Lab4 - MakeLab

Lab形式

本次Lab对应CSAPP中的链接单元，包含5个tasks加一个荣誉课程必做的task，在所有task文件夹以外有一个测试脚本，使用

```
sh TestAll.sh
```

运行测试，将所有的输出截图（一个截图可能截不下）放入报告中。最终提供修改过的本次lab的文件与一个报告，报告内容主要是在完成每个task时遇到的问题与解决办法，除此之外每个task可能包含一个或多个问题，对问题的回答也需要写在报告之中，问题当然也可以去网上搜索可能的解释，回答不用每个都太详细，很多一句话就可以，当然你想详细的话也可以～（没有篇幅加分之类的东西）
本次lab需要大家写的代码量很小，也不像前几个lab一样烧脑，但由于这是第一个与现实操作系统充分接轨的lab，可能需要大量的搜索和学习。

Lab的预先准备

ubuntu18.04默认没有安装make和gcc。之前的lab可能已经让你们装过gcc了，所以使用以下命令安装make工具

```
sudo apt install make
```

什么是make和Makefile

在软件开发中，make是一个工具程序（Utility software），通过读取叫做“Makefile”的文件，自动化建构软件。它是一种转化文件形式的工具，转换的目标称为“target”；与此同时，它也检查文件的依赖关系，如果需要的话，它会调用一些外部软件来完成任务。它的依赖关系检查系统非常简单，主要根据依赖文件的修改时间进行判断。大多数情况下，它被用来编译源代码，生成结果代码。然后把结果代码连接起来生成可执行文件或者库文件。它使用叫做“Makefile”的文件来确定一个target文件的依赖关系，然后把生成这个target的相关命令传给shell去执行。（from Wikipedia）
换言之，Makefile包含了一系列的规则和命令，在含有Makefile的文件夹内，执行如下命令来使用Makefile内部的规则和指令：

```
make <target>
```

其中，<target>表示在Makefile中定义的数个目标之一，定义的格式如下：

```
[target] <target name> : [prerequisites] ...
    command
    ...
```

其中target标签是可选的，例如如下定义也是可行的：

```
main: main.cpp
    g++ main.cpp -o main
```

在上述Makefile所在的目录在执行make main命令时，make程序先检查main.cpp是否有更新，如果前提条件里出现了不是文件而是其他目标名称的前提，则递归检查其他目标是否有更新，如果所有前提条件都没有更新，则本次执行不进行操作，如果前提条件里其他的目标有更新，则先执行有更新的目标下命令。最后执行调用的目标下的命令（在这里是main），命令使用键进与Makefile的指令进行区分，带有缩进的行最终是由shell来执行的，make内部执行的指令在Makefile中是没有缩进的，命令不限于编译命令，任何能够在shell里执行的命令比如rm -rf *(🤔如果要尝试这个命令一定记得快照且不要sudo)都可以执行。
同时，Makefile中也可以使用变量，直接使用如下格式就可以定义变量：

```
<var name> = <value>
```

所有的变量值都是字符串值，使用时，只需要${<var name>}就可以精确的展开变量值到变量所处位置。例如如下代码：

```
foo = c
prog.o : prog.$(foo)
    g$(foo)$(foo) -S$(foo) prog.$(foo)
```

等价于：

```
prog.o : prog.c
    gcc -c prog.c
```

当然不会有正常人这么写。
变量可以在make运行时通过如下格式进行值的指定：

```
make <target> <var name>=<value>
```

Makefile包含的内容非常丰富，并不是所有内容我们都会在这次lab中使用到，本次lab的重点也并不全在编写Makefile，所以对Makefile简单的介绍就到这里啦。想要继续了解Makefile的同学可以阅读这个链接，或自行查阅Makefile的官方标准文档，对于每个task额外需要的Makefile知识，将在每个task的描述文档里进行描述。

Task 0

描述

在这个task里需要更改或删除部分代码，请修改main.cpp，不要修改some.h和some.cpp。
这个task就是“简单”地编译好一个三个文件的C++程序，编译C++程序我们一般使用g++命令。如下的代码能够将main.cpp编译为可执行文件main：

```
g++ main.cpp -o main
```

