Linux for Embedded Systems

Lab 3 - Report

By Noman Noor (id: 302343)

Current Date: 20/05/2022 (dd/mm/yy)

PROBLEM DESCRIPTION

Task: Make a Rescue OS and a Utility OS (which runs a file server built using a framework like flask, tornado, etc.) and selection of which OS to use using buttons on the lab RPi4 extension board. Also, indication of the current OS being used via LEDs.
The RPi's boot partition needs to be properly populated with the chainloading U-Boot and it's configuration enabling for the LED and Button behavior described earlier.
The SD Card of the lab RPi4 also needs to be properly partitioned to have one partition which is flashed with the rootfs image for the Utility OS, meanwhile the other is simply formatted to be used as secondary storage (used by server to store files).

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
```

Step 1: Setting up Rescue (Adminstrative) Buildroot Image:

Basic Configuration:
Basic configuration remains the same initramfs based system we have been building so far in labs 1 and 2. However, we include some additional packages for playing around with the filesystem and so that it can act as a proper recovery OS.

Additional Packages: (aside from trivial ones contained in every other package and not directly relevant to the current task)

1. **BusyBox** (contains various tools like mkfs (ext2, ext4, etc.), fsck, tar, gzip, dd, etc.) *[Found in "Target Packages"] (I also clicked Show packages also provided by busybox in that menu).*
2. **Util-linux and its "sub-modules"** for various libraries and programs**.** Some useful modules were pre-selected for it but I selected the following manually: **libfdisk**, **fsck**, **more**, and **mount/unmount**. *[Found In "Target Packages/System Tools"]*
3. **dosfstools** for fat partition related tools (like mkfs.fat and fsck.fat) for management of the boot partition which is in vfat (vfat is a type of fat) *[Found In "Target Packages/Filesystem and Flash utilities"]*
4. **fsck.fat**, **mkfs.fat**, and **mtools**. [*Found In "Target Packages/Filesystem and Flash utilities"]*

5. **e2fsprogs** (includes mkfs.ext2,etc. by default) and its **resize2fs** submodule. [*Found In "Target Packages/Filesystem and Flash utilities"]*
6. **gptfdisk**, including its "interactive gdisk" sub-option (auto-selected) for manipulating devices, creating/deleting partitions, etc. [*Found In "Target Packages/Hardware handling"]*
7. *Note: You may include the full "bash" shell if needed for anything because Buildroot comes with a limited shell by itself. [Found In "Target Packages/Shell and utilities"]*

IMPORTANT NOTE:
We also edit this configuration to build the bootloader, but that'll be explained separately in the section about said bootloader.

Step 2: Setting up Utility (Server) Buildroot Image:

Basic Configuration:
In this case, the basic configuration deviates only in the packages installed and in being a initramfs based system like the server above in the fact that we don't use initramfs and hence also build the root filesystem (also known as its abbreviation "rootfs"). The packages included are also different as the use-cases for both systems are totally different.

Filesystem Configuration:
1. Ensure that "initramfs" is disabled. *[In "Filesystem images" menu]*
2. Enable **ext2/3/4 root filesystem.** *[In "Filesystem images" menu]*
    a. Under that select "**ext4**" as the "**ext2/3/4 variant"**.
    b. Set "**rootfs**" as the "**filesystem label**".
    c. Set "**300M**" as the "**exact size**". (You can also set any other size you like (e.g. 500m), but I estimated this number to be enough for my use.)
    d. You can also add a compression method if you require/want it, but then the flashing process would need to have a pipe between the source to gizip and a pipe from that to dd in order to decompress before flashing. I didn't need any of this, so I didn't add it.
3. We also set up a rootfs overlay in order to have the /server and the /etc/init.d directories containing the content where they need to be and have what they need to have. Using this we transfer the whole server application which will then be in the directory /server and the script to start the application on startup (as described in the section related to the server application). We do this by:
    a. Setting "**../Overlay**" for "**Root filesystem overlay directories**" and ensuring that the Overlay directory (containing the required /server and /etc/init.d directories) is one directory above/right outside the buildroot folder of the utility image.

Additional Packages: (aside from trivial ones contained in every other package and not directly relevant to the current task)

1. BusyBox and other trivial functions (contained in every image we've built so far).
2. "**python3**" for running our server. *[Found In "Target Packages/Interpreter languages and scripting"]*
    a. Include the following "**External Python Modules**":
        i. **python-flask and every package that starts with "python-flask"**
        ii. I also included "**python-tornado**" just in case I wanted or needed to switch. But it wasn't needed so it is unused and **may be not included**.
        iii. "**python-werkzeug**" is auto-selected, so we just check if it is indeed selected because it is a direct dependency of the current version of the server I have built.
3. *Note: You may include the full "bash" shell if needed for anything because Buildroot comes with a limited shell by itself. [Found In "Target Packages/Shell and utilities"]*

Step 3: Building and Configuring the Bootloader (U-BOOT):

Since in user mode of the default booting process of the lab RPi4 we have to load somehow one of the two new images, we make another u-boot and chainload it to be able to do what we have to do. Then, we boot into our u-boot by setting it as the user image (and not pressing any button on the lab RPi4 to load into the "user image" which happens to be our u-boot) which we can then use to select our image.
The way I have set it is, if you DON'T PRESS the second-from-left button, our chainloaded u-boot will boot the Rescue/Administrative OS and turn the third-from-left LED. If you DO PRESS the second-from-left button, our chainloaded u-boot will boot the Utility/Server OS. (*Note: All physical directions are from when the device is oriented in such a way that the buttons are directly in front of you and the LEDs are right above that*).

Basic BUILDROOT Configuration:

In the Rescue OS related Buildroot configuration, we go to the "**Bootloaders**" section and there we enable "**u-boot".** Then we ensure the build system is Kconfig, U-Boot Version is a version compliant with our configuration file (will talk more about it in a later subsection) [*Note: 2022.01 was the version used to make this specific image*]. We don't use any specific Custom U-Boot patches.
For "**Board defconfig**" we enter "**rpi_4**". None of the other u-boot options were changed.

In "Host utilities" section, we ensure, that "host u-boot tools" or any other tools (e.g. maybe mkimg for host) are enabled (if required).

U-BOOT CONFIGURATION:

We achieve the intended behavior (explained above; In the introductory paragraph of the u-boot section) by writing a U-Boot boot script (refer to the boot.txt for the actual content of the script). The image is then converted to an image “**boot.scr**” using the “mkimg” tool on your host machine. mkimg isn’t installed on the lab PC so I had to use the one Buildroot had built. In fact I have copied and included it in the image to ensure the build script will work.

The command used was:
(while working directory is the directory containing boot.txt and mkimg)

