Bash Examples

Dr James Martin
Associate Professor
School of Computing
Clemson, SC
jmarty@clemson.edu

Last update: 3/16/2018

This documents scripts used in homework assignments or that are located in codeReleased/V1/bash

We begin by looking at a script that was developed for Q2, HW2, CPSC4240/6240 Spring 2018. This was used to run a program that measured the transfer rate over an unnamed pipe. The three parameters were
Param 1: number of iterations or times we run the program
Param 2: The pipe size (an experimental parameter)
Param 3: The amount of data that was transferred
The script is meant to do multiple runs all using the same parameter settings – this is useful when operating over a system that exhibits randomness… we can analyze aspects of the results to better assess (and model) the randomness.

Script 1 go.sh , example: ‘./go.sh 10 1000000 1000000000’

```
1. iterations=1
2. sleepTime=0.01
3. PSize="1000000"
4. i="0"
5. TSize="1000000000"
6. chunkSize=1000000
7. echo "Entered($0): number args: $# "
8. if [ "$#" -eq 3 ]
9. then
10. iterations=$1
11. PSize="$2"
12. TSize="$3"
13. fi
14. if [ "$#" -eq 2 ]
15. then
16. echo
17. iterations=$1
18. PSize="$2"
19. fi
20. if [ "$#" -eq 1 ]
21. then
```

```
22. iterations=$1
23. fi
24. echo "$0 $# : params: iterations:$iterations pipesize:$PSize  transferSize:$TSize"
25. #careful - make sure spaces are there
26.
27. while [ $i -lt $iterations ]
28. do
29. resultLine="$i pipesize:$PSize chunksize:$chunkSize transferSize:$TSize "
30. ./serverfifo2 $PSize $chunkSize > /dev/null  &
31. sleep 0.01
32. clientResultLine=`./clientfifo2 $TSize $chunkSize | grep Gbps | awk '{printf "%s %s \n",$5, $6}'`
33. resultLine+=" $clientResultLine "
34. echo "$resultLine "
35. i=$[$i+1]
36. done
37. exit
```

Lines 1-25 is all setup and init code. Each iteration of the while loop runs the server in background mode, and then the client using the specified transfer size (in bytes) and chunk size (bytes).
The while loop (lines 27-36) loops for the correct number of iterations. The variable resultLine is used to hold the iteration status result line. The output of running the program is placed in a string variable clientResultLine. By running the pipeline surrounded by backticks causes it to run before the first part of the command line….allowing the results to be placed in the string variable.

```
clientResultLine=`./clientfifo2 $TSize $chunkSize | grep Gbps | awk '{printf "%s %s \n",$5, $6}'`
```

Why would this following not work ?

```
clientResultLine=./clientfifo2 $TSize $chunkSize | grep Gbps | awk '{printf "%s %s \n",$5, $6}'
```

or more generally, clientResultLine=./clientfifo2 100000000 1000000
Bash parses this into clientRes
clientResultLine=./clientfifo2 and executes, then tries to excecute 100000000 thinking it is a shell token.
Note that this will work: resultLine=$(./clientfifo2 1000000000 1000000)

Script 2 is a similar script as script 1- developed for running iterations of the same program. It is different from Script 1 as it iterates across two variables. The outerloop (line 3) iterates over different settings of transfer sizes. The innerloop iterates over different choices of the pipe size.

