EE5012 UNIT 3 – Tutorial

1. Conditional Expressions in Bash

1.1 Comparisons

Variables and values can be compared and program branch decisions can be made based on the results of the comparisons. Some arithmetic comparisons can include the following relational operators:

<=	less than or equal to
>=	greater than or equal to
<	less than
>	greater than
= =	equal
! =	not equal
&&	logical AND
||	logical OR

1.1.1 The if-statement and conditional blocks

A test will determine whether something is true or false. The **test** command is very useful in program control constructs, such as the **if** statement.

A Bash command on termination results in an exit code, which represents a return value that is an integer between 0 and 255. A **0 (zero)** exit code denotes **success**, and a positive number denotes some **failure**. The actual exit code number is application-specific.

There are commands that are intended to *test* some condition or conditions and to return an appropriate *exit* status. The **test** command is designed to do just this and can be used as follows to test the truth of some expression and will *exit* with the status determined by EXPRESSION:

test EXPRESSION

The command can also be called using the following equivalent syntax, which is more commonly used:

[EXPRESSION]

We will look at simple examples of the **test** command to check if a file called *big_file* exists in the home directory. The **-e** means to test if the file exists.

test -e ~/big_file

or

[-e ~/big_file]

Both of the above commands have the same meaning, Note the **necessary requirement** to always use spaces following the opening [bracket and preceding the closing] bracket.

NOTE ON STYLE

The **[-e ~/big_file]** format is preferred to the **test -e ~/big_file** syntax.
However, it far better to use the **[[-e ~/big_file]]** format which will be presented soon in the text.

We will look at some practical examples using such test operations by studying the **if** statement, which will execute a command (or commands) and check the exit code for success or otherwise. Here is the general outline form of the **if** statement:

```
if TEST-COMMANDS
then CONSEQUENT-COMMANDS
fi
```

There is a much preferred style where the ‘then’ keyword is written on the same line as the ‘if’ and the semicolon is required as follows:

```
if TEST_COMMANDS; then
   CONSEQUENT-COMMANDS
fi
```

Note that the **then** and **if** are considered to be separate statements and that is why they are separated by a semicolon when written on the same line.

Here is a simple example using an **if** statement that includes the **if-then-else** construct:

```
if true;  then
  echo "true!"
else
  echo "false!"
fi
```

Note the use of **‘true’** in the first line of the example which is a **built-in** command that always evaluates to true.

The following code checks if the file **big_file** exists in the user’s home directory. Here is a sample program using this test and illustrates the **if-then-else** construction:

```
if  [ -e ~/big_file ];  then
   echo “Yes, big_file is there”
else
  echo “No, big_file is not there”
fi
```

Here is a simple example that includes the **elif** (else if) construct:

```
my_status=Clown
if [ $my_status = "student" ]; then
  echo "You have so much to learn!"
elif [ $my_status = "teacher" ]; then
  echo "You must be kind to students!"
else
  echo "Well Mr. $my_status - you probably think you know it all by now!"
fi
```

2. Expressions with Logical Operators

It is important to note that we do not always need the **if** statement to test conditions. Actions can be performed based on the success of a previous command by using control operators, which are **&&** and **||,** that respectively represent a logical **AND** and a logical **OR**. This gives us the concept that is known as conditional execution. Here is an example command:

```
$  [ $name = "Ann" ]  &&  echo "Her name is $name"
```

This will check the result of the test, and if the result is true (exit code 0), then Bash will execute the echo command and return a true (exit code 0) after the successful echo. If the test had failed Bash would skip the echo command and return a non-zero exit code. Try this out on the command line as follows and you will see there is an echo output because the test is true:

```
$  hristo@hristo-lubuntu18:~$  name="Ann"
$  hristo@hristo-lubuntu18:~$ [ $name = "Ann" ]  &&  echo "Her name is $name"
Her name is Ann
```

Now try this and the test is false; so there is no echo:

```
$ name="Clare"
$  [ $name = "Ann" ]  &&  echo "Her name is $name"
```

So, we now know that the above conditional execution example is equivalent to the **if** statement as follows:

```
if  [  $name = "Ann"  ];  then echo "Her name is $name";  fi
```

Now we will look at a simple example of conditional execution that uses the **OR** logic:

```
mkdir  ~/accounts  ||  echo "Could not make that directory"
```

If the directory **~/accounts** does not already exist, this command will make a directory called accounts in the user’s home directory and the **mkdir** command will return true (exit code 0). No further command will be executed, i.e. the echo command will not be executed.

However, if the directory **~/accounts** does already exist then the **mkdir** returns false and the **echo** command is executed.

Note – it is always good practice to do a check on making a new directory, as the script should know of any failures to make the directory.

