Program optimization & Linking I

朱家启 徐梓越 许珈铭

2023.11.15

Optimization (CS:APP Ch. 5)

徐梓越

Content

- 前情提要：如何编写编译器友好的程序
- 如何利用并行性

  Modern process design 处理器硬件

  Data independent 数据的独立性
- SIMD

优化的本质

	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

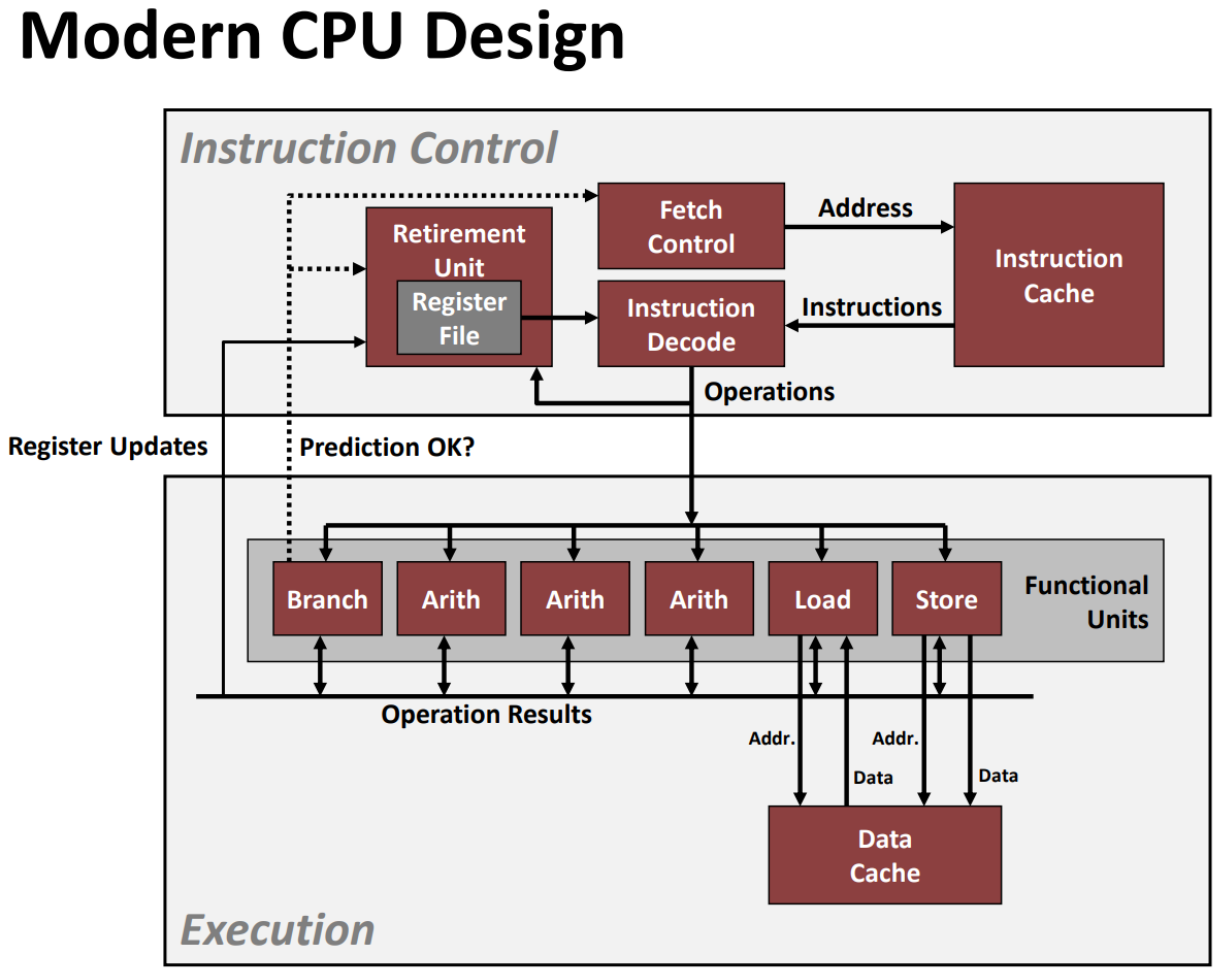
- CPE 每元素的周期数
- Basic Optimization：减少重复性
- 并行性：当处理一串数据时，我们希望能像pipeline，同时处理多个数据

