DHA Suffa University
CS 206 – Operating Systems – Lab
Fall 2017

Lab 07 – Functions in Shell Scripting & Introduction to Processes

Twitter: @oslabatdsu

Objective(s):

- Learn about Functions in Shell
- Arguments in a Shell Function
- Return Value of a Shell Function
- What are Processes?
- Basic Information about Processes
- Starting a Process
- Background and Foreground Processes
- Killing a Process

Functions in Shell

A function in Shell can be defined as follows:

```
function_name()
{
}
```

E.g:

```
#Defining the function
Greet()
{
    echo Hello
}
#Invoking the function
Greet
```

Point to remember: You should only invoke the functions after it has been defined.

Arguments in a Shell Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by $1, $2 and so on.

E.g.

```
#Defining the function
Add()
{
    echo `expr $1 + $2`
}
#Invoking the function
Add 8 6
```

Returning a value from a function

```
#Defining the function
Add()
{
    temp=`expr $1 + $2`
    return $temp
}
#Invoking the function
Add 7 5
sum=$?
echo The Sum is $sum
```

$? Depicts the value returned by the last command. Assigning that value to a variable means you are catching the value returned by the function that you invoked in the previous command.

Processes:

When you execute a program on your Unix system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in Unix, it creates, or starts, a new process. When you tried out the **ls** command to list the directory contents, you started a process. A process, in simple terms, is an instance of a running program.

The operating system tracks processes through a five-digit ID number known as the **pid** or the **process ID**. Each process in the system has a unique **pid**.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

Starting a Process

When you start a process (run a command), there are two ways you can run it –

- Foreground Processes
- Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the **ls** command. If you wish to list all the files in your current directory, you can use the following command.

```
$ls ch*.doc
```

This would display all the files, the names of which start with **ch** and end with **.doc** −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc   ch02-1.doc
```

The process runs in the foreground, the output is directed to my screen, and if the **ls** command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in the foreground and is time-consuming, no other commands can be run (start any other processes) because the prompt would not be available until the program finishes processing and comes out.

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

```
$ls ch*.doc &
```

This displays all those files the names of which start with **ch** and end with **.doc** –

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc   ch02-1.doc
```

Here, if the **ls** command wants any input (which it does not), it goes into a stop state until we move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and the process ID. You need to know the job number to manipulate it between the background and the foreground.

Press the Enter key and you will see the following –

```
[1]   +   Done                ls ch*.doc &
$
```

The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows −

```
$ps
PID       TTY      TIME        CMD
18358     ttyp3    00:00:00    sh
18361     ttyp3    00:01:31    abiword
18789     ttyp3    00:00:00    ps
```

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example −

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fields displayed by **ps -f** command –

S.No.	Column & Description
1	**UID** User ID that this process belongs to (the person running it)
2	**PID** Process ID
3	**PPID** Parent process ID (the ID of the process that started it)
4	**C** CPU utilization of process
5	**STIME** Process start time
6	**TTY** Terminal type associated with the process
7	**TIME** CPU time taken by the process
8	**CMD** The command that started this process

There are other options which can be used along with **ps** command –

S.No.	Option & Description
1	**-a** Shows information about all users
2	**-x** Shows information about processes without terminals
3	**-u** Shows additional information like -f option
4	**-e** Displays extended information

Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows –

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here, the **kill** command terminates the **first_one** process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows –

```
$kill -9 6738
Terminated
```

Parent and Child Processes

Each unix process has two ID numbers assigned to it: The Process ID (pid) and the Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check the **ps -f** example where this command listed both the process ID and the parent process ID.

Sleep Call

Sleep call stops and wait till the argument number of seconds to run the next command. Example:

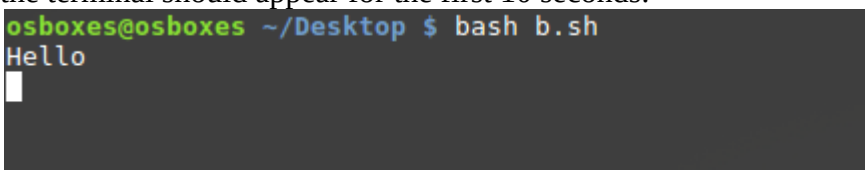
```
echo "Hello"
sleep 1
echo "Wait..."
sleep 1
echo "for..."
sleep 1
echo "it..."
sleep 1
echo "World"
```

Running a process in Background:

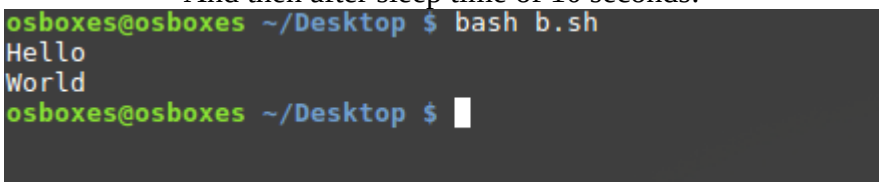
To run a process in background, we use an ampersand sign in the end of process call. For Example, consider the following script:

```
echo "Hello"
sleep 10
echo "World"
```

