UNIX008 May 2000

Advanced UNIX Techniques

Lee Ann Lee
Revised by Wendy Kajiyama

Welcome

Welcome to the Unix system of the University of Hawai‘i Information Technology Services (ITS). This document will show you some advanced techniques on using the system. If you have trouble getting started, we recommend that you read the following ITS documents (in this order):

Getting Started with UNIX from O‘ahu	UNIX003
Getting Started with UNIX from Kaua‘i, Mau‘i, Moloka‘i, Lana‘i and Hawai‘i	UNIX004
About UNIX	UNIX006/UNX020
Using UNIX: Learning the Basics	UNIX007
Introduction to Pine	UNIX013
Getting Started with Pico	UNIX011/UNX130

These documents are available from the ITS Help Desk at Keller 105 or can be viewed on the web at www.hawaii.edu/itsdocs.

Conventions Used in this Document

`Courier` Text printed in the `Courier` typeface denotes text that you type into the computer and/or text displayed by the computer. Note that the remainder of the conventions apply only to `Courier` text.

`^c` Stands for a control character.

A control character is produced by holding down the **`Ctrl key`** and pressing any other key.

The `c` in the `^c` stands for a character. Thus, `^s` stands for the control character generated by holding down the **`Ctrl key`** and pressing the s key.

`bold` Whenever a screen of computer text is used as an example, the text in bold denotes the buttons on the keyboard that you press.

`italic` Means you should replace whatever is *italicized* for an actual instance of the item being described.

`[item]` Means that `item` is not required (optional).

Example: `/home/1/charles% pico [`*`filename`*`]`
`Press` **`Return key`**
means that you should not type `/home/1/charles%.` You start typing with the word pico followed by a space and then, instead of typing the word filename, you should type the name of the file you wish to use with the `pico` command and then press the **`Return key`**. Note that filename can be omitted.

File Privacy

FILE MODE

Unix files can be assigned a desired level of privacy. In other words, it is possible to designate what type of access to your files. This relationship is called the file's mode. The following example shows you how to display file modes:

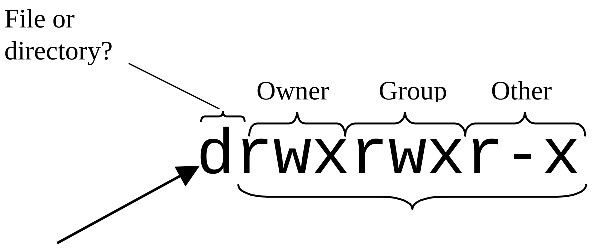
```
uhunix% ls -l
Press Return key
total 2
 drwxrwxrwx      2 diana 512 Jun 30 16:08 pubfile
 -rwx- - - - - - 1 diana 46 Jun 30 16:08 privfile

        file modes
```

Unix files can be read (read access), modified (write access) and/or executed (execute access). The letters `r`, `w`, and `x` in a file mode denote these types of access respectively.

When looking at a file's mode, it is also necessary to classify Unix users into these categories:

owner:	the user that created the file
group:	other users in the same user group as the owner's
other:	all other users



d	(is a directory)
-	(is a file)

r	(can be read by this type of user)
w	(can be modify by this type of user)
x	(can be run by this type of user)
-	(this type of access not permitted of this type of user)

Examples:

`drwxrw-r-`	`d`	=	file is a directory
	`rwx`	=	owner has no restrictions.
	`rw-`	=	group can read and modify this directory.
	`-r-`	=	others can only read this directory.

`-rw-r--r--`	File is not a directory, can be read by all, but can only be modified by the owner.
`drwx------`	Directory is completely private to owner.

chmod

The file mode can be changed using the `chmod` command:

```
     uhunix% chmod user ± mode filename
     Press Return key
```

The command adds or removes access privileges of type `mode` from users of type `user` for all files/directories specified by `filename`. Arguments for `chmod` are:

`user:` `a`=all users, `u`=owner, `g`=group, `o`=others
`±:` `+`=add, `-` = remove
`mode:` `r`=read, `w`=write, `x`=execute

Examples:

```
uhunix% chmod o-w *
Press Return key
```

removes write permission from all other users for all files in current working directory.

```
uhunix% chmod a-x mathnotes#100
Press Return key
```

will ensure that the file called `mathnotes#100` is not an executable file.