The [[command

We have seen that the **test** command can have an alternative syntax called '[' command, which is based on the square brackets []. This syntax is generally preferred over the '**test**' syntax. However, there is a more versatile version that is called the '[[' command, which used doubly squared bracket syntax. Thus, the following three commands, which each compares two strings, are functionally equivalent:

```
if  [ $string_x = $string_y ]; then echo "They are equal!"; fi

if  test $string_x = $string_y; then echo "They are equal!"; fi

if  [[ $string_x = $string_y ]]; then echo "They are equal!"; fi
```

However, the third line uses the ' [[' syntax and this form supports more advanced features which are not evident from this simple example. It is recommended to always use the [[..]] style.

The '[[' command was introduced in the **Korn** shell but because of its added features it was adopted by the Bash shell developers.

An example feature of the '[[' command that does not exist in the '[' command is pattern matching. Here is a test that uses pattern matching:

```
if  [[  $filename  =  *.html ]];  then echo "File name suggests a HTML file!"
```

A list of some of specific tests that are supported by Bash (for all formats of test) are listed in the table below.

Specific test	**Condition for TRUE**
-e FILE	file exists
-f FILE	file is a regular file
-d FILE	file is a directory
-h FILE	file is a symbolic link
-p PIPE	pipe exists
-r FILE	file is readable by you
-s FILE	file exists and is not empty
-t FD	FD is opened on a terminal
-w FILE	file is writable by you
-x FILE	file is executable by you
-O FILE	file is effectively owned by you
-G FILE	file is effectively owned by your group

FILE -nt FILE	first file is newer than the second
FILE -ot FILE	first file is older than the second
-z STRING	string is empty (length is zero)
-n STRING	string is not empty (length is not zero)
STRING = STRING *(== works also)*	the first string is identical to the second
STRING != STRING	first string is not identical to the second
STRING < STRING	first string sorts before the second
STRING > STRING	first string sorts after the second
EXP -a EXP	both expressions are true (AND)
EXP -o EXP	either expression is true (OR)
! EXP	Inverts the result of the expression (NOT)
INT -eq INT	both integers are identical
INT -ne INT	integers are not identical
INT -lt INT	first integer is less than the second
INT -gt INT	first integer is greater than the second
INT -le INT	first integer is less than or equal to the second
INT -ge INT	first integer is greater than or equal to the second

In addition to the list of specific tests listed in the table above, some additional tests are supported by the [[command as follows; this is not an exclusive list:
EXP && EXP:
This is similar to the '-a' operator (**AND**) in the **test** command but the second expression is not evaluated if the first one results as false.

EXP || EXP:
This is similar to the '-o' operator (**OR**) in the **test** command but the second expression is not evaluated if the first one results as true.

STRING = PATTERN:
Here *pattern matching* is supported so this is not a simple string comparison as in the **test** command. The test returns true there is a glob pattern match. Note STRING == PATTERN is valid syntax also.

STRING =~ REGEX:
This results as true if there is a string match on the regex pattern.

Testing for command exit status using 'if'

Note, **$?** is the built-in shell variable for the exit status of whatever last command was used.

The following example will echo a **1** if the file ~/hobbies does not exist, it will echo a **0** if the file does exist:

```
$ [[ -e ~/hobbies ]] ; echo $?
```

Sometimes it is useful to check the exit status using an **if** statement as follows:

```
$ if [[ $? -ne 0 ]] ; then echo "output of previous command is false" ; fi
```

We could have used this syntax in the above statement: **if [[$? != 0]]** …..

2.1 Numeric Comparisons

Numeric comparisons are also supported in the testing using the follow operators to act on integer operands as listed in the table above: -eq, -ne, -lt, -gt, -le and -ge. Here is a small code example:

```
echo "Input your age"
read  your_age
if [[ $your_age -ge 18 ]] ; then
  echo "You can vote!"
fi
```

A little later on, we will learn that if it more desirable to perform arithmetic tests using the double parentheses construct ((…)) to encapsulate the condition, allowing us to use this more intuitive syntax for tests, and this will be the recommended style for arithmetic conditional testing:

```
 if (( $your_age  >=  18 )); then
 echo "You can vote!"
 fi
```

2.1.1 The “case” statement

The **case** statement construct is of the general form that is common with many popular computer programming languages.

Let’s assume we want to write a program where the logic branch will depend on the content of a variable. We could use the **if** statement to do this. Here is an example that looks for a result on testing against a glob. The user is asked to type in three or more characters to identify a particular academic subject, and the intended subject is then selected by the program.

```
#! /bin/bash
echo "Select subject: type at least the three first characters for subject in lower case"
read subj

if [[ $subj = mat* ]]; then
   echo 'You selected: Mathematics!'
elif [[ $subj = sci* ]]; then
   echo 'You selected: Science!'
elif [[ $subj = eng* ]]; then
   echo 'You selected: Engineering!'
elif [[ $subj = lan* ]]; then
   echo 'You selected: Languages!'
elif [[ $subj = his* ]]; then
   echo 'You selected: History!'
elif [[ $subj = geo* ]]; then
```

```
    echo 'You selected: Geography!'
else
    echo 'Sorry - your subject is not in our curriculum.'
fi

exit 0
```

