

Test Lab

Linux Shell / GNU Make

Embedded Linux

Julia Desmazes
Michael Nissen

October 17, 2017

Command line, environment variables, and user permissions

1. In the bash prompt, what is the meaning of the character `~` ?

The tilde (`~`) character refers to the users home directory. The full path is `/home/user`.

2. Explain the behaviour of running in the order `VV=3`, `export VV`, `bash`, `unset VV`, `exit` and finally `echo $VV` ?

Environment variables is a value, named at runtime, that can affect the way running processes behave. A running process will be able to query the value of an environment variable for example, to discover a location to store files. When creating an environment variable, it will persist in the shell in which it was initialised.

When exporting the created environment variable by using `export VARNAME`, it sets the environment variable not only for current shell, but all other processes started from that shell.

- Set local variable `VV` to 3
- Export local variable as environment variable
- Enter the bash shell
- Unset the local variable
- Exit the bash shell
- Display the environment variable

3. How to run `/home/user/ls` instead of `/usr/bin/ls` automatically by typing `ls` without changing the behaviour of the other commands?

An easy way to achieve this kind of behaviour, is to create a shell alias. A shell alias is simply a shortcut to reference a command. It can be used to avoid typing long commands or as a mean to run increase efficiency. In this example, we want to run the `/home/user/ls` command instead of the builtin one located in `/usr/bin/ls`. As the `/usr/bin` folder is added to the path of the shell, one is able to run any script from that folder by simply entering the name of that script.

As we don't necessarily want to add the `home/user` folder to the path, we can create an alias to the specific file we need.

One can create an alias using `echo` and redirecting it to the shell configuration file:

```
echo alias 'ls="/home/user/ls"' >> .zshrc
```

 where the last file is your shell configuration file.


```

1 #!/bin/bash
2
3 #init i
4 i=1
5 #read line by line
6 for line in $(cat /etc/passwd)
7 do
8     #increment i
9     i=$((i+1))
10
11     #parse and modify output stream with sed
12     echo $i:"$line" | sed 's/^\[^\:]*\):\[^\:]*\):\[^\:]*\):\[0-9]*\):\[0-9]*\):
13     :\[^\:]*\).*\/\1 Username: \2 ,Password: encrypted , UID: \4, GID: \5, Home: \6/'
14 done

```

Code Snippet 1: RegEx in Shell Script

GNU Make

In order to compile both natively and cross-compile for our RaspberryPi we check at the start of our makefile for the variable `CROSS_COMPILE`. If set, we compile with the cross compilation toolchain that we have stored in tools from our linux kernel compilation lab located in `/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabi-hf-`. If not, this variable will not be set and we will be using the machines standard compiler: `gcc` or `g++`, depending on the value given to `TMPGCC` at the start of the makefile.

To access `CROSS_COMPILE` in the makefile we have to make sure the variable is exported to the bash `export CROSS_COMPILE`.

The rest of the makefile is very standard; we generate an object files based on their `.c` files and link them together to create our executable main. To clean we select all object file `*.o` and delete them as well as our generated executable.

```

1 # if we wanted to cross compile for rpi we would use the toolchain we have downloaded for
2   the kernel compilation : arm-linux-gnueabi-gcc, but to use it we would have to
3   specify a path. In our case we have exported the path to the folder in the variable
4   CROSS_COMPILE
5
6 TMPGCC=gcc
7
8 ifeq (${CROSS_COMPILE}, '')
9     echo "CROSS_COMPILE is not set";
10    GCC=${TMPGCC}
11 else
12     # to avoid recursive referencing to GCC variable
13     GCC=${CROSS_COMPILE}${TMPGCC}
14 endif
15
16 ADD=minmax
17 DEPS=minmax${HEADER_SUFFIX}
18
19 all: main
20
21 main: main.o minmax.o
22     ${GCC} -o main minmax.o main.o
23
24 main.o: main.c minmax.h
25     ${GCC} -c main.c
26
27 minmax.o: minmax.c minmax.h
28     ${GCC} -c minmax.c
29
30 clean:
31     rm -f *.o main

```

Code Snippet 2: MinMax Makefile

Managing Process, System Call `fork()`

The system call `fork()` is used to create processes and returns a process ID. When calling `fork()`, it creates a new process which becomes a *child* process of the caller. After a child process has been created,

both the *child* and *parent* process will execute the next instruction following the `fork()` system call. Therefore, one has to distinguish the difference between the two processes. This can be done by testing the process ID returned by the system call.

If `fork()` returns a negative value, the creation of a child process was unsuccessful. It will return a zero to the newly created *child* process, and it will return a positive value, the `pid` of the *child* process, to the parent.

Therefore it is simple to check what which is the child process - this can be seen in Code Snippet 1 below.

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main() {
6
7     // The process ID
8     pid_t pid;
9
10    // Store the process ID
11    pid = fork();
12
13    // Check for the process ID
14    if (pid == 0) {
15
16        // Child process
17
18        // Create a new fork
19        pid = fork();
20
21        // Check for the process ID
22        if (pid == 0) {
23
24            // Grand child process
25            printf("Child of Child");
26        } else {
27
28            // Child process
29            printf("Child");
30        }
31    } else {
32
33        // This is the parent process
34        printf("Parent");
35    }
36 }
```

Code Snippet 3: Managing system calls