实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023.12.15	上机课时间	周五 3-6
实 验 名 称	**实验环境介绍**				

Exercise 部分：

Exercise 0.1

> **Exercise 0.1** 在 bash 中分别输入
>
> - echo “Hello Linux”
> - bash –version
> - ls
>
> 三条命令，简单思考其回显结果 ■

回显结果如图 1：

```
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo "Hello Linux"
Hello Linux
jovyan@6d6d781799e0:~/ouc21030031009-lab$ bash -version
GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
jovyan@6d6d781799e0:~/ouc21030031009-lab$ ls
boot  drivers  fork_test.c  fs  gxemul  hello_os_dir  include  include.mk  init  lib  Makefile  mm  readelf  tools  user
jovyan@6d6d781799e0:~/ouc21030031009-lab$
```

图 1　bash 回显结果

Exercise 0.2

> **Exercise 0.2** 执行如下命令，并查看结果
>
> - echo first
> - echo second > output.txt
> - echo third > output.txt
> - echo forth >> output.txt

执行命令如图 2 所示。

```
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo first
first
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo second > output.txt
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo third > output.txt
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo forth >> output.txt
jovyan@6d6d781799e0:~/ouc21030031009-lab$
```

图 2　执行命令

output.txt 中的内容如图 3 所示。

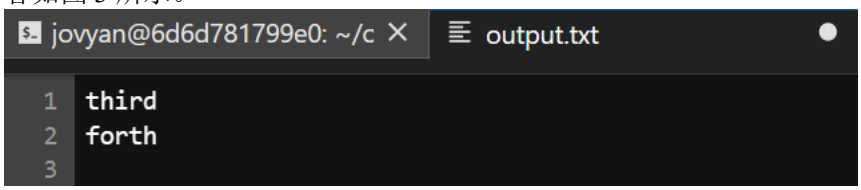


图 3　output.txt 中的内容

Exercise 0.3

Exercise 0.3

- 在/home/jovyan/kernel/learnGit（已 init）目录下创建一个名为 README.txt 的文件。这时使用 git status > Untracked.txt 。
- 在 README.txt 文件中随便写点什么，然后使用刚刚学到的 add 命令，再使用 git status > Stage.txt 。
- 之后使用上面学到的 Git 提交有关的知识把 README.txt 提交，并在提交说明里写入自己的学号。
- 使用 cat Untracked.txt 和 cat Stage.txt，对比一下两次的结果，体会一下 README.txt 两次所处位置的不同。
- 修改 README.txt 文件，再使用 git status > Modified.txt 。
- 使用 cat Modified.txt ，观察它和第一次 add 之前的 status 一样吗，思考一下为什么？

在执行第一次 git add 之前，README.txt 文件仍然位于工作区，尚未添加到暂存区（Stage），因此不是一个已修改的文件，也未被 Git 跟踪。当你在执行第一次 git add 后修改文件内容时，该文件已经被跟踪，但还没有将最新的更新添加到暂存区，因此暂存区中没有这个最新版本的 README.txt 文件。因此，执行 cat Modified.txt 会显示该文件已经被修改。

Exercise 0.4

Exercise 0.4

- 找到我们在/home/jovyan/kernel/learnGit 下刚刚创建的 README.txt，没有的话就新建一个。
- 在文件里加入 **Testing 1**，add，commit，提交说明写 1。
- 模仿上述做法，把 1 分别改为 2 和 3，再提交两次。
- 使用 git log 命令查看一下提交日志，看是否已经有三次提交了？记下提交说明为 3 的哈希值[a]。
- 开动时光机！使用 git reset --hard HEAD^ ，现在再使用 git log，看看什么没了？
- 找到提交说明为 1 的哈希值，使用 git reset --hard <Hash-code> ，再使用 git log，看看什么没了？
- 现在我们已经回到过去了，为了再次回到未来，使用 git reset --hard <Hash-code> ，再使用 git log，我胡汉三又回来了！

■

[a]使用 git log 命令时，在 commit 标识符后的一长串数字和字母组成的字符串

git reset 确实可以用于版本回溯，通过 HEAD 可以回到以前的某个版本。然而，在回溯到过去版本后，想要再返回到之后的版本相对复杂，不能直接通过 HEAD 返回，而需要通过输入相应的 commit hash 值来定位以前的版本。需要注意的是，当回溯到过去版本时，使用 git log 查看版本历史记录可能无法显示最新版本的内容信息，因为回溯时 HEAD 已经移动到过去的某个版本。