The above code makes some redundant comparisons. It would be neater to use the **case** statement. In a **case** statement each choice is a pattern followed by a right **')'** bracket, then a code block is executed if there is a match. Note, two semicolons are used to end the code block (so as not to confuse this with a statement separator). Each case plus its associated commands is called a **clause**. The matching check is stopped when one case is successfully compared. The ***** pattern can be used at the end to match any case that has not been matched earlier. Each **case** statement is ended with the **esac** statement. Here is the above example again but this time it is written using a **case** statement.

```
#! /bin/bash
echo "Select subject: type at least three first characters for subject in lower case"
read subj

case $subj in
  mat*) echo 'You selected: Mathematics!' ;;
  sci*) echo 'You selected: Science!' ;;
  eng*) echo 'You selected: Engineering!' ;;
  lan*) echo 'You selected: Languages!' ;;
  his*) echo 'You selected: History!' ;;
  geo*) echo 'You selected: Geography!' ;;
  *) echo 'Sorry - your subject is not in our curriculum.' ;;
esac

exit 0
```

In a **case** statement we can use the **'|'** symbol to separate multiple patterns. We could have used this clause in our example above:

```
  mat* | sci*) echo 'You selected: Mathematics with Science.' ;;
```

2.1.2 Arithmetic Conditionals

Arithmetic tests can be made using the double parentheses construct ((…)) to encapsulate the condition. The common relational operators, such as that found in the C language, are supported. For example the following expression returns an exit status of 0, i.e. the expression is true:

```
((  (5 > 2)  &&  (6 <= 9)  )); echo $?
```

So, in an **if** statement we can use this example:

```
$ if  (( (x == y) && (x == z) )); then echo "true"; fi
```

However for a string comparison we could not use the ((…)) construct, we would use:

```
if  [[ "$x" == "$y" && "$a" == "$z" ]]
```

3. Conditional Loops

Now we will learn how to write scripts that use control loops for repetitive tasks using **for**, **while** and **until** loops, where the **until** loop is a variant of the **while** loop. Previously we learned about making basic decisions in scripts, but very often we need to repeat operations and we need to use a loop for that. The **for** loop is used to go through a list of items sequentially. The **while** loop is used where it is not known in advance how many times a loop needs to be repeated.

3.1 The “for” loop

The general format of the **for** construct is:

```
for   var  in  list; do
  commands
done
```

The loop is repeated for each item in **list**, setting the loop index variable **var** to each item in turn. Here **list** can be any list of items: words, numbers, or strings. The list can be literal or generated by some command. Note the following, less favoured, syntax is also valid where the semicolon is not used because the **do** is started on a new line.

```
for   var  in  list
do
  commands
done
```

The **for** loop begins by assigning the first item in the **list** to the variable **var** and it then executes the **command** or list of **commands**. Then the next item in the **list** is assigned to the variable **var** and the command, or list of commands, is executed again. The process continues until each item in the **list** has been processed. Here is an example to display a line three times:

```
for i in {1..3};  do
echo "This is $i iteration!"
done
```

In the above example brace expansion was used to form the list but there are other syntax options; the following syntax examples will all give the same output.

```
for i in {1..3}; do echo "This is $i iteration"; done
for i in 1 2 3; do echo "This is $i iteration"; done
```

```
for i in $(seq 1 3); do echo "This is $i iteration"; done
```

An alternative and very useful syntax for the **for** loop is as follows, which is a style more often used in common programming languages (e.g. in the C language):

```
for  ((  expression;  expression;  expression  ))
```

The first arithmetic expression is evaluated, the loop is repeated so long as the second arithmetic expression is successful, and the third arithmetic expression is evaluated at the end of each iteration. Here is an example to list a line three times as in the above examples:

```
for (( i=1; i < 4 ; i++ )); do
echo "This is $i iteration!"
done
```

In the following example a script program will accept a command line argument to specify a directory and will list the number of lines in each file within that directory. The example illustrates a useful application that uses a **for** loop to process a list of files, using the * wildcard.

```
#!/bin/bash
# List the number of lines in each file of a specified directory

for  file  in  $1/* ; do
  echo  "There are $( wc -l  < "$file" ) lines in $file"
done

exit 0
```

Here is an example output of the script, assume the script is names ‘*line_numbers*’:

```
hristo@hristo-lubuntu18:~/EE5012$ ./line_numbers /home/hristo/EE5012/hobbies/
There are 4 lines in /home/hristo/EE5012/hobbies//model_cars
There are 15 lines in /home/hristo/EE5012/hobbies//model_planes
```

3.2 The “while” loop

The general format of the **while** loop construct is:

```
while  [ test something ] ; do
  commands
done
```

The following script program uses a while loop and will echo five lines to the terminal.

```
#! /bin/bash
num=5
count=1
```

```
while   ((  $count  <=  $num  )) ; do
        echo “This is line number $count”
        ((count++))
done
exit 0
```

