COMP SCI 35L NOTES

Linux Shell and Shell Scripting

Linux and Files

Linux is an operating system (OS), like Windows or MacOS, that acts as an interface between hardware and user applications.

- Linux is written in C and makes use of various C libraries.
- Linux itself is based on the Unix operating system. There are various distributions ("distros") of Linux that add commonly used packages and applications on top of the Linux kernel.

Operating systems like Linux are useful because they allow for the management of **persistent** data - that is, data that remains even after a device loses power. This data is represented through files, and these files as a whole are managed through a file system. Each file is uniquely identifed by an **inode**. Linux utilizes a tree-based file system. Directories are effectively containers for files (a container that maps file names to files), and these directories may contain directories within themselves (subdirectories), thereby creating a tree structure. The root directory on a Linux system is **/**, and from this root directory, all other files can be found.

Linux users should be familiar with some common directories and their usages:

- **/bin** contains executable programs (this includes commands) that are part of the Linux operating system, such as cp, cat, ls, etc.
- **/dev** contains *representations* of external devices such as flashdrives, keyboards, etc. and are typically used in programs that need to interact with such devices
- **/lib** contains library files essential to the system
- **/etc** contains system configuration files and initialization scripts
- **/home** contains the home for the system's root user (administrator)
- **/usr** contains files for programs that are intended for the user and are not vital to the core system
    - **/usr/bin** contains executable programs that are not essential to Linux

A link is a way to access a file. Most links that you will see on Linux are **hard links**, meaning that they link the specified name of the file and the *actual data of the file itself*. On the other hand, a **soft link**, also known as a **symbolic link**, stores the path to a file but does link to the actual file itself - you think of them as pointing to hard links.

- Soft links can link to a file or a directory. Hard links cannot link to directories (as this could create a loop in the tree stucture of the file system which then messes with any navigation commands like `find`).
- Deleting a soft link does not delete the original file. To delete the original file, you must delete all hard links to that file.

Linux is a multiuser operating system. The SEAS Linux server, as an example, contains the files and programs for all students in the class, each of whom have their personal directory. Linux has many mechanisms to prevent users from messing with other users' data. For example, only the **super user** has unrestricted access to the files of all users (to execute a command as a super user, you can do `sudo COMMAND` but this requires entering the password for the system). Another mechanism involves file permissions. In Linux, a file can be read, written, and/or executed, and the permission to do each of these things can be managed with

commands such as `chmod`. These permissions consider groups such as the **owner** of the file, members of a **group** that are associated to a file by the superuser, and all **other** users that don't fit the prior two definitions.

Linux is also a multiprocess operating system. Many processes can run in the background. You can run a command in the background by including a `&` after the command. You can put a background task in the foreground using the `fg PID` command, where PID is the process ID of the task (more on that later). Similarly, you can use `bg PID` to put a task running in the foreground to the background.

Commonly Used Commands

Commands are effectively just executable programs that use the shell as an interface. To run an executable file or command that is in the directory you are currently in, you must prepend that command with `./`. Commonly used commands are found through the system PATH, which is an environment variable that lists all locations that Linux should look for when a command is entered (i.e. if you use `cp` then the system will probably look for it in /bin or /usr/bin, both of which are already on the system's PATH). Running a command without `./` causes the system to search for that command in the PATH. When developing commands or executable programs meant to use system-wide, it is often useful to append that command to a directory that is already listed in the PATH or to append a new directory containing that command to the system's PATH so that it can be run from anywhere.

- Ctrl-C to stop a process
- Ctrl-Z to suspend a process

Piping and Redirection

- Commands can be chained together via piping. `COMM1 | COMM2` performs COMM1 and uses its output as the input for COMM2
- The contents of a file can also be used as an argument for a command through `COMM1 < FILE` which is known as input redirection. Most commands effectively do the same thing when they take a file argument.
    - `COMM <&-` closes the standard input (`&-` after a stream generally closes that stream)
- The output of a command can be redirected to a file through `COMM1 > FILE`. This inserts the output of COMM1 into FILE - if content already exists, then it is overriden. Overriding the file content can be avoided by using `>>` instead of `>`, which appends the content to the file instead.
    - `>` is the same as `1>`. This is the standard output stream and is typically where the program displays intended content.
    - `2>` is the standard error stream and is typically where the program displays error messages.
    - To redirect stdout to one file and stderr to another: `COMMAND 1> OUTPUTFILE 2> ERRORFILE`
    - To redirect stdout and stderr to the same file: `COMMAND > FILE 2>&1`. This redirects stdout to FILE and then redirects stderr to stdout, which is already going to FILE.
- Piping and redirection can be difficult to do for commands that may take multiple arguments, such as `comm`. In this case, it may be better to use **process substitution**, which takes the output of a process(es) and uses it as the input of another process. `COMMAND <(PROCCESS 1) <(PROCESS 2)`
    - Example: `diff <(ls DIRECTORY1) <(ls DIRECTORY2)` performs the diff command using the output of `ls DIRECTORY1` as one argument and the output of `ls DIRECTORY2` as another argument
    - This feature works in Bash but not in sh, so be careful when using it