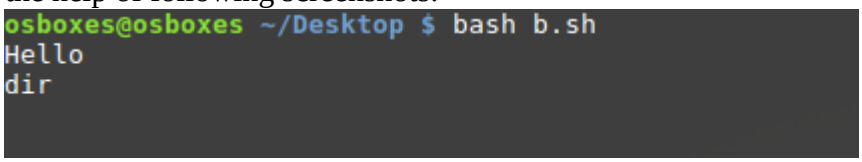
When we run the above script in foreground (i.e. the way we have done in last sessions), the following view of the terminal should appear for the first 10 seconds:



And then after sleep time of 10 seconds:



The point to remember is that running a process in foreground does not allow any other process to be run concurrently. For example, if we want to, say, show directories present on desktop (i.e. running a command of *dir*), the command won't be executed until the foreground process running of bash is completed. Although we could be able to write the command and press enter, but the command of *dir* would have to wait until the foreground process completes its execution. This whole process can be understood with the help of following screenshots:



This *dir* would wait until the execution of *b.sh* is completed i.e. “World” is printed. Therefore, after ten seconds and the completion of *b.sh*, the following screen will be shown in which dir (which was queued for execution) will run.

```
osboxes@osboxes ~/Desktop $ bash b.sh
Hello
dir
World
osboxes@osboxes ~/Desktop $ dir
a.sh  b.sh
osboxes@osboxes ~/Desktop $
```

However, we can overcome this wait state by making the bash script run in background, which would allow us to run other commands and processes while a process (bash script, in this case) runs in background. It can be done only by adding an ampersand sign at the end of the command you want to run in the background.

```
osboxes@osboxes ~/Desktop $ bash b.sh&
[1] 3477
Hello
osboxes@osboxes ~/Desktop $
```

The following screenshot shows that while *b.sh* is running in the background, *dir* has been executed. This screenshot is taken between the ten seconds execution time of *b.sh.*

```
osboxes@osboxes ~/Desktop $ bash b.sh&
[1] 3494
osboxes@osboxes ~/Desktop $ Hello
dir
a.sh  b.sh
osboxes@osboxes ~/Desktop $
```

After ten seconds, no matter what we are doing in the terminal, “World” will be printed on the screen as follows:

```
osboxes@osboxes ~/Desktop $ bash b.sh&
[1] 3614
osboxes@osboxes ~/Desktop $ Hello
dir
a.sh  b.sh
osboxes@osboxes ~/Desktop $ World
```

Lab Task:

Create a shell script to do the following:

- Print Hello n times, read n from user. Every Hello should print after three second sleep time and every Hello should tell the iteration number (starting from 1) E.g. Hello 1, Hello 2.. upto Hello n.
- After writing the shell script, run the script in background. Keep clearing the screen after every hello printing.

Killing a Process running in background

When a process is run in background, its process number and PID is shown on the terminal before executing the process. In order to kill a process running in background, you need to do the following two tasks:

- Search for the process ID (by running the command: pgrep wget)
- Kill the process by using its PID shown on terminal when the process was started OR you can always look for the PID of the process you want to kill using the command of ps.

For Example:

```
osboxes@osboxes ~/Desktop $ bash b.sh&
[1] 3695
osboxes@osboxes ~/Desktop $ Hello
pgrep wget
osboxes@osboxes ~/Desktop $ kill 3695
[1]+  Terminated              bash b.sh
osboxes@osboxes ~/Desktop $
```

Lab Assignment 07

1. **Write a Shell Script (using recursive function) to print the first n Fibonacci numbers. (input n by user)**
2. **Modify Lab task and add the following functionality to it:**

- **Keep asking user in every cycle (after 2 seconds sleep time) if he/she wants to kill the bash process before completing the printing of Hello cycles, if so, kill the bash process.**

Submission Instructions:

1. **Number your .sh files as question number e.g. Q1.sh, Q2.sh, etc. (Q is in upper case)**
2. **Create a new folder named cs152abc where abc is your 3 digit roll #. e.g. cs152111.**
3. **Copy all the .sh files into this folder.**
4. **Right-click on the folder you created and create a zip file by selecting the option**
   - **"Send to" and selecting "Compressed (zipped) folder" [for windows].**
   - **"Create Archive" and change option to ".zip" instead of ".tar.gz" and click on "Create". [for linux]**

   Now make sure a zip file named cs152abc.zip is created e.g. *cs152111.zip*.
5. **Upload the assignment solution on LMS under the assignment named Lab 05 Assignment – XX, where XX is your section name.**
6. **Due Date for assignment submission is Oct 14, 2017.**