The following script example will keep looping until a valid string match is found for the requested name.

```
#! /bin/bash

while   [ "$my_name"  !=  "John Smith" ]  ; do
   echo  "Guess my name"
   read  my_name
   echo  "You guessed $my_name"
done
echo "You guessed it correctly!"

exit 0
```

3.3 The “until” loop

The **until** loop is similar to the while loop. It will execute commands until the defined test becomes true. The **until** is not used very often, the **while** loop is much more widely used in practice. So you may be asking, 'Why bother having the two different kinds of loops? In some cases the logic of a program is easier to read if it is phrased with **until** rather than **while**. Here is a simple example script that uses an **until** loop and will echo 5 lines to the terminal, similar to the earlier script that was based on a **while** loop.

```
#! /bin/bash
num=5
count=1

until   (( $count  >  $num )) ;   do
        echo "This is line number $count"
        ((count++))
done
exit 0
```

3.4 Break and Continue

The **continue** statement causes a loop iteration to be terminated and the next iteration then begins. In the example below a script program will list the number of lines in each file of a specified directory, however, if any file of type ‘directory’ is encountered, that file will not be processed for line counting, and the loop continues to the next iteration.

Do not forget to provide a directory name as an argument is calling this program.

```
#!/bin/bash
# List the number of lines in each file of a specified directory,
# if any directory file is encountered - do not line count it.
# The $# represents the positional parameter
if [ $# != 1 ]; then echo "You must provide a single target dir name. Try again!"; exit; fi

for  file  in  $1/* ; do
   if [  -d "$file" ]; then
      echo "The $file is a directory so will not do line count on it!"
      continue
  fi
echo  "There are $( wc -l < "$file" ) lines in $file"
done

exit 0
```

The **break** statement causes a program to exit the script before its normal ending. Consider a feature extension to the above program where if a file of type '*pipe*' is found then the script is exited.

```
#!/bin/bash
# List the number of lines in each file of a specified directory.
# If any directory file is encountered - do not line count it.
# If a file of type pipe is found then exit the script.

if [ $# != 1 ]; then echo "You must provide a single target dir name. Try again!"; exit; fi


for  file  in  $1/* ; do
   if [  -p "$file" ]; then
      echo "Sorry $file is a pipe so script will exit now!"
      break
  elif [  -d "$file" ]; then
      echo "The $file is a directory so will not do line count on it!"
      continue
  else
    echo  "There are $( wc -l < "$file" ) lines in $file"
  fi
done

exit 0
```

From the above we see the **continue** statement is used to exit a loop iteration; but the **break** statement is used to exit the script completely

3.4.1 Conditional loops and I/O redirection

We saw how to use control loops by testing command results or by reading user input. It is also possible to specify a file from which to read input and in that way to control the loop.

For example the following style can be used to read a file line by line and the loop will terminate when there are no more lines to be read.

```
while  read i;  do echo $i;  done  <  /dir/filename
```

As an example, suppose we have a file called '*list_file_owners*' and it contains a list of entries where each line contains a file name and the respective file owner, as follows:

File: *list_file_owners*

```
joe     file_A
bill    file_B
root    file_C
joe     file_D
bill    file_E
joe     file_F
don     file_G
```

Now assume we are asked to write a script that will display how many files belong to each individual owner; for example **joe** has three files. Here is an example solution that uses a control loop that reads its input from a file.

```
#! /bin/bash
# Display number of files for each owner

# Make an unique list of owners in uniq_list
sort list_file_owners | awk '{print $1}' | uniq  >  uniq_list

# Make simple column titles
echo -e "\nOwner  \t Num_files \n"

# Loop to read each owner and count number of entries
while  read  owner;  do
   x=$( grep "$owner" list_file_owners | wc -l )
   echo  -e   "$owner  \t  $x"
done  <  uniq_list

rm   uniq_list
exit 0
```

The result from the above script will be as follows:

```
$ ./example

Owner Num_files

bill    2
don     1
joe     3
root    1
```

4. Shell script functions

A function is a short script that can be called any number of times within the full script. The Bash shell supports the use of functions, which can be called by the function name. Functions in Bash, like many programming languages, are a useful way to reuse code. The function name must be unique within the script. The commands that make up a function are executed as regular commands. A function is executed within the shell in which it has been declared, i.e. a new process is not created to interpret the commands. A function can be defined in one of two ways as follows:

```
fun_name () {
  command list
}
```

Alternatively, the **function** keyword can be used as follows:

```
function  fun_name {
   command list
}
```

The first style above, that does not use the function keyword, is much preferred and should always be used. The other style is presented here so that the reader can read older code. The function definition must exist in the script before any calls are made to the function. The function name should be descriptive of whatever task is required of the function.

In general programming languages arguments are passed to the function as listed inside the parentheses (), but in Bash the parentheses are used only to state that the declared identifier is a function and thus no arguments will ever exist inside the parentheses.