Regex Overview

- Regular expressions (regex) are a sequence of characters that specify a search pattern in text. Many Linux commands such as `sed`, `grep`, etc. make use of regular expressions.
    - When using commands that make use of regular expressions, enclose the regular expressions in single quotations ' '. The shell will interpret certain special characters in double quotations (such as '$' and '{' and '}'), but it will treat anything in single quotations as literal (other than single quotes themselves), so there won't be interference with the shell's special characters and regex special characters
    - Most commands by default use Basic Regular Expressions (BRE), which treats special characters such as '?' '+', '{' '}', and '|' as literal (so they might literally match '?' instead of matching 0 or 1 instance of a pattern which '?' normally does in Regex) so to use them specially you must escape them with a '\'. Extended Regular Expressions do not treat these characters as literal, so to use them literally you must escape them with a '\'.
        - To use a command with Extended Regular Expressions, usually use the `-E` flag with the command.

- Syntax:
    - `CHARACTERS` will match those sequence of characters. These characters can include spaces, so `MATCH THIS` is different from `MATCHTHIS`.
        - i.e. `ful` will match the "ful" in "fruitful", "wishful", etc.
    - `[abc...]` will match any character that is in the brackets - a or b or c
        - Ranges can be specified `[a-z]` will match all lower case letters
        - Special type identifiers can also be used, such as `[[:upper:]]` (note the extra pair of brackets)
            - Common types: `[:upper:]`, `[:lower:]`, `[:alpha:]`, `[:digit]`, `[:blank:]`, `[:punct:]`
        - i.e. `[da*]` will match the 'd' in "drown", the entire word "dad", and the 'a' and '*' in "star*"
    - `[^abc...]` will match any character that is NOT in the brackets (the '^' must be at the beginning of the bracket expression as otherwise it will try to match '^' as an option)
    - `(...)` groups expressions
    - `a|b` matches either a or b. This can also be used for groups
    - `^` is an anchor representing the start of the line
        - i.e. `^start` will only match "start" if is at the beginning of a line
    - `$` is an anchor representing the end of the line
        - i.e. `end$` will only match "end" if it is at the end of a line
    - `.` matches any character except a newline
        - i.e. `.ad` will match "mad", "sad", "Dad", "@ad", but not "ad", "\nad" (where \n is a newline).
    - `(PATTERN)*` matches 0 or more instances of the preceding PATTERN
        - i.e. `fo*bar` will match 0 or more instances of 'o' since it is right before the '*', so "fbar", "fobar", "foobar", etc. matches
        - i.e. `(fo)*bar` will match 0 or more instances of 'fo' since it is grouped accordingly, so "bar", "fobar", "fofobar" match but "fbar" and "fofbar" does not.
    - `(PATTERN)+` will match 1 or more instances of the preceding PATTERN
    - `(PATTERN)?` will match 0 or 1 instance of the preceding PATTERN
    - `(PATTERN){NUM}` will match NUM instances of the preceding pattern

    - `(PATTERN){NUM, }` will match NUM or more instances of the preceding pattern
    - `(PATTERN){NUM1, NUM2}` will match between NUM1 and NUM2 instances of the preceding pattern

- To get information about any command, do `man COMMAND`, outputs a manual page for that command.