It is also possible to set multiple access privileges by issuing several user±mode’s separated by commas:

```
uhunix% chmod g-r, g-w, o-r, o-w *
Press Return key
```

umask

Each time a file is created, its initial file mode is defined by applying the current value of the **file creation mask**.

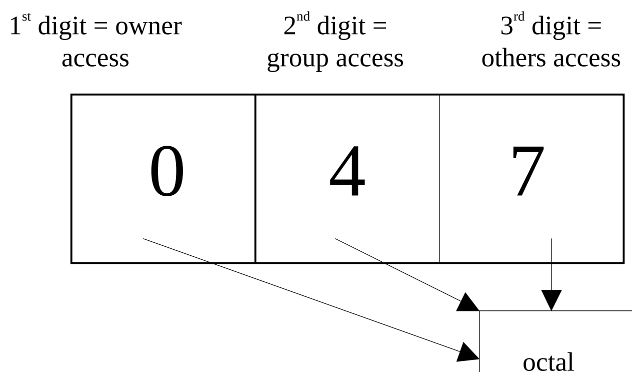
To see the current value of the file creation mask:

```
uhunix% umask
Press Return key
77
```

To change the value of the file creation mask:

```
     uhunix% umask newmask
     Press Return key
```

where `newmask` is a 3-digit octal number with the following meaning:



octal digit	directory mode	file mode
0 =	rwx	rw-
1 =	rw-	rw-
2 =	r-x	r--
3 =	r--	r--
4 =	-wx	-w-
5 =	-w-	-w-
6 =	--x	---
7 =	---	---

The Alias Mechanism

The `alias` command lets you assign another name to a command or a set of commands. For example, the following alias is in each user's `.cshrc` file:

```
     alias dir 'ls –sFC'
```

The line above tells the C shell that `dir` is now another name (an alias) for `ls –sFC`. This means that typing `dir` becomes synonymous with typing `ls –sFC`.

Adding options and arguments to an aliased command is simple. For example,

```
     uhunix% dir –a
     Press Return key
```

is equivalent to

```
     uhunix% dir *.txt
     Press Return key
```

is equivalent to

```
     ls –sFC *.txt
     Press Return key
```

It is also possible to place arguments in the middle of an aliased command. Use `\!*` as a placeholder for the argument:

```
if fi is defined as
alias 'fi find . –name \!* -print' then
uhunix% fi myfile
Press Return key is equivalent to
uhunix% find . –name myfile –print
Press Return key
```

You can also attach a series of commands to a single alias by separating commands with a semicolon (;).This feature is also available from the shell.

```
uhunix% alias pwdls 'pwd; ls –sFC'
Press Return key
```

The semicolon in the example above separates the `pwd` (print working directory) and ls commands. After you create the `pwdls` alias, typing it displays both the current working directory name and the files in it.

To display the aliases that are currently active, type:

```
uhunix% alias
Press Return key
```

To remove an alias, use the `unalias` command:

```
uhunix% unalias nameofalias
Press Return key
```

You should place any useful aliases in your `.login` file. You do this by using any Unix text editor to insert the alias line in `.login`. It is not a good idea, to have more than one or two dozen of aliases in your `.login` file since the time needed to interpret each line lengthens the login process.

Manipulating Input/Output

INPUT/OUTPUT (I/O)

Most programs and commands expect input and/or generate output. For most commands, input is from a file and output is generated to the screen. The screen is known as standard output (stdout). If no files are specified as input, input will be taken from the standard input (stdin), the keyboard. For example, the `cat command` is generally used at `cat filename` where output (the contents of the file) is displayed on your screen. If you used the `cat` command without specifying an input file, the `cat` command will expect your input to come from the keyboard (you must terminate the keyboard input with `^d`):

```
uhunix% cat
Press Return key
this is input from the keyboard
Press Return key
this is input from the keyboard
instead of from a file.
Press Return key
instead of from a file.
^D
uhunix%
```

REDIRECTING I/O

Your can redirect your output to a file (instead of sending it to your screen) by using a redirection pointer (> or >>). If you want to copy the contents of `file1` to `file2`, you could use:

```
uhunix% cat file1 > file2
Press Return key
```

