Lab4_report

测试截图：

```
a1341543541@ubuntu:~/ICS/lab/Lab4$ sh TestAll.sh

Task0
g++ main.cpp some.cpp -o main
测试成功！
测试成功!0
测试成功!1
测试成功!2
测试成功!3
测试成功!4
能调用这个函数？

Task1
g++ main0.cpp function0.cpp function1.cpp shared.cpp -o main0
task1
斐波拉契数列第10项为:144
g++ main1.cpp function0.cpp function1.cpp shared.cpp -o main1
task1
斐波拉契数列第20项为:17711
g++ main1.cpp function0.cpp function1.cpp shared.cpp -DDEBUG -o main1
task1 测试模式开启
task1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
斐波拉契数列第20项为:17711
```

```
Task2
cd \A && make A
make[1]: Entering directory '/home/a1341543541/ICS/lab/Lab4/Task2/A'
g++ -c A.cpp
ar -r libA.a A.o
ar: creating libA.a
make[1]: Leaving directory '/home/a1341543541/ICS/lab/Lab4/Task2/A'
cd \C && make C
make[1]: Entering directory '/home/a1341543541/ICS/lab/Lab4/Task2/C'
g++ -c C.cpp -o C.o
ar -r libC.a C.o
ar: creating libC.a
make[1]: Leaving directory '/home/a1341543541/ICS/lab/Lab4/Task2/C'
g++ main.cpp C/libC.a B/libB.a A/libA.a -o main
   text    data     bss     dec     hex filename
  23027     744     288   24059    5dfb main
A
Name in B is 我是B哒
B
C
Name in A is 我是A哒

Task3
cd A && make A
make[1]: Entering directory '/home/a1341543541/ICS/lab/Lab4/Task3/A'
g++ -shared -fpic A.cpp -o ../libA.so
make[1]: Leaving directory '/home/a1341543541/ICS/lab/Lab4/Task3/A'
cd C && make C
make[1]: Entering directory '/home/a1341543541/ICS/lab/Lab4/Task3/C'
g++ -shared -fpic C.cpp -o ../libC.so
make[1]: Leaving directory '/home/a1341543541/ICS/lab/Lab4/Task3/C'
g++ main.cpp ./libA.so ./libB.so ./libC.so -o main
   text    data     bss     dec     hex filename
   3389     768      96    4253    109d main
A
Name in B is 我是B哒
B
C
Name in A is 我是A哒
```

```
Task4
g++ -c some.cpp -o some.o -fPIC
g++ -c main.cpp -o main.o -fPIC
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -o main /usr/lib/gcc/x86_64
-linux-gnu/9/../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-gnu/9/
../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/9/crtbeginS.o -L
/usr/lib/gcc/x86_64-linux-gnu/9 -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_
64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-li
nux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/
gcc/x86_64-linux-gnu/9/../../.. main.o some.o /usr/lib/gcc/x86_64-linux-gnu/9/c
rtendS.o /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crtn.o -lc
测试成功！
测试成功!0
测试成功!1
测试成功!2
测试成功!3
测试成功!4
能调用这个函数？

Task5
weak_password
login successful
8413347494854412998
15264931958976647432
```

Task0:

一开始把main.cpp文件里面的 #include"some.h" 改成了 #include"some.cpp" ，后来发现不对，当时助教说不小心把正确的Task0代码放上去了，才发现不对。答案应该是把notATest()函数去掉。

Q1:

这个代码的错误是在main.cpp中使用了没有定义没有声明的函数notATest()，此时编译也无法通过。

Task1:

做这个Task时，遇到的第一的问题是使用ifeq时，总是会报错。一开始将ifeq前面加了tab，终端报错有语法错误；后来知道了使用tab后的命令是由shell执行的，去掉tab后发现有missing separator错误，后来查询后发现ifeq与左括号间需要一个空格。由于是否开启DEBUG模式的函数是在shared.cpp中定义的，因此可以在shared.cpp中定义宏

```
g++ shared.cpp -DDEBUG
```

会提示缺少函数定义，发现是因为g++命令会去尝试编译链接shared.cpp。于是可以在可以DEBUG时使用

```
g++ shared.cpp -DDEBUG -o shared.o
g++ main1.cpp function0.cpp function1.cpp shared.o -o main1
```

但在Makefile中的clean中没有remove .o文件的命令，因此猜测可以省略第一步，将全部一起定义DEBUG宏，但这样定义会不会重复了?