Go2.sh 1

```
1.  #!/bin/bash
2.  # Runs the clientfifo2 and serverfifo2 programs
3.  #   Iterates the pipeSize
4.  chunkSize=100000
5.  transferSize=10000000
6.  iterations=1
7.  sleepTime=0.01
8.  if [ "$#" -ne 1 ]
9.  then
10. echo
11. echo "Usage: $0 <iterations>"
12. echo " using iterations of $iterations "
13. else
14. iterations=$1
15. fi
16. i="0"
17. factor="10"
18. Pfactor="10"
19. TSize="10000"
20. startPipeSize=1000
21. PSize="1000"
22. #careful - make sure spaces are there
23. while [ $i -lt $iterations ]
24. do
25. #inner loop iterate over pipe size
26. PSize="1000"
27. for j in $(seq 1 $iterations); do
28. PSize=$[$PSize * $Pfactor]
29. resultLine="$i,$j pipesize:$PSize chunksize:$chunkSize transferSize:$TSize "
30. #echo "run:$i,$j pipesize:$PSize chunksize:$chunkSize transferSize:$TSize "
31. ./serverfifo2 $PSize $chunkSize > /dev/null  &
32. sleep 0.01
33. clientResultLine=`./clientfifo2 $TSize $chunkSize | grep Gbps | awk '{printf "%s %s \n",$5, $6}'`
34. #./clientfifo2 $TSize $chunkSize | grep Gbps | awk '{printf "%s %s \n",$5, $6}'
35. #clientResultLine=`./clientfifo2 $TSize $chunkSize | grep Gbps`
36. # experiment+=`./clientfifo2  "$i" | tail -n 1 | awk '{printf "%s, ", $2}'`
37. resultLine+=" $clientResultLine "
38. echo "$resultLine "
39. done
40. TSize=$[$TSize * $factor]
41. i=$[$i+1]
42. done
43. exit
```

An outer loop (line 23-44) performs the desired number of iterations. The inner loop (line 27-39) performs a number of iterations each varying the pipesize.

script 2 : countdown.sh This illustrates the use of counting with integer and strings.

The first function declares count 'typeset count'
The second function uses string math.

Function 1- line 4 is doing a test based on [$count –gt 0] . This requires line 7 ((count=count-1)) to make sure the math is done based on integers.
Function 2 – line 16 is doing a test based on [$i –lt 4]. This requires line 18 i=$[$i+1] which increments the string number.

```
1.  function count_down () {
2.  typeset count
3.  count=$1
4.  while [ $count -gt 0 ]
5.  do
6.  echo "$count..."
7.  ((count=count-1))
8.  sleep 1
9.  done
10. echo "Blast Off."
11. }

12. #using character math
13. function count_down2 () {

14. i="0"

15. #careful - make sure spaces are there
16. while [ $i -lt 4 ]
17. do
18. i=$[$i+1]
19. echo "iteraton $i "
20. done

21. }
```

Script - sysmon.sh

```
1.  #!/bin/bash
2.  #  System Health Monitor
3.  #   sysmon.sh
4.  #
5.  #  Usage;  sysmon.sh interval  interface
6.  #         interval:  time in seconds between samples
7.  #         interface: the network interface name (e.g., eth0)
8.  #
9.  #   Note: this script requires the bc tool (apt-get install bc)
10. #

11. interval=$1
12. interface=$2
13. if [ "$#" -ne 2 ]
14. then
15. echo
16. echo "Usage: $0 <interval> <interface>"
17. echo
18. exit 1
19. fi

20. declare -i tempOut=0
21. declare -i tempIn=0

22. while true; do
23. clear
24. echo "Interval (sec): "$'\t'"$interval"
25. echo "Inteface: "$'\t'"$interface"
26. echo -n "Number of Cores: "$'\t'
27. cores=$(grep pro /proc/cpuinfo -c)
28. echo $cores CPU Utilization
29. echo -n "5 Minute Avg CPU Utilization (%): "$'\t'
30. util=$(cat /proc/loadavg | awk '{print $1}')
31. echo "scale=1; (($util*100)/$cores)/1" | bc # >>  cpuUtilBash.txt

32. # - Bandwidth Consumption
33. echo -n "Outbound BW (bits): "$'\t'
34. outbound1=$(awk -v interface="$interface" -F'[: \t]+' \
35. '{ sub(/^ */,"");
36. if ($1 == interface) print $10;
37. }' /proc/net/dev)

38. let "tempOut = outbound1 + 0"
39. let "outbound1 -= outbound0"
40. let "outbound0 = tempOut + 0"
```

```
41. let "bpsOut = outbound1 * 8"

42. echo "$bpsOut" # >> bitsOutBash.txt

43. echo -n "Inbound BW (bits): "$'\t'
44. inbound1=$(awk -v interface="$interface" -F'[: \t]+' \
45. '{ sub(/^ */,"");
46. if ($1 == interface) print $2;
47. }' /proc/net/dev)

48. let "tempIn = inbound1 + 0"
49. let "inbound1 -= inbound0"
50. let "inbound0 = tempIn + 0"
51. let "bpsIn = inbound1 * 8"
52. echo "$bpsIn" # >> bitsInBash.txt
53. sleep $interval
54. done
```