The following is a simple script program that uses a function called **greet**:

```
#! /bin/bash
# Define the ‘greet’ function
greet ()  {
  my_name=$USER
  echo "Hello $my_name"
}
```

```
# The main program – just calls ‘greet’ a couple of times.
greet     # call the ‘greet’ function
greet     # call the ‘greet’ function

exit
```

The script will give the following outcome; assume the name of the script is ‘*greet*’:

```
hristo@hristo-lubuntu18:~/EE5012/function$ ./greet
Hello hristo!
Hello hristo!
```

Next we will learn how to pass arguments to a function.

4.1 Passing arguments to functions

Arguments are passed into the function in a similar manner to passing command line arguments to a script. The arguments are stated directly after the function name on calling a function. In the actual function the arguments are accessible as positional parameters $1, $2 etc. The following example passes a single argument to a function:

```
#! /bin/bash
# Define the ‘name_greet’ function
name_greet () {
   echo "Hello $1"
}

# The main program – just call ‘name_greet’ a couple of times.

name_greet  Ringo     # call the ‘name_greet’ function
name_greet  George    # call the ‘name_greet’ function

exit
```

The script will give the following outcome; assume the name of the script is ‘*greet1*’:

```
hristo@hristo-lubuntu18:~/EE5012/function$ ./greet1
Hello Ringo!
Hello George!
```

The script example was very simple, passing just a single argument.

Note, the positional parameters passed to a function are not the same ones passed to the actual full script program. On executing a function, the arguments passed to the function become the positional parameters, as seen within the function. The parameter **$#**, which indicates the number of positional parameters, is updated to reflect this change.
The positional parameter **$0** is not changed and is the name of the actual script program. When a function is executing, the shell variable $FUNCNAME is set to the name of the function. On completion of a function, the values of the script program’s positional parameters and the parameter **$#** are restored.-e

4.2 Returning a value from a function

Bash does not return a value from a function in the same sense as functions can return values in most general programming languages. Rather, in Bash a return status can be set, which is similar to a command exiting with an exit status to indicate success or otherwise. A function uses the optional **return** statement to return the exit status.
The **return value** defaults to the exit status of the last command that was executed in the function, where typically a return status of **0** indicates **success**; a non-zero value indicates an **error**. If an **exit** statement is used within a function, then the entire script program is exited, regardless of how deeply nested a function may be.
The function's exit status is accessible using the special **$?** shell variable.

When the **return** statement is executed within a function, the function then completes and then execution resumes with the next command following the function call.

Now, here is a simple example script program where two arguments are passed to a function, and the function calculates the product of the two arguments. The product value is accessed using a global variable. NOTE – generally using such global variables should be avoided.

```
#! /bin/bash
# Example showing function to return a product value using a global variable

# Create the 'product' function
product () {
  product_var=$(( $1 * $2 ))  # global variable since keyword ‘local’ is not used
}

# The main program
product 22 3     # product function called with two arguments
echo  "The answer is: $product_var"

exit
```

The script will give the following outcome; assume the name of the script name is ‘example’:

```
hristo@hristo-lubuntu18:~/EE5012/function$ ./calculate
The answer is: 66.
```

Note, using global variables is often undesirable, which leads us to a short discussion on the scope of a variable in Bash.

4.3 Scope of function variables

By default a variable that is declared in Bash is global, meaning that it is visible everywhere within the script program. A variable can be declared as a local variable by using the keyword **local** when first declaring the variable. If we declare a local variable inside a function, then that variable is seen only within that function. The keyword **local** is used as follows:

```
local  name_of_var=<var_value>
```

Usually in computer programming it is good practice to use local variables inside functions, as it is safer so as to avoid the possibility of inadvertently changing the value of a variable by some other code in the overall program. Consider the following example script:

```
#! /bin/bash
# demonstrate local and global variables

some_func () {
  local  name1='Imposter'
  name2='Pretender'
  echo "Within the function my two friends are $name1 and $name2! "
}

name1='Punch'
name2='Judy'

echo "Before calling the function my two friends are $name1 and $name2! "

some_func
echo "After calling the function my two friends are $name1 and $name2! "

exit
```

Here is the expected output for the script; assume the name of the script is ‘local_var’:

```
hristo@hristo-lubuntu18:~/EE5012/function$ ./local_var

Before calling the function my two friends are Punch and Judy!

Within the function my two friends are Imposter and Pretender!

After calling the function my two friends are Punch and Pretender!
```

Examine the logic of the script to see why the function was able to globally change the **$name2** variable but the value of the **$name1** variable was not changed outside of the function, as this is a local variable.

5. Monitoring Processes with “ps”, “top” and other similar bash commands

The UNIX/Linux operating system has multitasking features. Multiple processes can be run at the same time and processes can communicate with one another, using **pipes** and **signals** to provide IPC (**I**nter**p**rocess **c**ommunication).