If you want to concatenate several files, you could use:

```
uhunix% cat file1 file2 file3 > newfile
Press Return key
```

To append a file to the bottom of another file, use >>:

```
uhunix% cat bottomfile >> appendeefile
Press Return key
```

In a similar manner, if a program normally expects input from stdin (the keyboard) you could redirect the input to come from a file instead:

```
uhunix% mail username < letterfile
Press Return key
```

PIPES

There may be situations when you want to use the output from one command as input into another. Piping output from one program into another is done using the pipe symbol |. For example, if you wanted to see if a particular `username` was currently logged-in, you could use:

```
uhunix% w | grep username
```

Another example of using pipes can be seen when you want to print a `man` page which `man` displays a la `more` with some reverse video text included. To remove the reverse video text and page breaks form the `man` page and then send it to the printer, you could do the following:

```
uhunix% man mail | cat | lpr –Plj4si
```

COMBINING I/O REDIRECITON AND PIPES

You can use both I/O redirection and pipes in the same command line to manipulate input and tailor output to your specifications:

```
uhunix% ls | grep vi > vi.files
Press Return key
```

The above examples creates a file called `vi.files` which contains a list of all files in the current directory that contain `vi` in their names.

Evaluating the example from left to right we see that the output of `ls` (a list of all files in the current directory) is piped through the `grep` command. This means that the output of ls is the input for the `grep vi` command. The `grep` program displays all those lines of input that contain the pattern `vi` usually to the screen, but since the redirection character > is present, the output will be redirected to a new file called `vi.files`.

Job Control

A **job** is a batch of one or more commands typed in one line at the uhunix% prompt. In other words, all commands separated by either a vertical bar `(|)` or a semicolon `(;)` that were typed before pressing **`Return key`** constitute a job. For example,

```
uhunix% cd ..; ls
Press Return key
```

is one job with two commands (`cd` and `ls`). However, if we had typed them separately, we would have two jobs of one command each:

```
uhunix% cd ..
Press Return key
uhunix% ls
Press Return key
```

Unix is a multi-talking system which allows you to run several jobs at a time. Jobs can either be in foreground, in background, or suspended. A job is said to be running in **foreground** if you must wait for the job to finish executing before you can enter another command. In contrast, a job running in **background** does not have to finish executing before you can enter another command. A **suspended** job does not do any processing. It is in a state of complete inactivity. All commands typed at the uhunix% prompt run in the foreground. To run a process in the background put an ampersand (&) at the end of the command line:

```
uhunix% who &
Press Return key
[1] 5291
uhunix%
```

As seen above, the results of the `who` command do not appear right after the command; rather, the uhunix% prompt appears after the `[1] 5291` message meaning that is now running in the background that you are ready to type another command. When a background process finishes, it signals you like so:

```
[1] Done              who
```

Unlike a suspended job, a background job is still running and computing away.

To suspend a foreground job, type `^z`

```
uhunix% vi test
Press Return key
“test” [New file]
~
~
^Z
Stopped (signal)
uhunix%
```

To suspend a background job, type

```
uhunix% stop %jobnumber
Press Return key
```

Every job has a **job number** which uniquely identifies it. To view stopped and background jobs as well as their job numbers, use the `jobs` command:

```
uhunix% jobs
Press Return key
[1]  -Stopped (signal)    pine
[2]  +Stopped         vi test
[3]  Running          a.out
```

The numbers in square brackets `[1]` are the job numbers.

To terminate a job use the `kill` command:

```
uhunix% kill %jobnumber
Press Return key
```

You can resume a stopped job in either foreground (`fg`) or background (`bg`):

```
uhunix% fg [%jobnumber]
Press Return key
uhunix% bg [%jobnumber]
Press Return key
```

If you omit %jobnumber, it is assumed that you are referring to the last job in the queue. This is handy when you have only one stopped job and you wish to resume it.

