# UNX510/DPS918 - Unix BASH Shell Scripting

## Lecture 11 - Named Pipes; Process Substitution; Advanced Permissions

### Named Pipes

* pipes are one method of Inter-Process Communication in Unix
* the familiar "|" is called an anonymous pipe, and connects the output of one process to the input of another
* a named pipe also connects two processes, and consists of two pipes, one for input, one for output
* a named pipe is also known as a FIFO: "First In First Out"
* named pipes can be accessed by any local process which has required access permissions
* mkfifo command is used to create a named pipe
  + often put in the /tmp directory
  + delete with rm
* to use a named pipe, one process opens it for reading, the other process opens for writing
* for the following examples, ~/tmp will be used because each Matrix server has it's own /tmp directory:
* the first process creates the named pipe and gives read/write access to everyone, then it reads from the pipe and waits till another process writes to it:
* ==> mkdir ~/tmp
* ==> chmod 711 ~/tmp
* ==> mkfifo ~/tmp/unx510.fifo
* ==> chmod 666 ~/tmp/unx510.fifo
* ==> cat ~/tmp/unx510.fifo

\_

* the second process displays information about the pipe, and writes to it:
* ==> ls -l ~/tmp/unx510.fifo
* prw-rw-rw- 1 lczegel users 0 Jun 6 02:49 /home/lczegel/tmp/unx510.fifo
* ==> cat cars > ~/tmp/unx510.fifo

==> \_

* the first process successfully reads from the pipe, then writes to it and waits for another process to read it:
* ==> mkdir ~/tmp
* ==> chmod 711 ~/tmp
* ==> mkfifo ~/tmp/unx510.fifo
* ==> chmod 666 ~/tmp/unx510.fifo
* ==> cat ~/tmp/unx510.fifo
* 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
* ==> echo 'Wow, it works!' > ~/tmp/unx510.fifo

\_

* the second process successfully reads from the pipe:
* ==> ls -l ~/tmp/unx510.fifo
* prw-rw-rw- 1 lczegel users 0 Jun 6 02:49 /home/lczegel/tmp/unx510.fifo
* ==> cat cars > ~/tmp/unx510.fifo
* ==> cat ~/tmp/unx510.fifo
* Wow, it works!

==> \_

* the first process deletes the pipe:
* ==> mkdir ~/tmp
* ==> chmod 711 ~/tmp
* ==> mkfifo ~/tmp/unx510.fifo
* ==> chmod 666 ~/tmp/unx510.fifo
* ==> cat ~/tmp/unx510.fifo
* 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
* ==> echo 'Wow, it works!' > ~/tmp/unx510.fifo
* ==> rm ~/tmp/unx510.fifo

==> \_

* could also have used: mkfifo -m 666 ~/tmp/unx510.fifo

### Client-Server Applications

* an example of a simple server using a named pipe:
* fifo=~/tmp/log\_server\_pipe
* if [[ ! -p $fifo ]] && ! mkfifo -m 666 $fifo
* then echo "Error: could not create pipe" >&2
* exit 1
* fi
* while read log\_rec
* do
* echo $log\_rec

done < $fifo

* an example of a matching client:
* fifo=~/tmp/log\_server\_pipe
* if [[ ! -p $fifo ]]
* then echo "Sorry, log server not running" >&2
* exit 1
* fi
* printf "Please enter a comment: "
* read log\_record

echo "$log\_record" > $fifo

* note that the server often terminates when the read from the pipe mistakenly sees a client termination as an EOF
* the server can keep the pipe output open, so that the pipe doesn't close with a read error:
* fifo=~/tmp/log\_server\_pipe
* if [[ ! -p $fifo ]] && ! mkfifo -m 666 $fifo
* then echo "Error: could not create pipe" >&2
* exit 1
* fi
* while read log\_rec
* do
* echo $log\_rec

done < $fifo 3>$fifo

* alternatively, the server loop can be made independent of read failure:
* fifo=~/tmp/log\_server\_pipe
* if [[ ! -p $fifo ]] && ! mkfifo -m 666 $fifo
* then echo "Error: could not create pipe" >&2
* exit 1
* fi
* while :
* do
* if read log\_rec < $fifo
* then
* echo "$log\_rec"
* fi