In sysmon.sh, we note the following lines :

- Linex 42,43 - these declare integer variables which allows math updates such as linnes 70-73.
- Line 49 : cores=$(grep pro /proc/cpuinfo -c)
  This is setting the variable cores with a number based on :
  - grep pro /proc/cpuinfo –c
  - If we issue : grep pro /proc/cpuinfo it shows two lines
    - processor : 0
    - processor : 1
  - The grep with –c param is telling it just to return the number of lines….which is what we set core with. Represents the number of CPUs on the system.
- Lines 54-59 - sets the variable outbound1 with the name of the interface
  - Issuing a cat /proc/net/dev lists information related to each interface. On my system, I see:

```
Inter-|   Receive                                                |  Transmit
face |    bytes    packets errs drop fifo frame c
                 ompressed multicast|bytes
                                                                 packets errs drop fifo colls carrier compressed
  enp0s17: 76833794   74670   0   0   0   0       0       0 11041451   34971   0   0   0   0     0       0
  enp0s8:      0      0   0   0   0   0       0       0   75049    465   0   0   0   0     0       0
  lo:  364171    3705   0   0   0   0       0       0  364171    3705   0   0   0   0     0       0
```

The following documents the script parseFile.

Invocation: ./pasrseFile inputFile.txt delim1 delim2
This shows the contains of each line that match delim1 and 2 – shows what is in beween these.

```
#!/bin/bash

usage () {
  echo "Usage: [data filename] [beginning delimiter] [end delimiter] "
  echo "    Example 1: ./parseFile.sh  data1.dat ack ', win' "
  echo "    Example 1: ./parseFile.sh  ping.rawdata time= ms"

#cat $dataFName | grep -o "$begin$regex2$end" | sed "s/$end//g" | sed "s/$begin//g"
}

regex2=".*"

if [ $# -lt 3 ]; then
   echo "$0 [ERROR] missing operand"
   usage
   exit -1
fi

dataFName=$1
begin=$2
end=$3

echo "parseData:  file:$dataFName;  begin:$begin;  end:$end; "
#grep -o "$begin$regex$end"  : this will produce the pattern with begin and end  delimiters
#sed "s/$end//g" | sed "s/$begin//g"  : this strips off the delims
cat $dataFName | grep -o "$begin$regex2$end" | sed "s/$end//g" | sed "s/$begin//g"
#grep -o "$begin$regex$end"  : this will produce the pattern with begin and end  delimiters
#sed "s/$end//g" | sed "s/$begin//g"  : this strips off the delims

~
For example, if an input file ping.rawdata:
jjm@jjm-VirtualBox:~/courses/codeReleased/V1/bash/parseFile$ cat ping.rawdata
PING netlabserver1.clemson.edu (130.127.49.48) 56(84) bytes of data.
64 bytes from netlabserver1.clemson.edu (130.127.49.48): icmp_seq=1 ttl=59 time=2.79 ms
64 bytes from netlabserver1.clemson.edu (130.127.49.48): icmp_seq=2 ttl=59 time=2.54 ms
64 bytes from netlabserver1.clemson.edu (130.127.49.48): icmp_seq=3 ttl=59 time=4.24 ms
64 bytes from netlabserver1.clemson.edu (130.127.49.48): icmp_seq=4 ttl=59 time=3.80 ms
64 bytes from netlabserver1.clemson.edu (130.127.49.48): icmp_seq=5 ttl=59 time=3.77 ms
64 bytes from netlabserver1.clemson.edu (130.127.49.48): icmp_seq=6 ttl=59 time=2.69 ms
And we wanted to display just the RTTss:
: ./parseFile.sh  ping.rawdata time= ms"
```