Example:

```
uhunix% jobs
Press Return key
[1]   - Stopped (signal)    pine
[2]   + Stopped        vi test
[3]    Running         a.out
uhunix% kill %2
Press Return key
[2]    Terminated      vi test
uhunix% stop %3
Press Return key
[3]   + Stopped (signal)    a.out
uhunix% fg
Press Return key
pine
Press Return key
```

Using the History Feature

The history feature allows you to re-execute a previeously issued command, modify a preiously executed command, and keep a log of commands executed (even between sessions). The default number of commands to be saved is 100. When you logout, history will save these 100 commands in a file named .history in your home directory.

COMMANDS RELATED TO HISTORY

The examples for this section make up a continuous session. That is, the meaning of the commands shown for one example depends on what was typed in the previous example.

`history` -List all previous commands in history.

```
uhunix% history
Press Return key
1 cd
2 cat test.c
3 ls
```

`!!` -Re-execute the previous command.

```
uhunix% !!
Press Return key
history
1 cd
2 cat test.c
3 ls
4 history
```

`!!chars` -Append chars to the previous command then execute it

```
uhunix% ls
Press Return key
math.notes   calc   calc.c
uhunix% !!math.notes
Press Return key
ls math.notes
week1.note   week2.note
week3.note
```

`!n` -Execute the nth command.

```
 uhunix% !2
 Press Return key
 cat test.c
 /* This is file test.c */
 #include <stdio.h>
 main ()
{
      printf (“Testing\n”)
 } /* end of test.c */
```

`!-n`

-Execute the nth previous command.

```
uhunix% history
 Press Return key
 1 cd
 2 cat test.c
 3 ls
 4 history
 5 history
 6 ls
 7 ls math.notes
 8 cat test.c
 uhunix% ! –8
 Press Return key
 cd
```

`!chars`

-Re-execute the most recent command beginning with chars.

```
uhunix% !ca
Press Return key
cat test.c
/* This is file test.c ^C%
```

`^old^new`

-Modify the previous command replacing old with new.

```
uhunix% rm matj/*;rmdir matj
Press Return key
matj/: No such file or directory
rmdir: matj: No such file or
uhunix% ^matj^math
Press Return key
rm math/*; rmdir math
```

PARAMETERS ASSOCIATED WITH HISTORY

```
command: ptroff-ms –t /T1/doc/awk > awk.PS
word #:     0    1  2    3     4      5
```

Examples given below assume that the above command is the last command issued.

`!$` -Last word of last command

e.g. `awk.PS`

`!*` -All words of last command except for the command itself

e.g. `–ms –t /T1/doc/wk > awk.PS`

`:n` -Word n of specified command (using commands related to history)
`cat !ptroff:3` would result in
`cat /T1/doc/awk` being executed

`:n-$` -Words n through last word of specified command

`cat !ptroff:3-$` executes
`/T1/doc/awk > awk.PS`

`:p` -Print command line – don’t execute it; used to verify command

`cat !ptroff:p` would just display
`ptroff –ms-t/T1/doc/awk>awk.PS`

Using Shell Scripts

THE SHELL

Many erroneously think that Unix is the program that displays the % prompt and the processes the commands typed at the prompt. Actually, what they see is the **shell**, a program which acts as an intermediary between the user and the **kernel** (Unix itself). Unix uses a shell called the **C Shell**.

Whenever the shell is ready to accept user input, it displays a prompt. The default prompt for the C Shell is %. After a command is entered, the shell will call the appropriate programs typed in the command line. For example, if you typed `ls`, the shell will search for and execute `ls`, a program which displays the contents of the current working directory. Also, the shell, like any other Unix program, makes system calls. For example, whenever you type `rm` *`filename`*, the shell will make a system call to `unlink`, the actual Unix function that removes a file.

SHELL SCRIPTS

Commands typed at the shell can only be entered one at a time. You must wait for the shell to display the % prompt which tells you that only then can you enter another command. Many shell commands can be typed and saved to a file and then executed as a single program. These files are called **shell scripts**. In addition to the commands available at the % prompt, scripts have other features (such as loop control) that are found in any programming language.