如果希望再次返回到最新的版本，必须在回溯之前记录最新版本的 commit hash 值。这意味着在进行每次版本回溯之前，需要慎重考虑，并确保重要的文件没有遗漏，以免造成不可逆的丢失。一种明智的做法是，在回溯之前创建一个新的分支，以保存当前版本，以防需要后续恢复。这有助于在版本控制过程中更好地管理和保护文件。

Exercise 0.5

Exercise 0.5 仔细回顾一下上面这些指令，然后完成下面的任务

- 在 /home/jovyan/kernel 下新建分支，名字为 Test
- 切换到 Test 分支，添加一份 readme.txt，内容写入自己的学号
- 将文件提交到本地版本库，然后建立相应的远程分支。

命令执行过程如下两图：

```
jovyan@6d6d781799e0:~/ouc21030031009-lab$ git checkout -b test
M       boot/start.S
M       fs/fs.c
M       fs/fsformat.c
M       fs/ide.c
M       fs/serv.c
M       include.mk
M       init/init.c
M       lib/env.c
M       lib/kclock.c
M       lib/print.c
M       lib/sched.c
M       lib/syscall.S
M       lib/syscall_all.c
M       lib/traps.c
M       mm/pmap.c
M       mm/tlb_asm.S
M       readelf/readelf.c
M       tools/scse0_3.lds
M       user/fd.c
M       user/file.c
M       user/fork.c
M       user/fsipc.c
M       user/pipe.c
M       user/sh.c
M       user/syscall_wrap.S
Switched to a new branch 'test'
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo "21030031009" > readme.txt
```

图 4　练习 5.1

```
jovyan@6d6d781799e0:~/ouc21030031009-lab$ echo "21030031009" > readme.txt
jovyan@6d6d781799e0:~/ouc21030031009-lab$ git add readme.txt
jovyan@6d6d781799e0:~/ouc21030031009-lab$ git commit -m "Add readme.txt with student ID"
[test 1009ed0] Add readme.txt with student ID
 1 file changed, 1 insertion(+)
 create mode 100644 readme.txt
jovyan@6d6d781799e0:~/ouc21030031009-lab$ git push origin test
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 298 bytes | 298.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To 192.168.130.193:ouc21030031009-lab
 * [new branch]      test -> test
jovyan@6d6d781799e0:~/ouc21030031009-lab$
```

图 5　练习 5.2

成功创建 test 分支，并添加了一份 readme.txt，内容为我的学号 21030031009。

Exercise 0.6

Exercise 0.6 1、在 fibo.c 中使用 c 语言实现输出斐波那契数列前 n 位的程序（n 为评测时的输入数据，输出用逗号分隔，例如 1 1 2 3 5 ）

2、完善 Makefile，要求使用 make 指令可实现 fibo.c 的编译链接，可执行文件名为 fibo

3、在文件夹 sh_test 中新建文件 hello_os.sh，要求通过指令 bash hello_os.sh xxx xxx.c 可以在 sh_test 文件夹下创建新文件 xxx.c，该.c 文件的内容为当前目录文本文档 xxx 的第 8、32、128、512、1024 行的内容提取。例如：代码补全后，在 src/sh_test 文件夹使用指令 bash hello_os.sh file hello_os.c 可以在当前文件夹下生成文件 hello_os.c（若已有 hello_os.c，则将其覆盖），其内容为 src/sh_test/file 第 8、32、128、512、1024 行的内容提取。

4、将完成后的 src/fibo.c、src/Makefile、src/sh_test/hello_os.sh 依次拷贝到 dst/fibo.c、dst/Makefile、dst/sh_test/hello_os.sh 下。

完成后 $CGUSERID-lab 下的文件树如图 ■

```
jovyan@6d6d781799e0:~/ouc21030031009-lab$ git checkout lab0
Switched to branch 'lab0'
Your branch is up to date with 'origin/lab0'.
jovyan@6d6d781799e0:~/ouc21030031009-lab$ git branch
  Test
* lab0
  lab1
  lab2
  lab3
  lab4
  lab5
  lab6
  test
jovyan@6d6d781799e0:~/ouc21030031009-lab$ cd src
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ vi fibo.c
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ cat fibo.c
#include<stdio.h>

int fibo(int n){
    int i, fibo_0, fibo_1 = 0, fibo_2 = 1;
    for(i = 1; i < n; i++){
        printf("%d ", fibo_2);
        fibo_0 = fibo_1;
        fibo_1 = fibo_2;
        fibo_2 = fibo_0 + fibo_1;
    }
    return fibo_2;
}

int main(){
    int var;
    scanf("%d", &var);
    printf("%d", fibo(var));
    return 0;
}
```