The pipeline cat's the data file to grep which searches for $begin$regex2$end
For this file, this returns time=2.79ms. We use sed to remove the ms and sed to remove the time=.

Download the set of data files from our web site. Several of the questions will have you use the files. After you unpack , you should see a directory PB1 (practice Bash #1) with three subdirs each containing a number of non-zero size files. The following shows the results from a 'ls –Rlt'

```
drwxrwxr-x 2 jjm jjm 4096 Feb 10 01:22 myDirC
drwxrwxr-x 2 jjm jjm 4096 Feb 10 01:22 myDirB
drwxrwxr-x 2 jjm jjm 4096 Feb 10 01:22 myDirA

./myDirC:
total 48
-rw-rw-r-- 1 jjm jjm  1865 Feb 10 01:22 myFile1.txt
-rw-rw-r-- 1 jjm jjm  1865 Feb 10 01:22 myFile2.txt
-rw-rw-r-- 1 jjm jjm  1865 Feb 10 01:22 myFile3.txt
-rw-rw-r-- 1 jjm jjm  1865 Feb 10 01:22 myFile4.txt
-rw-rw-r-- 1 jjm jjm  1865 Feb 10 01:22 myFile5.txt
-rw-rw-r-- 1 jjm jjm 22645 Feb 10 01:22 wordsList3.txt
-rw-rw-r-- 1 jjm jjm   660 Feb 10 01:22 userdata.dat

./myDirB:
total 44
-rw-rw-r-- 1 jjm jjm   426 Feb 10 01:22 myFile1.txt
-rw-rw-r-- 1 jjm jjm   426 Feb 10 01:22 myFile2.txt
-rw-rw-r-- 1 jjm jjm   426 Feb 10 01:22 myFile3.txt
-rw-rw-r-- 1 jjm jjm   426 Feb 10 01:22 myFile4.txt
-rw-rw-r-- 1 jjm jjm 22645 Feb 10 01:22 wordsList2.txt
-rw-rw-r-- 1 jjm jjm   660 Feb 10 01:22 userdata.dat

./myDirA:
total 40
-rw-rw-r-- 1 jjm jjm   182 Feb 10 01:22 myFile1.txt
-rw-rw-r-- 1 jjm jjm   182 Feb 10 01:22 myFile2.txt
-rw-rw-r-- 1 jjm jjm   182 Feb 10 01:22 myFile3.txt
-rw-rw-r-- 1 jjm jjm 22645 Feb 10 01:22 wordsList1.txt
-rw-rw-r-- 1 jjm jjm   660 Feb 10 01:22 userdata.dat
```

Question : We issue a grep of some sort on a file call file.txt: grep ‘regexp’ file.txt

The contents of file.txt consists of the following 10 lines. The file does not contain the line numbers – each line of the file begins with the first letter or digit after the line number, period, and space. You will be asked if a number of grep invocations will result in a match. You should identify the line number of any line in the data file would match. We answer the first question to provide an example.

1. hello 1234567 hello
2. 1234567
3. 8655081888
4. 865-6081888
5. 864-508-1821
6. 864-5081821
7. 864-128-1829
8. 864-1281829
9. HELLO 864-128-1830

10. HELLO 864-1281830

What matches the following?

- grep 1888 file.txt
  - Answer: this matches two of the lines (the 3rd and 4th).

