Lab1实验报告

一、实验思考题

1、Thinking1.1

1. ls -l以长格式的形式查看当前目录下所有可见文件的详细属性。显示的内容包括文件名，文件类型、权限、硬连接数、所有者、组、文件大小和文件的最后修改时间等。
2. mv test1.c test2.c是将test1.c改名为test2.c。
3. cp test1.c test2.c是将test1.c内容复制给test2.c。
4. cd .. 是回到上级目录。

2、Thinking1.2

grep "^#define" test.c可以查找test.c中所有以"#define"开头的行并输出，所以可以用来查找宏定义。
grep "函数名" test.c就可以在test.c中查找所有出现函数名的行。grep "函数名()" test.c可以在test.c中查找所有出现"函数名()"的行，即函数定义处。

3、Thinking1.3

gcc -Wall 会在编译时输出所有警告信息。
gcc -Werror 把所有的告警信息转化为错误信息，并在告警发生时终止编译过程 。
使用gcc -Wall可以在得到警告信息和编译好的程序，可以运行一下试试效果。而用gcc -Werror则不会编译成功

4、Thinking2.1

1. 如果小明从来没有**add**和commit过**printf.c**，那就没办法了。如果只**add**而没有commit过，可以使用git checkout -- **printf.c**恢复。如果**add**与commit过，也可以使用git checkout -- **printf.c**恢复。

2. 既然，小红使用了git rm命令，说明小明至少已经**add**了**print.c**。如果小明只**add**了**printf.c**，那就没办法恢复了。如果已经commit过了，可以先用git reset HEAD **printf.c**,然后用git checkout -- **printf.c**恢复。

3. 使用 git rm --cached Tucao.txt即可。

5、Thinking2.2

第1、3个论述不正确第4个正确，我在github上创建了一个仓库，添加了一个名为test1的分支，然后在本地用 git **clone**命令，master和test1都被**clone**并检出了，而且工作区默认分支是处于master分支，如图：

```
zwb@since MINGW64 /h/Githouse/master (master)
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/test1

zwb@since MINGW64 /h/Githouse/master (master)
$ git checkout test1
Branch test1 set up to track remote branch test1 from origin.
Switched to a new branch 'test1'

zwb@since MINGW64 /h/Githouse/master (test1)
$
```

第2个论述正确，只有执行git **push**才会把本地版本库已经commit的工作同步到远程仓库。在本地使用这些命令与远程仓库没有产生交流。

二、实验难点

在实验一的4个实验中，前三个相对简单，只要找到正确的路径或地址，然后在ppt和指导书的引导提示下就可以做出来。而最后一个修改**print.c**的实验难度要大很多，需要阅读三个程序的代码并认真理解其结构和作用才能明白到底要干啥。对于这三个程序，我对一些细节还不是很清楚，只是大概明白它们之间的调用关系。调用关系如图：

其中printf()和myoutput()在printf.c中，lp_Print()在print.c中，printcharc()在consol.c中。

printcharc()将字符的地址送到PUTCHAR_ADDRESS，应该是起着最基础的输出作用。在printf()中，printf()调用lp_Print()，并将myoutput()、fmt和可变参ap数传入lp_Print()作为参数。其中*fmt就是我们所熟知的printf()函数的参数中双引号内的部分,包含有输出格式、输出宽度、精度、字符补齐等内容。ap则包含着所有输出值。在接下来的对lp_Print()的理解过程中，printf()的格式十分重要，可以说是思路所在。printf()格式如下：

```
%[flags][width][.precision][length]specifier
```

lp_Print()的结构和与其它函数调用关系是这样的：

假设此时需要输出的是数num。要达到将其输出的目的，首先要知道它的输出宽度、实际宽度不足时的填充字符和整数类型这些信息。又因为是用**myoutput**()输出的，而**myoutput**()的参数为(arg, buf, length)，其中arg的作用不是很明白，而buf和length分别是要输出的由ap转化成的字符数组和输出宽度。所以需要得到buf和length。在**lp_Print**()中，调用了**PrintNum**()(若是输出字符串或字符，则调用PrintString()或**PrintChar**())来求length和buf。对于**PrintNum**(PrintString()、**PrintChar**())，它的参数还需要num的实际宽度、填充字符、正负等，这些就需要在**lp_Print**()中求得，而这便是我们需要做的。