5.1 Monitoring Process Execution

To see what processes are running, type the **ps** command (see **man** and **info** pages for a description of the **ps** command). Note, the **ps** command’s options may be a little different for other flavours of UNIX/Linux shells.

For example type:

ps au

A list of process activity is shown, something like as follows:

```
$ ps  au

USER       PID    %CPU  %MEM   VSZ   RSS   TTY     STAT  START    TIME    COMMAND
root       3321    0.0    0.0    1876   408   tty1    Ss+    2007        0:00   /sbin/mingetty tt
root       3340    0.0    0.0    2484   408   tty2    Ss+    2007        0:03   /sbin/mingetty tt
donal      17205   0.0    0.0    4420  1468   pts/2   Ss     08:31       0:25   -bash
joe        19168   0.0    0.0    2928   776   pts/2   R+     09:30       0:00   ps au
```

The various fields can be described as follows:

USER	Name of the process user
PID	Process ID number
%CPU	What percentage of the CPU the process is using
%MEM	What percentage of memory the process is using
VSZ	Virtual memory usage
RSS	Real memory usage
TTY	Terminal associated with USER
STAT	The current state of the process
START	Time when the process started
TIME	Total CPU usage time
COMMAND	Name of process

The **STAT** field indicates the status of each process. Some of the codes are as follows:

R	Running or runnable
S	Sleeping but interruptible (for a short time, waiting for an event to complete)
L	Waiting to acquire a lock
Z	Zombie (terminated but not reaped by its parent process)
D	Uninterruptible sleep (woken from outside, usually I/O)
T	Stopped by job-control signal

t	Stopped by debugger during tracing
X	Dead (should never be seen)

For **BSD** format and when the **stat** keyword is used, additional characters have the following meaning:

+	Foreground process group with control of the terminal
s	Process is a session leader
l	Multithreaded process
L	Process has pages locked in memory
<	High-priority process (not nice to other users)
N	Low-priority process (nice to other users)

Now we will demonstrate a really busy program which will take up a lot of the processor's time. We will write a simple program, called **busy_wait**, which is spinning in a loop, doing nothing except using up valuable processor time, since the condition that it is waiting on will never happen.

Create the **busy_wait** shell script program below, make it executable (using **chmod**).

The **busy_wait** program

```
#!  /bin/bash
# Meaningless program – but useful to demonstrate the busy-wait concept

while true ; do
  ((  x++  ))
done
```

Run this program as a **background** process by typing:

```
./busy_wait  &
```

Note, the **&** symbol following a command tells the system that the program is to be run in the background. The shell can continue to run in the foreground.

Now type **ps au** and see what percentage CPU utilisation (%CPU) that this **busy-wait** program is using (probably close to 100% utilisation).

Run the **busy_wait** program a few more times by typing **busy_wait &** again.

Now type **ps au** and this will give a display something like the following:

```
USER   PID   %CPU  %MEM    VSZ    RSS   TTY    STAT START  TIME COMMAND
root   1092  0.4   1.9     672004 78132 tty7   Ssl+  09:00  0:49 /usr/lib/xorg/Xorg -core
root   1096  0.0   0.0     23288  1980  tty1   Ss+   09:00  0:00 /sbin/agetty
hristo 2396  0.0   0.1     29812  5180  pts/0  Ss    09:07  0:00 bash
hristo 3545  101   0.0     19992  1268  pts/0  R     11:58  0:24 /bin/bash ./busy_wait
hristo 3546  03    0.0     19992  1236  pts/0  R     11:58  0:21 /bin/bash ./busy_wait
hristo 3547  100   0.0     19992  1240  pts/0  R     11:58  0:19 /bin/bash ./busy_wait
```

This is a simple example of multitasking; a number of programs (or a number of copies of the same program) are running at the same time, sharing the processor's time.

You can **kill** any one of these processes by using the **kill** command, e.g. **kill 3545** where 3545 is the process PID number. Now kill all of these **busy_wait** processes.

5.1.1 The “top” utility

There is a useful utility called **top** that will periodically display the process activity. Try this by typing **top**. Run the **busy_wait** script and monitor it in the **top** display. The **top** utility’s data is updated on a periodic basis, so you can watch this activity on the screen. Your output should be of the following format:

```
Tasks: 147 total,   4 running, 100 sleeping,   0 stopped,   0 zombie
%Cpu(s): 75.1 us,  0.0 sy,  0.0 ni, 24.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4039172 total,  3434036 free,   279968 used,   325168 buff/cache
KiB Swap:   960756 total,   960756 free,        0 used.  3540844 avail Mem
PID   USER    PR   NI   VIRT     RES    SHR S    %CPU    %MEM    TIME+     COMMAND
2126  hristo  20   0    19992    1184   1044 R   100.0   0.0     0:23.94   busy_wait
2127  hristo  20   0    19992    1100   956 R    100.0   0.0     0:20.45   busy_wait
2128  hristo  20   0    19992    1136   992 R    100.0   0.0     0:20.95   busy_wait
102   root    20   0    0        0      0 I      0.3     0.0     0:00.09   kworker/u8:2
2129  hristo  20   0    48904    3768   3180 R   0.3     0.1     0:00.03   top
1     root    20   0    225004   8908   6868 S   0.0     0.2     0:00.93   systemd
```

6. Signals

A program will need to be able to deal with unpredicted external events. Such events are often referred to as interrupts, as they interrupt the normal flow of a program. At a higher level, UNIX/Linux sends **signals** to processes to indicate that some type of event has occurred.