```
./mkimage -T script -C none -n 'Start script' -d boot.txt boot.scr
```

If you have mkimage installed on the PC, you can simply use:
(*while working directory is the directory containing boot.txt)*

```
mkimage -T script -C none -n 'Start script' -d boot.txt boot.scr
```

This is how we obtain our **boot.scr** file which we will be using later.

Step 4: THE SERVER APPLICATION (made with Python and Flask):

Note: The port used for the server is 8989. (may be changed in source code easily in parameters of app.run(......) function).

Building and Modifying/Coniguring the Server:

I made a flask server which serves a “File server” front-end to enable upload and download of files as asked for in the task description. The server requires authentication to access the file server, and to do that the default user password combination is:
Username: admin
Password: admin

This may be edited, removed, or more may be added near the line 10 (userList["admin"]="admin") of Utility/Overlay/Server/server.py.

The default file server is started in the Utility/Overlay/Server/files directory but may be easily modified to use any directory by replacing all the references to “./files” in Utility/Overlay/Server/server.py to the absolute or relative path of the required directory. I was asked to change this in the lab, but since I couldn't test on an actual lab RPi4, I decided against it. (I am not sure of the path of the third partition of /dev/mmcblk0 created later on, where it should be actually hosted.)

The file server also only serves files with extensions that have been allowed. Currently, those are: 'zip','txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif', and 'gz. However, this list may be edited near the line 5 of Utility/Overlay/Server/server.py (ALLOWED_EXTENSIONS = {'zip','txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif','gz'})

Running at startup (using init.d script) and transferring to filesystem:

To run the program at start up, we create the init.d script Utility/Overlay/etc/init.d/S99fileserver (using !#bin/sh which happens to be a leaner version of bash for us). The contents of the script are only:

```
#!/bin/sh

ifconfig
python3 /server/server.py &
```

Ifconfig is just called for usefulness in letting you know the script has run and also the ip you can use to access the RPi4 when the server has started. The “&” next to the python3 command to run the server is used to run the process in background. We don’t need to use “nohup” because the server is run in the main shell of the OS when booting and as such won’t be closed until the OS itself is closed or if the server hits a serious bug (Currently, there are no reasons to believe there might be such a situation/bug).

The Overlay for the transfer of the fileserver and init.d script are already set in the configuration of the Utility OS, so we don’t need to do anything more for that.

Step 5: SETTING UP THE BOOT PROCESS FOR LAB RPi4

I won't discuss the actual method of transferring used beyond mentioning that I used wget on the RPi4 while using the Python http.server module on the host PC because this is trivial and mostly unchanged from Lab 1 and 2.

The steps to setting up the chainloading booting are:

Boot into recovery OS already provided on the lab RPi4 by pressing the first button on the left (when the orientation is such that the buttons are below/ahead of the LEDs).

Mount /dev/mmcblk0p1 to /mnt and change working directory to /mnt:

```
mount /dev/mmcblk0p1 /mnt
cd /mnt
```

Then, transfer the U-Boot image from /Rescue/{buildroot-dir-name}/Output/Images to /mnt/user with the filename "Image". Also transfer the device tree (I think extension is .dtb) and commandline.txt of the Rescue build as needed (from their respective locations in the Rescue OS' Output/Images directory).

Then, transfer the respective Image file from each Rescue and Utility builds naming them Image.rescue and Image.utility respectively.

Then, in /mnt, download the "**boot.scr**" we made earlier in the U-Boot section.

Step 6: SETTING UP THE FILESYSTEM

(Note: Please be careful of the blocksize of the SD card, so as to avoid wearing it out more than necessary.)
(Note 2: For more info on using fdisk try the manpage or searching the internet. One resource I found with a quick search was https://support.huaweicloud.com/intl/en-us/eu-west-0-usermanual-evs/evs_01_0033.html**)**

With this, we are mostly done and just need to deal with the partitioning and formatting of /dev/mmcblk0 for rootfs and the extra partition for server.

For this part, we can use either the Recovery OS provided on the lab RPi4 by default or the Rescue system. I used the Rescue OS I built but the default Recovery OS provided on the lab RPi4 might have more tools, so one might want to use that.

<u>Partitioning /dev/mmcblk0 to get /dev/mmcblk0p2 and /dev/mmcblk0p3:</u>
We use **fdisk** on the device **/dev/mmcblk0** to delete the previous partition /dev/mmcblk0p2.

/dev/mmcblk0p2:
Then create a new partition (partition 2) of the same size as the rootfs filesystem image (and starting from the first unused block on the sdcard as we don't have any use for spaces between partitions nor will there be any uses for it in the future). This partition will be the partition we will flash our rootfs image to so we leave it as is for now.

/dev/mmcblk0p3:
Now, we create another partition (partition 3) of any size we want (I typed nothing to achieve the 'default', which is the same as all of the unpartitioned storage). This partition will start right after the end of Partition 2 we just made and end right at the end of the SD card (because size was specified to be all unpartitioned space).

For both:
We also make sure the correct flags are set on both new partitions for the filesystem (which is Linux and I think it is set by default by fdisk).

<u>Formatting /dev/mmcblk0p3:</u>
Since I chose the ext2 filesystem, we use the following command: (blocksizes may be need to be chosen specifically to prioritize the health of the SD card)

```
mkfs.ext2 /dev/mmcblk0p3
```

Or

```
mkfs.ext4 -t ext2 /dev/mmcblk0p3
```

Or

```
mkfs -t ext2 /dev/mmcblk0p3
```

You may also mount /dev/mmcblk0p3 later to create a directory for the file server or just have the file server point to the root of /dev/mmcblk0p3.

<u>Flashing /dev/mmcblk0p3 with the rootfs filesystem image:</u>
While the Python http.server is running on the Host machine in the Utility/{buildroot-dir}/Output/Images directory:

After making sure that the rootfs image (uncompressed) and the /dev/mmcblk0p2 partition have the same exact size (if they don't a tool like "truncate" might help to truncate the image to be the same size as the partition). They should always have the same size as long as the instructions in here were followed properly because the instructions in this file take into account the fact that both size needs to be same (i.e. in buildroot config and in fdisk we use the same size for the image and partition respectively).

On the lab RPi4, we enter the following command :

```
wget http://{host_ip_address}:{http_server_port}/rootfs.ext4 -O - \
| dd of=/dev/mmcblk0p2 bs=4096 \
/
```

That's a single multi-line command to save space. To write it in one line, we would just type:

```
wget -O -{host_ip_address}:{http_server_port}/rootfs.ext4 | dd
of=/dev/mmcblk0p2 bs=4096
```

If you used compression, for example for gzip you would need to add gzip in between wget and dd (and change rootfs.ext4 to the compressed file's filename) which would have to be connected to dd and wget via pipes ("|") and other such modifications.

You might also be able to do something like directly using gzip, gunzip, etc. to directly flash but I am not sure. So, I recommend against it unless you have done proper research which indicated that it should work.

Also, here's an interesting link about flashing in general:
https://ubuntuforums.org/showthread.php?t=1540873

: FINISHED :

Now you may boot the required OS as described in the introductory paragraph of the U-Boot/Bootloader section.