**Compilation On Different Targets**

Document outlining the compilation steps on different targets (x86 Ubuntu, Raspberry Pi, and BeagleBone), follow these general steps:

1. **Development Environment Setup**:

* For X86 Ubuntu, ensure the Native-compilation tools installed.
* For Raspberry Pi and BeagleBone, ensure that you have cross-compilation tools installed on your development machine. You will need the appropriate toolchains for ARM architecture.

1. **Set Up Cross-Compilation Toolchain**:

* Follow the instruction of **ELA-Lab-Exercise-007-Building-A-Cross-Compilation-Toolchain** to set up the cross-compilation Toolchain. This might involve downloading and configuring the toolchain specific to your target platform (x86 Ubuntu, Raspberry Pi and BeagleBone).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **Build steps for X86**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## **Step 1: Environment Setup**

1. **Development Environment Setup**:
   * Ensure that you have a C compiler (such as GCC) installed on your Ubuntu system.
2. **Navigate to Your Code Directory**:
   * Open a terminal and navigate to the directory containing your C code and Makefile.

## **Step 2: Compilation & Verification**

1. Run the **make** command to compile your code:

$ make

1. This will execute the compilation process defined in your Makefile and generate the executable binary file.
2. Verify that the compilation was successful by checking for the presence of the generated binary file and Obj file.

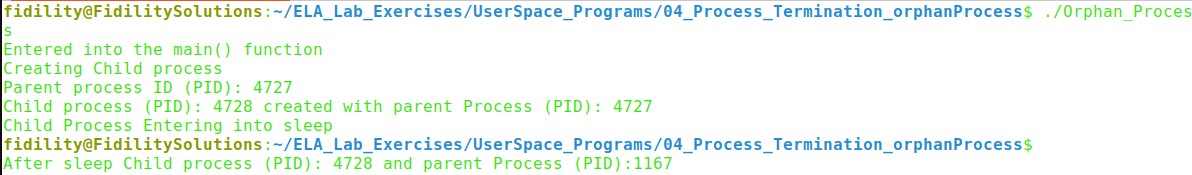


## **Step 3: Running on Platform**

1. Once you're in the correct directory, execute the generated executable file using the **./filename** command. Replace **filename** with the name of your executable file.

$ ./filename

1. The output will be as given below:



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **Build steps for BBB**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## **Step 1: Environment Variables Setup**

1. **Set the ARCH and CROSS\_COMPILE** **environment Variables**

* The ARCH environment variable specifies the target architecture for compilation, in this case, ARM.
* The CROSS\_COMPILE environment variable specifies the prefix for the cross-compiler binaries.

$ export ARCH=arm

$ export CROSS\_COMPILE=arm-linux-gnueabihf-

1. **Set the PATH to the Cross-Toolchain:**

* The PATH environment variable defines the directories where the system looks for executable files.

$ export PATH=${HOME}/ela\_lab\_exercises/bbb\_build/toolchain/gcc-linaro-7.5.0-2019.12-x86\_64\_arm-linux-gnueabihf/bin/:$PATH

## **Step 2: Compilation & Verification:**

1. Run the **make** command to compile your code using the cross-compilation toolchain:

$ make

1. Verify that the cross-compilation was successful by checking for the presence of the generated binary file.

## **Step 3:** **Transfer Binary file to target**

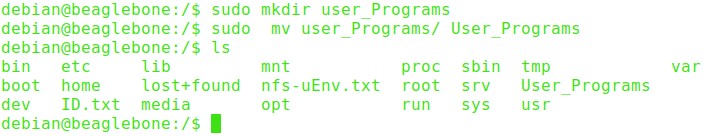
1. Boot the board from SD card and login into the target
2. Set the connection between the host and the target using the below command in the host

$ssh target\_name@target\_ip\_address

Example: $ssh [root@10.10.3.233](mailto:root@10.10.3.233)

**Note:** To get the ip\_address give **ifconfig** command from target terminal

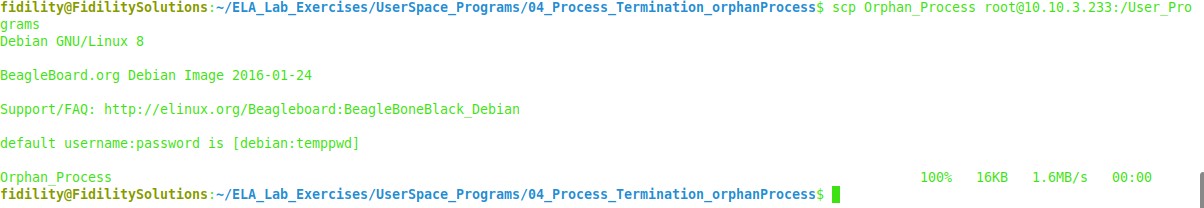
1. Create a directory in the target named User\_Programs



1. To copy the files from the host to the target directory give the SCP command in the host

$ scp <binary\_file> <username>@<ip\_address>:<destination\_directory>

Example: $scp Orphan\_Process [root@10.10.3.233:/User\_Programs](mailto:root@10.10.3.233:/User_Programs)



