33. #define LOG\_NOTICE(fmt , args...) \
34. log\_message(LOG\_PRI\_NOTICE, fmt , ##args)
35. #define LOG\_DEBUG(fmt , args...) \
36. log\_message(LOG\_PRI\_DEBUG, fmt , ##args)
37. #define LOG\_TRACE(fmt,args...) \
38. log\_trace(\_\_FILE\_\_ , \_\_LINE\_\_ , \_\_FUNCTION\_\_ , fmt ,## args)

41. #endif
42. static log4c\_category\_t \*log\_category = NULL;
44. int log\_open(const char \*category)
45. {
46. if (log4c\_init() != 0)
47. {
48. printf("log\_open fail\n");
49. return -1;
50. }
51. printf("log\_open is %s ok\n",category);
52. log\_category = log4c\_category\_get(category);
53. return 0 ;
54. }
56. void log\_message(int priority , const char \*fmt , ...)
57. {
58. // printf("this a log\_message\n");
59. va\_list ap;
61. assert(log\_category != NULL);
63. va\_start(ap, fmt);
64. log4c\_category\_vlog(log\_category , priority , fmt , ap);
65. va\_end(ap);
66. }

代码 8‑5 修改deflateInit2\_()函数支持liblog4c库

1. int ZEXPORT deflateInit2\_(strm, level, method, windowBits, memLevel, strategy,
2. version, stream\_size)
3. z\_streamp strm;
4. int level;
5. int method;
6. int windowBits;
7. int memLevel;
8. int strategy;
9. const char \*version;
10. int stream\_size;
11. {
12. log\_open("zlib");
13. LOG\_DEBUG("deflateInit2\_ exe");
14. void\*dlcpress= dlopen("/home/hessen/da200-compress/lib/libcprss.so",RTLD\_LAZY|RTLD\_GLOBAL);
15. if(dlcpress == NULL)
16. {
17. LOG\_DEBUG("dlcpress open error");
18. }else {
19. LOG\_DEBUG("dlcpress open success");
20. }

代码 8‑6 修改后的deflate()

1. int ZEXPORT deflate (strm, flush)
2. z\_streamp strm;
3. int flush;
4. {
5. LOG\_DEBUG("deflate exe"); // 日志记录内容
6. LOG\_DEBUG("deflate begin | avail\_in = %d",strm->avail\_in);
7. LOG\_DEBUG("deflate begin | avail\_out = %d",strm->avail\_out);
8. LOG\_DEBUG("deflate begin | flush = %d",flush);



13. strm->next\_in += strm->avail\_in; //消费掉输入数据
14. strm->avail\_in = 0;
16. strm->next\_out += filesize; //产生出“空”的输出数据
17. strm->avail\_out -= filesize;


21. return Z\_STREAM\_END;
23. }
    * 1. 使用libcprss.so库

代码 8‑7 实现deflate()压缩的FPGA加速功能

1. int ZEXPORT deflate (strm, flush)
2. z\_streamp strm;
3. int flush;
4. {
5. int fd\_log = open("/home/hadoop/zliblog",O\_RDWR|O\_CREAT,0777);
6. write(fd\_log,"deflate begin\n",15);
7. void \* lib\_so = dlopen("/home/hadoop/libcprss.so",RTLD\_LAZY | RTLD\_GLOBAL);
8. if(lib\_so == NULL)
9. {
10. write(fd\_log,"dlopen error",20);
11. close(fd\_log);
12. return Z\_ERRNO;
13. }
14. int (\*compress)();
15. compress = (int(\*)())dlsym(lib\_so,"ex\_compress");
16. }
17. int have;
18. int ret = compress(strm->next\_out,strm->avail\_out,strm->next\_in,strm->avail\_in,&have,0);
19. if(ret == 0)
20. {
21. write(fd\_log,"compress sucess",25);
22. }
23. ｝

代码 8‑8 replacelib.sh

1. #! /bin/bash
2. cp libz.so.1 /home/hadoop/bigdata/hadoop/lib/native
3. cd /home/hadoop/bigdata/hadoop/lib/native
4. hadoop checknative
   1. 功能验证
      1. 准备输入文件
      2. zlib原生库的压缩
      3. libcprss.so库的压缩
   2. 小结

# 附录