|  |
| --- |
| Git commands that are fundamental to the Git workflow such as merging, rebasing, pulling, fetching, and so on, and basic concepts of branches. |

nano/touch *testing.txt*

git add *testing.txt*

git commit -m “Add testing.txt” / git commint -am “Add testing.txt”

git branch *testing*

git checkout *testing*

git log --oneline --decorate

git log --oneline --decorate --graph --all

git checkout master

git merge testing

git status

* If you want to see which files are unmerged at any point after a merge conflict, you can run git status

git branch

git branch -v

* To see the last commit on each branch

git branch --merged

* To see which branches are already merged into the branch you’re on

git branch -d

git branch -D

git ls-remote [remote]

git remote show [remote]

* For remote branches as well as more information

git remote -v

git fetch <remote>

* To synchronize your work with a given remote, you run a git fetch <remote> command (in our case, git fetch origin). This command looks up which server “origin” is (in this case, it’s git.ourcompany.com), fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.

git pull = git fetch + git merge FETCH\_HEAD

git push --set-upstream origin master

원격저장소 브랜치

|  |
| --- |
| Slides ~64/87  For the Unix/Linux part of the assessment, you should be able to use  • The filters covered so far:  cat, sort, head, tail, uniq, grep (but only through slide 40)  • The other commands covered:  echo, basename, dirname, find, ls, seq  • and I/O redirection operators |

$whoami

* *whoami* is an unusual command - it displays your username just in case you forgot

$date

* *date* command displays the current time and date

$echo

* *echo*displays the strings that follow it on the command-line

$wc

* *wc* is short for **w**ord **c**ount; it displays the number of lines, words, and characters in the file(s) whose names follow it on the command-line

$ls

* *ls* command displays the contents of a directory

$which ls

* If a command is an executable, the *which* command will display the (absolute) pathname of that executable

$type pwd cal

* *type*command (which is built-in) displays the type(s) of its arguments

$man ls

* To access the manual page for a command, type man followed by the command name

File and Directory Processing

***ls***

|  |
| --- |
| · This lists information about files and directories  · Syntax: ls [option] … [file] …  · If no files are given, it lists information about the current directory  · Some very useful options:  -l produce a “long” listing  -a show all files, including hidden ones  -F append trailing character to classify type  -t sort by time of last modification, most recent first 최송 수정 시간을 기준으로  -R recursively list subdirectories 하위의 서브디렉토리의 내용도 순차적으로 표시  -u 최종 수정 시간 대신 최종 액세스 시간(-t 또는 -l과 함께 사용)  [Examples]  · List all files in current directory with most recent first, classifying  **$ls -atlF**  · List all files sorted by access time  **$ls -atulF** |

***basename***

|  |
| --- |
| · It strips leading directory path and suffix from filenames  · Syntax: basename pathname [suffix]  · It outputs pathname with the leading path removed and, if it has a suffix matching suffix with  that removed also  [Examples]  **$basename /usr/bin/sort** sort  **$basename include/stdio.h .h** stdio \* .h = 뒤에 파일명 제거하는 거  **$basename include/stdio.h .foo** stdio.h *# i.e., suffix does not match*  **$basename ~/hunter/cs395.86\_s19/blogs/stewart-weekly -weekly** stewart |

***dirname***

|  |
| --- |
| · It strips the filename from a pathname, leaving the directory path  · Syntax: dirname pathname [pathname] …  · It outputs each pathname with its last non-slash component and trailing slashes removed  [Examples]  **$dirname /usr/bin/sort** /urs/bin  **$dirname /usr/bin/gcc /urs/lib/gcc /urs/share/man/man1/gcc.1.gz**  /usr/bin  /usr/lib  /usr/share/man/man1  **$dirname ~/hunter/cs395.86\_s19/blogs/stewart-weekly**  /home/stewart/hunter/cs395.86\_s19/blogs \* ~는 실제 홈폴더 경로로 변경됨 |

