**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**

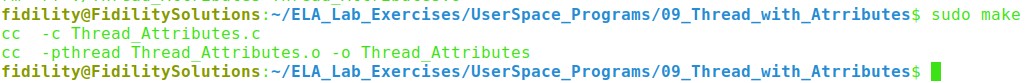
1. **Development Environment Setup**:
   * Ensure that you have a C compiler (such as GCC) installed on your Ubuntu system.
2. **Navigate to Your Code Directory**:
   * Open a terminal and navigate to the directory containing your C code and Makefile.

## **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.

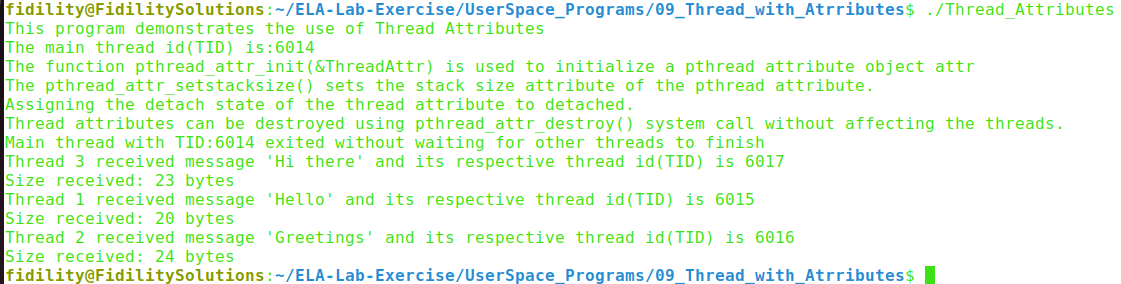


## **Step 3: Running on Platform**

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**

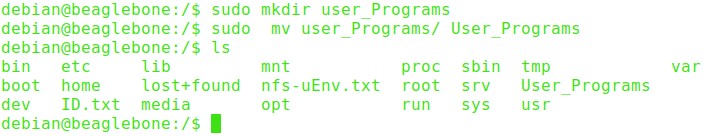
1. Boot the board from SD card and login into the target
2. Set the connection between the host and the target using the below command in the host

$ssh target\_name@target\_ip\_address

Example: $ssh [root@10.10.3.233](mailto:root@10.10.3.223)

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

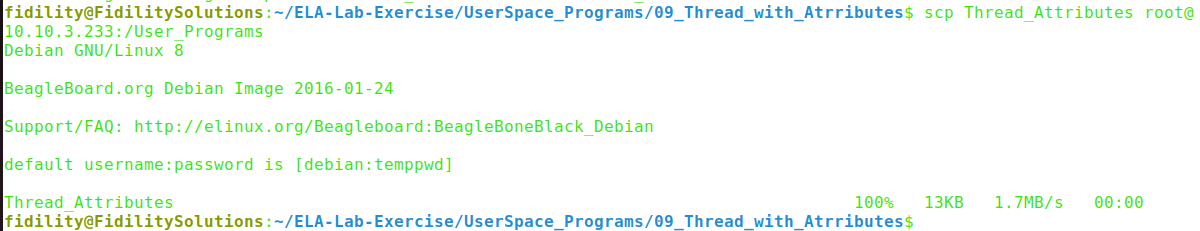
1. Create a directory in the target named User\_Programs



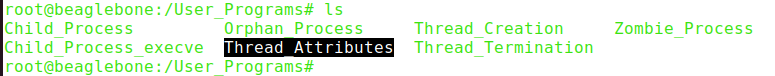
1. To copy the files from the host to the target directory give the SCP command in the host

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

Example: $scp Thread\_Attributes [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

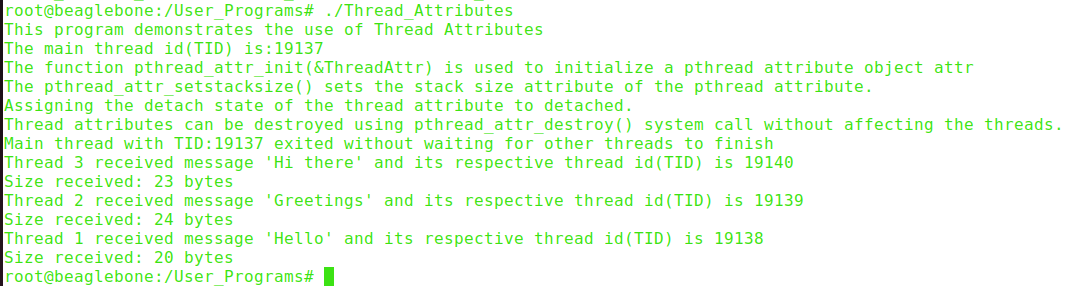


## **Step 4: Running on Platform**

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=arm64

$ 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

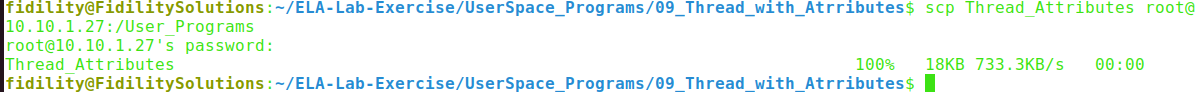
1. Create a directory in the target named User\_Programs



