编译原理
调研报告1

梁宸
1511186

October 10, 2017

Abstract

这是一篇有关包含预处理、编译、汇编、链接的整个编译过程的调研报告, 主要调研方式是看文档、看书、看博客、看stackOverflow.

关键词: 预处理、编译、汇编、链接、宏、GENERIC、Gimple、基本块、控制流、数据流、RTL、目标文件、重定位、段、符号解析.

Contents

1 引言

作为生产的重要工具, 编译器是我们天天都能见到的好伙伴. 虽然对编译各个过程曾有耳闻, 但一直没有较为全面的了解. 本次调研相应编译原理课程号召, 以实事求是为指导思想, 理论与实践结合, 观察了例程各个阶段的输出, 并简单了解了背后的过程及原理.

2 实验环境

本实验处理器指令集为x86-64, 前半部分环境为操作系统为Windows10 64bit, 所用编译器为MinGW w64项目移植的GCC 6.3.0, 进程标准为POSIX, 异常标准为SEH, revision为#1; 后半部分环境为虚拟机下的Ubuntu

3 C标准翻译过程

C11的标准规定了翻译过程的8个阶段:

1. 字符映射: 源文件依照不同的定义[1]映射为一个字符集. 此外三联符(trigraph)[2]会被替换为相应的符号.

2. 行连接: \续接的行被合并为一行, 这使得物理行变为逻辑行.

3. 分词: 预处理器将输入文件分词, 得到**预处理单词(preprocessing token)**[3]和空格序列.

4. 预处理: 所有**预处理指令(preprocessing directive)**被执行, **宏**被展开, _Pragma运算符表达式被执行. 任何包含头文件都被递归地从阶段1开始处理. 完成后, 所有预处理指令都被删除

5. 字符集映射: 原字符集的每一个成员以及字符常量和字面常量中的每一个生成控制字符的序列[4]都会被转为执行时字符集的相应成员, 如果做不到, 就按规定的方式转为别的字符, 而不是空(NUL).

6. 字符串连接: 相邻的字面常量被合并, 例如"ABC""DEF"合并为"ABCDEF".

7. 翻译: 每个预处理单词被转换为单词, 它们会在语法上和语义上被分析, 最后被翻译为翻译单元.

8. 链接: 所有外部的对象和函数被解引用, 外部库被链接进来.所有的翻译单元被被集合到一起, 构成一个能独立被执行的程序镜像.

[1]比如UTF-8或者ASCII.
[2]有些古老的键盘无法输入诸如[这种符号, 于是用三个相连的特定符号—称为三联符— 代替
[3]"预处理语言"的单词.
[4]比如B̂序列表示控制字符STX(Start of Text)

4 预处理器

4.1 简介

C语言的预处理器(C preprocessor, 也可缩写为cpp)是一个宏处理器, 在开始编译之前被编译器所调用. 它的原本用途只是用于处理C, C++ 和Objective-C的源码, 但在过去也经常被滥用作一般文本处理器. 它在遇到不符合C 词法的输入时会终止.

4.2 工作流程

4.2.1 初始处理

在最开始, 预处理器进行一些文本的转换, 在概念上这些转换应该是严格串行的, 但C预处理器实际上将他们一起执行. 这些转换大概对应C标准中描述的翻译的前三个阶段.

1. 输入文件被读进内存, 并被划分成行
2. 如果三联符编译选项被启用, 那么会进行相应的符号替换.
3. 把被续的行合并成一行.
4. 所有注释被替换为单空格.

4.2.2 分词

在文本转换之后, 输入文件被转换为预处理单词的流, 它基本上与C编译器的语义符号对应, 但还是有一些不同. 空格用于分割单词, 而不是单词. 单词之间原则上不需要空格分割: 预处理器会贪心地将字符串识别为单词. 但为了避免歧义, 通常会使用空格. 一旦分词完成, 词之间的边界就不会再改变, 除非使用##运算符连接. 预处理后的单词就是编译器会接收的单词, 编译器不会再次分词[5].

```
1  #define foo() bar
2  foo()bar
```

(a) 例1输入

```
1  # 1 "test.cpp"
2  # 1 "<built-in>"
3  # 1 "<command-line>"
4  # 1 "test.cpp"
5
6  bar bar
```

(b) 例1输出

如上例, 字符串'foobar'中有一宏foo, 被识别、展开后被空格分隔开.

[5]预处理器的单词和编译器的单词有不同, 所以不保证输出能被编译器认为有效

预处理器的单词可以分为四类:

1. **标识符**: 预处理器的标识符定义与C基本一致, 但不会区别大部分C的保留关键字, 预处理器可能只有一个保留关键字:defined.
2. **数(preprocessing number)**: 数字以一个可选的句号开始, 中间必须有一个数字位, 最后可以有任意的数字、字母、下划线、固定串、指数位.
3. **字面常量(string literal)**: 字面常量是字符串常量、字符常量或者头文件名[6].
4. **符号**: 和C语言的符号基本一致, 也存在针对为老式计算机提供的转移符号.

4.2.3 预处理语言

分词后的文本可能会被直接传到C编译器, 但是如果含有预处理语言, 则还需要先转换. 这一过程对应C标准的翻译第四阶段, 通常也被认为是预处理器的分内工作.

预处理语言由要执行的指令和要展开的宏构成, 他们最主要的功能是:

- **包含头文件**: 它们会替换文件中的头文件声明.
- **宏展开**: 你可以定义宏, 它们是任意C代码片段的缩写. 预处理器会把所有的宏展开为对应的代码. 有些宏是预先定义好的.
- **条件编译**: 你可以根据某些变量来决定是否把代码包含到程序中.
- **行控制**: 如果你想组合或者重新编排一些源文件到一个中间文件, 你可以用行控制来告诉编译器其中每一行来自哪里.
- **诊断**: 你可以在编译时检测错误或者提出错误和警告.

除了预先定义的宏以外, 所有的功能都由预处理指令触发, 它以#开始, 允许两侧有空格, 接着是一个标识符, 表明指令名. 这两部分都不能来自于宏展开, 否则不被识别为指令. 预处理指令是固定的, 程序不能定义自己的指令. 有些指令需要参数, 参数必须以空格隔开, 如#define.

4.3 例程输出分析

用cpp命令和-dI参数可以获得例程2的预处理结果[7], 其中-dI参数指示预处理器附加原始的#include指令在输出文件的相应位置.

值得注意的是, 预处理器将所有预处理指令在输出文件中表示为一个空行. 为预处理器每行输出的格式为:

linenum filename flags

[6] #include的参数
[7] 一般预处理输出文件的扩展名可以是.i, 但C语言标准并没有做出规定.

```cpp
// 2017-09-18 20:34:59
// test code #1 for Compilers homework #1




#include <iostream>

using namespace std;


int main()
{
    int i, n, f;

    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    cout << n << endl;
    return 1 - 1;

}
```

(c) 例2输入

```
# 1 "H1_1.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "H1_1.cpp"




# 1 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/iostream" 1 3
# 55 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/iostream" 3

# 56 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/iostream" 3

# 1 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 1 3
# 199 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 3

# 199 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 3
namespace std
{
  typedef long long unsigned int size_t;
  typedef long long int ptrdiff_t;


  typedef decltype(nullptr) nullptr_t;

}
# 221 "C:/Program Files/mingw-w64/x86_64-6.3.0-posix-seh-rt_v5-rev1/mingw64/lib/gcc/x86_64-w64-mingw32/6.3.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 3
namespace std
{
  inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
namespace __gnu_cxx
{
  inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
```

(d) 例2输出

意思是, 接下来的内容来自filename文件的linenum行, flags为标志位:

- 1: 打开了一个新文件.
- 2: 返回了一个文件(结束了一个文件之后).
- 3: 来自一个系统头文件, 因此应禁止一些警告.
- 4: 接下来的内容应该被看作在一个隐含的C语言外部块里[8].