图 6　编写 fibo.c 文件

```
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ ls
fibo  fibo.c  Makefile  sh_test
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ vi Makefile
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ cat Makefile
fibo:fibo.c
        gcc -o fibo fibo.c
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ make
gcc -o fibo fibo.c
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ ls
fibo  fibo.c  Makefile  sh_test
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ ./fibo
8
1 1 2 3 5 8 13 21jovyan@6d6d781799e0:~/ouc21030031009-lab/src$
```

图 7　编写 Makefile 文件并测试 fibo

```
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ cd sh_test
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ ls
file  hello_os.c  hello_os.sh
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ vi hello_os.sh
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ cat hello_os.sh
#!/bin/bash
sed -n 8p $1 > $2
sed -n 32p $1 >> $2
sed -n 128p $1 >> $2
sed -n 512p $1 >> $2
sed -n 1024p $1 >> $2
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ ls
file  hello_os.c  hello_os.sh
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ bash hello_os.sh file hello_os.c
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ ls
file  hello_os.c  hello_os.sh
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ cat hello_os.c
#include<stdio.h>
int main(){
        printf("This is Test!\n");
        return 0;
}
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$
```

图 8　sh_test 部分

```
jovyan@6d6d781799e0:~/ouc21030031009-lab/src/sh_test$ cd ..
jovyan@6d6d781799e0:~/ouc21030031009-lab/src$ cd ..
jovyan@6d6d781799e0:~/ouc21030031009-lab$ ls
boot  drivers  dst  include  init  lib  output.txt  readelf  src  tools  user
jovyan@6d6d781799e0:~/ouc21030031009-lab$ cp -rf src/* dst/
jovyan@6d6d781799e0:~/ouc21030031009-lab$ ls dst
fibo  fibo.c  hello_os.sh  Makefile  sh_test
jovyan@6d6d781799e0:~/ouc21030031009-lab$
```

图 9　复制 src 到 dst

Thinking 部分：

Thinking 0.1

> Thinking 0.1　通过你的使用经验，简单分析 CLI Shell，GUI Shell 在你使用过程中的各自优劣（100 字以内）

CLI Shell

①CLI 通常更加高效，特别适用于需要快速执行特定任务或操作的情况。通过命令输入，用户可以直接在键盘上输入命令，无需通过鼠标点击来执行操作。

②CLI 通常占用的系统资源较少，对系统的负担相对较小。这使得它适用于资源受限或远程连接的环境，如服务器管理。

③CLI 易于自动化，可以通过脚本编写自动化任务，提高工作效率。命令行的输出也更容易被其他程序解析和处理。

④对于熟悉命令行的用户，CLI 提供了强大的控制能力，但对于不熟悉的用户，学习曲线可能相对较陡。

GUI Shell

①GUI 提供了直观的图形界面，通过图形元素如按钮、菜单和窗口，用户可以更容易地进行交互。这使得 GUI 更适合初学者或不熟悉命令行的用户。

②GUI 通过图形元素展示信息，使得数据更容易理解。对于一些需要可视化信息的任务，如文件管理、图形编辑等，GUI 更为直观。

③对于非技术用户，GUI 通常具有较为平缓的学习曲线。用户可以通过点击、拖拽等方式完成任务，而无需记忆和输入复杂的命令。

④对于一些较为复杂的操作，GUI 可以提供更直观的方式，而不需要用户记忆和输入大量的命令。

Thinking 0.2

Thinking 0.2 使用你知道的方法（包括重定向）创建下图内容的文件（文件命名为 test），将创建该文件的命令序列保存在 command 文件中，并将 test 文件作为批处理文件运行，将运行结果输出至 result 文件中。给出 command 文件和 result 文件的内容，并对最后的结果进行解释说明（可以从 test 文件的内容入手） ■

