ICS Seminar Week8 Prep

朱家启 徐梓越 许珈铭

2023.11.13

Rules

remainder <- ordinal number in WeChat Group % 4

for all questions do

    if question number % 4 == remainder then

        you should work on it

    end

end

8、在链接时，对于什么样的符号一定不需要进行重定位？

A. 不同 C 语言源文件中定义的函数

B. 同一 C 语言源文件中定义的全局变量

C. 同一函数中定义时不带 static 的变量

D. 同一函数中定义时带有 static 的变量

C

6. 在链接时，对于下列哪些符号需要进行重定位？

(1) 不同 C 语言源文件中定义的函数

(2) 同一 C 语言源文件中定义的全局变量

(3) 同一函数中定义时不带 static 的变量

(4) 同一函数中定义时带有 static 的变量

A. (1)(3)　　B. (2)(4)　　C. (1)(2)(4)　　D. (1)(2)(3)(4)

C

9. 下列关于静态库链接的描述中，错误的是(　　)

A. 链接时，链接器只拷贝静态库中被程序引用的目标模块

B. 使用库的一般准则是将它们放在命令行的结尾

C. 如果库不是相互独立的，那么它们必须排序

D. 每个库在命令行只须出现一次即可

D

2．判断下面关于静态链接的说法是否正确。

(1) (　　)链接时，链接器会拷贝静态库(.a)中的所有模块(.o)

(2) (　　)链接时，链接器只会从每个模块(.o)中拷贝出被用到的函数。

(3) (　　)链接时，如果所有的输入文件都是.o 或.c 文件，那么任意交换输入文件的顺序，都不会影响链接是否成功。

(4) (　　)链接时，通过合理地安排静态库和模块的顺序，每个静态库都可以在命令中出现至多一次。

X
X
√
X

4． 以下关于静态库链接的描述中，正确的是：

A．链接时，链接器会拷贝静态库中的所有目标模块。

B．使用库的时候必须把它们放在命令行的结尾处。

C．如果库不是相互独立的，那么它们必须排序。

D．每个库在命令行只须出现一次即可。

C

8． 下面关于链接的说法，正确的是：

A. Linux 链接器在处理多重定义的同名弱符号时，选择链接时遇到的第一个符号

B. 链接发生在源代码编译之后、可执行目标程序运行之前

C. C 程序静态局部变量和静态全局变量都在 ELF 可重定位目标文件的.data 段

D. 链接器构造可执行目标文件时，只复制静态库里被应用程序引用的目标模块

D

10.在 foo.c 文件中的函数外，如果添加如下一条语句：

```
static int count = 0xdeadbeef;
```

那么它在编译为 foo.o 后，会影响到 ELF 可重定位目标文件中的除.text 以外的那些字段？（　　）

A. .rodata

B. .data,.symtab,

C. .data,.symtab,.rel.data

D. .rodata,.symtab,.rel.data

B

5. 在 foo.c 文件中包含如下代码：

```
int foo(void) {
    int error = printf("You ran into a problem!\n");
    return error;
}
```

经过编译和链接之后，字符串"You ran into a problem!\n"会出现在哪个段中？

A. .bss
B. .data
C. .rodata
D. .text

C

/* 编译系统 */

1. （ ）c语言的编译步骤依次是预处理、编译、汇编、链接。其中，预处理阶段主要完成的两件事情是头文件包含和宏展开。
2. （ ）假设当前目录下已有可重定位模块 main.o 和 sum.o，为了链接得到可执行文件 prog，可以使用指令 ld -o prog main.o sum.o

/* 静态链接 */

3. （ ）链接时，链接器会拷贝静态库(.a)中的所有模块(.o)。
4. （ ）链接时，如果所有的输入文件都是.o或.c文件，那么任意交换输入文件的顺序都不会影响链接是否成功。
5. （ ）c程序中的全局变量不会被编译器识别成局部符号。

/* 动态链接 */

6. （ ）动态链接可以在加载时或者运行时完成，并且由于可执行文件中不包含动态链接库的函数代码，使得它比静态库更节省磁盘上的储存空间。

√
√
X
√
X
√

7. （ ）动态库通常被编译成位置无关代码。
8. （ ）通过代码段的全局偏移量表 GOT 和数据段的过程链接表 PLT，动态链接器可以完成延迟绑定 (lazy binding)。

