Lab 4
Kernel Patching and Cross-Compilation for RPi
Embedded Linux

Julia Desmazes
Michael Nissen

October 27, 2017

1 Download / Set the good version of Linux sources

To execute the tasks for this report, the first step is to get the Raspberry Pi cross-compilation tools, and specific version of the linux kernal that one wants to patch. This is going to be done using `git clone`, which ofcourse assumes a working installation of `git`.
To get the latest version of the kernel source code and the cross-compilation tools, the following commands with be copied to the terminal:

```
host$ git clone https://github.com/raspberrypi/tools.git
host$ git cone -b rpi-4.4.y https://github.com/raspberrypi/linux.git
```

One now has to make sure to get the same configuration as is used by ones Raspberry Pi during the kernel compilation, which can be done by checking out the `git branch` that corresponds to the hash of the Linux kernel running on the Raspberry Pi.
This will be done by by running the shells script seen in Code Snippet 1. This shell script will grep the firmware hash found on the Raspberry pi, and then get the `git hash` from the github link relative to the hardware hash.

```
#!/bin/bash

PLATFORM=`uname -s`

if [ ${PLATFORM} = "Darwin" ]
then
    CMD="gunzip -c"
elif [ ${PLATFORM} = "Linux" ]
then
    CMD="zcat"
else
    echo "Sorry, the platform ${PLATFORM} is not supported !!!"
    exit -1
fi

FWHASH=`${CMD} /usr/share/doc/raspberrypi-bootloader/changelog.Debian.gz | grep -m 1 'as
    of' | awk '{print $NF}'`
#echo "Firmware Hashcode: fwhash = $FWHASH"

LINUXHASH=`wget -qO- https://raw.github.com/raspberrypi/firmware/$FWHASH/extra/git_hash`

echo "Linux Hashcode: linuxhash = $LINUXHASH"
```

Code Snippet 1: Shell script to get Linux kernel git hash

When one has figured out the `git hash`, one can change directory to the linux git repository and checkout the respective brach by the following command:

```
host$ cd linux
host$ git checkout <git_hash>
host$ make clean -mrproper
```

At line 3, we are cleaning kernel tree from all unneeded files of the first version which is a recommended practice prior to each kernel compilation!

2 Patch the Kernel

Before starting to patch the kernel, one should make sure to identify the Linux sources that one is trying to install. This can be done using the `head` command, which will look like this:

```
host$ head Makefile

 # Ouput generated by running head on the Makefile #
 VERSION = 4
 PATCHLEVEL = 4
 SUBLEVEL = 21
 EXTRAVERSION =
 NAME = Blurry Fish Butt
```

Now that one know the specific version of the Linux kernel, it is time to download the latest `PREMEPT-RT` patch that corresponds to ones kernel version - if follows the format: `version.patch-level.sub-level`. To download the patch from the patch from `kernel.org` using the `wget` command like so:

```
wget https://cd.kernel.org/pub/linux/kernel/projects/rt/4.9/patch-4.9.47-rt47.patch.gz
```

Now one can actually start patching the kernel sources by executing the following commands in succession:

```
host$ gunzip patch-<version.patch-level.sub-level>-rt<last>.patch.gz
host$ cat patch-<version.patch-level.sub-level>-rt<last>.patch | patch -p1
```

One can now create a folder named `rt-modules` under your project folder - at the same level as the `linux` and `tools` folders, and export an environment variable `INSTALL_MOD_PATH` that points to that folder:

```
host$ mkdir rt-modules
host$ cd rt-modules
host$ export INTALL_MOD_PATH=$PWD
```

Here `$PWD` is the path to the current directory.