在这个task中，先修改错误的代码，再在Makefile中使用g++命令编译程序。

问题

1. 代码中的错误是什么？

Task 1

描述

在这个task里不需要更改cpp或者h了，只需要写Makefile即可。在给出的Makefile中，包含了两个编译目标：main0和main1。分别是以main0.cpp为入口文件和以main1.cpp为入口文件。这两个文件都引用了function0与function1,function0与function1都引用了shared。在function0中定义的代码，针对整个工程是否#define DEBUG而有着不同的运行逻辑。因此我们要求main1目标能够指定变量debug为True或者False，来开启这个“调试开关”。（注意大小写）
总的来说，task1需要大家修改Makefile文件，使的能够使用make main0命令来编译main0，make main1命令来编译main1，make main1 debug=True命令来开启funciton0中的DEBUG模式。

预备知识：

在g++中，使用如下命令相当于在源代码中#define <var name>:

```
g++ <source> -D<var name>
```

例如：

```
g++ main.cpp -DDEBUG
```

相当于#define DEBUG
在Makefile中，可以使用如下命令字符串之间相等的条件判断：

```
ifeq (<string 1>,<string 2>)
    [commands]
else
    [commands]
endif
```

当然Makefile中的变量也可以填入进去，毕竟是精确展开。

问题

1. 为什么两个function.h都引用了shared.h而没有出问题？本来有可能出什么问题。
2. 如果把shared.h中注释掉的变量定义取消注释会出什么问题？为什么？（使用TestAll.sh测试之前一定记得把注释加回去！）
3. 通常使用shared.h中另外被注释掉的宏命令(#开头的那些行)来规避重复引用的风险，原理是什么？取消这些注释之后上一题的问题解除了吗？不管解没解除背后的原因是什么？（使用TestAll.sh测试之前一定记得把注释加回去！）

Task 2

描述

在这个task里，我们需要编译两个静态链接库，将这两个静态链接库与一个编译好的，没有源代码的静态链接库与main.cpp编译链接，最终生成一个可执行文件。这个task也比较简单，在学习了相关的预备知识之后相信你也很容易上手～
在A目录内，有A.cpp A.h 和 Makefile。修改Makefile在该目录内编译出静态链接库libA.a，同样的，在C目录内编译出libC.a。在主目录中，调用A和C目录中的Makefile编译出libA.a 和libC.a 并与B目录中的libB.a一起，编译main.cpp并链接成一个可执行文件。

预备知识

有些时候Makefile会因为工程的扩大而过于庞大，我们会希望减小单个Makefile内指令的数量。很自然的，就像写代码一样，我们希望能够将一个Makefile拆分成多个，很遗憾的是，每一个目录只能有一个Makefile文件，所以大多数项目将不同部分的代码分装在不同的目录下。每一个目录指定一个Makefile。以期将这个目录里的文件例如源代码转换成其他的文件格式例如静态链接库。如何在顶层的Makefile里启用子目录里的Makefile呢，其实很简单，只需使用如下命令：

```
cd <path to folder> && make <target name>
```

这也很好解释，就是进入那个文件夹再make而已。
使用如下命令将一个或多个编译过的文件打包为静态链接库：

```
ar -r <lib name> [<filename> ...]
```

例如ar -r libB.a B.o将编译好的B.o打包为静态链接库。在Linux系统中，通常使用lib<name>.a来命名静态链接库
在gcc或g++中，使用静态链接库非常容易，可以简单理解为静态链接库与源代码的地位一样，例如如下命令：

```
g++ libA.a main.cpp -o main
```

就可以把当前目录下的A静态链接库和main.cpp编译并链接在一起。

问题

1. 若有多个静态链接库需要链接，写命令时需要考虑静态链接库和源文件在命令中的顺序吗？是否需要考虑是由什么决定的？
2. 可以使用size main命令来查看可执行文件A所占的空间，输出结果的每一项是什么意思？