Q1:

因为在shared.h中只有函数的声明而没有函数的定义；本来可能会出现重定义的问题。

Q2:

会出现重定义问题。这是因为function0.h与function1.h都包含了shared.h，而main0.cpp或者main1.cpp中都包含了function0.h与function1.h，因此在main0.cpp中或者main1.cpp中FOO变量产生了重定义。

Q3:

原理就是普通的if语句的逻辑，如果这个文件被include过了，那么就以及有了对应的宏了，就不再include了，否则就include。问题没有得到解决。这是因为#ifndef语句是为了防止在编译过程中被多次定义，但是main0.cpp或者main1.cpp与function0.cpp、function1.cpp、shared.cpp分开编译，每个编译后的文件都只包含了一次FOO变量，但链接时有重定义。

Task2:

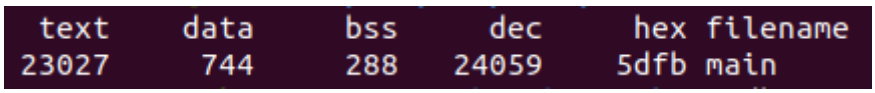
遇到的主要问题是一开始不知道怎么把不同目录下的静态链接库与main.cpp链接起来，后来发现了可以跳转到目录中去进行寻找静态链接库来链接。

Q1:

需要考虑静态链接库和源文件的顺序。这是由静态链接过程中，对静态链接库文件和对目标文件的不同操作决定的。需要将源文件放在前面，这样先找到源文件中的未解析符号，再由链接库去匹配这些符号；否则一开始把就静态链接库放在前面，可能导致源文件中需要用到的未解析符号被提前抛弃掉了。

Q2:

输出如图所示：



text表示已编译的机器代码，下面的数字表示占用的大小。data表示以及初始化的全局以及静态变量。bss表示未初始化的全局和静态变量以及初始化为0的全局与静态变量。dec表示所有数字之和的10进制表示，dec表示所有数字之和的十六进制表示。

Task3:

写的过程几乎没遇到什么问题，但是发现

```
g++ main.cpp -L. libA.so libB.so libC.so -o main
```

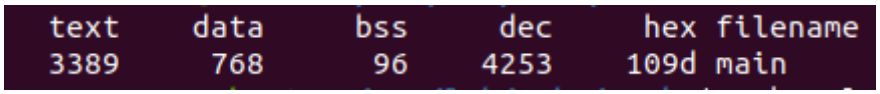
可以链接通过，在执行时却报错。

Q1:

先查找是否存在环境变量LD_LIBRARY_PATH或LD_PRELOAD，如果存在先在其中查找；再查看库高速缓存文件 /etc/ld.so.conf；最后查找默认路径/lib和/usr/lib。

Q2:

这是得到的结果图片：



可以显然看出占用的空间总体变小了。text的变化最大，共享库中的text节不会加载到可执行文件中了。而共享库中的data节与bss节中的内容也不会加载到可执行文件中，因此bss节也减小了，但是data节增大了，因为可执行文件中还要再data节开始的地方创建了一个GOT表占用了一定的空间。而可执行文件的text节也创建了一个PLT表，但相比下这个表的大小远远小于共享库不被加载入可执行文件中的text节大小。

Q3:

-fPIC指示编译器生成与位置无关的代码。如果不加-fPIC,则加载.so文件的代码段时,代码段引用的数据对象需要重定位, 重定位会修改代码段的内容,这就造成每个使用这个.so文件代码段的进程在内核里都会生成这个.so文件代码段的copy。但在Linux中试验了一下发现也能通过也能正常运行程序，并且调用size main得到的节的大小一样，可能是因为：

1. g++默认开启-fPIC开关
2. loader使代码位置无关
3. -shared选项包含了-fPIC选项。

Q4:

在Linux下不同版本之间有着不同的版本号，各个带有不同版本号的共享库共存，并采用SO-NAME(Shortfor shared object name)的命名机制来记录共享库的依赖关系。每个共享库都有一个对应的“SO-NAME”(共享库文件名去掉次版本号和发布版本号)。在Linux系统中，系统会为每个共享库所在的目录创建一个跟SO-NAME相同的并且指向它的软连接(Symbol Link)。这个软连接会指向目录中主版本号相同、次版本号和发布版本号最新的共享库，使得所有依赖某个共享库的模块，在编译、链接和运行时，都使用共享库的SO-NAME，而不需要使用详细版本号。

Task4:

使用ld命令还需要链接一些系统目标文件，显然标准C库是需要的，于是尝试了

```
ld -lc main.o some.o -o main
```

然后就有报错提示：ld: warning: cannot find entry symbol _start; defaulting to 0000000000401050，得到的main文件无法执行。去网上查找资料，发现是因为ld手动链接时需要包含的库文件很多，可以使用-v选项来运行查看具体的步骤：

```
g++ -v -o main main.o some.o
```

得到的内容很多，但关键的链接操作细节如下：

```
 /usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-linux-
gnu/9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper -
plugin-opt=-fresolution=/tmp/ccbYVErR.res -plugin-opt=-pass-through=-lgcc_s -
plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lc -plugin-opt=-pass-
through=-lgcc_s -plugin-opt=-pass-through=-lgcc --build-id --eh-frame-hdr -m
elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-
64.so.2 -pie -z now -z relro -o main /usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-
gnu/9/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-gnu/9 -L/usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib
-L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -
L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/9/../../.. main.o some.o -
lstdc++ -lm -lgcc_s -lgcc -lc -lgcc_s -lgcc /usr/lib/gcc/x86_64-linux-
gnu/9/crtendS.o /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/crtn.o
COLLECT_GCC_OPTIONS='-v' '-o' 'main' '-shared-libgcc' '-mtune=generic' '-
march=x86-64'
```

最终编译器 g++ 调用链接器 collect2 来完成链接工作。为什么是 collect2 而不是 ld 呢？实际上 collect2 是对 ld 的封装，最终还是要调用 ld 来完成链接工作。与 ld 不同的是在完成链接工作后，collect2 还会对链接结果做一些处理，主要是收集所有与程序初始化相关的信息并且构造初始化的结构。简单来说，可以把 collect2 看作是链接器 ld。因此collect2后面的内容就是ld后面的内容。将后面一段直接copy到Makefile中的ld后面，可以得到正确的结果。再进行一些删减，程序也能正常运行。

Q1：

添加了/lib64/ld-linux-x86-64.so.2，这是动态链接器，用于运行时动态链接程序。

添加了libc.so.6，这是C语言标准库。

添加了linux-vdso.so.1，VDSO就是Virtual Dynamic Shared Object，就是内核提供的虚拟的.so，这个.so文件不在磁盘上，而是在内核里头。内核把包含某.so的内存页在程序启动的时候映射入其内存空间，对应的程序就可以当普通的.so来使用里头的函数。这样，随内核发行的libc就唯一的和一个特定版本的内核绑定到一起了。注意，VDSO只是随内核发行，没有在内核空间运行，这个不会导致内核膨胀。这样内核和libc都不需要为兼容多个不同版本的对方而写太多的代码，引入太多的bug了。

Q2：

是的，如果不考虑动态链接器的不同版本的话，一个操作系统只需要一个动态链接器。因为一个动态链接器实际上是一种特殊的动态链接库，他也可以被加载到现在正在运行的程序当中，不过其能够自己而重定位自己的全局变量和静态变量。

Task5：

历时5天写出来的一个Task。一开始想到的是栈溢出攻击，后来查看了login的汇编之后，太多复杂的函数调用，可能还开启了防护，就放弃了这个想法；然后想到了猜值，去visual stdio查看了hash的函数定义后，估算了一下复杂程度，觉得暴力破解要跑很久，猜密码的觉得太复杂，于是去进行打桩。

一开始进行对hash的调用的打桩，由于c++的符号命名规则比c复杂很多，去查看了vs里面的hash的实现，copy过来并去掉一些不属于linux的定义，将其重载仿函数的返回值值修改为3983709877683599140 ，再加上std的命名空间，测试发现构造的hash调用后的符号与原来的符号完全一致，但是将其设置为LD_PRELOAD后，发现并不能调用，用

```
ldd --dyn-sym login
```

检查了login的动态符号表后发现并没有hash相关的符号，因此只能去打桩strcmp函数。

login的动态符号表里面有strcmp函数，因此一开始想想类似构造hash一样去构造一致的符号，后来发觉strcmp是来自c语言库的函数，其符号就是strcmp，于是构造动态链接库的文件也必须是c文件。重写了strcmp函数，直接使其返回值为0，但是程序运行中不断有崩溃，甚至影响了正常的终端输出与使用。可能在进行源码中出现的strcmp之前，也执行了很多次strcmp进行初始化之类的操作，于是加入一个输出来观察进行想要改变结果的strcmp之前之后执行了多少次，发现第10次调用strcmp时才是进行hash值的比较，于是加入一个静态变量，记录调用次数，调用次数为10时返回0，其余时候正常输出。