**UNX510/DPS918 - Unix BASH Shell Scripting**

**Lecture 2 - Introduction to Scripting; More Commands**

**Introduction to Scripting**

* a shell script is simply a collection of commands in a file, similar to a DOS bat file
* to run a script by just using the script name:
  + the script must have r and x permission
  + the script must be in a directory that's included in your PATH, otherwise a path must be provided
  + a directory for scripts can be included in your PATH, in a startup-script, so that the scripts can be easily executed regardless of your current directory, for example:

PATH=$PATH:~/scripts

* you may specify which shell the script should run in
  + #!/bin/bash
    - must be first line in script, starting in first column, specifying absolute path of the required shell
    - ensures that script will run in correct shell regardless of which shell user is using
    - if not used, script runs in current shell type
    - on any other line, "#" indicates a comment - provides information about program, but is ignored by shell
* here is a typical structure for a script:
* #!/bin/bash
* # Program name: firstShellScript
* # Author name: Joe Student
* # Date created: Oct. 9, 2011
* # Date updated: May 9, 2013
* # Description:
* # This is a short script used to
* # demonstrate basic scripting principals.
* # It simply displays two lines of output.
* echo "Hello, you have successfully run your first script."

echo "Congratulations!"

* + to run this, give it execute permission
  + then enter firstShellScript or ./firstShellScript if the current directory is not in your path
* another simple sample script:
* ==> cat sample.script # Display script
* echo "This script was executed on:"
* date
* ==> sample.script # Run script
* This script was executed on:
* Mon Jan 25 21:44:58 EST 2010
* ==> \_

**Variable Assignment**

* user-created variable names must begin with a letter
  + eg. school=Seneca (no spaces around = )
* precede variable name with $ to retrieve value - called variable substitution
* ==> school=Seneca
* ==> echo "My school is $school"
* My school is Seneca
* ==> \_

* read will read one line from standard input and assign it to variables, for example:
* ==> cat read.example
* echo -n "Please enter your name: "
* read name
* echo "Hello $name"
* ==> read.example
* Please enter your name: Josephine Student
* Hello Josephine Student
* ==> \_

* read var1 var2 var3 - will read one line and assign first word to var1, second word to var2, and remaining words to var3

**Command Substitution**

* `cmd` - will use the output of a command as a string (note the backward single quotes)
* ==> currentDirectory=`pwd`
* ==> echo "My current directory is $currentDirectory"
* My current directory is /home/lczegel/test
* ==> echo "Today is `date`"
* Today is Mon Oct 10 17:31:41 EDT 2011
* ==> \_

* Bash and Korn shell also allow $(cmd) - allows nested substitutions
* ==> who | wc -l
* 27
* ==> echo "There are $(who | wc -l) users on the system"
* There are 27 users on the system
* ==> whoami
* lczegel
* ==> grep ^$(whoami): /etc/passwd
* ==> ypcat passwd | grep ^$(whoami):
* lczegel:x:6444:500:Les Czegel:/home/lczegel:/bin/bash
* ==> echo "My home directory is $(ypcat passwd | grep ^$(whoami): | cut -d: -f6)"
* My home directory is /home/lczegel
* ==> \_

* + this example was done on Matrix, so "ypcat passwd" must be used because /etc/passwd does not contain ordinary userids
  + ypcat displays the values of keys from the NIS (Network Information System) database
  + ypcat -x displays the available keys
  + ypmatch can be used to display records with specific keys, for example:
    - ypmatch lczegel passwd gives similar results to "ypcat passwd | grep lczegel"