/* 加载 */

9. （ ）_start 函数是程序的入口点。
10. （ ）地址空间布局随机化(ASLR)不会影响代码段和数据段间的相对偏移，这样位置无关代码才能正确使用。

/* static/extern 关键字 */

11. （ ）函数内的被 static 修饰的变量将分配到静态存储区，其跨过程调用值仍然保持。
12. （ ）变量声明默认不带 extern 属性，但对函数原型的声明默认带 extern 属性。

√
X
√
√
√
√

7. C 源文件 f1.c 和 f2.c 的代码分别如下所示，编译链接生成可执行文件后执行，输出结果为（ ）

A. 100　　B. 200

C. 201　　D. 链接错误

```
// f1.c
#include <stdio.h>
static int var = 100;
int main(void)
{
    extern int var ;
    extern void f() ;
    f() ;
    printf("%d\n", var) ;
    return 0;
}
```

```
//f2.c
int var = 200;

void f()
{
    var++;
}
```

A

8. C 源文件 m1.c 和 m2.c 的代码分别如下所示，编译链接生成可执行文件后执行，结果最可能为（ ）

```
$ gcc -o a.out m2.c m1.c ; ./a.out
0x1083020 _____ _____
```

A. 0x1083018, 0x108301c　　B. 0x1083028, 0x1083024

C. 0x1083024, 0x1083028　　D. 0x108301c, 0x1083018

```
// m1.c
#include <stdio.h>

int a1 ;
int a2 = 2 ;
extern int a4 ;

void hello()
{
    printf("%p ", &a1);
    printf("%p ", &a2);
    printf("%p\n", &a4);
}
```

```
//m2.c
int a4 = 10 ;

int main()
{
    extern void hello() ;

    hello() ;
    return 0 ;
}
```

D

7. C 源文件 f1.c 和 f2.c 的代码分别如下所示：

```
// f1.c
#include <stdio.h>
void f();
int x ;
int main(void)
{
   x = 1;
   f() ;
  printf("%x\n", x) ;
  return 0;
}
```

```
//f2.c
float x ;
void f()
{
  x = 2 ;
}
```

运行下面的命令后得到的结果是：

```
$ gcc f1.c f2.c
$ ./a.out
```

A. 1　　B. 2　　C. 3f800000　　D. 40000000

D

8. 文件 f1.c 和 f2.c 的 C 源代码如下图所示：

```
//f1.c
#include<stdio.h>
extern float x;
extern void f();
int main()
{
   f();
   printf("%d\n",(int)x);
   return 0;
}
```

```
//f2.c
int x = 0;
void f()
{
   x++;
}
```

已知这两个文件在同一个目录下，在该目录下用“gcc -Og -o f f1.c f2.c”编译，然后用“./f”运行，这个过程中会出现的情况是：

A. 编译错误
B. 输出0
C. 输出1
D. 链接错误

B

11.（ ）c 源文件 f1.c 和 f2.c 的代码分别如下所示，那么运行结果为?

```c
// f1.c
#include <stdio.h>
static int var;
int main() {
  extern void f(void);
  f();
  printf("%d", var);
  return 0;
}
```

```c
// f2.c
extern int var;

void f() { var++; }

int var = 100;
```

A. 0　　B. 1　　C. 101　　D. 链接时出错

A

12. ( )在 gcc-7 编译系统下，以下的两个文件能够顺利编译并被执行。在 x86-64 机器上，若某次运行时得到输出 0x48\n，请你判断这个 16 进制的 48 产生自？

```
// f1.c
void p2(void);

int main() {
  p2();
  return 0;
}
```

```
// f2.c
#include <stdio.h>
char main;

void p2() {
  printf("0x%x\n", main);
}
```