1. Find the file is copied into the target directory User\_Programs

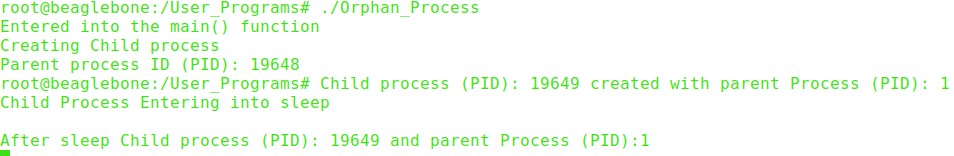


## **Step 4: Running on Platform**

1. Once you're in the correct directory, execute the generated executable file using the **./filename** command. Replace **filename** with the name of your executable file.

$ ./filename

1. The output will be as given below:



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **Build steps for Raspberry Pi 4B**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## **Step 1: Environment Variables Setup**

1. **Set the ARCH and CROSS\_COMPILE environment Variables**

* The ARCH environment variable specifies the target architecture for compilation, in this case, ARM.
* The CROSS\_COMPILE environment variable specifies the prefix for the cross-compiler binaries.

$ export ARCH=aarch64

$ export CROSS\_COMPILE=aarch64-linux-gnu-

1. **Set the PATH to the Cross-Toolchain**

* The PATH environment variable defines the directories where the system looks for executable files.

$ export PATH=${HOME}/ela\_lab\_exercises\_rpi/rpi\_build/toolchain/gcc-linaro-7.5.0-2019.12-x86\_64\_aarch64-linux-gnu/bin/:$PATH

## **Step 2: Compilation & Verification:**

1. Run the **make** command to compile your code using the cross-compilation toolchain:

$ make

1. Verify that the cross-compilation was successful by checking for the presence of the generated binary file.

## **Step 3: Transfer Binary file to target**

1. Boot the board from SD card and login into the target.
2. Set the connection between the host and the target using the below command in the host.

$ssh target\_name@target\_ip\_address

Example: $ssh [root@10.10.1.27](mailto:root@10.10.1.27)

**Note:** To get the ip\_address give **ifgonfig** command

1. Create a directory in the target named User\_Programs



1. To copy the files from the host to the target directory give the SCP command in the host

$ scp <binary\_file> <username>@<ip\_address>:<destination\_directory>

Example: $scp Orphan\_Process [root@10.10.1.27:/User\_Programs](mailto:root@10.10.1.27:/User_Programs)



1. Find the file is copied into the target directory User\_Programs

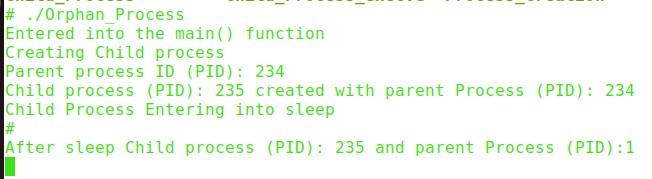


## **Step 4: Running on Platform**

1. Once you're in the correct directory, execute the generated executable file using the **./filename** command. Replace **filename** with the name of your executable file.

$ ./filename

1. The overall output will be as given below:



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **Understanding Processes Using /proc Interface and ps Command**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* In Linux, the **/proc** filesystem serves as a virtual interface to kernel data structures. It provides valuable insights into various system parameters, including detailed information about running processes.
* Combined with the **ps** command, users can effectively monitor and manage processes during runtime.

## **Using /proc Interface**

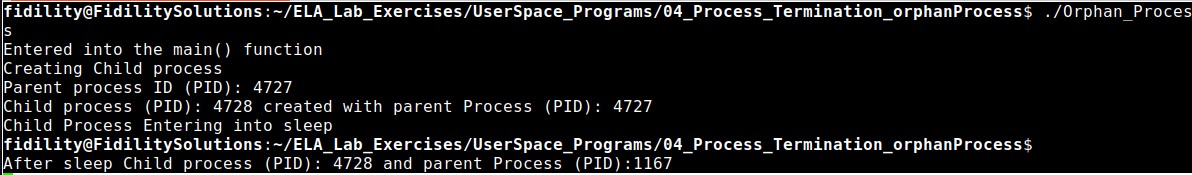
1. First, let's identify the PID (Process ID) of a running process, for example, in our case the process id for Orphan\_Process program is 4728
2. # Now, let's use the `cat` command to read information about this process from `/proc`( cat /proc/$pid/status).

$ cat /proc/4728/status

1. The **/proc/[PID]/status** file contains detailed status information about the process, including its state, memory usage, CPU usage, parent process ID, and more. So the below image showing the child process is an Orphan Process.



## **Using ps Command**

1. The **ps -ef/ps aux** command provides a concise overview of all processes running on the system, displaying detailed information in a full-format listing.
2. The following image showing the child process entered sleep for long time so the parent process exited before child process in that case the kernel allocated new parent to child process and the process ids shown below. 
3. The image depicts entry of child process present in process table.