- grep "\<\d*" file.txt
  - Answer: this matches all 10 lines.

- grep "\<[A-Za-z].*" file.txt
  - Answer: this matches three lines with letters (lines 1,9 and 10)

Question: create a regular expression that matches any phone number in the form abc-def-ghij. For example it would match 864-128-1829 but would not match 864-1281829. Assume that the phone number is preceded by a white space.

Solution

- grep "\<\d*" file.txt
  - matches all 10 lines
- grep "\<[A-Za-z].*" file.txt
  - matches the three lines with letters (lines 1,9 and 10)

Q1. The following snippet of code is in a script called example1.sh. Note that it is similar to code from the bex3.sh script that we talked about in class.

```
#!/bin/bash
LIST="$(ls *.c)"
for i in "$LIST"; do
    echo $i
done
```

Assume we run the script from a directory that has the following files (I show the results of *ls*) :

```
$ > ls
client       Make.defines  server          UDPEcho.h       UDPEcho.tar.gz
DieWithError.c  Makefile     UDPEchoClient2.c  UDPEchoServer.c
DieWithError.o  Makefile.bak  UDPEchoClient2.o  UDPEchoServer.o
```

What would you expect to see on standard out when you run the script (i.e., './example1.sh') ?

Solution:
./example1.sh

DieWithError.c UDPEchoClient2.c UDPEchoServer.c

Q2. Explain the difference between the effects of the following two *basename* commands.

```
$ > cd /usr/local/bin
$ > basename pwd
$ > basename `pwd`
```

Solution:
pwd vs. actual path base: bin. In the first case, basename is applied to the string "pwd". Now, basename is supposed to strip path and suffix from a file name; in this case, there is none, so basename wrote "pwd". In the second case, the backticks cause the output from pwd to replace the command invocation, so basename sees the result from pwd, which was the string “/usr/local/bin”. So, basename stripped the leading path information off and wrote "bin".

Q3. Using a brief sentence, explain what the following line of script doing.

```
count=$(find "$directory" -type f -mtime "-$daysAgo" -user "$user" -print0 | tr -d -c '\0' | wc -c)
```

Solution:
For each user in the list, count represents the number of files that have been modified within the last $daysAgo

Q4. Bash's built-in arithmetic uses integers only, how can you calculate with floating point numbers instead of just integers?

Solution:
For most operations involving non-integer numbers, an external program must be used, e.g. bc, AWK or dc:

```
$ > echo 'scale=3; 10/3' | bc
$ > 3.333
```

The "scale=3" command notifies bc that three digits of precision after the decimal point are required. Same example with dc (reverse polish calculator, lighter than bc):

```
$ > echo '3 k 10 3 / p' | dc
$ > 3.333
```

k sets the precision to 3, and p prints the value of the top of the stack with a newline. The stack is not altered, though.

Q5. How do you read a file (data stream, variable) line-by-line (and/or field-by-field)?

Solution:
Don't try to use "for". Use a while loop and the read command:

```
while IFS= read -r line; do
   printf '%s\n' "$line"
done < "$file"
```

The -r option to *read* prevents backslash interpretation (usually used as a backslash newline pair, to continue over multiple lines or to escape the delimiters). Without this option, any unescaped backslashes in the input will be discarded. You should almost always use the -r option with read.

The most common exception to this rule is when -e is used, which uses Readline to obtain the line from an interactive shell. In that case, tab completion will add backslashes to escape spaces and such, and you do not want them to be literally included in the variable. This would never be used when reading anything line-by-line, though, and -r should always be used when doing so.

In the scenario above *IFS=* prevents trimming of leading and trailing whitespace. Remove it if you want this effect.

line is a variable name, chosen by you. You can use any valid shell variable name there.

The redirection < "$file" tells the while loop to read from the file whose name is in the variable file. If you would prefer to use a literal pathname instead of a variable, you may do that as well. If your input source is the script's standard input, then you don't need any redirection at all.