A. 垃圾值
B. main 函数汇编地址的最低字节按有符号补齐的结果
C. main 函数汇编地址的最高字节按有符号补齐的结果
D. main 函数汇编的第一个字节按有符号补齐的结果

D

第四题（10 分）

考虑如下两个程序（fact1.c 和 fact2.c）：

```
/* fact1.c */
#define MAXNUM 12
int table[MAXNUM];
int fact(int n);
int main(int argc, char **argv) {
    int n;
    table[0] = 0;
   table[1] = 1;
   if (argc == 1) {
      printf("Error: missing argument\n");
      exit (0);
   }
   argv++;
   if (sscanf(*argv, "%d", &n) != 1 || n < 0 || n >=
   MAXNUM) {
       printf ("Error: %s not an int or out of
   range\n", *argv);
       exit (0);
   }
   printf("fact(%d) = %d\n", n, fact(n));
}
```

```
/* fact2.c */
int* table;
int fact(int n) {
    static int num = 2;
    if (n >= num) {
        int i = num;
        while (i <= n) {
            table[i] = table[i-1] * i;
            i++;
        }
        num = i;
    }
     return table[n];
 }
```

（1）对于每个程序中的相应符号，给出它的属性（局部变量、强全局变量或弱全局变量），以及它在链接后位于 ELF 文件中的什么位置？（提示：如果某表项中的内容无法确定，请画 X）（6 分）

fact1.c

变量	类型	ELF Section
table		
fact		
num		

fact2.c

变量	类型	ELF Section
table		
fact		
num		

（2）对上述两个文件进行链接之后，会对每个符号进行解析。请给出链接后下列符号被定义的模块（fact1 or fact2）。（2 分）

	定义模块
table	
fact	
num	

（3）使用 gcc（命令：gcc -o fact fact1.c fact2.c）来编译之后得到的可执行文件是否能够正确执行？为什么？（2 分）

fact1.c

变量	类型	ELF Section
table	**weak global**	**.bss**
fact	**weak global**	**X（或.bss）**
num	**X**	**X**

fact2.c

变量	类型	ELF Section
table	**weak global**	**.bss**
fact	**strong global**	**.text**
num	**local**	**.data**

	定义模块
table	**不确定**
fact	**fact2**
num	**fact2**

不一定能正确执行。因为 table 在两个文件中都是 weak global，因此链接器会任选一个定义来解析 table。而因为在 fact2 模块中只给 table 预留了一个单字（4 字节）空间，因此如果选择了 fact2 来解析 table 的话，会出现 segmentation fault。【该问题已经过编译检查确认。】

第四题（10 分） 链接

考虑如下3个文件：main.c, fib.c和bignat.c：

```
/* main.c */
void fib (int n);
int main (int argc, char** argv) {
    int n = 0;
    sscanf(argv[1], "%d", &n);
    fib(n);
}
```

另外，假设在文件 bignat.c 中定义了如下两个函数 plus 和 from_int（具体定义略）：

```
int plus (int n, unsigned int* a, unsigned int* b, unsigned
int* c);
void from_int (int n, unsigned int k, unsigned int* a);
```

```
/* fib.c */
#define N 16

static unsigned int ring[3][N];

static void print_bignat(unsigned int* a) {
    int i;
    for (i = N-1; i >= 0; i--)
        printf("%u ", a[i]); /* print a[i] as unsigned int
        */
    printf("\n");
}
void fib (int n) {
    int i, carry;
    from_int(N, 0, ring[0]); /* fib(0) = 0 */
    from_int(N, 1, ring[1]); /* fib(1) = 1 */
    for (i = 0; i <= n-2; i++) {
        carry   =   plus(N,   ring[i%3],   ring[(i+1)%3],
        ring[(i+2)%3]);
        if (carry)
            {   printf("Overflow   at   fib(%d)\n",   i+2);
        exit(0); }
    }
    print_bignat(ring[n%3]);
}
```

1．（5 分）对于每个程序中的相应符号，给出它的属性（局部或全局，强符号或弱符号）（提示：如果某表项中的内容无法确定，请画 X。）

main.c

	局部或全局？	强或弱？
fib		
main		

fib.c

	局部或全局？	强或弱？
ring		
fib		
plus		

2．（3 分） 假设文件 bignat.c 被编译为一个静态库 bignat.a，对于如下的 gcc 调用，会得到什么样的结果（请选择）？

（A） 编译和链接都正确

（B） 链接失败（原因是包含未定义的引用）

（C） 链接失败（原因是包含重复定义）

命令	结果（A，B 或 C）
gcc -o fib main.c fib.c bignat.a	
gcc -o fib bignat.a main.c fib.c	
gcc -o fib fib.c main.c bignat.a	