Navigation Commands

- `pwd` prints the full path of the current directory
- `cd [DIRECTORY]` changes the current directory to DIRECTORY.
    - Omitting the argument changes the current directory to the default home directory for the user, which can also be accessed using **~**. So, `cd` is the same as `cd ~`.
    - The DIRECTORY argument can be an absolute path (a directory starting from the root directory all the way to the current directory) or a relative path (a directory relative to the current directory)
    - Using **.** as an option changes the current directory to the current directory (so nothing at all happens basically). Using **..** as an option changes the current directory to its parent directory.
- `ls` lists the contents of the current directory
    - `-a` flag lists any hidden files in the directory (such as . or ..)
    - `-l` flag lists all files in the directory in long format (displays **permissions, number of links, owner, owner group, file size, modification date, and file name**)
    - `-r` flag recursively lists the directory tree (so it will list all the contents of subdirectories and their subdirectories)
    - `-t` flag lists files by modification time
    - `-S` flag lists files by file size
- `which COMMAND` specifies the file location of COMMAND, as well as any aliases for it
- `command --version` usually lists the version that the command (or executable program) is running on
- `find [PATH] [OPTION] [EXPRESSION]` starting from PATH, find files that meet the criteria specified in OPTION, and perform EXPRESSION those found files
    - Common Options (for any numberic arguments NUM, +NUM is used for arguments greater than NUM, NUM is used for arguments equal to NUM, and -NUM is used for arguments less than NUM):
        - `-type T` where T is a character representing the type (d for directory, f for regular file, l for symbolic link, etc.) will only find files that match the type T
        - `-name NAME` will only find files that match NAME (this is the name of the file not the name of the entire directory path to the file)
        - `-iname NAME` case insensitive -name
        - `-inum NUM` will find file that has inode number NUM
        - `-executable` will find files that are executable
        - `-regex PATTERN` will find the **WHOLE PATH** that matches the Regex PATTERN
        - `-mtime NUM` will find files that were modified NUM*24 hours ago
        - `-mmin NUM` will find files that were modified NUM minutes ago
        - `-links NUM` will find files that have NUM links
    - Common Expressions:
        - `-delete` will delete each file found
        - `-exec COMMAND {} \;` will execute the command COMMAND with each file found as an argument

File Management Commands

- `touch FILENAME` updates the modification of FILENAME if it exists and creates an empty file named FILENAME if it otherwise does not already exist
    - To actually edit a file, use a text editor such as Emacs, Vim, or Nano
- `mkdir DIRNAME` creates a directory originating in the current directory
- `rm NAME` removes the file NAME
    - `-r` will recursively remove the directory NAME, meaning that it will also remove everything under the directory **BE CAREFUL WHEN USING THIS**
- `rmdir DIRNAME` removes the empty directory DIRNAME; if DIRNAME is not empty, then the command fails
- `mv SOURCE DESTINATION` moves the file SOURCE to DESTINATION. If DESTINATION is a directory, the file will now a subdirectory of DESTINATION. IF DESTINATION is a file, then SOURCE will be renamed to SOURCE (if SOURCE already exists, then it will be overriden).
- `cp SOURCE DESTINATION` copies the file SOURCE to DESTINATION. If DESTINATION is a directory, the copied file will be a subdirectory of DESTINATION. Otherwise, a new file is created.
- `ln TARGET LINKNAME` creates a link to TARGET with LINKNAME. Creates a hard link by default
    - `-s` flag creates a a symbolic link DESTINATION will be copied to the current directory.

System Management Commands

- `chmod PERMISSIONS FILE` changes the permissions of FILE to PERMISSIONS
    - Permissions can easily be set using the symbolic mode, where the character representing the group (u = owner, g = group, o = other, a = all), a symbol (+ for adding permissions, - for removing permissions, or = for setting permissions exactly as written), and a character representing the permission (r = read, w = write, x = execute) are all specified
        - i.e. `chmod u+rwx` gives reading, writing, an executing permissions from the user
        - i.e. `chmod ug-x` removes executing permissions from the user and group
        - i.e. `chmod u=rwx,g=rx,o=r`
    - Permissions can also be set up in octal mode, specifying a three-digit number with each digit between 0 and 7. The first digit is the permissions for the owner, the second digit is the permissions for the group, and the third digit is the permissions for other.
        - 0: No permissions (---)
        - 1: Execute permission (--x)
        - 2: Write permission (-w-)
        - 3: Execute and write permission (-wx)
        - 4: Read permission (r--)
        - 5: Read and execute permission (r-x)
        - 6: Read and write permission (rw-)
        - 7: All permissions (rwx)
        - i.e. `chmod 420 testfile` will give the user reading permission, the group writing permission, and other users no permissions (r---w----)
- `ps` outputs the current processes. By default, it outputs the processes running in the current shell. The output format is as follows: **Process ID, Terminal Type, Time Running, Command Launching Process**
    - `-e` flag outputs all running processes
    - `-T` flag outptus all processes associated with the terminal
    - `-H` flag outputs a tree for each process (its subprocesses)

- `df` displays a report on the system disk usage
- `kill SIGNAL PID` sends the SIGNAL to the process having the process ID PID
  - `-9` signal is the signal to kill
  - `-15` signal is the signal to terminate
- `zip ZIPFIlE FILE1 ... FILE2` zips FILE1 ... FILE2 into ZIPFILE
- `unzip ZIPFILE` unzips ZIPFILE