这对程序本身（数据依赖）和硬件（功能单元）有要求

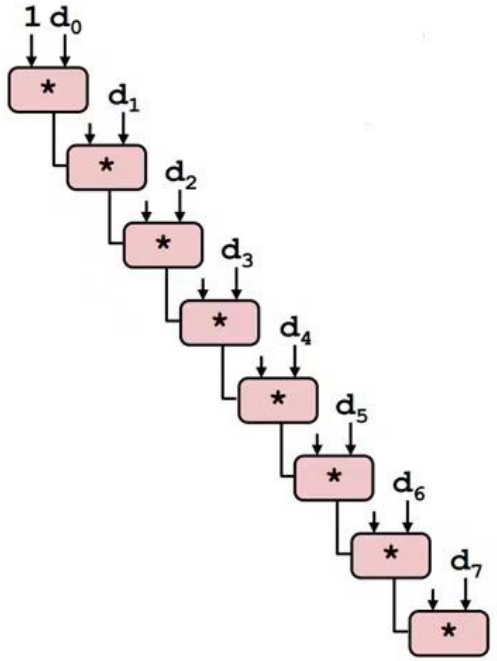
例：a+=1, a*=3, a+=1, a*=3…… **vs** a+=b[1], a+=b[2], a+=b[3]……

Hardware

- 相比于一次读取一条指令，采用超标量乱序执行（Superscalar）
- CPU尽可能读取多的指令序列
- 因为指令之间不存在互相依赖，因此可以并行。



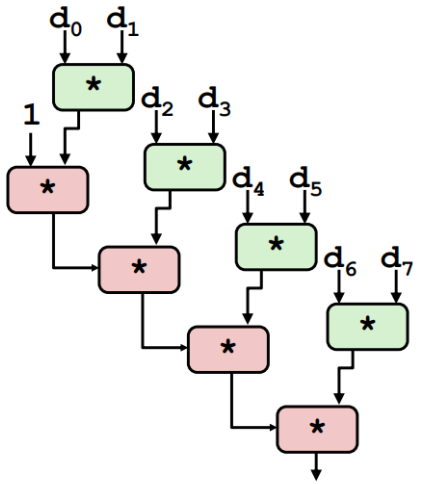
Loop Unrolling 2x1



循环代码，需要依顺序运行

Loop Unrolling 2x1a

```
x = x OP (d[i] OP d[i+1]);
```

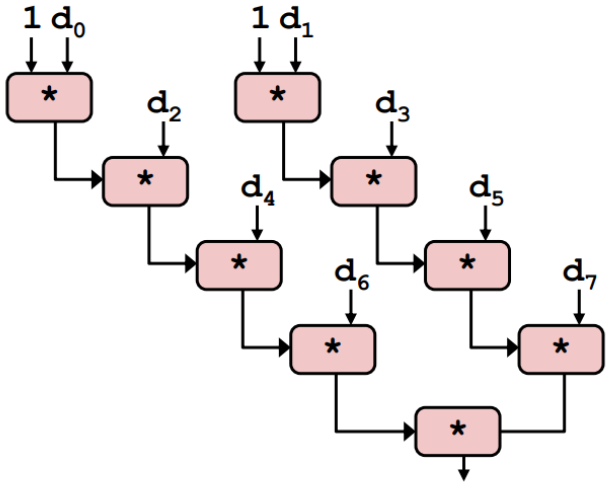


用结合律打破顺序，减少x OP的次数，实现优化

而**浮点型**？？

Loop Unrolling 2x2

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



借用2个functional unit来实现打乱顺序，多条并行

Unrolling and Accumulation

- a. 循环展开的问题：内循环中有多少op a op d[i] op… 展开L个
- b.有多少接口来存计算结果，并行a1，a2… K个（数据以来关系小）
- 以K为因子展开一个长度为L的数组

Limitations

- Add units, load units limited 展开多后，对访存器的压力（代码长度增加）
- Large overhead for short length 对于n大小的预测，倘若对于较小的n可能作用相反

5.11 限制因素

Updating of accumulator acc0 in 20 x 20 unrolling

```
vmovsd  40(%rsp), %xmm0
vmulsd  (%rdx), %xmm0, %xmm0
vmovsd  %xmm0, 40(%rsp)
```

- 寄存器溢出：

如果并行度超过了寄存器数量，就会导致寄存器溢出，编译器会将部分临 时值放到内存，比如栈，这会带来额外开销 （超出16个寄存器）

- 分支预测错误处罚：

当分支预测错误后需要重新填充流水线而增加的时钟周期。对于分支预测的可预测性很好的，处罚可以大致忽略。对于可预测性差的，可以改用更容易被转化为条件数据传送的代码。

- 解决办法：

1.不要过分关心可预测的分支

```
/* Rearrange two vectors so that for each i, b[i] >= a[i] */
void minmax1(long a[], long b[], long n) {
    long i;
    for (i = 0; i < n; i++) {
        if (a[i] > b[i]) {
            long t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

```
/* Rearrange two vectors so that for each i, b[i] >= a[i] */
void minmax2(long a[], long b[], long n) {
    long i;
    for (i = 0; i < n; i++) {
        long min = a[i] < b[i] ? a[i] : b[i];
        long max = a[i] < b[i] ? b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

