BUAA_OS_笔记

Lab0

常见指令

- 任何文件查找内容：Ctrl+F；man中查找内容：/需要查找的内容，其中直接按 &是列出所有存在该内容的行。**在用/的前提下，可以用 n 获取下一个匹配。 vim中搜索和匹配也是如此。**
- chmod +x 增加可执行权限
- -a 显示隐藏文件（文件名开头是.的文件），例如 ls -a
- 新建目录：mkdir，其中 mkdir -p 创建不存在的目录下的新目录；新建文件：touch，后面可以跟着多个文件，创建多个文件
- **-r：递归做目录中的所有内容**。例如 cp -r, rm -r
- rm -f 删除不提示
- cat 显示文件内容
- head -n 15 file 显示 file 文件的前15行；tail -n 15 file 显示后15 行。`-n` 选项通常用于指定某个数值参数。
- echo $? 查看上一条指令执行结果

Makefile 与 gcc

关于 .PHONY 和 all：

`.PHONY` 关键字用于声明**伪目标**，即不依赖于实际文件的目标。 这样可以避免与同名文件发生冲突，提高 `Makefile` 的可靠性。

`all` 通常是 `Makefile` 中的**默认目标**，通常出现在 `Makefile` 的第一行。其实就是不知道取什么名字（例如编译），而且默认make进行执行（不加任何选项）时就用all。例如：

```
all: function.c test.c
    gcc function.c test.c -o test

.PHONY: clean

clean:
    rm -rf *.o test
```

*.o 代表删除所有以 .o 结尾的文件。

`.PHONY: clean` 确保 `make` 在执行 `clean` 目标时，即使当前目录下有一个名为 `clean` 的文件，`make` 仍然会执行这个规则，而不会误认为它是一个文件。**一定别忘了冒号！！**

分段编译：改变一个时，不会重新编译。只会重新链接。

```
all: function.o test.o
    gcc function.o test.o -o test

function.o: function.c
    gcc function.c -o function.o

test.o: test.c
    gcc test.c -o test.o

.PHONY: clean

clean:
    rm -rf *.o test
```

常用选项：

-o 意思是指定输出文件名。

-c 意思是只编译不链接。生成 .o 文件，而不是最后的可执行文件。

-I 意思是指定头文件**目录**。告诉编译器去哪里查找 `#include` 指定的头文件。

选项	作用
`-o` `<文件名>`	指定输出文件名称
`-c`	只编译 `.c` 文件为 `.o` 目标文件，不进行链接
`-I` `<目录>`	指定额外的头文件搜索路径

例如：

```
gcc -I ../include -c fibo.c -o fibo.o
```

提醒一下，如果题目说要生成 .o 文件，那么一定要增加 -c 选项！ `.o` 文件是 **目标文件（Object File）**，是源代码 `.c` 编译后的中间产物，还没有完成链接。它是**机器代码**，但**不能直接运行**。它还没有包含主函数入口（比如 `main()` 的链接信息），也没有和其他库链接。

$(MAKE) -C 递归

🌟 **语法结构：**

```
$(MAKE) -C <directory> [target]
```

- `$(MAKE)`：这是一个 **自动变量**，表示当前使用的 `make` 程序（通常是 `make`，但也可能是 `gmake`、`nmake` 等）。使用 `$(MAKE)` 可以确保递归调用使用的是同一个 `make` 程序。
- `-C <directory>`：表示**切换到某个子目录下**执行 Makefile 文件（也就是 `cd <directory>`，然后在那个目录执行 make）。
- `[target]`：可选，表示要在子目录中构建的目标（如 `all`、`clean` 等）。

举例说明：

```
all: child_dir
    $(MAKE) -C child_dir        # 执行子目录child_dir中第一个构建目标

.PHONY: clean
clean:
    rm -rf *.o test
    $(MAKE) -C child_dir clean  # 执行子目录中 make clean
```