**Quoting**

* one reason for quotes is so that string values can contain spaces and special chars
* three styles of quoting:
  + single quotes ' ' (strong quotes - don't allow variable and command substitution)
  + double quotes " " (weak quotes - allow variable and command substitution)
  + backslash \ (quotes the next character only)
* quoting examples:
* ==> school=Seneca # Assign a variable
* ==> mySchool=my school is $school # This will cause an error
* bash: school: command not found
* ==> mySchool="My school is $school" # Weak quotes, allowing substitution
* ==> echo $mySchool
* My school is Seneca
* ==> mySchool='My school is $school' # Strong quotes, not allowing substitution
* ==> echo $mySchool
* My school is $school
* ==> mySchool=My\ school\ is\ $school # Equivalent of weak quotes
* ==> echo $mySchool
* My school is Seneca
* ==> mySchool=My\ school\ is\ \$school # Equivalent of strong quotes
* ==> echo $mySchool
* My school is $school
* ==> dirListing=$(ls -l) # Command substitution
* ==> echo $dirListing # Spacing is not preserved
* total 0 -rw------- 1 lczegel users 0 Oct 10 17:54 file1 -rw------- 1 lczegel use
* rs 0 Oct 10 17:54 file2 -rw------- 1 lczegel users 0 Oct 10 17:54 file3
* ==> echo "$dirListing" # Spacing is preserved
* total 0
* -rw------- 1 lczegel users 0 Oct 10 17:54 file1
* -rw------- 1 lczegel users 0 Oct 10 17:54 file2
* -rw------- 1 lczegel users 0 Oct 10 17:54 file3
* ==> \_

**Variables**

* $0 is name used to execute script, including any path specified
* $1 to $9 are first nine positional parameters, or command line arguments
* ==> cat arguments.example1 # Display script
* echo "\$1 is $1"
* echo "\$2 is $2"
* ==> arguments.example1 lion tiger bear # Run script
* $1 is lion
* $2 is tiger
* ==> arguments.example1 elephant # Run script again
* $1 is elephant
* $2 is
* ==> \_

* can access beyond $9 by using set braces, eg. ${10} will access 10th argument
* $\* represents all parameters, as a single string
* $@ represents all parameters, as separate strings
* $# contains number of parameters
* shift shifts parameters left, so that the first one (which was $1) disappears, the second one which was $2 becomes $1, etc.
* example of $#, $\*, and shift :
* ==> cat arguments.example2 # Display script
* echo "\$1 is $1"
* echo "\$2 is $2"
* echo "\$\* is $\*"
* echo "\$# is $#"
* shift
* echo "After shift:"
* echo "\$1 is $1"
* echo "\$2 is $2"
* echo "\$\* is $\*"
* echo "\$# is $#"
* ==> arguments.example2 lion tiger bear # Run script
* $1 is lion
* $2 is tiger
* $\* is lion tiger bear
* $# is 3
* After shift:
* $1 is tiger
* $2 is bear
* $\* is tiger bear
* $# is 2
* ==> \_

* shift nn, where nn is a number, has the same effect as that number of shift commands
* set with arguments sets $n variables, eg. set a b c sets $1 to a, $2 to b, and $3 to c, sets $4 etc. to null
* ==> set lion tiger bear
* ==> echo "\$1 is $1"
* $1 is lion
* ==> echo "\$2 is $2"
* $2 is tiger
* ==> \_

* of course, if any original script arguments are still needed, they should be saved into other variables before the set command is executed
* ${var} can be used instead of $var - useful if followed by alphanumerics - eg $var35 vs. ${var}35
* $$ is the PID (process ID) number of the current process - useful for naming temporary files
  + useful for multiple users running the same program, because PID is unique for each user
  + also, use a directory such as "/tmp" - everyone has read/write permission, and the system administrator will usually make sure that there is lots of disk space available
* ==> echo $$
* 18118
* ==> ls -l > /tmp/output.$$
* ==> ls -l /tmp | grep $(whoami)
* ==> ls -ld /tmp
* lrwxrwxrwx 1 root root 9 Feb 27 2007 /tmp -> /var/tmp/
* ==> ls -lL /tmp | grep $(whoami) # -L dereferences symbolic link
* -rw------- 1 lczegel users 152 Oct 10 18:42 output.18118
* ==> rm /tmp/output.$$
* ==> \_