Useful Filtering Commands

- `echo ARGUMENT` prints out ARGUMENT (does not read argument)
  - This will also print out shell expansions (i.e. `echo *` will print out all items in the current directory since '*' is expanded by the shell)
  - `echo` is usually used to pipe in a specified text to another command or file (i.e. `echo "Hello" | grep 'ello'`)
- `cat FILE1...` concatenates multiple files (if they are listed) and then prints them out. If there are no arguments, then standard input is read
  - `cat` might be used in the context of a script to pipe into another command (i.e. `cat | head` will use the standard input as an argument for head)
- `tac FILE1...` does the same as cat, but prints the text in reverse-line order
- `grep PATTERN FILE...` prints the lines in FILE(s) that match the specified PATTERN (in regex)
  - `-E` flag uses Extended Regular Expression (ERE)
  - `-v` flag prints out all lines that do NOT match the pattern
  - `-c` flag prints only a count of lines that match the pattern
  - `-i` flag performs a case insensitive match
  - `-n` flag displays the matched lines and their line numbers
- `sort FILE` sorts the file by line
  - `-b` flag ignores leading blanks
  - `-d` flag considers only blanks and alphanumerical characters
  - `-f` flag ignores case (converts to uppercase)
  - `-g` flag compares according to numerical values (as opposed to string comparisons)
  - `-r` flag reverses the sort
- `head -n NUM FILE` prints out the first NUM lines (just `head` defaults to the first 10)
- `tail -n NUM FILE` prints out the last NUM lines (just `tail` defaults to the last 10)
- `wc FILE` prints out the number of lines, words, and bytes of FILE
  - `-l` flag just prints out the number of lines
  - `-w` flag just prints out the number of words
  - `-c` flag just prints out the number of bytes
  - `-m` flag just prints out the number of characters
- `tr SET1 SET2 FILE` translates the characters from SET1 to SET2 in FILE. Translate works well with replacing single characters with single characters. Outputs the edited version of FILE.
  - To specify a range of characters, just use a - (i.e. `A-Z`).
  - `-C` flag complements the characters in SET1
  - `-s` flag squeezes characters replaced via SET2 so that they don't repeat
- `sed SCRIPT FILE` stream editor that acts according to SCRIPT for instances in FILE, where SCRIPT can contain regex patterns; outputs the edited version of FILE
  - `sed` is a scripting language with certain commands; `p` prints, `q` exits the editor, `s` substitutes (using `/` as a delimiter between the item being substituted and the substitution i.e.

`s/REPLACEME/WITHTHIS/`), `d` deletes
    - `-E` flag uses Extended Regular Expressions
    - `sed 's/PATTERN/PATTERN2/'` replaces the first occurence in the line of PATTERN with PATTERN2 `sed 's/PATTERN/PATTERN2/n'` replaces the nth occurence in the line of PATTERN with PATTERN2
    - `sed 's/PATTERN/PATTERN2/g'` replaces all occurences of PATTERN with PATTERN2
    - `sed 'NUM s/PATTERN/PATTERN2/'` replaces the first occurence only on the NUM line
    - `sed s/PATTERN//g` deletes all instances of PATTERN
    - `-n` flag prints out just the lines that have been altered
- `comm FILE1 FILE2` compares FILE1 and FILE2, assumed to be sorted, line by line and outputs in a three-column format: (1) lines unique to FILE1, (2) lines unique to FILE2, (3) lines contained in both files
    - `num` flag suppresses the specified column (i.e. `-12` only shows lines shared by both files)
- `uniq FILE` assuming FILE is sorted, prints out FILE with duplicate lines filtered out
    - `-d` only prints lines which were repeated
- `diff FILE1 FILE2` outputs differences between FILE1 and FILE2.