注意，-C 后面的目录必须存在，且里面要有 Makefile。

sed 行级编辑

-n 安静模式，不修改源文件，只输出有修改的内容，否则输出完有修改的内容后会把所有内容全输出一遍。

-i 修改源文件。

注意 -e 链接的使用。

注意是双引号！单引号中无法引用脚本参数！

打印行：

```
sed -n '1p' file.txt            # 打印第一行
sed -n '1,3p' file.txt          # 打印第一行到第三行
sed -n -e '1p' -e '3p' file.txt # 打印第一行和第三行
```

增：

```
sed -i '1a666' file.txt     # 在源文件第一行后面加一行666
sed -i '1c666' file.txt     # 把源文件第一行替换为666
```

改：

```
sed -i 's/a/b/' file.txt    # 把每行第一个a替换成b，s表示替换
sed -i 's/a/b/g' file.txt   # 把每行所有a替换成b
sed -i '3s/a/b/g' file.txt  # 把第三行所有a替换成b
```

删：

```
sed -i 's/a//g' file.txt    # 内容删除：把所有a删除。相当于把所有a替换成空
sed -i '2d' file.txt        # 整行删除：删除第二行
```

注意**转义的使用**。如 `sed -i 's/\.\/dir/dir/g' lab0_exam` 就是把所有的 `./dir` 替换成 `dir`，也就是删除当前目录标记符。

awk 列级编辑

要注意的地方：

- 默认空格分隔
- 在 `awk` 中，字段编号是从 **1 开始** 的，而不是从 0 开始！

表达式	含义
`$1`	第一列（第一个字段）
`$2`	第二列
`...`	...
`$0`	整行的全部内容（原样输出整行），**相当于什么都没有做！**

- 注意格式，print以花括号包裹。

```
awk -F, '{print $2}' file.txt   # -F后是分隔符，也就是以','作为列区分的标志。如果以空格区分，可以不打-F。print $2代表打印第二列。
awk -F, '{print NR, $0}' file.txt   #显示行号(Number of Row)，打印整行内容！
```

完整示例：打印带行号的第一行到命令行：

```
awk '{print NR, $0}' code/modify.sh | sed -n '1p'
```

grep

grep 默认的输出结果是含有要搜索的内容的整行的完整内容。

如果用 -n 选择打印行号，则输出每一行的格式是：**行号:行内容。**

要掌握 -i -n 的用法。

```
# 查找文件中line
grep line file.txt
grep line* file.txt     # 以line开头的所有
grep -i line file.txt   # 忽略大小写
grep -n line file.txt   # 打印行号
grep -in line file.txt
grep -r -n linw file.txt# 递归寻找
```

注意如果要递归寻找整个目录下的所有子目录的所有文件（扫盘），则必须要加 -r！否则不会进入子目录！ -r -n 可以简写成 -rn。

简单记住，**凡是 grep，一定要 -r。通常也要 -n 输出行号。**

进阶示例：

```
grep -n $2 $1 | awk -F: '{ print $1 }' > $3
```

在文件 $1 中找到含有字符串 $2 的行，把所有行号输出到文件 $3 中。可以看到，**这里省略了 awk 中最后一个参数（文件名），而是以管道的方式输入数据，以重定向的方式输出。**要深刻理解并熟悉这种用法。

Bash shell

简易脚本，执行一群指令：
如：

```
#!/bin/bash
# balabala
echo "Hello World!"
```

bash helloworld.sh即可执行

变量与参数
一般变量：
定义与修改 ： a=1
使用 ： $a, ${a}，双引号内可使用

```
echo "$1"
echo "$2"
echo "the number of parameters is $#"
```