Q6. How do you print the n'th line of a file?

Solution:

```
$ > sed -n "${n}p" "$file"
$ > head -n "$n" "$file" | tail -n 1
$ > awk "NR==$n{print;exit}" "$file"
```

Q7. How do you concatenate two variables? How do I append a string to a variable?

Solution: There is no (explicit) concatenation operator for strings (either literal or variable dereferences) in the shell; you just write them adjacent to each other:

```
$ > var=$var1$var2
```

If the right-hand side contains whitespace characters, it needs to be quoted:

```
$ > var="$var1 - $var2"
```

If you're appending a string that doesn't "look like" part of a variable name, you just smoosh it all together:

```
$ > var=$var1/.-
```

Otherwise, braces or quotes may be used to disambiguate the right-hand side:

```
var=${var1}xyzzy
# Without braces, var1xyzzy would be interpreted as a variable name

var="$var1"xyzzy
# Alternative syntax
```

Q8. How do you run a command on all files with the extension .gz?

Solution:
Often a command already accepts several files as arguments, e.g.

```
$ > zcat -- *.gz
```

On some systems, you would use gzcat instead of zcat. If neither is available, or if you don't care to play guessing games, just use gzip -dc instead.

The -- prevents a filename beginning with a hyphen from causing unexpected results.

If an explicit loop is desired, or if your command does not accept multiple filename arguments in one invocation, the for loop can be used:

```
for file in ./*.gz
do
   echo "$file"
   # do something with "$file"
done
```

To do it recursively, use find:

```
$ > find . -name '*.gz' -type f -exec do-something {} \;
```

If you need to process the files inside your shell for some reason, then read the find results in a loop:

```
while IFS= read -r file; do
   echo "Now processing $file"
   # do something fancy with "$file"
done < <(find . -name '*.gz' -print)
```

Q9. How do you use numbers with leading zeros in a loop, e.g. 01, 02?

Solution:
Bash version 4 allows zero-padding and ranges in its BraceExpansion:

```
# Bash 4 / zsh
for i in {01..10}; do
   ...
```

To Bash version earlier than 4.0, or a non-Bash shell:

```
# Bash / ksh / zsh
for i in 0{1,2,3,4,5,6,7,8,9} 10
```

```
do
   echo "$i"
done
```

In Bash 3, you can use ranges inside brace expansion (but not zero-padding). Thus, the same thing can be accomplished more concisely like this (prefer):

```
# Bash 3
for i in 0{1..9} 10
do
   echo "$i"
done
```

Q10. How do you let two unrelated processes communicate?

Solution:
Two unrelated processes cannot use the arguments, the environment or stdin/stdout to communicate; some form of inter-process communication (IPC) is required. e.g. Process A writes in a file, and Process B reads the file. This method is not synchronized and therefore is not safe if B can read the file while A writes in it. A lockdir or a signal can probably help.

Using locker: *mkdir* can be used to test for the existence of a dir and create it in one atomic operation; it thus can be used as a lock, although not a very efficient one.

Script A:

```
until mkdir /tmp/dir;do # wait until we can create the dir
   sleep 1
done
echo foo > file        # write in the file this section is critical
rmdir /tmp/dir         # remove the lock
```

Script B:

```
until mkdir /tmp/dir;do #wait until we can create the dir
   sleep 1
done
read var < file        # read in the file this section is, critical
echo "$var"            # Script A cannot write in the file
rmdir /tmp/dir         # remove the lock
```

Using signals: Signals are probably the simplest form of IPC:

ScriptA:

```
trap 'flag=go' USR1 #set up the signal handler for the USR1 signal
# echo $$ > /tmp/ScriptA.pid #if we want to save the pid in a file

flag=""
while [[ $flag != go ]]; do # wait for the green light from Script B
   sleep 1;
```

```
done
echo “we received the signal”
```

You must find or know the pid of the other script to send it a signal using kill:

```
#kill all the other script
pkill -USR1 -f ScriptA

#if ScriptA saved its pid in a file
kill -USR1 $(</var/run/ScriptA.pid)

#if ScriptA is a child:
ScriptA & pid=$!
kill -USR1 $pid
```

The first 2 methods are not bullet proof and will cause trouble if you run more than one instance of scriptA.

