一、选择题

1. D
2. A

二、非选择题

有下面两个程序。将它们先分别编译为.o文件，再链接为可执行文件。请注意，本大题内前问信息在后问中均有效。

```
// m.c
#include <stdio.h>
void foo(int *);

int buf[2] = {1,2};

int main(){
  foo(buf);
  printf("%d %d", buf[0],
          buf[1]);
  return 0;
}
```

```
// foo.c
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void foo(){
  static int count = 0;
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
  count++;
}
```

Part A. **（20分）**请填写 foo.o 模块的符号表。如果某个变量不在符号表中，那么在名字那一栏打×；如果它在符号表中的名字含有随机数字，那么请用不同的四位数字区分多个不同的符号。对于局部符号，不需要填强符号一栏。

变量名	符号表中的名字	局部符号?	强符号?	所在节
buf	buf	No	No	UND/UNDEF
bufp0	bufp0	No	Yes	.data/.data.rel
bufp1	bufp1	No	No	COM/COMMON
temp	×			
count	count.1797	Yes		.bss

Part B. （15分）使用 gcc foo.c m.c 生成 a.out。 其节头部表部分信息如下。已知 main 和 foo 的汇编代码相邻，且 Ndx 和 Nr 都是指节索引。请补充空缺的内容。

Section Headers:

[Nr] Name	Type	Address	Offset	Size
[1] .interp	PROGBITS	00000000000002a8	000002a8	000000000000001c
[14] .text	PROGBITS	0000000000001050	00001050	0000000000000205
[16] .rodata	PROGBITS	0000000000002000	00002000	000000000000000a
[23] .data	PROGBITS	0000000000004000	00003000	0000000000000020
[24] .bss	NOBITS	0000000000004020	00003020	0000000000000010

Symbol Table:

Num:	Value	Size	Type	Bind	Ndx	Name
35:	0000000000004024	4	OBJECT	LOCAL	24	count.1797
54:	0000000000004010	8	OBJECT	GLOBAL	23	bufp0
59:	000000000000115a	78	FUNC	GLOBAL	14	foo
62:	0000000000004018	8	OBJECT	GLOBAL	23	buf
64:	00000000000011a8	54	FUNC	GLOBAL	14	main
68:	0000000000004028	8	OBJECT	GLOBAL	24	bufp1
51:	0000000000000000	0	FUNC	GLOBAL	UND	printf@@GLIBC_2.2.5

Part C. （4分）接 Part B. 回答以下问题。

a) 读取 .interp 节，发现是一个可读字符串 /lib64/ld-linux-x86-64.so.2。
由 .interp 大小可知要填 4 个字符。而这是动态链接器的绝对路径。

b) .bss节存储时占用的空间为 0 字节，运行时占用的空间为 16 字节。

Part D. （7分）现在通过 objdump -dx foo.o 我们看到如下重定位信息。

```
0000000000000000 <main>:
 0:   55                      push %rbp
...
10:   8b 15 00 00 00 00      mov  0x0(%rip),%edx  # 16 <main+0x16>
                     12: R_X86_64_PC32       buf
...
1e:   48 8d 3d 00 00 00 00   lea  0x0(%rip),%rdi  # 25 <main+0x25>
                     21: R_X86_64_PC32       .rodata-0x4
...
2a:   e8 00 00 00 00         callq  2f <main+0x2f>
                     2b: R_X86_64_PLT32      printf-0x4
...
```

假设链接器生成 a.out 时已经确定：foo.o 的 .text 节在 a.out 中的起始地址为 ADDR(.text)=0x11a8。请写出重定位后的对应于原本 main+0x10 位置的代码。

```
11b8: 8b 15 5e 2e 00 00      mov  0x2e5e(%rip),%edx
(注意addend=0，或者从程序代码知访问的是buf后一个元素，不要填成 5e2a)
```

而原本 main+0x1e 处的指令变成

```
11c6: 48 8d 3d 37 0e 00 00   lea  0xe37(%rip),%rdi
```

可见字符串“%d %d”在 a.out 中的起始地址是 0x2004。

Part E. （10分）使用 objdump -d a.out 可以看到如下 .plt 节的代码。

```
Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:  ff 35 9a 2f 00 00      pushq  0x2f9a(%rip)
           # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:  ff 25 9c 2f 00 00      jmpq   *0x2f9c(%rip)
           # 3fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:  0f 1f 40 00            nopl   0x0(%rax)

0000000000001030 <printf@plt>:
    1030:  ff 25 9a 2f 00 00      jmpq   *0x2f9a(%rip)
           # 3fd0 <printf@GLIBC_2.2.5>
    1036:  68 00 00 00 00         pushq  $0x0
    103b:  e9 e0 ff ff ff         jmpq   1020 <.plt>
```

a) 完成 main+0x2a 处的重定位。

```
11d2:       e8 59 fe ff ff          callq  1030 <printf@plt>
```

b) printf 的 PLT 表条目是 PLT[1]，GOT 表条目是 GOT[3]（均填写数字）。

c) 使用 gdb 对 a.out 进行调试。某次运行时 main 的起始地址为 0x5555555551a8，那么当加载器载入内容后而尚未重定位 printf 地址前，其 GOT 的内容是 0x555555555036。你填写的这个值是 动态（填 静态/动态）链接器设置的。而重定位后可以使用 disas *(long *)0x555555557fd0 读出 printf 动态链接进来的代码。**提示：**disas 是 gdb 中用于反汇编的指令。gdb 如果通过立即数直接访问内存地址，直接使用该数即可。如果需要从一个地址中读值并以此间接访问内存，可以使用 *(long *)0xImm 的格式，其中 Imm 表示该立即数。