3．（2 分）如果在文件 fib.c 中，程序员在声明变量 ring 时，不小心把它写成了：

```
static int ring[3][N];
```

会不会影响这些文件的编译、链接和运行结果？为什么？

main.c

	局部或全局？	强或弱？
fib	全局	弱
main	全局	强

fib.c

	局部或全局？	强或弱？
ring	局部	X
fib	全局	强
plus	全局	弱

命令	结果（A，B 或 C）
gcc -o fib main.c fib.c bignat.a	A
gcc -o fib bignat.a main.c fib.c	B
gcc -o fib fib.c main.c bignat.a	A

对编译、链接和执行结果都没有影响。因为 signed 和 unsigned 之间的转换不会改变整数的表示形式，因此函数调用会正常进行。

Chap2

3. 在int 类型值为32 位、用补码表示、采用算术右移，unsigned 类型值也为32 位的机器上，声明

```
int x, y;
unsigned ux = (unsigned)x;
unsigned uy = (unsigned)y;
```

则下列表达式中，当x 与y 取遍所有可能值时，可能为假的是

A. (x<y)==(-x>-y)

B. ~x+~y+1==~(x+y)

C. (ux-uy)==-(unsigned)(y-x)

D. ((x>>3)<<3)<=x

A

3.
```
int x = -2019;
int a = (x / 8) * 8;
int b = ((x + 7) >> 3) << 3;
int c = -(((~x)+1) >> 3) << 3;
```

a, b, c 大小关系是什么?

A. a=b, b=c

B. a=b, b>c

C. a<b b<c

D. a<b, b=c

A

0. 下面程序的输出是（　　）

```
int main() {
    int x = 0xbadbeef >> 3;
    char y = (char)(x);
    unsigned char z = (unsigned char)(x);
    printf("%d %u\n", y, z);
    return 0;
}
```

A. -35 221

B. -35 35

C. -221 221

D. -221 35

A

2. 假设有下面 x 和 y 的程序定义

```
int x = a >> 2;
int y = (x + a) / 4;
```

那么有多少个位于闭区间[-8,8]的整数 a 能使得 x 和 y 相等？（　　）

A. 12

B. 13

C. 14

B

1、对于 IEEE 浮点数，如果减少 1 位指数位，将其用于小数部分，将会有怎样的效果？答：（　　）

A．能表示更多数量的实数值，但实数值取值范围比原来小了。

B．能表示的实数数量没有变化，但数值的精度更高了。

C．能表示的最大实数变小，最小的实数变大，但数值的精度更高。

D．以上说法都不正确。

C

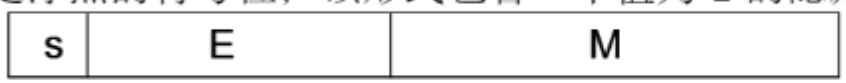
2． 浮点数的阶码没有设计成补码的形式的原因是：

A．为了加快四则运算　　B．为了方便排序

C．为了保存更多数值　　D．为了表示特殊值

B

2. 现有一个二进制浮点的表示规则，其中 E 为指数部分，3 比特，且 bias 为 3；
M 为小数部分，5 比特，采用二进制补码表示形式，且取值(½ ≤ |M|<1)；
s 是浮点的符号位；该形式包含一个值为 1 的隐藏位。



如果用该形式表示$+5_{10}$，s、E 和 M 分别是：

A. 0, 100, 01100
B. 0, 101, 00100
C. 0, 110, 11010
D. 0, 111, 10101

D

1． 下面关于 IEEE 浮点数标准说法正确的是（　　）

A．在位数一定的情况下，不论怎么分配 exponent bits 和 fraction bits，所能表示的数的个数是不变的

B．如果甲类浮点数有 10 位，乙类浮点数有 11 位，那么甲所能表示的最大数一定比乙小

C．如果甲类浮点数有 10 位，乙类浮点数有 11 位，那么甲所能表示的最小正数一定比乙小

D．"0111000"可能是 7 位浮点数的 NAN 表示

D