Task 3

描述

恭喜～做了超过一半啦！在这个task中，所有的代码都跟上一个task一模一样！与众不同的是，这次，我们要将A和C编译为动态链接库。和编译好的动态链接库B一起，编译main并使用动态链接的方法编译好程序。在这个task中，对于动态链接库如libA.so的位置不做要求，可以放在Task3目录内。

预备知识

编译动态链接库很简单，使用如下命令：

```
g++ -shared -fPIC B.cpp -o libB.so
```

就能够将B.cpp编译为动态链接库，在Linux系统中，通常使用lib<name>.so来命名动态链接库。
编译时链接动态链接库也很容易，语法与静态链接库一样。只不过重面链接的文件名字要改为动态链接库。

问题

1. 动态链接库在运行时也需要查找库的位置，在Linux中，运行时动态链接库的查找顺序是怎样的？
2. 使用size main查看编译出的可执行文件占据的空间，与使用静态链接库相比占用空间有何变化？哪些部分的哪些代码（也要具体到本task）会导致编译出文件的占用空间发生这种变化？
3. 编译动态链接库时-fPIC的作用是什么，不加会有什么后果？
4. 现在被广泛使用的公开的动态链接库如何进行版本替换或共存（以linux系统为例）？

Task 4

描述

说来你们可能不信，task4中的代码与task0中一模一样，因此，第一步需要把task0中错的代码也在这个task中一起改掉。
在这个task中，我们的目的也很简单，那就是编译好main这个程序，而与task0不同的是，在这个task中，不允许大家直接使用g++编译好整个完整的程序，而需要使用g++ -c来首先将C++代码编译为各自的elf文件（.o文件），再使用ld 命令手动进行链接。事实上在Makefile中，已经写好了两行g++ -c命令，只需要加上链接的命令就可以啦～
在链接中可能会遇到各种各样的问题，请大家自行搜索（推荐使用google或bing，百度很可能搜不到啥），并把这些问题与在搜索中学到的东西写到报告中。如果遇到一些困难可以和g++编译的结果进行对比。常用的对可执行文件进行属性检查的工具包括file，ldd等。

问题

1. 添加的动态链接库分别是什么，起什么作用？
2. 动态链接器一个操作系统中只需要一个吗？为什么？

Task 5(H)

描述

在Task5目录中，有一个已经编译好的，名叫login的程序。运行该程序之后能够输入一个密码字符串，如果密码与我们预先设定的密码一样，就会输出login successful：如果密码与预先设定的不同，就会输出incorrect password。这个task的任务是在不更改login程序的情况下，让login程序输出login successful，具体的方法自定～可以猜密码，也可以通过什么其他的方式，将进行的操作写入Task5目录下的Task5.sh中，注意不要更改执行login程序的代码，login程序除了login successful或者incorrect passwd还会输出两个数字，不用在意这两个数字，它们是作为检查手段存在在代码中的，为了减轻大家的负担，login.cpp的代码如下：

```
#include <functional>
#include <iostream>
#include <string>
#include <cstring>
#include <fstream>
#include <sstream>

using namespace std;

string output_file_hash(char* filename){
    ifstream f(filename);
    string fs;
    stringstream buffer_file;
    buffer_file << f.rdbuf();
    fs = buffer_file.str();
    return to_string(hash<string>{}(fs));
}

int main(){
    string passwd;
    cin>>passwd;
    size_t passwd_hash = hash<string>{}(passwd);
    string passwd_str = to_string(passwd_hash);
    if (strcmp(passwd_str.c_str(),"3983709877683599140") == 0){
        cout<<"login successful"<<endl;
        cout<<passwd_hash<<endl;
    }
    else{
        cout<<"incorrect password"<<endl;
        cout<<passwd_hash<<endl;
    }
    cout<<output_file_hash("./login")<<endl;
    return 0;
}
```

其中output_file_hash函数仅仅是将程序的一种hash值输出，对这个task没有什么作用。

问题

1. 简述这道题的解法