* $? is the exit status of the last command, 0 means successful
* ==> cat cars
* plym fury 77 73 2500
* chevy nova 79 60 3000
* ford mustang 65 45 17000
* volvo gl 78 102 9850
* ford ltd 83 15 10500
* Chevy nova 80 50 3500
* fiat 600 65 115 450
* honda accord 81 30 6000
* ford thundbd 84 10 17000
* toyota tercel 82 180 750
* chevy impala 65 85 1550
* ford bronco 83 25 9525
* ==> grep "chevy" cars
* chevy nova 79 60 3000
* chevy impala 65 85 1550
* ==> echo $?
* 0
* ==> grep "non-existing car" cars
* ==> echo $?
* 1
* ==> grep "mustang" cars-oops-wrong-file-name
* grep: cars-oops-wrong-file-name: No such file or directory
* ==> echo $?
* 2
* ==> \_

**Environment Variables**

* running a script invokes a child process
  + eg. try displaying the process id ($$) in the shell and in a script
    - the process id of the child process will be different than the parent
  + . before a script name will run it in the current process
    - since no child is created, the process id will be the same
    - . is a synonym for the source command
* ==> echo $$ # Display PID (process ID)
* 4462
* ==> cat environment.example1 # Display script
* echo $$
* ==> environment.example1 # Run script
* 20396
* ==> echo $$ # Display PID again
* 4462
* ==> . environment.example1 # Run script in current process
* 4462
* ==> echo $$ # Display PID again
* 4462
* ==> \_

* try creating a variable, then displaying and changing the variable in a script
  + the value of the variable does not get passed to the script
  + the value of the variable within the script does not get passed back to the parent
* ==> school=Seneca # Assign a value to variable "school"
* ==> echo "My school is $school" # Display the value of "school"
* My school is Seneca
* ==> cat environment.example2 # Display script
* echo "My school is $school"
* school="Ryerson University"
* echo "My school is $school"
* ==> environment.example2 # Run script
* My school is
* My school is Ryerson University
* ==> echo "My school is $school" # Display the value of "school" again
* My school is Seneca
* ==> \_

* export command can be used to pass the value of a variable to all child processes, making it an environment variable
  + the value of the variable gets passed to the script this time
  + the value of the variable within the script still does not get passed back to the parent
* ==> echo "My school is $school" # Display the value of "school"
* My school is Seneca
* ==> export school # Make "school" an environment variable
* ==> environment.example2 # Run script
* My school is Seneca
* My school is Ryerson University
* ==> echo "My school is $school" # Display the value of "school" again
* My school is Seneca
* ==> \_

* usual naming convention: environment variable names are UPPERCASE (local shell variables are normally lowercase or camelCase)
* some common environment variables:
  + PS1     - primary prompt
    - for example   
      PS1="\n ==> "   
      PS1='\w: '
  + PWD     - present working directory
  + OLDPWD  - previous working directory, used by cd -
  + HOME    - absolute path to user's home directory, similar to ~
  + HOST    - name of host
  + USER    - name of current user
  + TERM    - terminal type being emulated
  + PATH    - list of directories containing executables (programs)
    - colon-delimited list, try echo $PATH
    - for example, to add current directory to PATH: PATH=$PATH:.
    - more than one directory may contain a particular executable
      * directory list is searched left to right, first matching executable name is used
      * can use which command to identify which executable will be used
      * for example, which grep will show which executable will be used when grep is entered on command line
* can show all environment variables with env command
* can show all variables (local and environment) with set with no arguments