可以看到, 在开头的几行, 先进入了built-in文件, 在-dN的预处理参数下能看到在该文件里定义了许多预置的宏; 之后进入了command-line[9], 这里按照命令行的参数定义一些宏[10]; 接下来是正式的输入文件. 粗略看来#include命令事实上是按DFS的顺序执行的, 再加上标志位, linemarker 可以被认为是预处理的一个日志.

4.4 总结

预处理器的功能主要是两方面: 一是帮助分词, 编译器不再对预处理器的输出再分词; 二是在文本层面上做一些处理, 比如宏、头文件和条件编译, 这其实与编译无关, 将上述功能硬编码对编译器也是等价的, 但能极大提高编码的工作效率, 也因此在事实上成为必不可少的环节.

[8]不明白, 试了几个经典的implicit extern都没看到4出现
[9]我不确定这是不是文件
[10]命令行下cpp 命令可以使用参数预定义一些宏

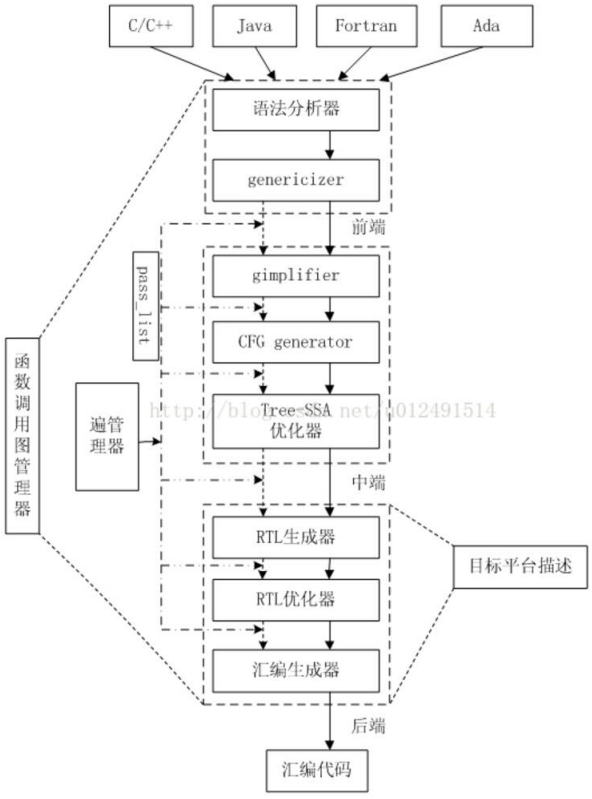
5 编译器

5.1 简介

编译器是GCC和核心部件, 也十分复杂, 所以我只能简单看看. GCC编译器事实上不再对预处理器的输出进行分词, 它会继续进行语法分析和语义分析, 并且在分析的各阶段中间结果中进行优化, 最后生成机器相关的汇编代码.

5.2 工作流程

5.2.1 流程图



(e) GCC流程图

GCC的编译器会将源代码先后翻译成多种中间语言, 并且在上面进行优化, 这称为一个pass.

5.2.2 语法树

编译器在预处理器的分词结果上进行语法分析, 分析结果的各个语法部件存到一个叫做**GENERIC**的树里, 不同的前端语言可能会使用不同的表示方法, 但GENERIC树支持GCC 支持的所有语言. GENERIC可以被看作Parser到Optimizer的中间桥梁.

```
@1      namespace_decl   name: @2       scpe: @3       srcp: <built-in>:0
                         dcls: @4
@2      identifier_node  strg: ::       lngt: 2
@3      translation_unit_decl
@4      function_decl    name: @5       type: @6       srcp: iostream:93
                         note: artificial              chain: @7
                         lang: C        link: static   body: @8
@5      identifier_node  strg: __tcf_0  lngt: 7
@6      function_type    size: @9       algn: 8        retn: @10
                         prms: @11
@7      function_decl    name: @12      type: @13      srcp: H1_1.cpp:28
                         note: artificial              chain: @14
                         args: @15      link: static   body: @16
@8      cleanup_point_expr type: @10      op 0: @17
@9      integer_cst      type: @18     int: 8
@10     void_type        name: @19      algn: 8
@11     tree_list        valu: @10
@12     identifier_node  strg: __static_initialization_and_destruction_0
                         lngt: 41
@13     function_type    size: @9       algn: 8        retn: @10
                         prms: @20
@14     function_decl    name: @21      type: @6       srcp: basic_string.tcc:229
                         note: artificial              chain: @22
                         lang: C        body: undefined
                         link: extern
@15     parm_decl        name: @23      type: @24      scpe: @7
                         srcp: H1_1.cpp:28             chain: @25
                         argt: @24      size: @26      algn: 32
                         used: 1
@16     cond_expr        type: @10      op 0: @27      op 1: @28
                         op 2: @29
@17     expr_stmt        type: @10      line: 93       expr: @30
@18     integer_type     name: @31      size: @32      algn: 128
                         prec: 128      sign: unsigned min : @33
                         max : @34
@19     type_decl        name: @35      type: @10      srcp: <built-in>:0
                         note: artificial
```

(f) 阶乘例程GENERIC输出

通过此例可以看到, GENERIC的输出实在是太太太大了, 这个程序共8000多行, 而且难以阅读. 值得一提的是, 似乎GCC并没有在GENERIC结果上进行优化, 虽然他们也认为这种树结构可能会适合某些特定的优化, 因此正在做这些工作.

5.2.3 Gimple化

GENERIC的结构较复杂, 而且不同语言的GENERIC表示结构不同, 所以随后, GENERIC结果被gimplifier递归地转化为**Gimple**, 通过设置中间变量, 它使所有指令都为三地址的, 并且像for、while等等所有控制指令都转为条件跳转, 单词作用域也取消. 这就使得语言相关、结构复杂的GENERIC结果变为了语言无关、通用的三地址指令序列[11], 这是一种较高层次的中间语言, 其目的在于明确代码的控制流和数据流.

[11]这有些像图灵机的定义, 能写能移能跳.

```
int main() ()
{
  int n.0;
  int n.1;
  struct basic_ostream & D.38872;
  int D.38873;

  {
    int i;
    int n;
    int f;

    try
      {
        std::basic_istream<char>::operator>> (&cin, &n);
        i = 2;
        f = 1;
        <D.38868>:
        n.0 = n;
        if (i > n.0) goto <D.35654>; else goto <D.38870>;
        <D.38870>:
        f = f * i;
        i = i + 1;
        goto <D.38868>;
        <D.35654>:
        n.1 = n;
        D.38872 = std::basic_ostream<char>::operator<< (&cout, n.1);
        std::basic_ostream<char>::operator<< (D.38872, endl);
        D.38873 = 0;
        return D.38873;
      }
    finally
      {
        n = {CLOBBER};
      }
  }
  D.38873 = 0;
  return D.38873;
}


void __static_initialization_and_destruction_0(int, int) (int __initialize_p, int __priority)
{
  if (__initialize_p == 1) goto <D.38879>; else goto <D.38880>;
  <D.38879>:
  if (__priority == 65535) goto <D.38881>; else goto <D.38882>;
  <D.38881>:
```

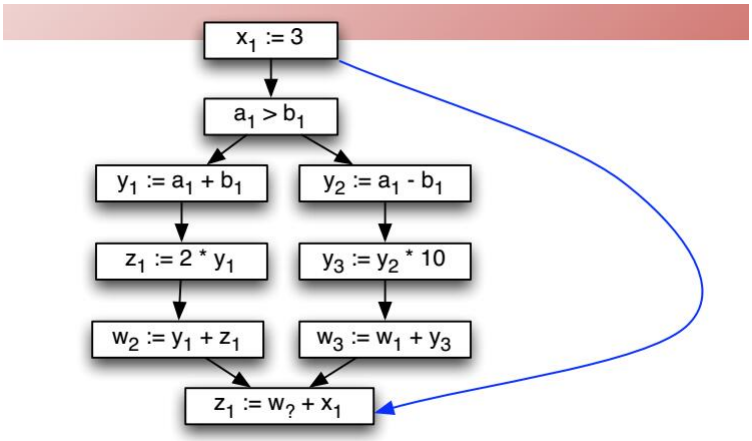
(g) 阶乘例程Gimple输出

可以看到, Gimple的输出与GENERIC相比十分接近C语言, 相当友好, 甚至能从中看出参数传递是引用还是赋值. 源程序的逻辑在try 块中, 可以看到全部都是三地址指令.