2.书写适合用条件传送实现的代码（条件数据传送而不是条件控制转移）

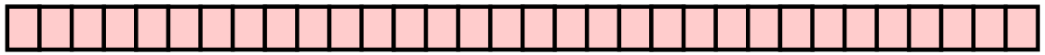
Single Instruction Multiple Data

Advanced Vector Extensions（AVX）

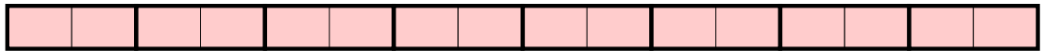
Programming with AVX2

YMM Registers

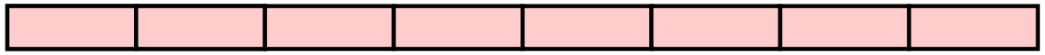
- 16 total, each 32 bytes
- 32 single-byte integers



- 16 16-bit integers

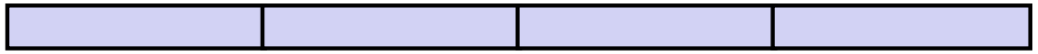


- 8 32-bit integers



- 8 single-precision floats

- 4 double-precision floats



- 1 single-precision float

- 1 double-precision float

对于本来一个寄存器可以8字节

X86机器上有 %xmm 16字节

现在新一代CPU上有 %ymm 32字节

Intel AVX512 64字节

Single Instruction Multiple Data 三种变体

- Vector architectures
  - 更容易被理解和编译的SIMD cons: 昂贵，贵在晶体管和DRAM的带宽（bandwidth）
- Multimedia SIMD instruction set extensions
  - 多媒体应用 运用AVX
- Graphics processing units (GPUs)
  - 系统处理器、系统内存、图像内存（system processor, system memory, graphic memory）（和Vector architecture区分）

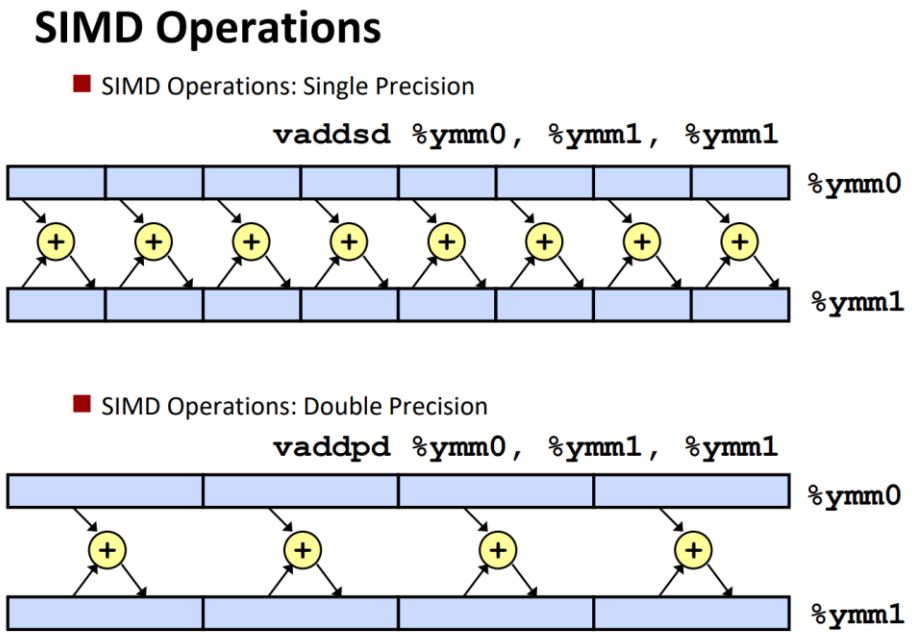
Double 为例

add src1,des1

add src2,des2

add src3,des3

add src4,des4



vaddpd %ymm0, %ymm1, %ymm1

vmulps (%rcx), %ymm0, %ymm1

- %ymm0 a0,a1,…,a7
- %rcx 储存着内存中的地址 指向一串8个单精浮点数 b0,b1…,b7
- ci←ai*bi 存入%ymm1

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	*0.50*	*3.00*	*3.00*	*5.00*
Throughput Bound	*0.50*	*1.00*	*1.00*	*0.50*
Vec Throughput Bound	*0.06*	*0.12*	*0.25*	*0.12*

- 应用于视频、音频、图像处理等方向
- Intel的编译器可以实现这样的优化，但GCC的编译器暂时不行，除非对GCC进行扩展编译

Linking I (CS:APP Ch. 7.1-7.7)