done

* note that with any of these examples, the pipe input can be redirected to either the "read" or the "while"
  + unlike a normal file, once something is read from a pipe, it is gone and will not be read again if the pipe is closed and reopened
* this client can accept the comment as a single argument, or as multiple arguments:
* fifo=~/tmp/log\_server\_pipe
* if [[ ! -p $fifo ]]
* then echo "Sorry, log server not running" >&2
* exit 1
* fi
* if [[ -z $1 ]]
* then
* read -p "Please enter a comment: " log\_record
* else
* log\_record="$\*"
* fi

echo "$log\_record" > $fifo

* note that the output of the log server could be redirected to a file, with write permission needed by the server, but not by the client

### Process Substitution

* a named pipe is internally generated, to connect to another process as either input or output
* <( command-list ) - output of the command list can be used as an input file
* can be similar to anonymous piping:
* ==> head -1 <(grep ford cars | tr "a-z" "A-Z")
* FORD MUSTANG 65 45 17000
* ==> grep ford cars | tr "a-z" "A-Z" | head -1
* FORD MUSTANG 65 45 17000
* ==> \_
* if a file needs to be redirected, so does a process substitution:
* ==> tr "a-z" "A-Z" < <(grep ford cars | head -1)
* FORD MUSTANG 65 45 17000
* ==> \_
* multiple command lists can be used as input to a command, this cannot be done with anonymous piping:
* ==> cp cars cars2
* ==> echo "ford - new car added" >> cars2
* ==> diff <(grep ford cars) <(grep ford cars2)
* 4a5
* > ford - new car added
* ==> \_
* >( command-list ) - output of a command can be used as input to the command list
* similar to piping, but output of a command can be sent to multiple command lists (using "tee"):
* ==> cat cars | tee >(awk '/ford/ {total+=$5} END {print "Total fords: " total}') \
* >(awk '/chevy/ {total+=$5} END {print "Total chevys: " total}')
* 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
* ==> Total chevys: 4550
* Total fords: 54025
* \_
* standard output of the "tee" can be redirected to /dev/null:
* ==> cat cars | tee >(awk '/ford/ {total+=$5} END {print "Total fords: " total}') \
* >(awk '/chevy/ {total+=$5} END {print "Total chevys: " total}') \
* >/dev/null
* ==> Total chevys: 4550
* Total fords: 54025
* \_
* the processes created are independent, execution order is indeterminate:
* ==> echo 1 | tee >(echo 2) >(echo 3) >(echo 4) > >(cat)
* 3
* 4
* 2
* ==> 1
* echo 1 | tee >(echo 2) >(echo 3) >(echo 4) > >(cat)
* 2
* 4
* 3
* ==> 1
* \_
* the processes can be synchronized, in this example using sleep:
* ==> echo 1 | tee >(sleep 2; echo 2) >(sleep 3; echo 3) >(sleep 4; echo 4) > >(sleep 5; cat)
* ==> 2
* 3
* 4
* 1
* echo 1 | tee >(sleep 2; echo 2) >(sleep 3; echo 3) >(sleep 4; echo 4) > >(cat)
* ==> 1
* 2
* 3
* 4
* \_

### Advanced Permissions

There are actually 12 permission bits, not just the 9 commonly used

* 1 - Set UID bit - when executing this file, use the userid of the file's owner (rather than the userid of the user executing the file)
* 2 - Set GID bit - when executing the file, use the groupid of the file (rather than the groupid of the user executing it)
* 3 - Sticky bit - permit only the owner of this directory, or the owner of a contained file, to delete or overwrite the file
* 4, 5, 6 - Owner read, write, execute permission
* 7, 8, 9 - Group read, write, execute permissions
* 10, 11, 12 - Other read, write, execute permissions

### Setting Permissions

Absolute method: **chmod** wxyz filename

* "wxyz" are octal digits
* first digit: Set UID has value of 4, Set GID has value of 2, Sticky bit has value of 1
* second digit: Owner read has value of 4, write has value of 2, execute has value of 1
* third digit: Group read has value of 4, write has value of 2, execute has value of 1
* fourth digit: Other read has value of 4, write has value of 2, execute has value of 1