5.2.4 Gimple上的优化

Gimple的上述特性, 使得它可以被抽象为一个**控制流图(Control Flow Graph)**, 它的每个结点是一个**基本块(Basic Block)**, 简单来说就是一段逻辑上一定被严格顺序执行的指令序列[12], 图中的边描述了基本块之间的指令跳转. 这里也被叫做中端或者树优化器, 基于这种性质良好的逻辑结构, 编译器可以在Gimple 上进行一些机器无关的优化, 主要如下:

1. **SSA(Static Single Assignment, 静态单赋值)**: 所有变量只会被赋值一次, 原变量的不同赋值被拆分为不同变量, 以版本号区分. 对于影响变量值的控制流, 在流的汇点增加一个Φ函数以合流. 在CFG中从每个变量的赋值到引用的结点增加一条SSA边, 这使得在CFG 中追踪数据流十分简单.



(h) SSA示例

2. **别名分析(Alias Analysis)**: 当两个变量指向内存的同一地址时, 这两个变量是别名的. 别名分析帮助找到别名的变量, 基于此进行优化, 可以被认为是数据流的精简???

```
1                                   p.foo = 1;
2                                   q.foo = 2;
3                                   i = p.foo + 3;
```

[12]逻辑上是因为, 中断和异常可能使指令跳转.

如上例, 如我们知道p和q一定是别名的, 那么这段代码就可被优化为:

```
1                                   p.foo = 1;
2                                   q.foo = 2;
3                                   i = 5;
```

SSA为别名分析提供了相当大的帮助.

3. **内存模型(Memory Model)**: 内存模型给出了内存可用区域的记号, 对**基于类型的别名分析(Type-based Alias Analysis)** 有帮助. 鉴于GCC文档都懒得写了, 我也不写了[13].

```
;; Function int main() (main, funcdef_no=1516, decl_uid=35643, cgraph_uid=417, symbol_order=419)

Removing basic block 10
Merging blocks 8 and 9
;; 2 loops found
;;
;; Loop 0
;;  header 0, latch 1
;;  depth 0, outer -1
;;  nodes: 0 1 2 3 4 5 6 7 8 9 10
;;
;; Loop 1
;;  header 4, latch 5
;;  depth 1, outer 0
;;  nodes: 4 5
;; 2 succs { 10 3 }
;; 3 succs { 4 }
;; 4 succs { 6 5 }
;; 5 succs { 4 }
;; 6 succs { 10 7 }
;; 7 succs { 10 8 }
;; 8 succs { 9 }
;; 9 succs { 1 }
;; 10 succs { }
int main() ()
{
  struct basic_ostream & D.38875;
  int f;
  int n;
  int i;
  int D.38873;
  struct basic_ostream & D.38872;
  int n.1;
  int n.0;

  <bb 2>:
  std::basic_istream<char>::operator>> (&cin, &n);

  <bb 3>:
  i = 2;
  f = 1;

  <bb 4>:
  n.0 = n;
  if (i > n.0)
    goto <bb 6>;
  else
    goto <bb 5>;
```

(i) 阶乘例程CFG输出

[13]这是因为他不写, 我抄起来不方便

CFG图的结果输出在.cfg文件;可以看到GCC对CFG还进行了找圈、求深度等方法来进行一系列高深莫测的优化(或统计?). 直观地想, CFG能在宏观上给优化提供很好的帮助, 比如如果我们发现一个程序的CFG 是不连通的!

```
int main() ()
{
  struct basic_ostream & D.38875;
  int f;
  int n;
  int i;
  int D.38873;
  struct basic_ostream & D.38872;
  int n.1;
  int n.0;
  int n.0_8;
  int n.1_11;
  struct basic_ostream & _13;
  struct basic_ostream & _14;
  int _16;

  <bb 2>:
  # .MEM_5 = VDEF <.MEM_4(D)>
  std::basic_istream<char>::operator>> (&cin, &n);

  <bb 3>:
  i_6 = 2;
  f_7 = 1;

  <bb 4>:
  # i_1 = PHI <i_6(3), i_10(5)>
  # f_2 = PHI <f_7(3), f_9(5)>
  # VUSE <.MEM_5>
  n.0_8 = n;
  if (i_1 > n.0_8)
    goto <bb 6>;
  else
    goto <bb 5>;

  <bb 5>:
  f_9 = f_2 * i_1;
  i_10 = i_1 + 1;
```

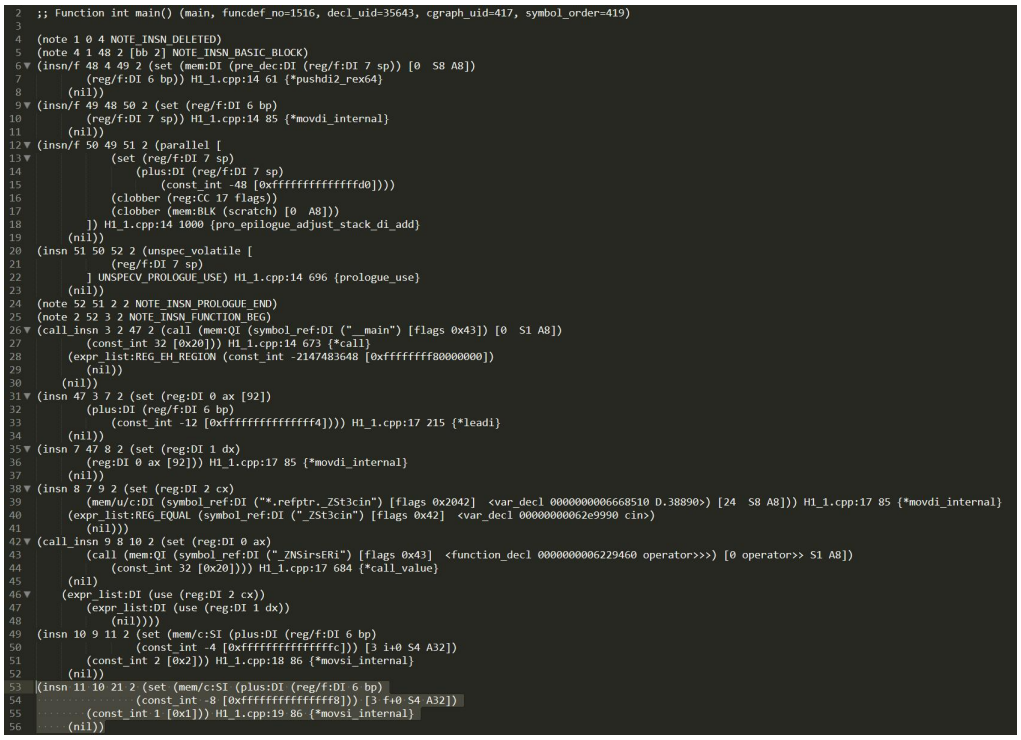
(j) 阶乘例程中端优化结果

用-vops参数可以得到含有**Memory SSA**信息的中端优化输出结果.fixup_cfg3. Memory SSA在用到内存的每条指令上方处增加了一个**虚指令**, 虚指令指示编译器由于**可能别名**, 这里**有可能**对变量进行了赋值(VDEF)或使用(VUSE), 据说这会很利于寄存器的分配.