朱家启

- Linker工作概览
- ELF结构
- 符号解析
- 重定位
- 可执行目标文件及其加载

Linker工作概览

(a) `main.c` — *code/link/main.c*

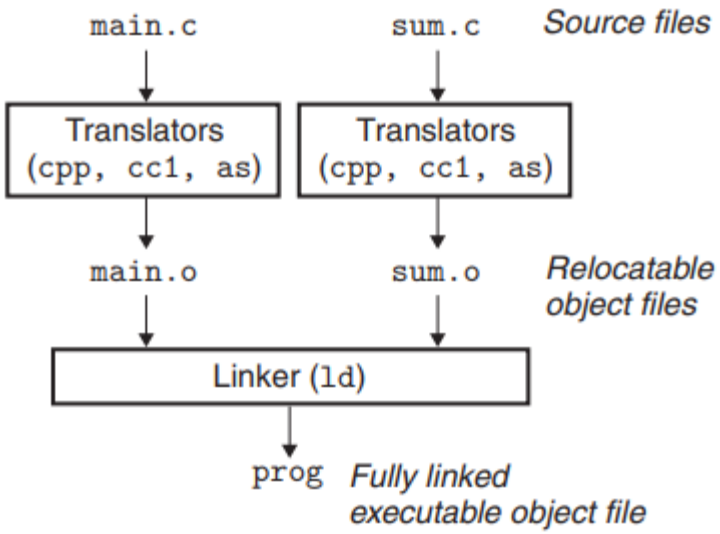
```
1   int sum(int *a, int n);
2
3   int array[2] = {1, 2};
4
5   int main()
6   {
7       int val = sum(array, 2);
8       return val;
9   }
```

code/link/main.c

(b) `sum.c` — *code/link/sum.c*

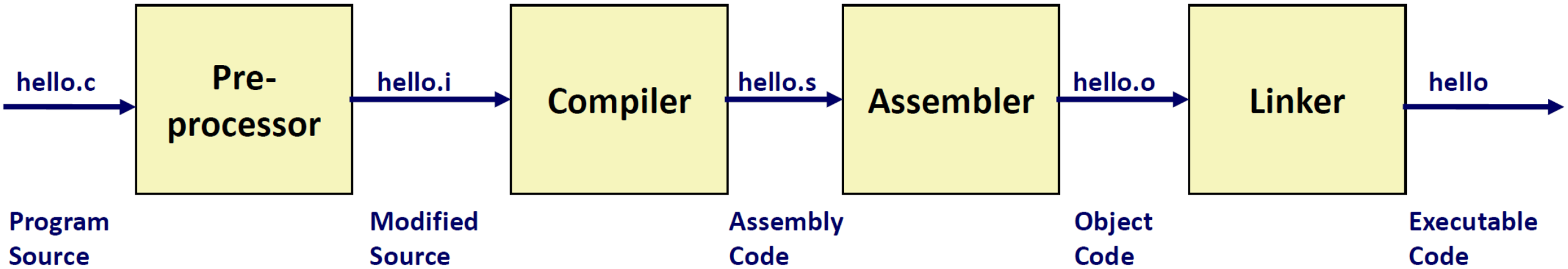
```
1   int sum(int *a, int n)
2   {
3       int i, s = 0;
4
5       for (i = 0; i < n; i++) {
6           s += a[i];
7       }
8       return s;
9   }
```

code/link/sum.c



GCC运行过程（编译器驱动程序）

- Pre-processor(cpp)：展开”#”，只做文本替换，本质是.c变.c
- Compiler(cc1):变成汇编
- Assembler(gas(as)):变成二进机器码，带“说明”（ELF文件）
- Linker(ld):主角，组合并变成可执行目标文件（加载器直接复制，不再变化）



Linker工作概览

- 符号解析:
  - 管定义和引用的符号，这里符号对应于一个函数、全局变量或静态变量
  - 目的是把符号引用和符号定义对应
  - 声明不是链接器管的
- 重定位
  - 把引用改向最终程序的正确位置