A signal is a short notification sent to a process to notify that process of a particular event. The signal interrupts the process so that it can act on this received signal. This provides a mechanism for handling asynchronous events in a system. There are a number of different kinds of signal, and each kind signal is associated with an integer identifier number and an associated symbolic name.

A signal is simply an event notification that is sent by some running process. Any process can send a signal to another process, as long as it has permission to do so. Signals do not carry any data. A process does not know what process sent the signal.
Signals are often used by the operating system kernel to notify a process that some event has occurred, without the process needing to poll for the status of the event. We say that the signal happens asynchronously.
When a process receives a signal, the corresponding **trap** code gets called. The Bash **trap** command allows the user to specify a command, or a function, that is to be executed when the shell receives a particular signal. The **CRTL-Z** and **CRTL-C** key inputs make direct use of signals. The **trap** command causes the shell to execute a specified command when a

numbered signal, or signals, arrive. If numerous signals arrive they are handled in numerical order.

6.1 Signal identifiers

POSIX has standardised the signal handling scheme for UNIX. Each signal has a symbolic name starting with the prefix SIG. For example SIGKILL is the signal sent when a process is forcefully terminated. Signals are all defined in the header file **<signal.h>**. The signals are pre-processor definitions that represent positive integers, i.e. each signal is associated with an integer identifier number as well as its symbolic name. The signal numbers start at 1 (SIGHUP). Signal number 0 has a special interpretation that we will not cover here.

There are many different signals defined. Use the **kill -l** command to see what signals are supported on your system. Here is an example (incomplete) output:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

For now we will consider some of the more common signals as listed in the table below.

Signal	**Number**	**Explanation**
SIGHUP	1	Hangup – controlling terminal no longer connected.
SIGINT	2	Interrupt – usually from keyboard (CTRL+C)
SIGQUIT	3	Quit – user has pressed a quit key (e.g. CTRL+\)
SIGKILL	9	Kill. Cannot be trapped or ignored. Fatal with no clean-up.
SIGTERM	15	Process termination signal. Default signal sent by kill

SIGHUP, the UNIX ‘hangup’ signal, is used to signal a process to say that the terminal is no longer connected. However, the SIGHUP signal is sometimes used as a general signal for other purposes. By default, the signal will terminate the process. By using the **trap** command, alternative action can be programmed to specify the response to the SIGHUP signal, or the response to other signals...

The **kill** command is used to send a signal to a running process, using the following syntax:

kill -(SignalNumber|SignalName) ProcessID

For example, if you wanted to send a SIGHUP signal (which is signal number 1) to the process with the PID, 4365, you could use the command:

```
kill  -HUP  4365
```

or, the equivalent command is:

```
kill  -1  4365
```

Note, it is generally much preferred to use signal names, rather than signal numbers, so as to be more portable for different versions of UNIX/Linux shells. Use the signal name without the leading 'SIG' for more general compatibility.

It is important to note that not all signals can be trapped. The SIGKILL signal, signal number 9, cannot be trapped. Consider the following command which uses signal **9**:

```
kill   -KILL  4365
```

This command will be received by process 4365 and a signal trap cannot be programmed to intercept the SIGKILL signal.

The SIGINT signal, which is signal number 2, is the **interrupt** signal which is generated from the user keyboard by using the CTRL+C keys, and sometimes by other defined keys as well.

The SIGQUIT signal, which is signal number 3, is also generated from the keyboard (usually) and has a special use.

The SIGTERM signal, which is signal number 15, is usually generated by another process. The kill command uses this signal by default, so if we issue the command **kill 19291**, the SIGTERM signal is sent to the process with PID number 19291.

6.2 Using a "trap"

The general syntax of the trap command is:

```
trap  'command'  <signal list>
```

Note the use of single quotation marks in the trap command. The 'signal list' is often a single signal, but can be a list of signals. When any one of the signals is received then the specified command is executed. The setting of a trap overwrites a previous trap on a given signal.