$n 为传入的第n个参数
$# 参数个数

定义数组	arr=(ele1 ele2 ele3)
长度	${#arr[*]}
元素	${arr[0]}
全体元素	${arr[@]} 或 ${arr[*]}
添加元素	arr+=(ele4)
删除元素	unset arr[1]
变量更新	let a=a+1 ((a=a+1)) a=$((a+1))

if 语句

注意格式为 if then - elif - then - ... - else - fi

```
#!/bin/bash
if (( $1 > $2 ))
then
        echo "first > second"
elif (( $1 < $2 ))
then
        echo "first < second"
elif (( $1 == $2 ))
then
        echo "first == second"
else
        echo "I don't know..."
fi
```

注意两个关键地方：

- 最后一个 else 之后没有 then！
- 注意 bash 中**两个整型变量的比较方法为 (())**！不要漏写。

小补充：

✅ Shell 中的 `if` 语法机制：在 Bash 或大多数 Shell 中，`if` 的判断**不是基于布尔值 true/false**，而是根据：

> **命令的退出状态（Exit Status）**

❗**重点来了**：

命令返回值（退出码）	含义	`if` 判断结果
`0`	成功（Success）	进入 `then` 分支 ✅
非 0（如 1、2）	失败（Failure）	进入 `else` 分支 ❌

while 语句

while - do - done

```
while condition
do
    # do something!
done
```

进阶操作

使用 && 可以连接两个指令。如 `mkdir Test && cp -r ./be_copied/ ./Test`。

$# 表示传入参数的个数。

思考1：Makefile 和 Bash 的区别

Makefile 是为了管理项目构建**过程**的自动化工具，而 bash 是通用的 shell 脚本**语言**。

特性	Makefile	Bash (Shell 脚本)
主要用途	自动化构建、编译、生成目标文件等	通用脚本：系统运维、任务自动化、安装脚本等
执行单位	以“规则”为单位，包含依赖关系和命令	以“命令行”为单位，顺序执行
是否使用 shell?	是的！每条命令都会用 shell（通常是 `/bin/sh`）执行	是的，整个脚本都在 shell 中运行
语法风格	自己的语法结构（规则、目标、依赖）+ shell 命令	全部是 shell 语法
TAB 缩进重要性	很重要：命令行必须以 TAB 开头！	不重要，空格或 TAB 都行
变量语法	`$(VAR)` 或 `${VAR}`（更像 GNU Make 的语法）	`$VAR` 或 `${VAR}`（shell 语法）
控制结构（if/for）	需要用 shell 语法，且注意写在一行或用 `\` 换行	完整支持 if/for/while 等结构
重定向 >、管道 |	是 shell 命令的一部分，可以使用	完整支持

思考2：管道和重定向的区别

重定向是写文件或读文件的，管道是用于两个命令之间传输数据的。二者有本质区别。

符号	作用	举例	意义
`>`	**重定向输出到文件**	`echo "hello" > out.txt`	把 `echo` 的输出写入 `out.txt`，**覆盖原内容**
|	**传输数据，将 第一个程序的输出作为第二个程序的输入**	`grep -n $2 $1 | awk -F: '{print $1}' > $3`	把 grep 后的输出（行号：内容）作为awk指令 的输入

拓展：

输入重定向。符号是 < 。一个**文件**作为另一个文件的输入。也就是说，**出现在 < 右侧的，一定是文件而非表达式！**

这里我们不得不提到两种常见的题目需求：

1. 利用 file1 计算 file2，并将结果输出到 file3 中。这个时候可以用以下语句一行搞定：

```
./file2 < file1 > file3
```

2. 将某某表达式（如 `file` 文件第8行）作为 `programA` 的 `stdin`，将结果覆盖输出到 `outputA` 文件中。经典错误做法是 `programA < sed -n '8p' file > outputA`。错误的原因在于**输入重定向出现在 < 右侧的，一定是文件而非表达式！**正确的做法是使用管道：

```
sed -n '8p' file | ./programA > outputA
```

还有大坑是不能写 `| programA`，否则会把 `programA` 理解成类似于cd、ls这种指令。**执行文件，都要先加可执行权限，然后用 ./ 符号。**