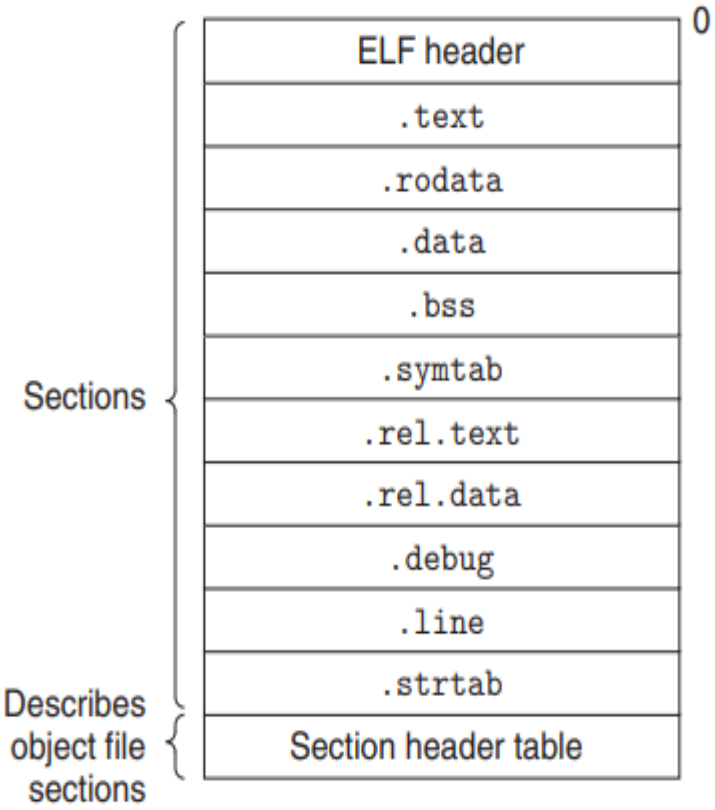
ELF结构

目标文件类型

- 可重定位目标文件(.o)：二进制。（linker的食材）
- 共享目标文件（.so)：特殊的前者，允许动态加载。（特种食材）
- 可执行目标文件(a.out)：二进制，可直接复制进内存执行。（做好的菜）

ELF(Executable and Linkable Format)

- ELF header:字长，字节顺序，文件类型，机器类型
- 段头部表：页大小，段内存地址，……
- .text：代码
- .rodata：跳转表，格式串……
- .data：已初始化的全局变量和静态变量
- .bss：未初始化的静态变量或初始化为0的
  全局和静态变量（不占空间）
- .symtab：符号表，后面重点介绍
- .rel.text：.text重定位信息（给linker的指示）
- .rel.data：.data重定位信息
- .strtab：字符串表，里面有符号表和节的名字
- 节头部表：每节的偏移和大小
- Q：计算机想要找.data节时是怎么找的？



Linker Symbols（存在.symtab中）

- 全局符号：
  - 自己定义，外部可引用
- 外部符号：也是全局符号
  - 外部定义，自己引用
- 局部符号：
  - 自己定义，自己引用

- Static必然为局部符号
- Static在.bss/.data中
- 链接的“局部”不是程序的“局部”
- 静态局部变量？

符号表(symbol table, symtab)

- Name: 名字在字符串表(strtab)中的字节偏移
- Type：我是函数还是变量
- Binding：局部还是全局符号
- Section: 所在的节的index
- 特殊的节
- Value: 一般是离所在节开头的偏移，COMMON-对齐限制，a.out/.so-虚拟地址
- Size: 数据大小

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

Figure 7.4 **ELF symbol table entry.** The `type` and `binding` fields are 4 bits each.

节与伪节(pseudosection)

- 每个符号都分配到某个节
- 三种伪节在节头部表中无条目
- 伪节只在.o中有，a.out没有
- ABS：不该被重定位（main.c)
- UNDEF：未定义（本地引了但本地没定义）
- COMMON：未初始化

COMMON与.bss区别：

理解符号解析并背图

	Global Variables	Static Variables
Uninitialized	COMMON	.bss
Initialized to Zero	.bss	.bss
Initialized to Non-Zero	.data	.data

符号解析

符号解析在干啥?

```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

static-local.c

- 原则：拼好的程序中，每个符号有且只有一个定义
- 每个模块中每个局部符号只有一个定义（编译器检查过了）
- 模块中多个静态变量，不能重名，x, x.1, x.2（编译器干过了）
- 跨模块的引用的勾连（链接器干）
- 处理跨模块重名（链接器干）

- 编译器：处理食材 链接器：炒菜

重整（c++,Java)

- 同名，但不同参数列表
- Eg. Print()可以是void(int), void(char), void(String)
- 处理办法：多叫几个名字

- 名字命名规则：Foo::bar(int,long) 重整为 bar__3Fooil
- 3：”Foo”这个名字长度为3
- i：第一个参数是int；l：第二个参数是long
- bar: 方法名

解析多重定义的原则

- 全局符号要么强，要么弱
- 强：函数和已初始化的全局变量
- 弱：未初始化的全局变量（extern可以强制转成弱符号，类比函数的声明）
- Linux链接器规则：
- 1.（同名字的）强符号最多一个
- 2. 有强有弱，选强的
- 3. 全弱，随机挑

“智力有限”的链接器

```
int x=7;
int y=5;
p1() {}
```

```
extern double x;
p2() {}
```

Undefined behaviour. No link error.
Writes to **x** in **p2** may overwrite **y**!