Shell scripts are useful when repetitively executing the same set of C shell commands or to string together some other commands to generate some specific output. For example, if you are writing a computer program and need to keep a working backup copy after each editing session and want to compile, link, and execute the program each time, you could create a shell script that looks like this:

```
uhunix% cat editrun.sample
Press Return key
# sample shell script editrun.sample

echo 'start of script editrun.sample'
cp sample.c oldsample.c
vi sample.c
cc -o sample sample.c
sample
echo 'end of script editrun.sample'
uhunix%
```

The above shell script will allow you to backup, edit, compile and run any file named `sample.c` that is in the current working directory. To add flexibility to it, you could use the `$argv` variables:

```
uhunix% cat editrun
Press Return
# sample shell script editrun

echo 'start of script editrun'
cp $argv [1].c old$argv[1].c
vi $argv [1].c
cc -o $argv [1] $argv [1].c
echo 'end of script editrun'
uhunix%
```

To make a script executable, change the file’s mode:

```
      uhunix% chmod u+x scriptfile
      Press Return key
```

The editrun script allows you to specify any filename (without the `.c` extension) in the script execution. for example:

```
      uhunix% editrun progfile
      Press Return key
```

copies the contents of `progfile.c` to a file called `oldprgfile.c`, opens `progfile.c` with the `vi` editor, compiles `progfile.c` into an executable file called `progfile` and then executes `progfile`.

Customizing you Unix Environment

To facilitate the tailoring of the Unix environment to your specifications, two types of variables are used: **shell variables** and **environment variables**. Shell variables keep values that can affect the behavior of the shell (e.g., the maximum number of commands to save for history), whereas environment variables keep values describing a user's work area (e.g., the current working directory).

SHELL VARIABLES

To display the current values for you shell variables, use `set`:

```
uhunix% set
Press Return key
argv        ()
cwd         /home/1/charles
history     100
home        /home/1/charles
ignoreeof
noclobber
notify
path        (/usr/ucb /bin /usr/bin /etc …
prompt      %
savehist    200
shell       /bin/csh
status      0
term        vt100
user  charles
```

Shell variables are defined also with the `set` command.
For example:

```
uhunix% set history = 10
Press Return key
```

instructs the shell to save at most 10 commands in the `.history` file. There are other variables that only need to be set and done care about what value they receive:

```
uhunix% set ignoreof
Press Return key
```

tells the shell to ignore `^d` as a logout command. There is an `unset` command to reverse the process for these type of variables:

```
uhunix% unset ignoreof
Press Return key
```

ENVIRONMENT VARIABLES

Use `printenv` to view the current values of your environment variables:

```
uhunix% printenv
Press Return
HOME= /home/1/charles
SHELL= /bin/csh
TERM= vt100
USER= charles
PATH= /usr/ucb: /usr/bin: etc: /usr/local: …
```

Environment variables can be defined using the `setenv` command. For example:

```
     uhunix% setenv TERM vt100
     Press Return key
```

MAKING YOUR CUSTOM ENVIRONMENT PERMANENT

There are special files in your directory in which you can place these `set` and `setenv` commands to avoid having to redefine the variables each time you login or start a new process. The file names are: `.cshrc`, `.login`, and `.logout`.

Each time you login, your `.cshrc` file is executed first followed by your `.login` file. The commands stored in your `.cshrc` file are executed every time you run a shell script whereas your `.login` file is executed only when you login. The `.logout` file is executed when you logout. Here are samples for each of these files:

```
uhunix% cat .cshrc
Press Return key
#    @(#)cshrc  1.11  89/11/29   SMI
umask 077
set notify
set history = 100
set path = ( /usr/ucb /bin /usr/bin /etc …
```

```
uhunix% cat .login
Press Return key
#    @(#) .login     Login 1.7  89/90/5         SMI
# tset -I -Q
echo -n 'Default';
setevn TERM 'tset -Q –'
sty erase ^H
alias dir 'ls –sFC'
alias h history
alias cd 'cd \!*;set prompt = "$cwd>"'
cd
uptime
```

```
uhunix% cat .logout
Press Return key
echo 'You are now logged out of uhunix'
date
```

Note: These files are not different from any other shell scripts except for the special meaning the shell assigns to their names.

For additional assistance, please contact the ITS Help Desk at (808) 956-8883,
e-mail help@hawaii.edu, or fax (808) 956-2108.

The ITS Help Desk is located in Keller 105 on the UH Mānoa Campus.
The ITS Help Desk home page is located: www.hawaii.edu/help

The University of Hawaiʻi is an equal opportunity/affirmative action institution.