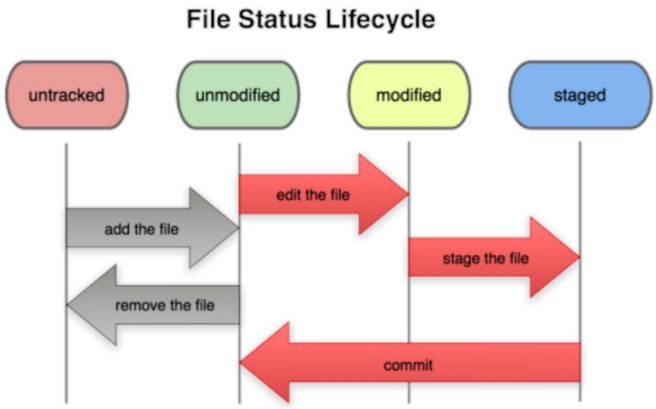
command 文件将 test 中的第一行通过'>'重定向到 test 文件，此后每一行通过'>>'追加在 test 文件后；result 文件则是执行 test 文件中每一行的命令后的结果。

echo echo Shell Start 与 echo 'echo Shell Start' 效果一致；

echo echo \$c>file1 与 echo 'echo \$c>file1' 效果不同，前者是将'echo $c'重定向到 file1 中，后者是将‘echo $c>file1’输出到控制台。

Thinking 0.3

Thinking 0.3 仔细看看这张图，思考一下箭头中的 add the file 、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢？ ■



add the file == git add

stage the file == git add

commit == git commit

Thinking 0.4

Thinking 0.4

- 深夜，小明在做操作系统实验。困意一阵阵袭来，小明睡倒在了键盘上。等到小明早上醒来的时候，他惊恐地发现，他把一个重要的代码文件 printf.c 删除掉了。苦恼的小明向你求助，你该怎样帮他把代码文件恢复呢？

- 正在小明苦恼的时候，小红主动请缨帮小明解决问题。小红很爽快地在键盘上敲下了 git rm printf.c，这下事情更复杂了，现在你又该如何处理才能弥补小红的过错呢？

- 处理完代码文件，你正打算去找小明说他的文件已经恢复了，但突然发现小明的仓库里有一个叫 **Tucao.txt**，你好奇地打开一看，发现是吐槽操作系统实验的，且该文件已经被添加到暂存区了，面对这样的情况，你该如何设置才能使 Tucao.txt 在不从工作区删除的情况下不会被 git commit 指令提交到版本库？

对于小明删除的代码文件 printf.c，可以通过 Git 提供的版本控制来进行恢复。可能步骤为：

git log //查看 git 版本历史

git checkout <commit_id> -- printf.c //恢复文件

git add printf.c //提交更改

git commit -m "恢复 printf.c 文件"

对于小红使用 git rm 删除文件的情况，也可以通过 Git 进行恢复：

git log //查看 git 版本历史

git checkout <commit_id> -- printf.c //恢复文件

git reset HEAD printf.c //撤销删除操作

git add printf.c //提交更改

git commit -m "恢复 printf.c 文件"

至于 Tucao.txt 文件，如果你希望在不从工作区删除的情况下不提交到版本库，可以使用以下步骤：

git reset Tucao.txt //取消跟踪

git add . //提交更改

git commit -m "忽略 Tucao.txt 文件"

Thinking 0.5

Thinking 0.5 思考下面四个描述，你觉得哪些正确，哪些错误，请给出你参考的资料或实验证据。

1. 克隆时所有分支均被克隆，但只有 HEAD 指向的分支被检出。
2. 克隆出的工作区中执行 git log、git status、git checkout、git commit 等操作不会去访问远程版本库。
3. 克隆时只有远程版本库 HEAD 指向的分支被克隆。
4. 克隆后工作区的默认分支处于 master 分支。

1. **错误。**克隆时，默认只有 HEAD 指向的分支被检出。如果要克隆所有分支，可以使用 --mirror 选项。
2. **正确。**在克隆出的工作区中执行 git log、git status、git checkout、git commit 等操作不会直接访问远程版本库。这些操作都是在本地仓库中进行的。
3. **错误。**克隆时会默认克隆远程版本库的所有分支，不仅仅是 HEAD 指向的分支。
4. **不一定。**这取决于远程版本库的默认分支设置。在 Git 中，默认分支的名字可以是任意的，不一定是 master。克隆后工作区的默认分支会与远程版本库的默认分支一致。通常情况下，远程版本库的默认分支是 master，但可以在远程版本库中修改默认分支的设置。

本次实验耗时 7 个半小时