```
char p1[1]={'1'};
```

```
extern void p1();
p2(){
    p1();
}
int main(){
    p2();
    return 0;
}
```

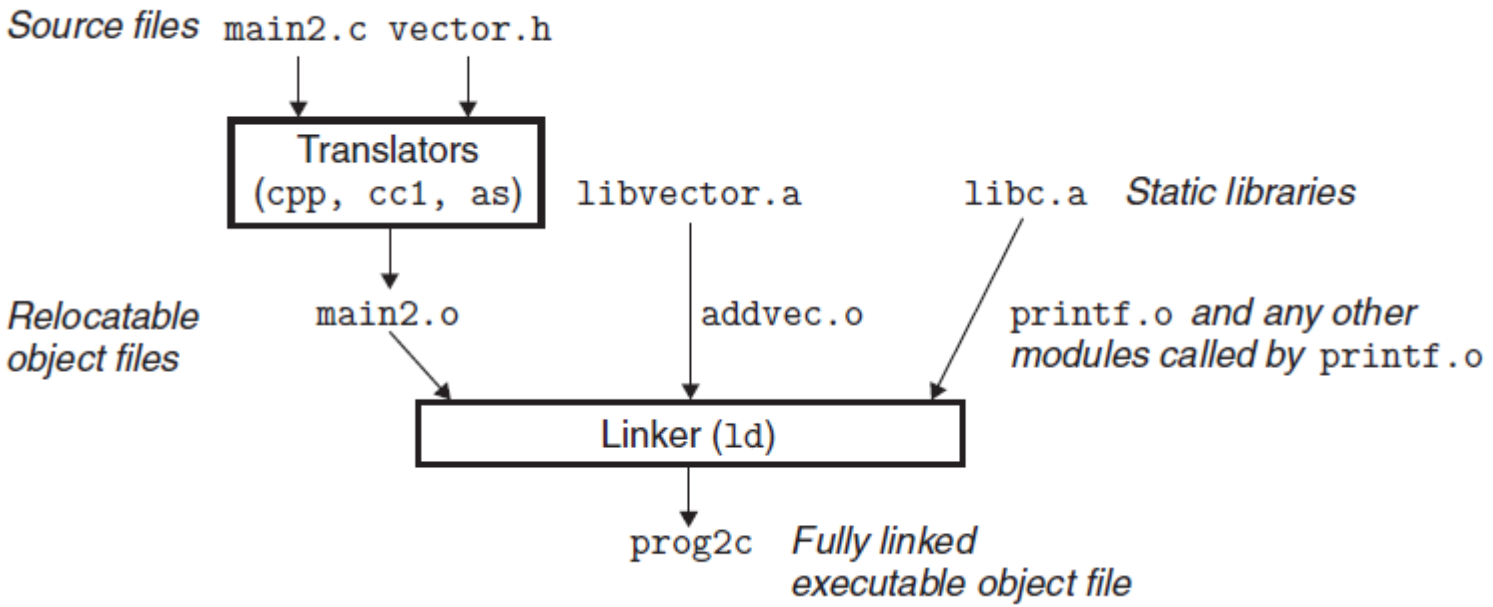
Undefined behaviour. No link error.
Call to p1 may crash!

- 链接器不管引用变量的类型，这可能会出错
- 解决方案：
- 1.能用static就用static（Recall：static是局部的，冲突只在全局）
- 2.非静态的全放header file，里面全加extern

与静态库链接

为什么要引入库的概念?

- 静态库以存档(archive)存放在磁盘中，是一组连接起来的.o文件的集合，后缀名为.a
- 当链接器构造可执行文件时，只复制静态库里被引用的目标模块



链接过程：维护三个集合E,U,D

E：要被合并形成可执行文件的.o的集合

U：引用了但未定义的符号

D：在前面输入文件中已定义的符号集合

顺序问题？

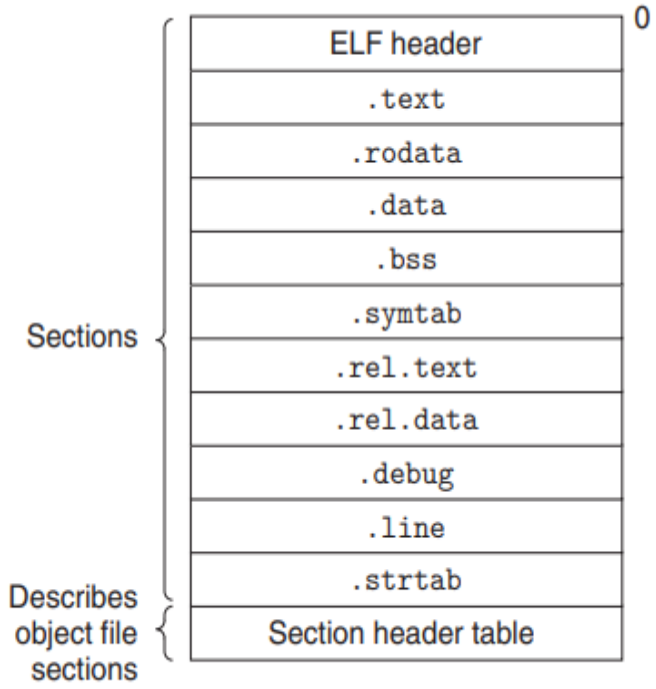
重定位

重定位在干啥?

- “合并同类项”
  - 把所有相同section拼在一起
- “修改路牌”
  - 把所有引用改到最终总程序的正确位置

重定位条目

- 留给链接器的说明，告诉它每个引用该怎么改
- 重定位类型：
- R_X86_64_PC32：用32位PC相对地址访问
- R_X86_64_32：用32位绝对地址访问



code/link/elfstructs.c