5.2.5 RTL

在最后一部分, GCC将Gimple翻译为**RTL(Register Transfer Language)**, RTL是一种相对底层的中间语言, 它把源代码的每条指令几乎逐条译成一种代数形式. RTL的主要元素是**RTX(RTL Expression)**, 是一种**S-表达式(S-Expression)**, 使用**波兰记法(Polish Notation)**, 即操作符在前, 操作数括号在

后[14].



```
;; Function int main() (main, funcdef_no=1516, decl_uid=35643, cgraph_uid=417, symbol_order=419)

(note 1 0 4 NOTE_INSN_DELETED)
(note 4 1 48 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn/f 48 4 49 2 (set (mem:DI (pre_dec:DI (reg/f:DI 7 sp)) [0  S8 A8])
        (reg/f:DI 6 bp)) H1_1.cpp:14 61 {*pushdi2_rex64}
     (nil))
(insn/f 49 48 50 2 (set (reg/f:DI 6 bp)
        (reg/f:DI 7 sp)) H1_1.cpp:14 85 {*movdi_internal}
     (nil))
(insn/f 50 49 51 2 (parallel [
            (set (reg/f:DI 7 sp)
                (plus:DI (reg/f:DI 7 sp)
                    (const_int -48 [0xffffffffffffffd0])))
            (clobber (reg:CC 17 flags))
            (clobber (mem:BLK (scratch) [0  A8]))
        ]) H1_1.cpp:14 1000 {pro_epilogue_adjust_stack_di_add}
     (nil))
(insn 51 50 52 2 (unspec_volatile [
            (reg/f:DI 7 sp)
        ] UNSPECV_PROLOGUE_USE) H1_1.cpp:14 696 {prologue_use}
     (nil))
(note 52 51 2 2 NOTE_INSN_PROLOGUE_END)
(note 2 52 3 2 NOTE_INSN_FUNCTION_BEG)
(call_insn 3 2 47 2 (call (mem:QI (symbol_ref:DI ("__main") [flags 0x43]) [0  S1 A8])
        (const_int 32 [0x20])) H1_1.cpp:14 673 {*call}
     (expr_list:REG_EH_REGION (const_int -2147483648 [0xffffffff80000000])
        (nil))
    (nil))
(insn 47 3 7 2 (set (reg:DI 0 ax [92])
        (plus:DI (reg/f:DI 6 bp)
            (const_int -12 [0xfffffffffffffff4]))) H1_1.cpp:17 215 {*leadi}
     (nil))
(insn 7 47 8 2 (set (reg:DI 1 dx)
        (reg:DI 0 ax [92])) H1_1.cpp:17 85 {*movdi_internal}
     (nil))
(insn 8 7 9 2 (set (reg:DI 2 cx)
        (mem/u/c:DI (symbol_ref:DI ("*.refptr._ZSt3cin") [flags 0x2042]  <var_decl 0000000006668510 D.38890>) [24  S8 A8])) H1_1.cpp:17 85 {*movdi_internal}
     (expr_list:REG_EQUAL (symbol_ref:DI ("_ZSt3cin") [flags 0x42]  <var_decl 0000000006 2e9990 cin>)
        (nil)))
(call_insn 9 8 10 2 (set (reg:DI 0 ax)
        (call (mem:QI (symbol_ref:DI ("_ZNSirsERi") [flags 0x43]  <function_decl 0000000006229460 operator>>>) [0 operator>> S1 A8])
            (const_int 32 [0x20]))) H1_1.cpp:17 684 {*call_value}
     (nil)
    (expr_list:DI (use (reg:DI 2 cx))
        (expr_list:DI (use (reg:DI 1 dx))
            (nil))))
(insn 10 9 11 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
                (const_int -4 [0xfffffffffffffffc])) [3 i+0 S4 A32])
        (const_int 2 [0x2])) H1_1.cpp:18 86 {*movsi_internal}
     (nil))
(insn 11 10 21 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
                (const_int -8 [0xfffffffffffffff8])) [3 i+0 S4 A32])
        (const_int 1 [0x1])) H1_1.cpp:19 86 {*movsi_internal}
     (nil))
```

(k) 阶乘例程RTL输出

用-fdump-rtl-all参数可以得到在RTL上进行的各个阶段的中间输出, 上图为RTL阶段的最后结果. GCC用**insns**[15] 表示各个函数中的RTL指令序列, 它是一个双向链表; 用**insn**表示一个实际的RTL指令, 上图中我们看到的每一段都是一个insn.

insn的第0个操作数表示这个insn的类型, 常见的有:

- **insn**: 表示这是一条指令.
- **call_insn**: 类似insn, 但同时表示这条指令可能是一个函数调用. 这提醒此处可能会有无法预测的寄存器和内存改变.
- **jump_insn**: 类似insn, 但同时表示这条指令可能是一个跳转, 比如从一个函数调用中返回.
- **barrier**: 像字面意思, 表示控制流不能经过这里, 比如紧接着在一个无条件跳转指令后面.

[14] 也因此叫做前缀记法

[15] 我并没有找到这个全名是什么, 猜测是Instruction N* S* N* Sequence

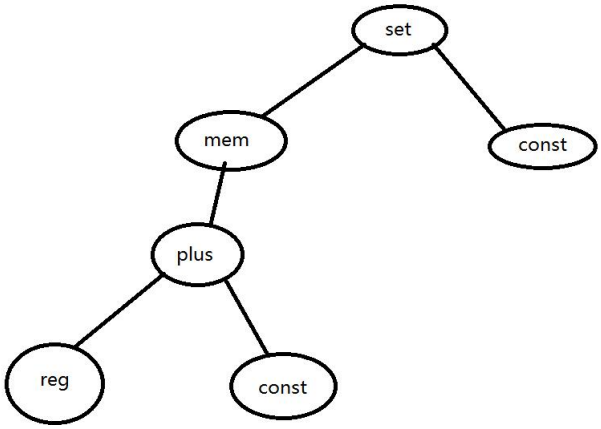
- **note**: 字面意思, 表示这是额外的调试和声明信息.

不同类型的insn后面几个操作数意义和结构不同, 以主要的insn类型为例: insn的第1个操作数指明这条insn的ID; 第2、3个操作数表示这条insn的前驱和后继的insn的id; 第4个操作数表示这条insn所在的基本块的id; insn第5个操作数表明这条指令的RTX; 后面未用的操作数记为nil.

以上图53行的insn为例, 它的RTX就是一个S-Expr的形式, 树是:

```
53  (insn 11 10 21 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
54                  (const_int -8 [0xfffffffffffffff8])) [3 f+0 S4 A32])
55          (const_int 1 [0x1])) H1_1.cpp:19 86 {*movsi_internal}
56      (nil))
```

(l) 上例insn



(m) 上例表达式树

这里我把具体的叶子操作数省略, 只留操作符, 可以看到这条指令描述的是一个对一个内存区域赋值的操作, 而且值得注意的是, reg操作中的寄存器是虚拟寄存器, 此时仍未完成实际的寄存器分配; 而mem 操作中的地址看起来很像重定向之前的虚拟地址了[16], 不知道是否会保留到链接环节; RTL中还有字长的概念, 这再一次表明了RTL已经是一种很接近汇编的中间语言了, 所以在RTL上的优化会比Gimple上的更偏向硬件, 至于具体的优化方法, 心累不写了.

5.2.6 机器表示

有了RTL, 其实再转为ASM已经很简单了, 直观地想, 因为RTX的良好格式, 只需针对RTX中的每个操作写出对应的ASM代码, 再基于宏去替换就好. 事实GCC的做法思路就是这样, 对于每个目标机器平台, 都有一个md文件指导对应的ASM代码生成. 具体就不介绍了.

[16]因为它很高.....