**Shell Arithmetic**

* expr is used to evaluate integer expressions in the Bourne shell (also works in Bash and Korn)
  + spacing is inflexible, and special characters must be quoted
  + eg. expr \( 2 + 3 \) \\* 5
  + eg. x=`expr $x + 1`
* let performs arithmetic in Bash and Korn shells
  + $ is optional in front of variables:   
    let x=(2+3)\*5
  + spacing is flexible if expression is within quotes:   
    let "x = x + 1"
* (( expression )) allows any spacing (works in Bash and Korn)
  + $ is optional, spacing is flexible:   
    ((x = (2 + 3) \* 5))   
    x=$(( x + 1 ))   
    echo $(((x + 6) / 3))

**Exiting From a Script**

* exit 0 - exit script immediately with an exit status of 0
  + normally exit value of 0 means success, any other value means failure
  + use 1 to 125 for failure, remaining values have reserved meanings
  + allows scripts to be called by other scripts with errors properly handled
  + most scripts should be terminated with an exit statement
    - **exception**: if a script is meant to change the environment, it must be invoked as part of the current process, using source or .
    - eg. a script to change to a different directory
    - if this script had an exit then the calling shell would be terminated

**More Commands**

* head -7 filename - same as head -n 7 filename - displays first 7 lines, 10 is default
  + head -n -7 - displays lines from the beginning till 7 lines before the end
* tail -7 filename - same as tail -n 7 filename - displays last 7 lines, 10 is default
  + tail -n +7 - displays from the 7th line till the end
* grep 'string' file - displays lines in file that contain the string (regular expression)
  + grep will be revisited in the lecture about regular expressions
* find - to find files matching specified characteristics
  + find . -name file\*                - lists pathname of any filenames beginning with "file", from the current directory and any subdirectories
  + find . -size +50k                 - lists pathname of any files larger than 50 kb, from the current directory and any subdirectories
  + find . -user alice -empty -delete - deletes empty files owned by user "alice"
  + find . -mmin -5                   - lists files modified less than 5 minutes ago
* cut is used to extract fields and characters from records
  + cut -f2 filename            - extract 2nd field from all records in file, using tab as delimiter (default)
  + cut -f2,5 filename          - extract 2nd and 5th field
  + cut -f1-3,5 filename        - extract 1st through 3rd and 5th fields
  + cut -c3-5 filename          - extract 3rd to 5th characters
  + echo $1 | cut -c2           - extract second character of first argument
  + cut -d: -f2,5 filename      - extract 2nd and 5th field, using colon as delimiter
  + cut -d' ' -f2,5 filename    - extract 2nd and 5th field, using space as delimiter
    - note that each space is used as a delimiter if fields are separated by multiple spaces
    - for example, try:
      * cut -d' ' -f5 cars
      * sed 's/  \*/\t/g' cars | cut -f5
      * sed 's/  \*/ /g' cars | cut -d' ' -f5
      * sed 's/  \*/\t/g' cars | cut -f1-3,5
      * sed 's/  \*/ /g' cars | cut -d' ' -f1-3,5
* sort filename - displays records in ascending order
  + by default uses dictionary (ascii sort) order, from first to last character in each record
  + sort -f filename          - sort ignoring case (fold to uppercase)
  + sort -k3 filename         - sort on 3rd field (default field delimiter is white-space)
    - note that each space is used as a delimiter if fields are separated by multiple spaces
    - can use -b option to ignore multiple spaces, for example, try:
      * sort -k2 cars
      * sed 's/  \*/\t/g' cars | sort -k2
      * sort -bk2 cars
  + sort -t: -k3 filename     - sort on 3rd field using colon as field delimiter
  + sort -rk3 filename        - sort on 3rd field in reverse (descending) order
  + sort -nk5 filename        - sort numerically on 5th field
  + sort -u filename          - sort records, drop duplicate records
  + note that -k3 will sort from the 3rd field to the end of the record, then continue from the beginning of the record if necessary
    - -k3,3 will sort the 3rd field only, then continue from the beginning
    - -k2,4 will sort the the 2nd to 4th field, then continue from the beginning
    - for example, try:
      * sort -k3 cars
      * sort -bk3 cars
      * sort -bk3,3 cars
      * sort -bk3,3 -k5 cars
      * sort -bk3,3 -nk5 cars
      * sort -nk3,3 -k5 cars
* tr is used to translate characters to different characters
  + **tr** a A < filename               - translate all characters "a" to "A"
  + **tr** "[a-z]" "[A-Z]" < filename   - translate lowercase "a" through "z" to uppercase
  + **tr** "a-z" "A-Z" < filename       - translate lowercase "a" through "z" to uppercase, different syntax (non-System V)
  + **tr** ':' ' ' < filename           - translate all colons to spaces
  + **tr** ' ' '\n' < filename          - translate all spaces to newline characters
  + **tr** 'abc' 'A' < filename         - translate 'a', 'b', and 'c' to 'A', the last character in the "to" string repeats
  + **tr** 'a-f' '1-3' < filename       - same as: tr 'abcdef' '123333'
  + **tr** -d '\n' < filename           - delete all newline characters
  + **tr** -c 'a-zA-Z' ' ' < filename   - change complement (all characters not specified) to a space
  + **tr** -cd 'a-z\n' < filename       - delete complement (all characters not specified)
  + **tr** -s ' ' < filename            - squeeze repeats, deletes duplicate adjacent characters:
  + ==> cat cars
  + plym fury 77 73 2500
  + chevy nova 79 60 3000
  + ford mustang 65 45 17000
  + volvo gl 78 102 9850
  + ford ltd 83 15 10500
  + Chevy nova 80 50 3500
  + fiat 600 65 115 450
  + honda accord 81 30 6000
  + ford thundbd 84 10 17000
  + toyota tercel 82 180 750
  + chevy impala 65 85 1550
  + ford bronco 83 25 9525
  + ==> cut -d' ' -f3-5 cars
  + fury
  + nova
  + mustang
  + gl
  + ltd
  + nova
  + 600
  + accord
  + thundbd
  + tercel 82
  + impala
  + bronco
  + ==> tr -s ' ' < cars | cut -d' ' -f3-5
  + 77 73 2500
  + 79 60 3000
  + 65 45 17000
  + 78 102 9850
  + 83 15 10500
  + 80 50 3500
  + 65 115 450
  + 81 30 6000
  + 84 10 17000
  + 82 180 750
  + 65 85 1550
  + 83 25 9525