```
1   typedef struct {
2       long offset;    /* Offset of the reference to relocate */
3       long type:32,   /* Relocation type */
4            symbol:32; /* Symbol table index */
5       long addend;    /* Constant part of relocation expression */
6   } Elf64_Rela;
```

code/link/elfstructs.c

重定位实例(P480-481)

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_X86_64_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_X86_64_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14      }
15  }
```

Figure 7.10 **Relocation algorithm.**

```
refaddr = ADDR(s) + r.offset; /* ref's run-time address */
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
```

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

code/link/main-relo.d

```
1  0000000000000000 <main>:
2     0:   48 83 ec 08              sub    $0x8,%rsp
3     4:   be 02 00 00 00           mov    $0x2,%esi
4     9:   bf 00 00 00 00           mov    $0x0,%edi        %edi = &array
5                       a: R_X86_64_32 array              Relocation entry

6     e:   e8 00 00 00 00           callq  13 <main+0x13>   sum()
7                       f: R_X86_64_PC32 sum-0x4          Relocation entry
8    13:   48 83 c4 08              add    $0x8,%rsp
9    17:   c3                       retq
```

code/link/main-relo.d

Figure 7.11 **Code and relocation entries from** `main.o`. The original C code is in Figure 7.1.

可执行文件结构

- 和可重定位文件相比变了什么？

多了

- .init：定义了小函数__init，用于初始化
- 段头部表及段的结构
- ELF文件头还包括程序的入口点

少了

- .rel.text, .rel.data等节

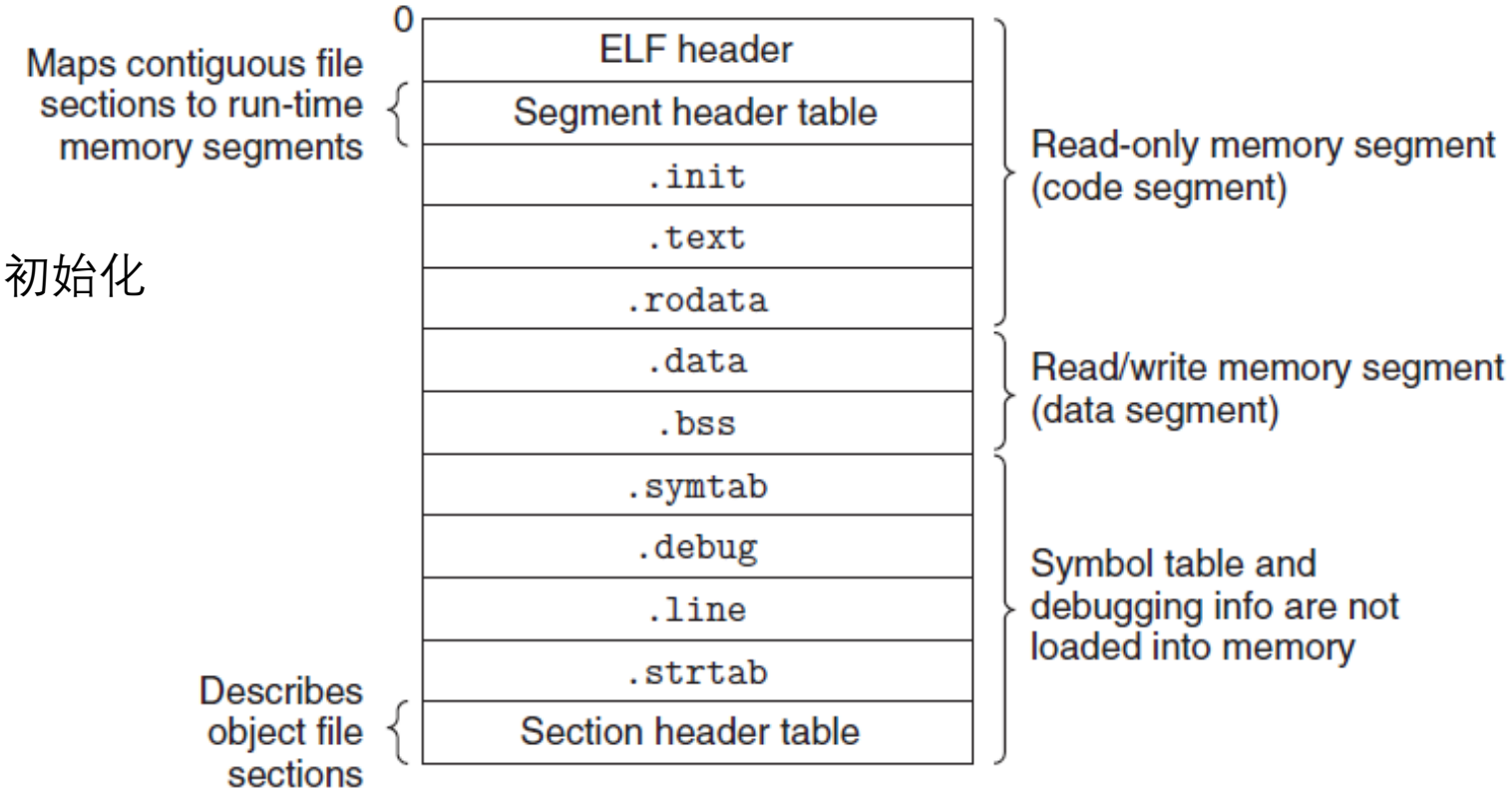
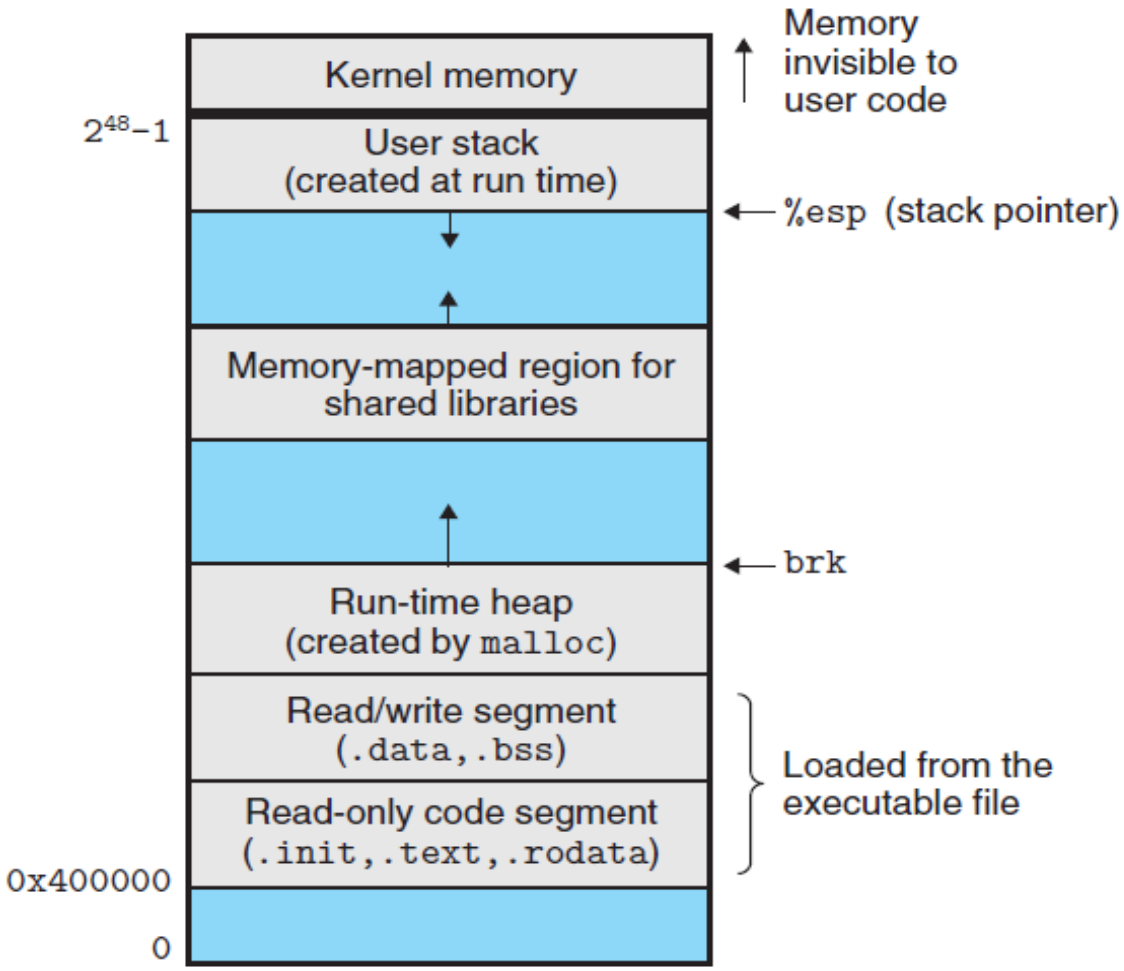


Figure 7.13 Typical ELF executable object file.

加载可执行文件

Figure 7.15
Linux x86-64 run-time memory image. Gaps due to segment alignment requirements and address-space layout randomization (ASLR) are not shown. Not to scale.



Practice

朱家启

The End