5.3 例程输出分析

说了这么多, GCC的优化还是很复杂, 看一下阶乘例程的无优化和3等级优化的输出:

```
11  main:
12  .LFB1516:
13      pushq   %rbp
14      .seh_pushreg    %rbp
15      movq    %rsp, %rbp
16      .seh_setframe   %rbp, 0
17      subq    $48, %rsp
18      .seh_stackalloc 48
19      .seh_endprologue
20      call    __main
21      leaq    -12(%rbp), %rax
22      movq    %rax, %rdx
23      movq    .refptr._ZSt3cin(%rip), %rcx
24      call    _ZNSirsERi
25      movl    $2, -4(%rbp)
26      movl    $1, -8(%rbp)
27  .L3:
28      movl    -12(%rbp), %eax
29      cmpl    %eax, -4(%rbp)
30      jg  .L2
31      movl    -8(%rbp), %eax
32      imull   -4(%rbp), %eax
33      movl    %eax, -8(%rbp)
34      addl    $1, -4(%rbp)
35      jmp .L3
36  .L2:
37      movl    -12(%rbp), %eax
38      movl    %eax, %edx
39      movq    .refptr._ZSt4cout(%rip), %rcx
40      call    _ZNSolsEi
41      movq    .refptr._ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_(%rip), %rdx
42      movq    %rax, %rcx
43      call    _ZNSolsEPFRSoS_E
44      movl    $0, %eax
45      addq    $48, %rsp
46      popq    %rbp
47      ret
```

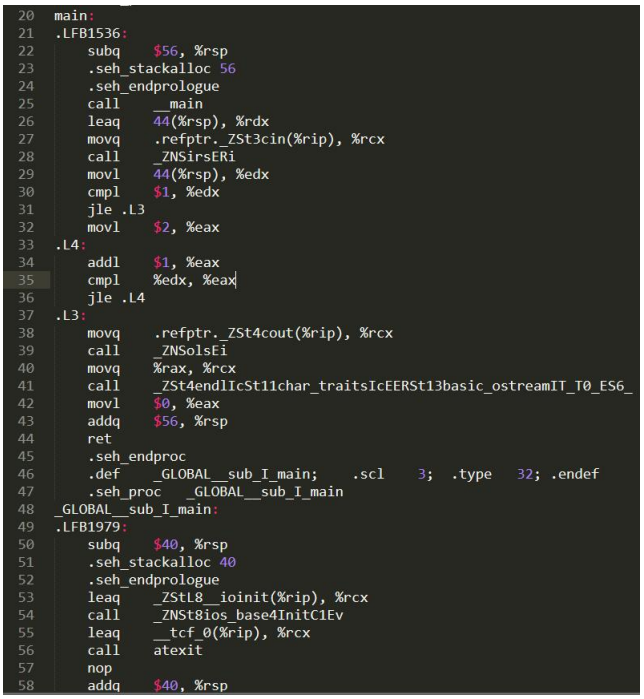
(n) 阶乘例程@O0

这个无优化的汇编代码还是很友好的, 先是函数调用的经典三板斧, 其中还顺便设置了异常栈, 再去call一个标准库里的cin代码, 那里会进行系统调用, 之前的edx和ecx也是为此设置的, 判断, 进入循环.....喜闻乐见的结果. 再看看O1的输出:

首先能看到它迫不及待地用初始值1先去和i比较, 算是体现了它考虑了字面常量; 再定睛一看, 这根本没算积啊! 仅仅是在数数! 一定是打开方式不对, 再看看O2的输出:

这连循环都优化没了! 这下就甭看O3了, 肯定也没了. 虽说有编译器能在编译期计算常量阶乘的江湖传闻, 但此例的阶乘依赖于运行时变量啊! 一开始我实在难以理解这个优化, 后来经同学提醒, 这里的原因在于源程序仅仅计算了阶乘f, 并没有后续. 考虑到前面提到的SSA 和数据流分析, 就不难想到, GCC可能用了某种回溯的方法, 从"有用的"[17]数据起, 在数据流中回溯找到所有影响的数据,

[17]怎么定义"有用"可能很难说全, 但起码要在生存期内被外部引用过一次吧, 这种函数栈上自生自灭的一定是没用的



```
main:
.LFB1536:
    subq    $56, %rsp
    .seh_stackalloc 56
    .seh_endprologue
    call    __main
    leaq    44(%rsp), %rdx
    movq    .refptr._ZSt3cin(%rip), %rcx
    call    _ZNSirsERi
    movl    44(%rsp), %edx
    cmpl    $1, %edx
    jle .L3
    movl    $2, %eax
.L4:
    addl    $1, %eax
    cmpl    %edx, %eax
    jle .L4
.L3:
    movq    .refptr._ZSt4cout(%rip), %rcx
    call    _ZNSolsEi
    movq    %rax, %rcx
    call    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    movl    $0, %eax
    addq    $56, %rsp
    ret
    .seh_endproc
    .def    _GLOBAL__sub_I_main;    .scl    3;  .type   32; .endef
    .seh_proc   _GLOBAL__sub_I_main
_GLOBAL__sub_I_main:
.LFB1979:
    subq    $40, %rsp
    .seh_stackalloc 40
    .seh_endprologue
    leaq    _ZStL8__ioinit(%rip), %rcx
    call    _ZNSt8ios_base4InitC1Ev
    leaq    __tcf_0(%rip), %rcx
    call    atexit
    nop
    addq    $40, %rsp
```

(o) 阶乘例程@O1

```
main:
.LFB1536:
    subq    $56, %rsp
    .seh_stackalloc 56
    .seh_endprologue
    call    __main
    movq    .refptr._ZSt3cin(%rip), %rcx
    leaq    44(%rsp), %rdx
    call    _ZNSirsERi
    movl    44(%rsp), %edx
    movq    .refptr._ZSt4cout(%rip), %rcx
    call    _ZNSolsEi
    movq    %rax, %rcx
    call    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    xorl    %eax, %eax
    addq    $56, %rsp
    ret
```

(p) 阶乘例程@O2

然后抛弃无关的分支. 事实上, GCC用了类似的叫**活跃变量分析()**的方法, 它是基于CFG的, 思路类似, 这里就不再赘述了. 总之, 将f定义为全局的, 就能看到它被正常的计算了:

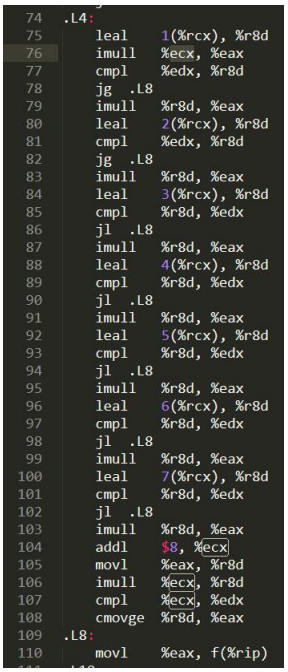
```
main:
.LFB1536:
    subq    $56, %rsp
    .seh_stackalloc 56
    .seh_endprologue
    call    __main
    movq    .refptr._ZSt3cin(%rip), %rcx
    leaq    44(%rsp), %rdx
    call    _ZNSirsERi
    movl    44(%rsp), %edx
    movl    $1, %ecx
    movl    $1, f(%rip)
    movl    $2, %eax
    leal    1(%rdx), %r8d
    cmpl    $1, %edx
    jle .L5
    .p2align 4,,10
.L6:
    imull   %eax, %ecx
    addl    $1, %eax
    cmpl    %eax, %r8d
    jne .L6
    movl    %ecx, f(%rip)
.L5:
    movq    .refptr._ZSt4cout(%rip), %rcx
    call    _ZNSolsEi
    movq    %rax, %rcx
    call    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    xorl    %eax, %eax
    addq    $56, %rsp
    ret
```

(q) 阶乘例程@O2魔改

可以看到全局的f存在了一个特定的数据段里. 并且似乎O1和O2代码差不多, 只是阶乘上限n从eax移到了r8d中, 而这两个寄存器都是32位的, 似乎也没什么区别[18]. 可能是因为这段程序很简单, 没什么优化的空间.