3 Configure Cross-compilation

Before starting the cross-compilation, one is going to specify the architecture for which to compile, the cross-compiler that ones wishes to use, and the kernel that one is trying to compile for. This is done by exporting some specific environment variables that is used by the makefile:

```
host$ export ARCH=arm KERNEL=kernel7
host$ export CROSS_COMPILE=$PWD/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian
    -x64/arm-linux-gnueaihf-
```

Now, to configure the kernel in an appropriate way, one can use the existing kernel configuration from the Raspberry Pi. This configuration file can be generatied using the `modprobe` command, and then copy that generated file from the Raspberry Py over to the `linux/` folder on the host machine using the `scp` command. Finaly, extract the content of the `config.gz` into a `.config` file:

```
host$ sudo modprobe configs
host$ scp pi@193.169.0.26:/proc/config.gz /home/user/kernel_labs/linux
host$ zcat -c config.gz > .config
```

Now that one has created a default `.config` file, it is time to costumise the kernel using the `sudo make menuconfig` which opens the GUI seen in Figure **??**. For the purpose of this lab assignment, we are going to enable the `Fully Preemptible Kernel` setting, and the `High Resolution Timer Support`.

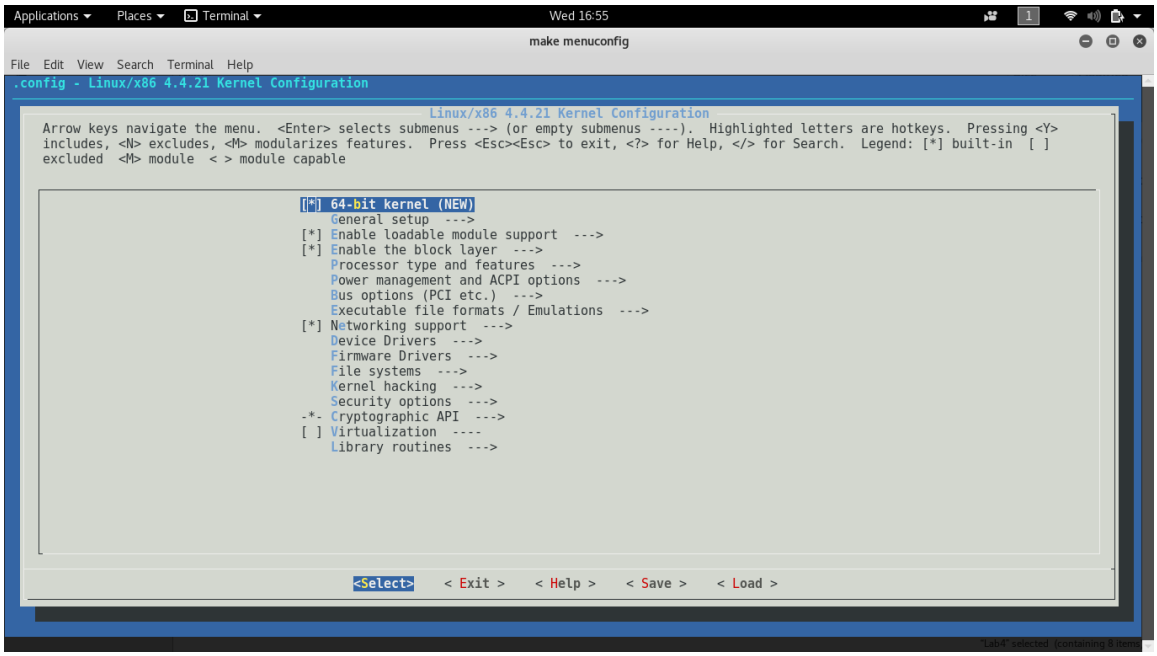

Figure 1: caption

4 Build the new Kernel and Modules

To build the kernel,and modules we have to compile our configured sources before they can be run by our pi. This is where all of our exported variables comme into play, they will be used in the compilation processe to assurw we are doing a cross compilation.
Once we know the number of availible CPU's by using the `nproc`, we need to build the kernel with it's file system and the modules.

```
host$ make -j < 2* number of CPU > zImage modules dtbs
```

Once the modules are compiled we can install them to the path scpecified by `$INSTALL_MOD_PATH`.

```
host$ make -j < 2* number of CPU > modules_intall
```