***find***

|  |
| --- |
| · It allows you to apply commands and actions to all files matching a set of search criteria in one or more subtrees of the Unix file system  · Syntax: find [starting-point…] [expression]  · starting-points are the directories to act as the roots of the hierarchies to search  · expression describes what is to be searched for; it includes search criteria as well as actions to perform  <Finding by file name>  **$find . -print = $find .**  · With no expression, find displays every file in the trees rooted at the starting-points  The default action is -print  **$find dir -name “\*.cpp”**  · Finding (printing paths to) all files in the directory dir whose name ends in .cpp  · Lesson: The \* is a shell wildcard that matches 0 or more characters, including the period if it is the first  character in the name  **$find dir -name “main.cpp”**  · Finding all files in dir whose name is exactly main.cpp  **$find dir -iname “\*.jpg”**  · Finding all files in dir whose name ends in any of .jpg, .JPG, .JPg, etc  · Lesson: -iname is a case-insensitive version of -name  **$find dir -iname “\*.jpg” -o -iname “\*.jpeg”**  · Finding all files in dir whose name ends in any of .jpg, .JPG, .jpeg, .JPEG, etc  · Lesson: Expressions return true or false. -iname is a test applied to each file as it is found. If the filename  matches, it returns true otherwise false. The -o is a logical OR-operator; its operands above are -iname  "\*.jpg" and -iname "\*.jpeg". If either is true then the filename passes the test.  <Finding by time stamp>  · Unix timestamps files with three stamps: time of last access, time of last modification, and time of last  change of status (file properties)  **$find dir -mmin -180**  · Finding files in dir that have been **modified** within the past 3 hours  · Lesson: -mmin expects an argument in minutes. The - in front of 180 means "less than"  **$find idr -mmin +180**  · Finding files in dir that have been modified more than 3 hours ago  **$find dir -amin 180**  · Finding files in dir that have been **accessed** (use -amin) exactly 3 hours ago  **$find dir -cmin -480**  · Finding files in dir whose **status changed** (use -cmin) within the past 8 hours: |

Continued on ***find***

|  |
| --- |
| <Finding by other properties>  · In Unix, properties such as size, type, permissions, user ownership, group ownership, and more, are stored  in a special structure called an inode. find can test any of these properties  **$find dir -size +500k**  · Finding files in dir larger than 500 Kilobytes (1024 bytes)  · The letter after the number can be c, w, k, M, or G. Guess what they stand for  **$find dir -executable**  · Finding files in dir that are executable  **$find dir -user stewart -a -group cs\_ossd**  · Finding files in dir that are owned by user stewart and group cs\_ossd  · Lesson: -a is the logical AND-operator  **$find dir -perm -002**  · Finding files in dir for which people other than the owner and the group have write access, whether or not  the owner or group does.  · It is dangerous to let anyone be able to write to a file. This looks for all such files.  **$find dir -perm 700**  · Finding files in dir for which the owner has read, write, execute permission and no one else has access of  any kind  **$find dir -type f**  · Finding files in dir that are regular files  · To find directories, replace f with d  <Taking actions when files are found>  · You can use add **actions** to expressions. These actions can be applied to the files for which the test returns true, or to a set of arguments that follow the action.  · Useful actions include -print, -prune, and -exec. There are many others.  -prune is used to prune the search, i.e., prevent it from descending the tree  -exec <command> executes the that follows it  **$find . -type f -exec file ‘{}’ \;**  · Run the file command on every regular file below the current directory  · **Lesson**: exec is followed by a command. {} after the command is replaced by the file that matched the test.  It must be written in quotes, and the semicolon must be escaped with backslash as shown.  **$find . -name “\*~” -execdir /bin/rm ‘{}’ \;**  · Remove every file whose names ends in ~ below the current directory (dangerous if you make a mistake) |