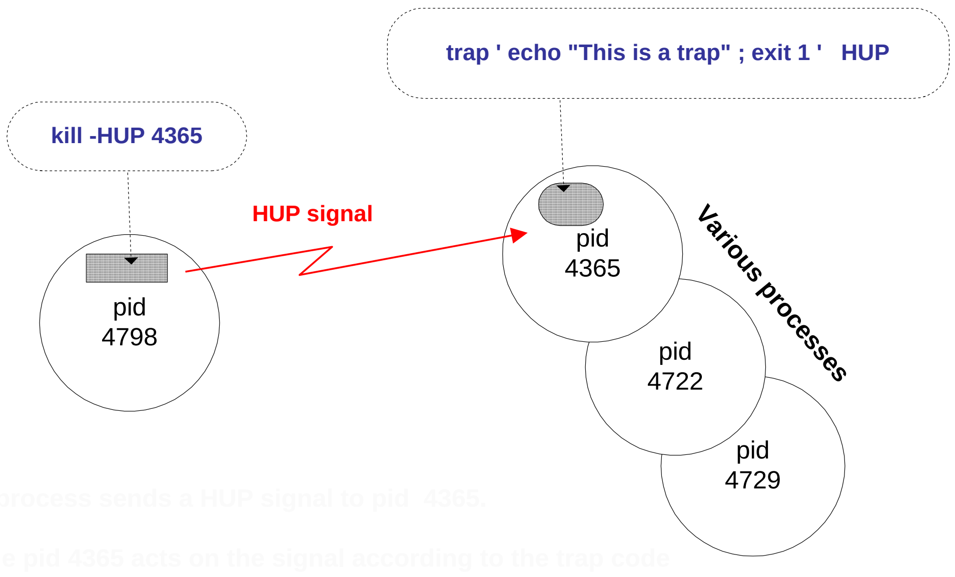
Here is a simple example program that uses the **trap** command. The program will continuously loop, but will exit with the message "**This is a trap**" displayed if a SIGINT signal is sent to this program.

```
#! /bin/bash
trap 'echo "This is a trap" ; exit'  INT

while true;  do
  echo  "I'm looping"
  sleep 1
done
```

Try out the above program; when it is running hit CTRL-C on the keyboard and you will see the altered behaviour of the CTRL-C operation.

The diagram below illustrates a HUP signal being sent (using **kill**) and a **trap** acting upon the receipt of that signal.


trap ' echo "This is a trap" ; exit 1 '   HUP
kill -HUP 4365
HUP signal
pid 4798
pid 4365
pid 4722
pid 4729
Various processes
A process sends a HUP signal to pid 4365.
The pid 4365 acts on the signal according to the trap code

Other examples for the trap command are listed below:

1) The trap command allows the use of the **signal number** or **signal name**:

trap ‘ echo “This is a trap” ‘   1   # uses signal number rather than signal name

2) The trap command using sequential commands:

trap ‘ echo “This is a trap” ; exit 1 ‘   1   # includes a second command

3) The trap command allows the use of more than one signal:

trap ‘ echo “This is a trap” ‘ SIGINT   SIGHUP   # either signal will trap

Some special cases of interest are as follows:

If the command is an empty string, then the signals are ignored, as with the following command:

trap ‘ ‘   SIGINT

This is a useful way to ignore signal interrupts. Note, the signal **9**, i.e. **SIGKILL**, cannot be ignored.

The following command will restore the default behaviour for each signal in the list:

trap - <signal list>

6.2.1 Trap using a function

The **trap** commands are often written as a single function. Consider the following example where on receipt of a SIGINT signal a program will remove all file in the home directory with the extension '.tmp'.

```
#! /bin/bash
# The function is written here before the main code
trap_function () {
  rm -f ~/*tmp              # remove files with extension .tmp
  echo "This program is closing now!"
  echo "I have removed all of your .tmp files!"
}

# Define response to a SIGINIT signal
trap 'trap_function ; exit'   SIGINT

# The main code is here - it is just a simple loop to simulate real activity
while true; do
  echo "I'm looping"
  sleep 1
done
```

EXAMPLE QUESTION

Consider the Bash script exhibit program as below.

```
# Exhibit program  D.H. 15/June/2015  ver. 1.0.0
#! /bin/bash

# The main code is here
./progB &       # start program progB in the background

# a simple loop to simulate real activity
while true ; do
   echo "I'm looping"
   sleep 1
done

wait     # wait for child to exit properly
exit
```

Modify this program so that it will include a **signal trap**. The **trap** will do the following:

i) Acts on receipt of a **SIGINT** signal (i.e. Ctrl+C from keyboard)
ii) Contains a function called **trap_function()**
iii) The **function** does the following:
   - displays (echoes) a simple message to say what is the **PID** for **progB**
   - sends a **TERM** signal to the running **progB** program
   - properly exits the script program without **orphaning** progB

NB: the shell variable **$!** *is always the process ID for the last background command*

SAMPLE ANSWER TO ABOVE

```
#! /bin/bash
# Solution to the above question
trap_function() {
  echo
  kill   -TERM   "$!"
  echo "I have just terminated progB who's PID was $! and I am now exciting!"
  wait                        # wait for child to exit properly
  exit
}

trap  'trap_function'  SIGINT

# The main code is here
./progB &       # start program progB in the background

# a simple loop to simulate real activity
while true ; do
   echo "I'm looping"
   sleep 1
done

wait     # wait for child to exit properly
exit
```