Q11. How do you find a process ID for a process given its name?

Solution:
Usually a process is referred to using its process ID (PID), and the ps(1) command can display the information for any process given its process ID, e.g.

```
$ > echo $$        # my process id
21796

$ > ps -p 21796
PID TTY          TIME CMD
21796 pts/5    00:00:00 ksh
```

But frequently the process ID for a process is not known, but only its name. Some operating systems, e.g. Solaris, BSD, and some versions of Linux have a dedicated command to search a process given its name, called pgrep(1):

```
$ > $ pgrep init
1
```

Often there is an even more specialized program available to not just find the process ID of a process given its name, but also to send a signal to it:

```
$ > pkill myprocess
```

Ubuntu/CentOS also provides pidof(1). It differs from pgrep in that multiple output process IDs are only space separated, not newline separated.

```
$ > pidof cron
5392
```

If these programs are not available, a user can search the output of the ps command using grep. The major problem when grepping the ps output is that grep may match its own ps entry (try: ps aux | grep init). To make matters worse, this does not happen every time; the technical name for this is a RaceCondition. To avoid this, there are several ways:

Using grep -v at the end:

```
$ > ps aux | grep name | grep -v grep
```

will throw away all lines containing "grep" from the output. Disadvantage: You always have the exit state of the grep -v, so you can't e.g. check if a specific process exists.

Using grep -v in the middle:

```
$ > ps aux | grep -v grep | grep name
```

This does exactly the same, except that the exit state of "grep name" is accessible and a representation for "name is a process in ps" or "name is not a process in ps". It still has the disadvantage of starting a new process (grep -v).

Using [] in grep:

```
$ > ps aux | grep [n]ame
```

This spawns only the needed grep-process. The trick is to use the []-character class (regular expressions). To put only one character in a character group normally makes no sense at all, because [c] will always match a "c". In this case, it's the same. grep [n]ame searches for "name". But as grep's own process list entry is what you executed ("grep [n]ame") and not "grep name", it will not match itself.

Q12. How do you find out if a process is still running?

Solution: The kill command is used to send signals to a running process. As a convenience function, the signal "0", which does not exist, can be used to find out if a process is still running:

```
$ > myprog &          # Start program in the background
$ > daemonpid=$!      # ...and save its process id

while sleep 60
do
   if kill -0 $daemonpid       # Is the process still alive?
   then
      echo >&2 "OK - process is still running"
   else
      echo >&2 "ERROR - process $daemonpid is no longer running!"
      break
   fi
done
```

NOTE: Anything you do that relies on PIDs to identify a process is inherently flawed. If a process dies, the meaning of its PID is UNDEFINED. Another process started afterward may take the same PID as the dead process. That would make the previous example think that the process is still alive (its PID exists!)

even though it is dead and gone. It is for this reason that nobody other than the parent of a process should try to manage the process.

This is one of those questions that usually masks a much deeper issue. It's rare that someone wants to know whether a process is still running simply to display a red or green light to an operator.

More often, there's some ulterior motive, such as the desire to ensure that some daemon which is known to crash frequently is still running. If this is the case, the best course of action is to fix the program or its configuration so that it stops crashing. If you can't do that, then just restart it when it dies:

```
while true
do
   myprog && break
   sleep 1

done
```

This piece of code will restart myprog if it terminates with an exit code other than 0 (indicating something went wrong). If the exit code is 0 (successfully shut down) the loop ends. (If your process is crashing but also returning exit status 0, then adjust the code accordingly.) Note that myprog must run in the foreground. If it automatically "daemonizes" itself, you are screwed.