Relative method: **chmod** who operation permission filename

* "who" can be u (user), g (group), o (other), or a (all)
* "operation" can be + (add), - (remove), or = (set)
* "permission" can be r (read), w (write), x (execute), s (set UID or set GID, depending on "who"), t (sticky bit)

### Displaying Permissions

**ls -l** displays only 9 characters for permissions, usually: **rwxrwxrwx**

* "x" for user will show "S" if file has set UID permission, or "s" if file has both set UID and execute permissions
* "x" for group will show "S" if file has set GID permission, or "s" if file has both set GID and execute permissions
* "x" for other will show "T" if file has sticky bit set, or "t" if file has both sticky bit set and execute permission
* example:
* ==> touch f0000 f2000 f2777 f4000 f4777
* ==> mkdir f1000 f1777 f7777
* ==> chmod 0000 f0000
* ==> chmod 1000 f1000
* ==> chmod 1777 f1777
* ==> chmod 2000 f2000
* ==> chmod 2777 f2777
* ==> chmod 4000 f4000
* ==> chmod 4777 f4777
* ==> chmod 7777 f7777
* ==> ls -l
* total 12
* ---------- 1 lczegel users 0 Jan 19 16:49 f0000
* d--------T 2 lczegel users 4096 Jan 19 16:50 f1000
* drwxrwxrwt 2 lczegel users 4096 Jan 19 16:50 f1777
* ------S--- 1 lczegel users 0 Jan 19 16:49 f2000
* -rwxrwsrwx 1 lczegel users 0 Jan 19 16:49 f2777
* ---S------ 1 lczegel users 0 Jan 19 16:49 f4000
* -rwsrwxrwx 1 lczegel users 0 Jan 19 16:49 f4777
* drwsrwsrwt 2 lczegel users 4096 Jan 19 16:50 f7777
* ==> \_

### Example of using Set UID

* I'll use the following file to show how set UID works. Note that the file has read permission only for my "lczegel" account:
* lczegel==> cat text.file
* This file has read permission only for the owner.
* A program with set UID permission will be used to read it.
* The program will be executable by group and others.
* By running the program, all users can read this file.
* lczegel==> ls -l text.file
* -r-------- 1 lczegel users 215 Jan 19 20:02 text.file
* lczegel==> \_

* I'll use the following simple C program to read and display the file. Note that the permissions allow all users to execute the program:
* lczegel==> cat text.reader.c
* #include <stdio.h>
* #include <stdlib.h>
* int main(void) {
* char buffer[80];
* FILE \*fp = fopen("/home/lczegel/permission.examples/text.file","r");
* if (fp <= 0) {
* perror("/home/lczegel/permission.examples/text.file");
* exit(1);
* }
* while(fgets(buffer, 80, fp) != NULL)
* printf("%s", buffer);
* exit(0);
* }
* lczegel==> cc text.reader.c -o text.reader -Wall
* lczegel==> chmod 711 text.reader
* lczegel==> ls -l text.reader
* -rwx--x--x 1 lczegel users 14322 Jan 19 20:34 text.reader
* lczegel==> \_

* I'll try to execute this program from my "les.czegel" account:
* les.czegel==> ~lczegel/permission.examples/text.reader
* /home/lczegel/permission.examples/text.file: Permission denied
* les.czegel==> \_

* Permission is denied, as expected. Now in my "lczegel" account I'll add set UID permission to the program:
* lczegel==> chmod u+s text.reader
* lczegel==> ls -l text.reader
* -rws--x--x 1 lczegel users 14322 Jan 19 20:34 text.reader
* lczegel==> \_

* Now I'll run the program from my "les.czegel" account again. Note that this account still can't read "text.file" directly:
* les.czegel==> cat ~lczegel/permission.examples/text.file
* cat: /home/lczegel/permission.examples/text.file: Permission denied
* les.czegel==> ~lczegel/permission.examples/text.reader
* This file has read permission only for the owner.
* A program with set UID permission will be used to read it.
* The program will be executable by group and others.
* By running the program, all users can read this file.
* les.czegel==> \_

* It works! Set GID works in a similar way.