Shell Scripting

- Shell scripting essentially boils down to making use of the shell's existing commands to create other scripts, as well as basic computer science structures such as variables, comparisons, and loops.
- When creating a shell script, include at the top of the script `#!/bin/sh` or `#!/bin/bash`, which indicates what environment the system should execute the script in (sh or Bash)

Expansions

- The shell will expand certain characters if they are not in quotes. Use single quotes to prevent characters from being expanded and instead literally interpret them (except for ' itself). Double quotes will also literally interpret most characters except for '$', '`', and ''.
- One form of expansion done by the shell is globbing, which looks for pattern (similar to regex, but not the same) in filenames. This involves the characters '?', '*'
    - `?` is used to match a single character
        - i.e. `echo exer?.html` will output exer1.html, exer2.html, exer3.html if they exist in the directory
    - `*` is used to match zero or more characters
        - i.e. `echo *.txt` will output all files in the directory that end in ".txt"
    - `[SET]` is used to match characters in the range SET (i.e. `[A-Z]`)
        - i.e. `echo exer[1-9].html` would output exer1.html, exer2.html, exer3.html if they are in the current directory
        - An '!' at the beginning of the set indicates a complement (i.e. `[!1-9]`)
- Variables (more on that next section) are expanded using the '$' character preceding the variable name. A '$' in a non-quoted or double-quoted expression will be expanded. In a double-quoted expression the expansion can be escaped with a `\`. In a single-quoted expression, the '$' character will be treated literally and will not expand
    - i.e. `x=2`
    - `echo $x` => `2`
    - `echo "$x"` => `2`
    - `echo '$x'` => $x

- The '~' character is also expanded to the user's home directory
- The output of a command can also be expanded into an expression using either backticks `COMMAND` or a dollar-sign followed by parentheses $(COMMAND)
    - i.e. `echo `ls`` => echoes the output of ls, which is the contents of the current directory
- Braces can be used to expand a range
    - i.e. `echo {1..10}` will output 1 2 3 4 5 6 7 8 9 10

Variables

- Variables are created and assigned a value in the format `VARNAME=content`. There must be no space in between the equals sign. The variable content can be any type - a string, an int, a floating point number
- The value of a variable can be obtained using the '$' character followed by the variable name (no space) `$VARNAME`
    - This can also be done using `${VARNAME}`, which acts as a disambiguation mechanism
        - cases such as `echo $VARNAME.txt` where the intention is to replace just the VARNAME portion with the variable will fail, so the bracket grouping should be used instead as `echo ${VARNAME}.txt`
- The shell has built in variables:
    - `$?` is the last command or program's return value
    - `$$` is the current script's process ID
    - `$#` is the number of arguments passed to the script
    - `$@` is all arguments passed to the script
    - `$1`, `$2`, `$3` ... `$9` are the positional arguments passed to the script (i.e. running a made script `./script 'first' 2 "three"` will have the positional arguments 'first', 2, and "three")
    - `$0` is the name of the script
- Variables can be assigned from standard input using the `read` command, which reads a line from standard input into the variable
- Normally, variables are scoped within the script that they are contained. A script or process can inherit a variable from its parent process or script using the `export VARNAME` command.

Comparisons

- The `test` command, also known as `[` is used to perform any sort of comparison. Comparisons are usually in the syntax `[ ARG1 ARG2 ARG2 ]`. Note that the spaces are important.
    - Common Tests:
        - `=` string equality
        - `!` logical not
        - `-eq` numeric equality
        - `-ne` numeric inequality
        - `-lt` less than
        - `-le` less than or equal
        - `-gt` greater than
        - `-gte` greater than or equal
        - `-z` string is zero length
        - `-n` string is not zero length
        - `-nt` newer than
        - `-d` is directory

        - `-f` is file
        - `-r` is readable file
        - `-w` is writable file
        - `-x` is executable file
        - `&&` logical AND
        - `||` logical OR
    - i.e. `[ "$x" -lt "0" ]` checks if x is less than 0
- If statements involve the test command, a then, and a fi to close the if. You can also use else statements.

```
if [ TEST ]
then
  IF_CODE
elif [ TEST ]
  ELIF_CODE
else
  ELSE_CODE
fi
```

    - Colons `;` can be used to join together lines to make more readable syntax:

```
if [ TEST ]; then
  IF_CODE; elif
elif [ TEST ]; then
  ELIF_CODE
else
  ELSE_CODE
fi
```

Loops

- Looping conventions are pretty similar to other program languages. Loop control statements such as `break` and `continue` can be used when shell scripting.
- 'while' loops have pretty straightforward syntax:

```
while [ TEST STATEMENT ]
do
  CODE
done
```

        - `while :` always evaluates to to true, so this can be used as a way to do infinite looping (or have a loop dictated by a `break` exit condition)
- 'for' loops operate on a list of arguments

```
for VARIABLE in LIST
do
  CODE
done
```

    - i.e. `for i in 1 2 3 4 5` will iterate through 1, 2, 3, 4, and 5
    - i.e. `for i in {1..5}` will do the same as above
    - i.e. `for i in $(seq 1 5)` will do the same as above
- The `do` statement in both above loops can be put immediately after the `for` clause if separated with a semicolon `;`