Q13. How do you redirect stderr to a pipe?

Solution:
A pipe can only carry standard output (stdout) of a program. To pipe standard error (stderr) through it, you need to redirect stderr to the same destination as stdout. Optionally you can close stdout or redirect it to /dev/null to only get stderr. Some sample code:

```
# Assume 'myprog' is a program that writes to both stdout and stderr.

# version 1: redirect stderr to the pipe while stdout survives (both come mixed)
$ > myprog 2>&1 | grep ...

# version 2: redirect stderr to the pipe without getting stdout (it's redirected to /dev/null)
$ > myprog 2>&1 >/dev/null | grep ...

# same idea, this time storing stdout in a file
$ > myprog 2>&1 >file | grep ...
```

Q14. How do you view periodic updates/appends to a file? (ex: growing log file)?

Solution:
tail -f will show you the growing log file. On some systems (e.g. OpenBSD), this will automatically track a rotated log file to the new file with the same name (which is usually what you want). To get the equivalent functionality on GNU systems, use tail -F instead.

This is helpful if you need to view only the updates to the file after your last view:

```
# Start by setting n=1
$ > tail -n $n testfile; n="+$(( $(wc -l < testfile) + 1 ))"
```

Every invocation of this gives the update to the file from where we stopped last. If you know the line number from where you want to start, set n to that.

Q15. How do you trim leading/trailing white space from one of variables in bash script?

Solution:

```
# Bash
shopt -s extglob
var="${var##*( )}"       # trim the left
var="${var%%*( )}"     # trim the right
```

Q16. How do you use find to search files based on names, times, sizes? How do you declare an action (such as rm, cat, rename) on the results?

Solution:

http://mywiki.wooledge.org/UsingFind

Q17. In Named Pipes, a fifo typically accept one-time write and read. For example,

```
$ > mkfifo myfifo
$ > cat < myfifo &
$ > echo 'a' > myfifo
```

This works, but cat dies after reading one line. (In fact, what happens is when the named pipe is closed by all the writers, this signals an end of file condition for the reader. So cat, the reader, terminates because it saw the end of its input.).

What if we want to write several times to the pipe without having to restart the reader? How to write several times to a fifo without having to reopen it?

Solution:
In the general case, you'll open a new FileDescriptor (FD) pointing to the fifo, and write through that. For simple cases, it may be possible to skip that step.

Grouping the commands: We have to arrange for all our data to be sent without opening and closing the pipe multiple times. If the commands are consecutive, they can be grouped:

```
$ > cat < myfifo &
$ > { echo 'a'; echo 'b'; echo 'c'; } > myfifo
```

Opening a file descriptor: It is basically the same idea as above, but using exec to have greater flexibility:

```
$ > cat < myfifo &
```

```
# assigning fd 3 to the pipe
$ > exec 3>myfifo

# writing to fd 3 instead of reopening the pipe
echo 'a' >&3
echo 'b' >&3
echo 'c' >&3

# closing the fd
exec 3>&-
```

Closing the FD causes the pipe's reader to receive the end of file indication. This works well as long as all the writers are children of the same shell.

Using tail: The use of *tail -f* instead of *cat* can be an option, as tail will keep reading even if the pipe is closed:

```
$ > tail -f myfifo &

$ > echo 'a' > myfifo
# Doesn't die
$ > echo 'b' > myfifo
$ > echo 'c' > myfifo
```

The problem here is that the process *tail* doesn't die, even if the named pipe is deleted. In a script this is not a problem as you can kill tail on exit. If your reader is a program that only reads from a file, you can still use tail with the help of process substitution:

```
$ > myprogram <(tail -f myfifo) &
# Doesn't die
$ > echo 'b' > myfifo
$ > echo 'c' > myfifo
```

Here, tail will be closed when myprogram exits.

\