Linux for Embedded Systems

Lab 1 - Report

By Noman Noor (id: 302343)

Table of contents???

PROBLEM DESCRIPTION

Task A: Understanding the “rescue-mode”

First, we have to explore the functionalities provided by the pre-implemented “rescue-mode” build pre-installed on our lab Raspberry Pis.
Especially, the following:

1. The possible ways to transfer files between the RPi and the workstation (in both directions).
2. The possibility of using pipes to unpack, “on-the-fly”, the files being transferred via the network to the RPi (and its usefulness when installing a user system with rootfs stored in a dedicated SD card partition).

Task B: Configuring and Building the System

We have to build a GNU/Linux image for our Raspberry Pi 4 (henceforth referred to as just “RPi”) using Buildroot. We’ll make one with initramfs enabled as there are no restrictions on using it. The requirements set forth for our build is:

1. RPi connects and disconnects to the network on plugging in an Ethernet cable (assuming everything is configured correctly on the other end), using DHCP to obtain the DNS, IP Address(es), and other connection parameters.
2. The system hostname should be set in the format “firstname_lastname” as listed on student ID (noman_noor in my case).
3. As soon as an Internet link/connection is established, it should set the correct time using NTP.
4. The root user should have a password.
5. There should be a second user, which should also have a password.
6. The build should include a PHP Interpreter.
7. We should write a PHP script that will upload some informative system logs (I chose */var/log/messages*) to a remote server.

Procedure to recreate the design from the attached archive:

In the directory where the contents of the archive were extracted, run the script build.sh:

```
$ chmod +x ./build.sh
$ ./build.sh
```

Task A: Understanding the "rescue-mode"

The rescue-mode is a pre-built (using Buildroot) and pre-installed operating system (uses the Linux kernel) provided on our lab RPis and has various useful functionalities. One of the great things about it is that it's configured to automatically connect to the network when an ethernet cable (correctly configured on the other side) is plugged in. As for users/accounts, it only has one user: root. The root user doesn't have a password set by default.

The rescue system includes tools to partition and format the SD card (***mkfs, fdisk**, etc.*), and it also includes tools for repairing filesystems (***fsck**, etc.*) as needed. It even includes partition management tools such as ***resize2fs*** for resizing partitions instead of simply deleting and making them again with the new size.

There are more things we could talk about here, but let's move on to the two specifically requested in the task.

1. Transferring Files:

The provided rescue-system has a small, but focused selection of tools for file transfers. We can either use:

a. **WGET**: given that the source computer is running an http.server in the source directory on it and that the server is visible/accessible to us via a network, we can use WGET to download the files from the source computer's source directory.
For example:
On source PC (in source dir):

```
$ python3 -m http.server
```

On Raspberry Pi

```
$ wget sourceip:sourceport/file_in_sourcedir
```

i. **WGET with direct flashing:**
Connecting WGET via a pipe to DD, we can directly flash filesystem images while downloading them instead of waiting for the download to finish and then flashing.

```
$ wget -O - 192.168.145.xxx:8000/rootfs.ext4 | dd
of=/dev/mmcblk0p2 bs=4096
#Here, "-O -" in front of wget sets its output to
redirect(“-”), and then the | redirects that output to dd
which uses it in the flashing instead of having an “if”
parameter.
```

b. **SCP:** In order to use SCP, we must set the password for our user (root) first. We do that by using the ***passwd*** command.
For example:
On your Raspberry Pi

```
$ scp remote_source_file remote_username@remote_ip:remote_source_path
```

i. **SCP with direct flashing:**
It might be possible to achieve something just like *WGET with flashing* from the previous point, but I haven't tested that yet.

c. **SSH/SCP from the remote computer:** We could also use SSH to copy files from our computer to a Raspberry Pi.
For example: one complicated command with “on-the-fly” compression and “on-the-fly” decompression would be:

```
$ ssh RPi_ADDRESS "cd /DEST_PATH;tar zcf - files" | tar  zxf
```

2. Transferring Compressed Files with “on-the-fly” Decompression:

Just like the “direct flashing” points from the above section, it is possible to use pipes for useful tricks when it comes to decompression. In order to save network bandwidth, we could send a compressed file and then have a decompression command connected via a pipe to the file transfer command to be able to get our file back. In fact, we could then connect it via a pipe to DD to flash a filesystem image to a partition (like in WGET with direct flashing).

An example using WGET:
On the source PC, we must have a compressed archive in the directory where our HTTP server is running.
Then, we run the following command on our RPi:

```
$ wget -O - "SOURCE_ADDRESS" | tar xz
```

Task B: Configuring and Building the System

1.) We initialize buildroot config to the default config for Raspberry Pi 4 by finding the required default config using the commands

```
$ make list-defconfigs
#lists alphabetically, but we can redirection to grep for finding
```

I found the following two to be relevant:

raspberrypi4_64_defconfig	- Build for raspberrypi4_64
raspberrypi4_defconfig	- Build for raspberrypi4

I used raspberrypi4_64_defconfig because it’s 64bit, and here is the command to for that:

```
$ make raspberrypi4_64_defconfig
```

2.) Now, we modify the configuration to what is recommended in the Lab Guide and asked for by the task.
We enter the configuration window by:

```
$ make menuconfig
```

Then, we set the following as per lab guide:

Build Options → Compiler cache location = ../ccache-br

Build options → Mirrors and Download locations → Primary download site =
“http://192.168.137.24/dl”

Toolchain → Toolchain type → External toolchain <- It was left unchanged as the default was correct.

Filesystem images → ext2/3/4 root filestystem → Compression method → gzip

Filesystem images → cpio the root filesystem → Compression method → gzip

As for the task’s requirements:

1. Automatic network connection and disconnection on plugging or unplugging ethernet cable respectively:
Target packages → Networking applications → connman
Target packages → Networking applications → dhcpcd
Target packages → Show packages that are also provided by busybox
Target packages → Networking applications → ifplugd

2. Setting Hostname:

System configuration → System hostname = “noman_noor”

3. NTP Time Synchronization:

Target packages → Networking applications → ntp → ntpd

4. **For running the required PHP script:**

We include:
Target packages → Interpreter languages and scripting → PHP
And
Target packages → Interpreter languages and scripting
→ PHP
Extensions -> ftp;
(we’ll need it for my particular script).
Which is later copied as explained the transferring files section.
One addition however, is that we need to run the following on the script in our RPi:

```
$ chmod +x phpscript_path
```

5. Setting default password for root:
System configuration → Root password = “root”

6. Adding a new user:
Following the following two sources:
https://buildroot.org/downloads/manual/manual.html#customize-users
https://buildroot.org/downloads/manual/manual.html#makeuser-syntax
I made a userstable.txt file in the ".." directory.
Then:
System configuration → Path to the users tables = "../userstable.txt".

The PHP script:

```
#!/usr/bin/php
<?php

$ftp_server = "speedtest.tele2.net";
$ftp_conn = ftp_connect($ftp_server) or die("Could not connect to
$ftp_server");
$login = ftp_login($ftp_conn, 'anonymous', '');
if (ftp_put($ftp_conn, "messages-log.txt", "/tmp/messages", FTP_ASCII))
 {
 echo "Successfully uploaded /tmp/messages";
 }
else
 {
 echo "Error uploading /tmp/messages";
 }
 ftp_close($ftp_conn);
#btw, make sure to have execution permission. Also, if ownership causes a
problem, just use  chmod +x!

?>
```