I/O Redirection Operations

***File Descriptors and Streams***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| · The index values in the array of open files belonging to the process are called **file descriptors**. The table  below summarizes the different views of the standard streams.   |  |  |  |  | | --- | --- | --- | --- | | File Descriptor | Stream | Associated Device | C Symbolic Name | | 0 | Standard Input | Keyboard | stdin | | 1 | Standard Output | Screen or Terminal Window | stdout | | 2 | Standard Error | Screen or Terminal Window | stderr | |

***Input Redirection***

|  |  |  |
| --- | --- | --- |
| · All shells allow for any stream to be disconnected from its default device and reconnected to or from files,  the streams of other processes, or other devices. This is called **I/O redirection**.  **$command < file**  · Attaching a file to the standard input is called **input redirection**. The < operator is the **input redirection**  **operator**  [Example]  **$cat < file1**  · The contents of file1 are redirected to the standard input of the cat command, which is not very useful, but  illustrates the idea  · The bc command can be used as a simple calculator. If you type bc on the command line, it waits for you to  enter an arithmetic expression such as 25 + 4 or something more complex. When you type it evaluates and  prints the expression.  $ 6 + 4  10  $ 12 ^ 2  144  · Suppose file1 contains the sequence of arithmetic expressions   |  | | --- | | 6 + 4  12 ^ 2 |   **$bc < file1**  · Run the command bc < file1 at the command prompt   |  | | --- | | 10  144 |   · This shows that file1 replaced bc's standard input |

***Output Redirection***

|  |
| --- |
| **$command > file**  · Attaching a file to the standard output is called output redirection. The > operator is the output  redirection operator  [Example]  **$cat file1 file2 file3 > combined\_file**  · Concatenates files file1, file2, and file3 into a new file named combined\_file.  **$bash: combined\_file: cannot overwrite existing file**  · If combined\_file already existed and the bash variable noclobber is set, the command fails and you will see  an error message.  **$cat file1 file2 file3 >| combined\_file**  · To overcome this, use the >| operator, which will forcibly replace the existing file |

***Error Redirection***

|  |
| --- |
| **$command 2> file**  · Attaching a file to the standard error is called **error redirection**. The > operator is also used to redirect standard error, but with a slight modification  · **No space** between 2 and >. 2 is the file descriptor for the standard error stream. In general, if *n* is a file descriptor, *n*> redirects the stream associated with it to the  **$command 2> error\_file > output\_file**  · To send the errors of a command to one file and the standard output to another, above is the simplest solution (there are a few ways to do this).  **$command 2>| error\_file >| output\_file**  · If either file exists and noclobber is set, it will fail; abobe is to overwrite them    **$command &> file**  · To redirect the standard output and the standard error to the **same** file |

***Redirection Examples***

|  |  |
| --- | --- |
| **$ls -R / 2>/dev/null > verybigfile**  · Some commands can produce many error messages. Sometimes you don't care about the errors. If you try  to display the *entire file system* (bad idea) using ls -R there will be many permission denied errors. The way  to discard them is like above.  · **Lesson**: /dev/null is a black hole; any data written to it is discarded.   |  | | --- | | $ cat > notes  blah blah blah  ^D (Control D to stop)  $ cat notes  blah blah blah |   · We can take advantage of the fact that **the cat command reads from standard input if it is not given a filename as an argument**, to create a plain text file while taking notes.  · We must type Control-D to terminate the cat command and close the new file named notes. |

***Appending***

|  |
| --- |
| **$command >> file**  · To append to a file (or to create it if it does not exist) use the append redirection operator >>.  · which adds the output of the command to the end of the file.  **$backup 2>> ~/.backup\_errlog >> ~/.backuplog**  · Sometimes you will find it useful to log the results of a command that you run, or perhaps log its errors.  Suppose backup is some backup command that you run every day.  · which sends the errors of the command to one log file and the standard output to another. |