==> \_

* + tr -s results can be duplicated using sed:
  + ==> sed -r 's/ +/ /g' cars | cut -d' ' -f3-5
  + 77 73 2500
  + 79 60 3000
  + 65 45 17000
  + 78 102 9850
  + 83 15 10500
  + 80 50 3500
  + 65 115 450
  + 81 30 6000
  + 84 10 17000
  + 82 180 750
  + 65 85 1550
  + 83 25 9525

==> \_

* wc filename - displays count of newlines, words, and bytes by default
  + wc -l filename     - displays number of newlines in file
  + wc -w filename     - displays number of words in file
  + wc -m filename     - displays number of characters in file
  + wc -c filename     - displays number of bytes in file
  + wc -L filename     - displays length of longest line in file
  + ==> cat words
  + Here are a bunch of words used to demonstrate various
  + commands. Some of the lines are quite short and some are quite long. However, they will all be
  + used by all the commands. And I've completely run out of things
  + to say, so maybe I'll just repeat this paragraph.
  + Here are a bunch of words used to demonstrate various
  + commands. Some of the lines are quite short and some are quite long. However, they will all be
  + used by all the commands. And I've completely run out of things
  + to say, so maybe I'll just repeat this paragraph.
  + ==> wc -L words
  + 96 words
  + ==> cat words | wc -L
  + 96

==> \_

* + wc -L results can be duplicated using awk:
  + ==> awk '{print length($0)}' words | sort -n | tail -1
  + 96
  + ==> awk 'length($0) > maxlength { maxlength = length($0)} END {print maxlength}' words
  + 96

==> \_

* + note that with multiple options, the output is always in the order "lwmcL"