做法如下：遍历指针fmt，如果找到'%'，就判断'%'后边的格式信息，找出padc、width、longFlag、ladjust等变量的值。代码如下：

```
width = 0;
if((*fmt)!='%'){
    if(*fmt=='\0'){
            break;
```

```
    }
        OUTPUT(arg, fmt, 1);
        fmt++;
        continue;
}
else{
    fmt++;
    if(*fmt=='0'){
        padc = '0';
        ladjust = 0;
        fmt++;
    }
    if((*fmt)>'0'&&(*fmt)<='9'){
        while((*fmt)>'0'&&(*fmt)<='9'){
            width = (*fmt) - '0' + width*10;
            fmt++;
        }
    }
    if((*fmt)=='l'){
        longFlag = 1;
    }
}
```

三、体会与感想

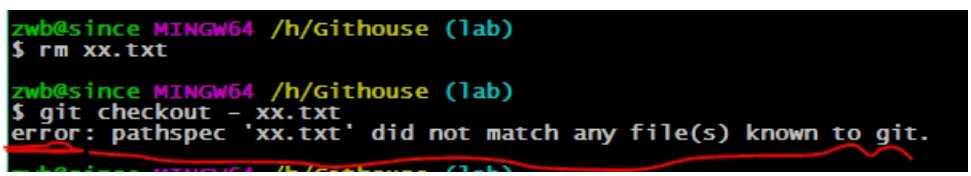
在这次试验中，我体会最深的是刚开始的手足无措与一点点探索的乐趣。在第一节实验课上，面对天书一般的指导书和爬满字符的linux终端，我脑袋里一团乱麻，不知道该怎么处理这么大的信息量。实在没法下手，只好细细研读指导书。对于内核、编译、ELF、Gxemul什么一无所知的我初看指导书比看外语书还艰难。看着看着，跟着指导书的思路，我仿佛明白了一点是在干嘛。后来又看了看ppt，发现ppt上已经写好了不少前三个Exercise的步骤，于是便比照着ppt做完了前三个Exercise。可是到了Exercise2.4，就又不知所措了，三个程序你调用我、我调用你让人摸不着头脑，而且还有不少没见过的语法。翻来覆去地看了一天，才大致理清了它们的关系，知道了应该补充些什么东西。总体来说，这次试验对我来说还是有些难度的，不过一点点地摸索，一点一点地解决问题还是很有乐趣的。

四、指导书反馈

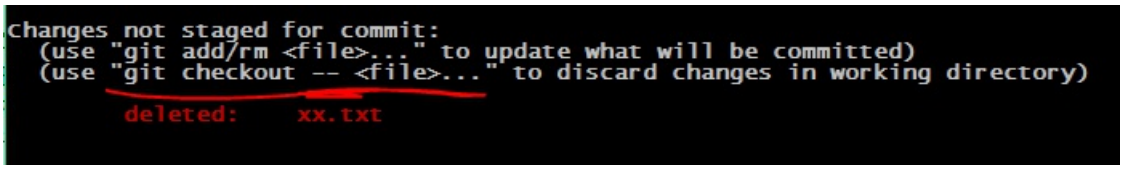
指导书中关于git checkout - <>的地方（如图）好像有一些问题。

git checkout – <file> 如果我们在工作区改呀改，把一堆文件改得乱七八糟的，发现编译不过了！只要我们还没 git add，我们就能使用这条命令，把它变回曾经美妙的样子。

在windows的git中，图中黄色标记的地方应该是“--”,否则会报错，如图：



用了git status命令，从提示中知道应该是git checkout -- <file>,如图：



虽然咱们实验不用windows系统，可是同学们有可能用windows练习git，所以我觉得最好说明一下差异。

git rm --cached <file>命令也是这样。