Pipes

|  |
| --- |
| · Attaching the standard output of one command to the standard input of another is done by creating a **pipe**. The | operator is the **pipe operator**.  · Syntax**: $command1 | command2**  · The real power of pipes is that they can be used with **filters**. Examples of pipes will follow the introduction of filters. |

Filters

***cat***

|  |  |  |
| --- | --- | --- |
| · “concatenates and prints" its list of text file arguments. It whizzes by on the screen, so use it  only to see small files  · The cat command is a technically a filter, but without options it does no transformation: its output is  exactly its input.   |  | | --- | | $ls . | cat -n  1 bash\_tutorial\_01.html  2 css/  3 img/  4 js/ |   · It does have some handy uses. The -n option numbers lines, -b numbers non-blank lines, -s *squeezes* blank  lines, and -v shows non-printing characters.  · lists the current directory and numbers the lines.   |  | | --- | | **$ls . | cat -n | tac**  4 js/  3 img/  2 css/  1 bash\_tutorial\_01.html |   · tac prints the lines in reverse order. I rarely have use for it. |

***head*** *and* ***tail***

|  |
| --- |
| · Simply put, head displays the first *N* lines of its input and tail, the last *N* lines. By default for both, *N* is 10.  To print a different number of lines, explicity use -N where *N* is a positive integer.  **$head -1 myfile**  · It displays just the first line of myfile  **$tail -1 myfile**  · It displays the last line.  **$head -4 myfile | tail -1**  · One way to print the *nth* line of a file  · It prints the 4th line of myfile.  **$man awk | head -4 | tail -1**  · To get the summary of any command from the man page for it since the summary is always line 4. |

***sort***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **$sort myfile**  · It will sort the text file named myfile and print it on standard output.  · But its exact behavior varies from one system to another; there are many different implementations  of sort.  · By default it uses the current locale settings (the *collating order* set in your environment\*). The major  difference in English is whether uppercase precedes lowercase or whether case is ignored.  · Locales are a subject way ahead of us. Suffice it to say that locale environment variables control how  information is displayed in the terminal, such as the language and character set, numbers, dates, times,  and more.  [Examples]  · Assume myfile contains the lines   |  | | --- | | a  b  A  B |   · Forcing the locale to be U.S. English and running sort  **$LC\_COLLATE="en\_US.UTF-8" sort myfile**   |  | | --- | | a  A  b  B |   · Forcing the locale to be the "C" locale and running sort  **$LC\_COLLATE=C sort myfile**   |  | | --- | | A  B  a  b |   · To force sort to ignore case and fold upper and lower case together, using GNU's sort, the option is -f.  **$LC\_COLLATE=C -f sort myfile**   |  | | --- | | a  A  b  B |   · sort by default will treat numbers like strings. For example, it will sort 1, 2, 10, 20 in this order: 1, 10, 2, 20.  · To tell sort to sort numerically, use -n.  · To tell sort to **reverse** its order, use -r.  · To tell sort to delete duplicate lines on output, use -u.  · sort can sort by **fields** in a line.  The GNU version uses the -k option to specify the specific key position (1- based).  The -t option tells sort what character keys in the line. By default sort uses whitespace. If a file has colon-  separated fields, and you want to sort numerically by field 2, use  **$sort -t':' -k2 -n myfile** |

***uniq***

|  |
| --- |
| **$sort mydata | uniq**  · uniq filters out matching adjacent lines from its input stream, sending unique lines to output. If the input  stream is sorted and has duplicates, this produces output with duplicates removed.  · It produces a stream of unique lines from the file mydata.  · It has several useful options as well:  -i - ignore case when trying to match  -c - prefix lines by the number of occurrences  -d - only print duplicate lines, one for each group  -u - only print the unique lines |