仔细观察可以发现一个很热的冗余操作: 每次循环都把阶乘积从ecx移到内存中. 由于这个操作很热, 所以这可能会成为O3优化的眼中钉:

[18] 一个可能的解释是eax比r8d更通用.



```
.L4:
    leal    1(%rcx), %r8d
    imull   %ecx, %eax
    cmpl    %edx, %r8d
    jg  .L8
    imull   %r8d, %eax
    leal    2(%rcx), %r8d
    cmpl    %edx, %r8d
    jg  .L8
    imull   %r8d, %eax
    leal    3(%rcx), %r8d
    cmpl    %r8d, %edx
    jl  .L8
    imull   %r8d, %eax
    leal    4(%rcx), %r8d
    cmpl    %r8d, %edx
    jl  .L8
    imull   %r8d, %eax
    leal    5(%rcx), %r8d
    cmpl    %r8d, %edx
    jl  .L8
    imull   %r8d, %eax
    leal    6(%rcx), %r8d
    cmpl    %r8d, %edx
    jl  .L8
    imull   %r8d, %eax
    leal    7(%rcx), %r8d
    cmpl    %r8d, %edx
    jl  .L8
    imull   %r8d, %eax
    addl    $8, %ecx
    movl    %eax, %r8d
    imull   %ecx, %r8d
    cmpl    %ecx, %edx
    cmovge  %r8d, %eax
.L8:
    movl    %eax, f(%rip)
```

(r) 阶乘例程@O3

O3优化是非常激进的, 它为了提高运行速度能做任何事. 可以看到O3优化后的代码基本牺牲了可读性, 代码长度明显增加, 指令也很新潮. 果然O3 的代码并没有每次都将阶乘的中间结果从寄存器中移到内存, 而是硬编码了8次循环, 之后进行一次转移; 用leal指令和基址寻址替换了add 做计数, 逻辑上是更绕了, 也许它认为这么做有利于指令并行?

但是一个重要的问题是, O3优化后的代码几乎每8次再flush进内存, 这么做虽然符合我们最终的结果, 但是从某种角度说, 这是不符合源程序的逻辑的, 因为通常编程时, 我们是期望变量在内存中的. 而且这会减少程序的鲁棒性, 虽然大部分人也不会相信内存, 而是会选择手动从内存flush到硬盘, 但对于能信任内存的环境下, 这种优化就显得拖累了, 这或许是GCC一般推荐O2的原因之一吧[19].

此外, GCC还有Os等等优化选项, 这些预设的选项从目标代码运行速度之外的别的角度去实现优化, 比如Os还限制目标代码长度, 这里不再赘述.

5.4 总结

编译器作为GCC的关键部件, 而优化部分更是核心, 非常复杂, 奇技淫巧极多, 文档不太全. 它先进行语法分析和语义分析, 生成GENERIC 作为结果. 然后转

[19] 更为主要的原因是O3的优化算法试图猜测程序走向, 风险很大, 而且编译时很依赖具体环境

换为Gimple和RTL, 分别在高层和底层进行优化, 其中最重要的手段是分析数据流和控制流.

根据不同的场景, 用户对目标程序的期望也不同, 比如目标代码运行速度、大小、编译时间等等, 这里就涉及到对优化手段的权衡; 同时, 过于激进的优化可能使目标代码丧失鲁棒性, 出现意料之外的错误.

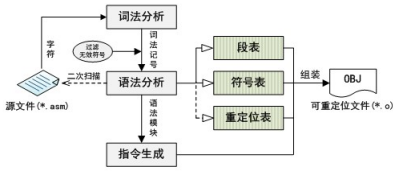
6 汇编器

6.1 简介

汇编器将编译器输出的汇编代码翻译为二进制的**目标文件(Object File)**. 汇编器并不是简单的解释器, 本质上讲它也是编译器. 生成的目标文件因为要考虑到下一步的链接和后续OS 的装载、执行, 所以要有严格的约定好的格式, 来描述数据的意义(元数据), 这主要由汇编器解析的汇编指令来实现. 目标文件是编译器和OS 的中间层, 这部分以后的结果与OS 环境有关, 并且考虑到Linux 下的工具比较多一点, 于是从此本实验环境为Ubuntu 14, 介绍也以Ubuntu 的结果为主, 但Windows 下的技术思路是类似的. 这部分大部分工作, 比如建立段表, 的目的都是为链接做准备, 但这个章节尽量不涉及链接.

6.2 工作流程

6.2.1 流程图



(s) 汇编器工作流程图

6.2.2 构建符号表和段表

20

汇编语言支持先引用后定义, 这需要对汇编代码进行两次扫描. 第一次记录每条指令在段内的偏移地址, 将所有定义的符号以及相关信息与它的偏移地址对应, 就构成了**符号表(Symbol Table)**; 将所有的段以及相关信息记录下来, 构成了**段表(Section Table)**[21].

[20]因为汇编器的词法分析与编译器类似, 并且比较简单. 故重点介绍语法分析.
[21]section翻译为段是不是会和segment产生混淆?

这一遍扫描其实比较简单, 只要根据一些标记, 比如EQU表达式、外部符号等等来维护一个当前扫描的地址指针即可. 最终得到的段表包含段名、段的偏移和大小; 符号表包含的信息要多一些, 比如符号名、符号大小、所属段名、段内偏移等等. 这些都是为了以后重定向所准备的, 目的在于用更"客观"的信息描述定义的符号, 例程的符号表读取如下[22]:

```
grrr@ubuntu:~/Documents$ objdump -t test.o

test.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000004 l     O .bss   0000000000000001 _ZStL8__ioinit
000000000000008e l     F .text  000000000000003e _Z41__static_initialization_and_destruc
tion_0ii
00000000000000cc l     F .text  0000000000000015 _GLOBAL__sub_I_f
0000000000000000 l    d  .init_array    0000000000000000 .init_array
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     O .bss   0000000000000004 f
000000000000008e g     F .text  000000000000008e main
0000000000000000         *UND*  0000000000000000 _ZSt3cin
0000000000000000         *UND*  0000000000000000 _ZNSirsERi
0000000000000000         *UND*  0000000000000000 _ZSt4cout
0000000000000000         *UND*  0000000000000000 _ZNSolsEi
0000000000000000         *UND*  0000000000000000 _ZSt4endlIcSt11char_traitsIcEERSt13basi
c_ostreamIT_T0_ES6_
0000000000000000         *UND*  0000000000000000 _ZNSolsEPFRSoS_E
0000000000000000         *UND*  0000000000000000 __stack_chk_fail
0000000000000000         *UND*  0000000000000000 _ZNSt8ios_base4InitC1Ev
0000000000000000         *UND*  0000000000000000 .hidden __dso_handle
0000000000000000         *UND*  0000000000000000 _ZNSt8ios_base4InitD1Ev
0000000000000000         *UND*  0000000000000000 __cxa_atexit
```

(t) 阶乘例程符号表

它的每列依次是:

1. 该符号在段内的偏移.
2. 符号作用域, l表local, g表global.
3. 描述符号的用途, f是file, F是function, d可能是default.
4. 符号类型, 如常见的bss、text、data, UND表示未定义的符号, ABS表示绝对非可重定位符号.
5. 符号大小, 未定义符号为0
6. 符号名

6.2.3 构建重定位表

重定位的动机是, 由于目标文件往往需要和别的目标文件链接到一起, 就无法得知符号链接后的绝对地址, 对符号的绝对引用就需重定向; 或者符号未定义就引用. 等等原因导致一些符号的引用必须要进行后续的地址修正[23], 而重定位表就是记录这些符号引用的表.

[22]确切的说, 这是目标文件中的符号表.
[23]所以重定位项也叫做fixup.

汇编器进行第二次扫描, 遇到每个符号的引用时, 根据已有的符号表和引用的方式, 来决定是否需要进行重定位, 比如未定义的符号需要重定位、使用绝对地址的引用需要重定位等等, 例程的重定向表读取如下[24]:

```
grrr@ubuntu:~/Documents$ readelf -r test.o

Relocation section '.rela.text' at offset 0x568 contains 16 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000001f  000e0000000a R_X86_64_32       0000000000000000 _ZSt3cin + 0
000000000024  000f00000002 R_X86_64_PC32     0000000000000000 _ZNSirsERi - 4
000000000031  000c00000002 R_X86_64_PC32     0000000000000000 f - 8
000000000043  000c00000002 R_X86_64_PC32     0000000000000000 f - 4
00000000004d  000c00000002 R_X86_64_PC32     0000000000000000 f - 4
00000000005d  00100000000a R_X86_64_32       0000000000000000 _ZSt4cout + 0
000000000062  001100000002 R_X86_64_PC32     0000000000000000 _ZNSolsEi - 4
000000000067  00120000000a R_X86_64_32       0000000000000000 _ZSt4endlIcSt11char_tr + 0
00000000006f  001300000002 R_X86_64_PC32     0000000000000000 _ZNSolsEPFRSoS_E - 4
000000000088  001400000002 R_X86_64_PC32     0000000000000000 __stack_chk_fail - 4
0000000000ac  00040000000a R_X86_64_32       0000000000000000 .bss + 4
0000000000b1  001500000002 R_X86_64_PC32     0000000000000000 _ZNSt8ios_base4InitC1E - 4
0000000000b6  00160000000a R_X86_64_32       0000000000000000 __dso_handle + 0
0000000000bb  00040000000a R_X86_64_32       0000000000000000 .bss + 4
0000000000c0  00170000000a R_X86_64_32       0000000000000000 _ZNSt8ios_base4InitD1E + 0
0000000000c5  001800000002 R_X86_64_PC32     0000000000000000 __cxa_atexit - 4

Relocation section '.rela.init_array' at offset 0x6e8 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000200000001 R_X86_64_64       0000000000000000 .text + cc

Relocation section '.rela.eh_frame' at offset 0x700 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
000000000040  000200000002 R_X86_64_PC32     0000000000000000 .text + 8e
000000000060  000200000002 R_X86_64_PC32     0000000000000000 .text + cc
```