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 Thread\_Attributes [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

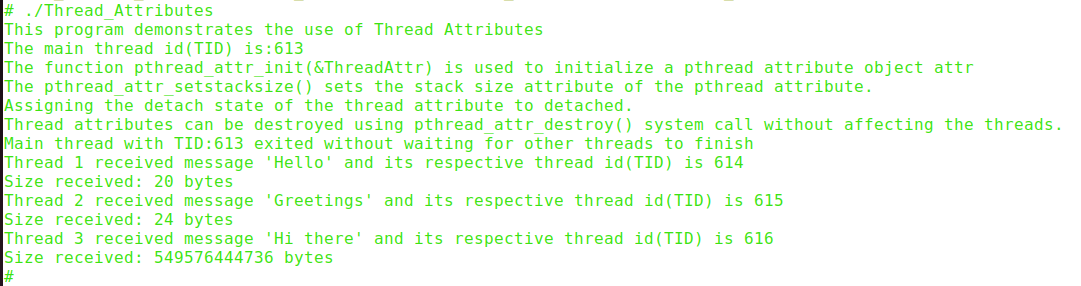


## **Step 4: Running on Platform**

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**

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

* 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.

## **Using /proc Interface**

1. First, let's identify the PID (Process ID) of a running process, for example, in our case the process id is 6014
2. # Now, let's use the `cat` command to read information about this process from `/proc`( cat /proc/$pid/status).

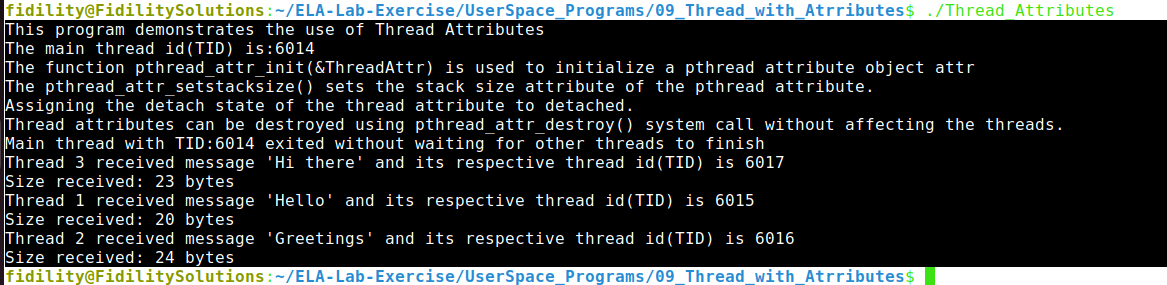
$ cat /proc/6014/status

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.

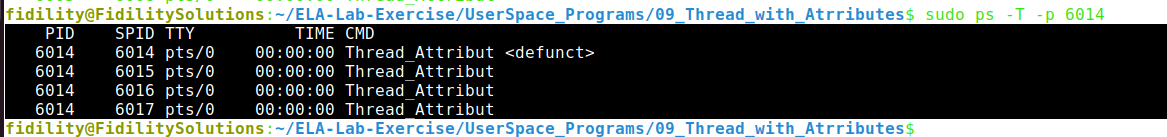


## **Using top Command**

1. The command `ps -T -p <TID>` displays thread information for a specific process ID. It shows details such as thread ID, CPU usage, and command associated with each thread.
2. The following image showing how thread attributes settled.



1. The image shows four Threads are running under process. Replacing `<TID>` with the actual process ID (6014) to retrieve relevant thread information. This command helps in monitoring and managing threads within a process efficiently.



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

# **Understanding Of Program Sequence**

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

Here are the step-by-step explanations of thread attribute program.

The agenda of the above program is to demonstrate the use of thread attributes in pthreads (POSIX threads) on Linux systems. Here's a breakdown of the program's agenda:

1. **Initialization of Thread Attributes:** The program starts by initializing pthread attributes using **pthread\_attr\_init().** This initializes a pthread attribute object which is used to specify various attributes for threads.
2. **Setting Thread Attributes**: It sets the stack size for the threads using **pthread\_attr\_setstacksize()** to 1MB. This customizes the stack size for the threads.
3. Setting Detached State: The program sets the detached state of the threads using **pthread\_attr\_setdetachstate().** Detached threads are those that operate independently and their resources are automatically released when they terminate without needing to be explicitly joined.
4. **Creating Thread Arguments:** It creates arguments for each thread, including a thread number, message, and message size.
5. **Creating Threads:** Using a loop, the program creates multiple threads (NUM\_THREADS) with the specified attributes and passes the thread arguments to each thread.
6. **Thread Function Execution:** Each thread executes the **threadfunction(**) function, which prints information about the thread attributes and performs specific actions based on the thread number.
7. **Exiting Main Thread:** The main thread exits without waiting for the detached threads to complete. This is because detached threads operate independently and do not need to be joined explicitly.
8. **Thread Attributes Destruction:** Finally, the program demonstrates how to destroy thread attributes using **pthread\_attr\_destroy()** without affecting the threads themselves.

In summary, the agenda of the program is to illustrate the creation and execution of detached threads with custom thread attributes, including stack size and detached state, and to show how the main thread can exit without waiting for the detached threads to finish their execution.