***Combining Filters***

|  |
| --- |
| · Putting some of this filters together, NYC Open Data has a database of baby names whose lines look like  this:  2011,FEMALE,HISPANIC,Geraldine,13,75  2011,FEMALE,HISPANIC,GIA,21,67  2011,FEMALE,HISPANIC,GIANNA,49,42  ...  **$sort -k4 babynames.csv | cut -d, -f4 | uniq -ic**  · Many names are repeated. We can get the frequencies of each name using this pipeline, assuming the file  is named babynames.csv:  · Where sort sorts using the 4th field, after which cut\* prints only the 4th field, then uniq filters out  duplicates and puts counts to the left of the names, ignoring case.  **$sort -k4 babynames.csv | cut -d, -f4 | uniq -ic | sort -k1 -nr | head -10**  · We can get the ten most frequent names like this. |

***grep***

|  |
| --- |
| · grep and egrep are given a pattern, called a **regular expression**, and use this to filter lines that match the  pattern. fgrep is a *fast*, fixed-string version that does not use patterns.  · Regular expressions are complex; mastering them is worth the effort because they are used  by vi, sed, ed, awk, grep, and egrep.  [Examples]  **$grep '\<cout\>' prog.cpp**  · It prints all lines containing the exact word cout in file prog.cpp.  **$grep -c '^ \*$' prog.cpp**  · It prints a count of the number of lines in prog.cpp that contain only blanks or no characters at all.  **$grep '\/\\*.\*\\*\/' prog.cpp**  · It prints all lines in prog.cpp that have C-style comments /\* ... \*/.  · **Lesson**: Enclose the pattern in single quotes to prevent the shell from interfering. |

***Regular Expression Rules***

|  |
| --- |
| · In the rules that follow, \0 denotes an empty string.  · The complete set of rules can be found in the **regex** man page in section 7, which defines the POSIX-  compliant regular expressions.  · Any sequence of characters matches itself: **abc** matches the string "abc".  · A regular expression followed by \* matches the concatenation of 0 or more strings each of which is  matched by the regular expression. \* is called the **closure** operator.  **- a\*** matches 0 or more a's:  \0, a, aa, aaa, ...  **- ab\*** matches a followed by 0 or more b's:  a, ab, abb, abbb, ...  **- ab\*ac\*** matches a followed by 0 or more b's followed by a followed by 0 or more c's:  aa, aba, aac, abba, abac, aacc, abbba, abbac, ...  · Use **\( \)** to group for applying \* to more than one character:  **- \(ab\)\*** matches 0 or more ab's  \0, ab, abab, ababab, ...  **$egrep '(ab)+' myfile**  **$grep -E '(ab)+' myfile**  · If you use **extended regular expressions** either by writing grep -E or by using egrep instead, you can use +,  the **positive closure** operator. It matches **1 or more** strings each of which is matched by the preceding  regular expression. You can also use ordinary parentheses for grouping.  · both match all lines in myfile that have 1 or more consecutive ab substrings.    **$grep '(ab)\*' myfile**  · Warning  · It only matches \0, (ab), (ab)(ab), ... because grep was used instead of egrep. |

Miscellaneous

***seq***

|  |
| --- |
| · It generates and prints sequences of numbers  · Its general form: $seq first increment last  · If first or increment are omitted, they default to 1. The separator is a newline, but the -s option  can be used to give it a different separator:  $ seq -s ‘ ‘ 1 10  1 2 3 4 5 6 7 8 9 10  $ seq -s ‘-‘ 10 2 20  10-12-14-16-18-20  $ seq -s “--” 20  1--2--3--4--5--6--7--8--9--10--11--12--13--14--15--16--17--18--19--20 |

***echo***

|  |
| --- |
| · It displays the words that follow it on the command line, as in:  $ echo “The future will be better tomorrow.”  The future will be better tomorrow  · It is useful because bash will do variable evaluation before giving the words to echo, as in:  $ echo “My username is $USER and I am using the $SHELL shell.”  My username is stewart and I am using the /bin/absh shell. |