(u) 阶乘例程重定向表

它的每列依次是:

1. 段内偏移.
2. 高四位表示该符号在符号表中的索引; 低八位表示重定位的类型, 与Type列一致.
3. 重定位类型, 之后在链接部分介绍.
4. 符号大小.

这里可能有人会问: 既然我们的OS和处理器提供了分段机制, 为什么还要对符号重定位呢? 直接切换段不就好了吗? 听起来很爽, 但如果这可行, 那么会导致程序在运行时需要反复在代码段和数据段之间切换, 并且段的切换开销不小, 这会使运行效率大大降低; 其次因为操作复杂和效率等原因, 像Unix这样的OS实际上并不使用分段机制来寻址, 而是从0开始安排虚拟地址, 只是用分段机制进行权限管理, 所以不可行.

6.2.4 二进制代码

汇编器在第二次扫描的同时生成二进制指令, 具体的翻译与机器有关, 这里不赘述. 值得一提的是翻译后的指令有些会在接下来的重定位里进行操作数的修正.

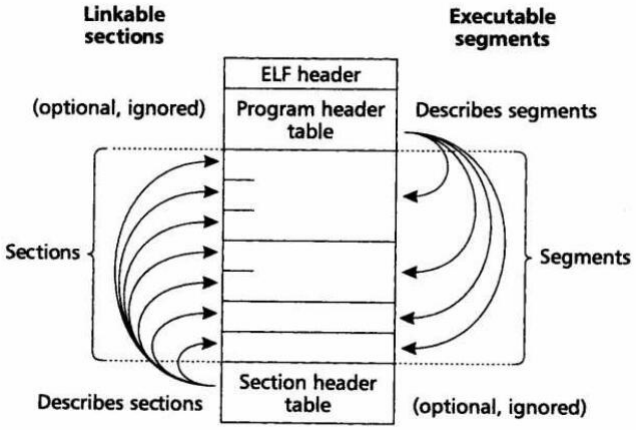
[24]确切的说,这是目标文件中的重定向信息,是重定向表加工后的.

6.2.5 组装目标文件

有了前面的准备工作和结果, 接下来只要按照约定的格式, 将结果组装为目标文件即可, 目标文件的介绍见下节.

6.3 目标文件

Linux下的目标文件是**ELF(Executable Linkable File)**格式的, 它的通用结构如图.



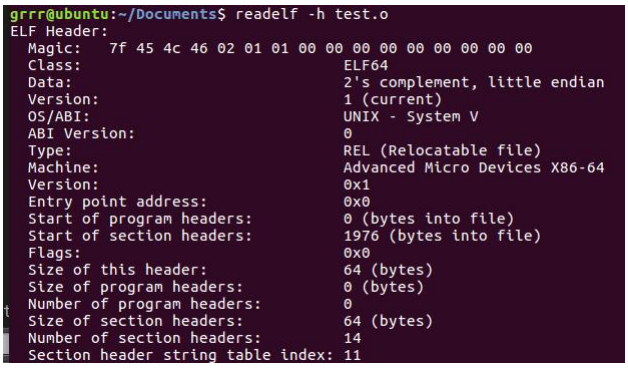
(v) ELF结构

图中给出了ELF文件的两种视角: 链接视角和执行视角, 链接视角认为ELF文件由一个ELF头、一个段表和它描述的一些段构成; 执行视角认为ELF文件由一个ELF头、一个程序头表和它描述的一些程序段构成. 但并不是所有ELF 文件都可执行和链接, ELF 文件有三个子类:

1. **可执行目标文件**: 人如其名, 可执行, 链接视角被忽略或者没有.
2. **可重定位目标文件**: 人如其名, 可链接, 执行视角被忽略或者没有.
3. **可共享目标文件** 能执行能链接, 可以从执行和链接两个视角识别文件, 通常是动态库.

GCC生成的目标文件通常是可重定位目标文件, 例程的ELF头读取如下图:

其中**Magic Number**实际上起到了类型标识的作用, 它告知OS这是一个ELF文件. 其余的项已经给出了描述, 如上所述, 这不是一个可执行的目标文件, 于是没有程序头和段. 接下来读取例程的区段表如下:



(w) 阶乘例程ELF头

其中常见的是text段即为代码段, data段即为数据段, bss段为未初始化的全局变量, 通常也会有rodata段即只读数据段, 这些段属于源程序, 会被真正加载到内存中; 上一步获得的符号表信息存在了SYMTAB 类型的段里, 重定位信息存在了RELA 类型的段里, 这些段是用于指导链接和加载的, 不会被加载到内存中.

6.4 总结

汇编器将汇编代码翻译并组织为可重定向的目标文件, 将原代码划分为各个段, 其中有符号表以及有关重定位信息的表, 这都是为了后续的链接和加载所做的准备工作.

7 链接器

7.1 简介

链接器可以将一些可重定位目标文件和共享目标文件组织成一个可执行目标文件或者共享目标文件, 对于GCC的最后阶段, 链接器通常只将可重定位的目标文件组织为可执行的目标文件. 链接器主要做了两件事, 一是在所有的输入模块中识别符号; 二是为这些符号分配虚拟地址, 修改它们的引用.

相比其他的工具, 链接器较为复杂, 与OS 关系更为密切, 技术性也更强. 除了在此处, 链接过程也可以在加载时或者运行时完成, 运行时链接也通常叫动态链接. 这章介绍的内容大部分是简单的原始情况, 比如没考虑有关异常和调试等等的段.

7.2 工作流程

7.2.1 符号解析

首先链接器需要对所有符号进行解析, 符号分为两种:

1. **强符号**: 对于C来说, 指函数或者初始化了的全局变量.

```
grrr@ubuntu:~/Documents$ readelf -S test.o
There are 14 section headers, starting at offset 0x7b8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       00000000000000e1  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000568
       0000000000000180  0000000000000018   I      12     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000121
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  00000124
       0000000000000005  0000000000000000  WA       0     0     4
  [ 5] .init_array       INIT_ARRAY       0000000000000000  00000128
       0000000000000008  0000000000000000  WA       0     0     8
  [ 6] .rela.init_array  RELA             0000000000000000  000006e8
       0000000000000018  0000000000000018   I      12     5     8
  [ 7] .comment          PROGBITS         0000000000000000  00000130
       0000000000000035  0000000000000001  MS       0     0     1
  [ 8] .note.GNU-stack   PROGBITS         0000000000000000  00000165
       0000000000000000  0000000000000000           0     0     1
  [ 9] .eh_frame         PROGBITS         0000000000000000  00000168
       0000000000000078  0000000000000000   A       0     0     8
  [10] .rela.eh_frame    RELA             0000000000000000  00000700
       0000000000000048  0000000000000018   I      12     9     8
  [11] .shstrtab         STRTAB           0000000000000000  00000748
       000000000000006a  0000000000000000           0     0     1
  [12] .symtab           SYMTAB           0000000000000000  000001e0
       0000000000000258  0000000000000018          13    12     8
  [13] .strtab           STRTAB           0000000000000000  00000438
       000000000000012e  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

(x) 阶乘例程区段头表

2. **弱符号**: 对于C来说, 指未初始化的全局变量.

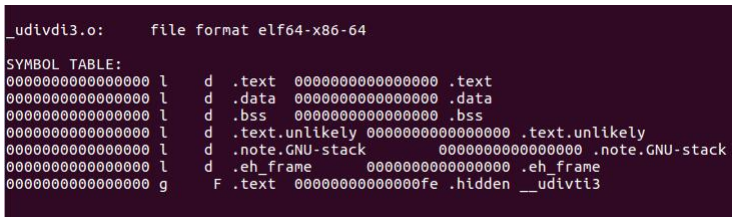
接下来还要涉及到静态库文件, 它的动机就是提供库... 库可以被看作打包的目标文件构成的归档文件, 在链接时作为输入参数. 这样在初始时就有了一些目标文件和一些库文件作为备选文件, 同时还要维护3 个链表: 所有实际参与链接的文件F、所有未定义符号U、所有已定义符号D. 接下来链接器的算法是一个循环过程:

每次迭代, 链接器按顺序检索每个备选文件, 如果它是目标文件, 则将它放入F, 同时检索它的符号表, 将其中的符号按定义分别放入U和D; 如果它是库文件, 依然检索它的符号表, 判断其中是否定义了U中的符号, 如果有, 则把它放入F, 并且将它的符号按定义放入U和D; 如果没有, 就跳过该文件.

迭代的终止条件是一个强符号被放入D两次, 这时链接失败, 出现了重复定义的错误; 或者一次迭代后三个链表不发生改变, 如果此时U不为空, 那么链接失败, 有符号未找到定义; 如果U为空, 那么链接成功, 此时所有的符号都找到了定义, F中的文件将会被组织为可执行文件.

值得一提的是, 这里就能解释头文件与静态库文件之间的关系了. 在预处理阶段, 头文件会以文本的形式包含进来, 只声明符号, 而不定义. 在链接阶段, 链接器在指定的一些库文件中搜索这些未定义的符号, 最终在对应的库文件里找到

定义, 加入进来. 这样做的好处一是无需每次重新编译方法的实现, 提高效率; 二是可以不公开源码. 实际的GCC静态库文件符号表部分如下:



(y) gcclib.a符号表节选

如上所说, 这是库文件包含的许多目标文件之一, 它似乎是不同类型整数和浮点数之间的除法运算函数. 它只有一个全局符号, 就是那个函数[25].

7.2.2 段重定位

既然到此为止已经确保每个符号引用都是已定义的了, 那么接下来就该重定位这些引用. 首先, 链接器将所有目标文件的对应段合并, 具体地说, 所有text段、data段、bss段合并成三个大段, 并且记录每个目标文件这三个段基址修正后的偏移, 分别记为TR、DR、BR, 比如一个目标文件原text段基址是a, 段合并后在新的大text段中段基址是b, 那么它的TR = b - a. 这里我们只考虑这三个段会被加载; 在更为复杂的情况下, 还会有有关异常、调试、动态链接等等的段被合并.

7.2.3 数据重定位

重定位作为数据的指针时, 情况比较简单, 只要将所引用的符号地址加上它所属的段的基址偏移就可以. 如text段中有一个对data段的数据引用, 那么只要在这个符号的地址上加DR即可.

7.2.4 指令重定位

以X86指令为例, 段内的call和近jump指令使用段内相对地址, 所以不需要重定位; 段之间的长jump指令需要考虑两个段的基址偏移, 比如从代码段跳转到数据段的指令, 段间相对地址要加上DR - TR.

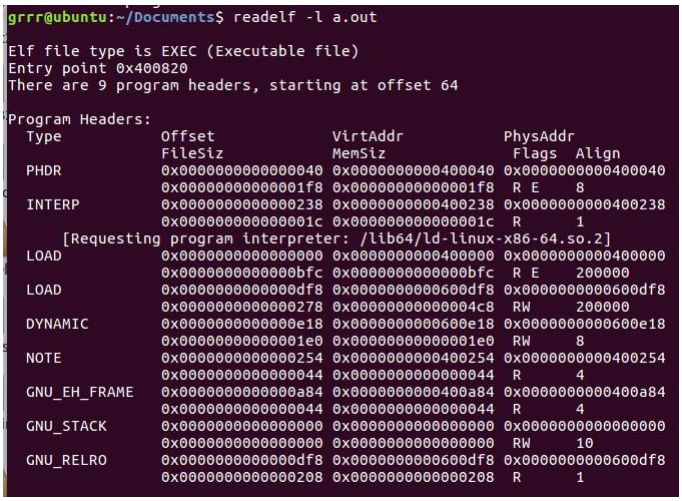
7.3 实际的链接

正如上面一再强调的, 实际的链接技术十分复杂. 首先, 使用动态与静态混合链接. 动机是像上述的静态链接方式会使得所有程序在内存中都拥有一份单独的库程序, 而有一些库是会被大量引用的, 这显然开销过大. 为了克服这点, 动态链接编译时只提供符号表保证所有符号都有定义, 并且库在内存中只有一个镜

[25]去年做作业的时候遇到过因32/64位版本问题GCC未找到此函数符号, 花了好久才解决, 因此对这个函数怀恨在心.

像, 在程序加载完成, 运行时, 遇到了库中的实现, 则由动态链接器找到对应的库, 并将它映射到程序的地址空间中[26].

此外, 由于处理器指令的复杂, 实际中的链接器处理重定位也不简单, 具体可见参考.



(z) 阶乘例程可执行文件程序头表

最终, 一个可执行文件指明了load段为要加载到内存的段, 并且在elf头中指明了入口地址. 这样加载器按此加载, 并将PC指向入口地址, 程序就得以运行. 注意的是其中有一段注明了需要链接解释器来动态链接.

7.4 总结

链接器将可重定位目标文件组织为可执行的目标文件, 目标文件中的符号都会找到定义, 所有引用符号的地址都被修正, 最后程序被组织为一些程序段, 一个程序头表来说明这些段, 一个ELF头说明整个文件. 好复杂.

8 总结

编译过程是这门课程的重点, 可以粗略的分为两部分, 以Gimple阶段为界. 前端与机器无关, 是在程序抽象的逻辑层面进行优化; 后端与机器有关, 在具体的指令集和处理器架构方面做并行化等优化. 作为本门课程的非重点, 汇编、链接和加载过程密切相关, 既涉及到具体的处理器架构, 又涉及到OS. 与前面的编译过程相比, 编译更理论化, 这部分更技术性, 二者同样都是十分复杂的过程.

[26]我觉得COW技术会使动态链接不仅限于库函数, 甚至数据也可以.

经过本次调研, 本人深刻认识到了编译过程的复杂. 稍微有了一点重视自己写的每一行代码的念头. 另外似乎这个报告用英文写更方便一点, 如果我在开头发现这点就好了.

References

[1] GCC Preprocessor Document: Output
https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html

[2] GCC Preprocessor Document: Overview
https://gcc.gnu.org/onlinedocs/gcc-7.2.0/cpp/Overview.html

[3] C Phase of Translation
https://en.wikichip.org/wiki/c/phases_of_translation

[4] GCC源码分析—SonicLing的博客
http://blog.csdn.net/sonicling/article/details/6706152

[5] Harvard CS252r Lec04 Slide

[6] Wikipedia Alias Analysis
https://en.wikipedia.org/wiki/Alias_analysis

[7] http://blog.csdn.net/u012491514/article/details/25000519

[8] https://en.wikipedia.org/wiki/S-expression

[9] https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

[10] http://blog.csdn.net/kmesg/article/details/6448042

[11] http://blog.csdn.net/freestyle4568world/article/details/49817799

[12] https://wenku.baidu.com/view/04090ebbeefdc8d376ee32bf.html

[13] http://blog.csdn.net/zdy0_2004/article/details/52333175

[14] 《Linkers and Loaders》

[15] http://www.cnblogs.com/fengyv/p/3775992.html

[16] http://www.cnblogs.com/xiaomanon/p/4210016.html