Our next step is to create is to replace bootable kernel image based with the ones in our compiled sources. We created this image with zImage but it still needs some post-processing to support the dtbs type file system. To do theys changes we use `mkknling` a small utilitay found in tools that essentially justs adds `boot-uncompressed.txt` and `args-uncompressed.txt` to our image and creats a new image named kernel7.img in our recently created directory
`boot`.

```
host$ mkdir $INSTALL_MOD_PATH/boot/
host$ ./scripts/mkknlimg ./arch/arm/boot/zImage $INSTALL_MOD_PATH/boot/$KERNEL.img
```

We also need to add our `.dtb` files, theys files are essentially a database that represents the hardware components on out pi, so they to will be necessary at boot. We also copy in the overlays folder.

```
host$ cp ./arch/arm/boot/dts/*.dtb $INSTALL_MOD_PATH/boot/
host$ cp -r ./arch/arm/boot/dts/overlays $INSTALL_MOD_PATH/boot
```

Now we should have a `boot` and an `lib` folder under our file location `$INSTALL_MOD_PATH`. We can now transfert it to our pi for merging and to make the transfert faster we will first create a archive of our directory.

```
host$ tar -cvzf kernel.tgz $INSTALL_MOD_PATH/*
host$ scp kernel.tgz pi@<IP adresse>:/tmp
host$ ssh pi@<IP adresse>
pi$ tar -zcvf /tmp/kernel.rgz /tmp/
```

Now we can start merging with the existing kernel on the pi, this processe is very simple, it envoloves coping recursively our files from boot and lib into the pi's own boot and lib folders.

```
pi$rm -rd /tmp/boot/overlays
host$ cp -rd /tmp/boot/* /boot/
host$ cp -rd /tmp/lib/* /lib/
```

At the end of the process we deactivate an option in our kernel for fast sd card interaction as it may cause problemes further on in our realtime kernel. Afterwars we can reboot and should have a realtime preemptive kernel.

```
pi$sudo nano /boot/cmdline.txt
##########################
add sdhci_bcm2708.enable_llm=0 at the end of the line
##########################
pi$sudo reboot now
```

5 Preemptible kernel check

To check if the new kernel is preemptive we check for two flags, the first is if the kernel type is said to be preemptive, this is the output we would get with `uname -a`. The second flag is located in file `/sys/kernel/realtime` if the file existis and if it's conents is 1 then we know the kernel is preemptive.

```
host$scp iffprempt.c pi@<IP address>:~/
host$ssh pi@<IP address>
pi$gcc -o iffpreempt ifpreempt.c
pi$chmod +x iffreempt
pi$./iffreempt
this is a PREEMPT RT kernel
```

6 Select the scheduling policy

The `chrt` command allows us to run a process with a specific scheduling policy and priority (when availible), it also allows us to see what availible options there are. If we wanted to lauch a preemptive processe we would use option `-r` and `-p` to define it's priority.

If we wanted to lauch a SCHED_FIFO process we would use the same commande but with oprion `-f` this time to specify FIFO.

```
pi$chrt -m
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority : 1/99
SCHED_RR min/max priority : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority : 0/0
#SCHED_PR
pi$chrt -r -p <priority from 1/99> <process name> <args>
#SCHED_FIFO
pi$chrt -f -p <priority from 1/99> <process name> <args>
```

7 Boot time optimization

One thing we can notice is how slow the pi's boot is, according to `systemd-analyze time` the boot sequence takes un an entire 33.927 seconds, 31.531 of thoses for userspace. In otherwords there are more than 31 seconds of the boot sequence on witch we can have an effect. What is intresting is that when we go more into detail we realise that the dhcpcd services use up an entire 16.374 seconds. To reduce this time it was decided that sinc dhcp services are responable (amoungs many other talsks) for managing dynamic ip attributions, a way to speed it up would be to set un a static ip.
To do this we included our static network configurations into `/etc/dhcpcd.config`.

```
#the interface we are currently using
interface eth0
#our current ip address
static ip_address=192.168.1.79/24
#my home rooters ip address
static routers=192.168.1.1
```

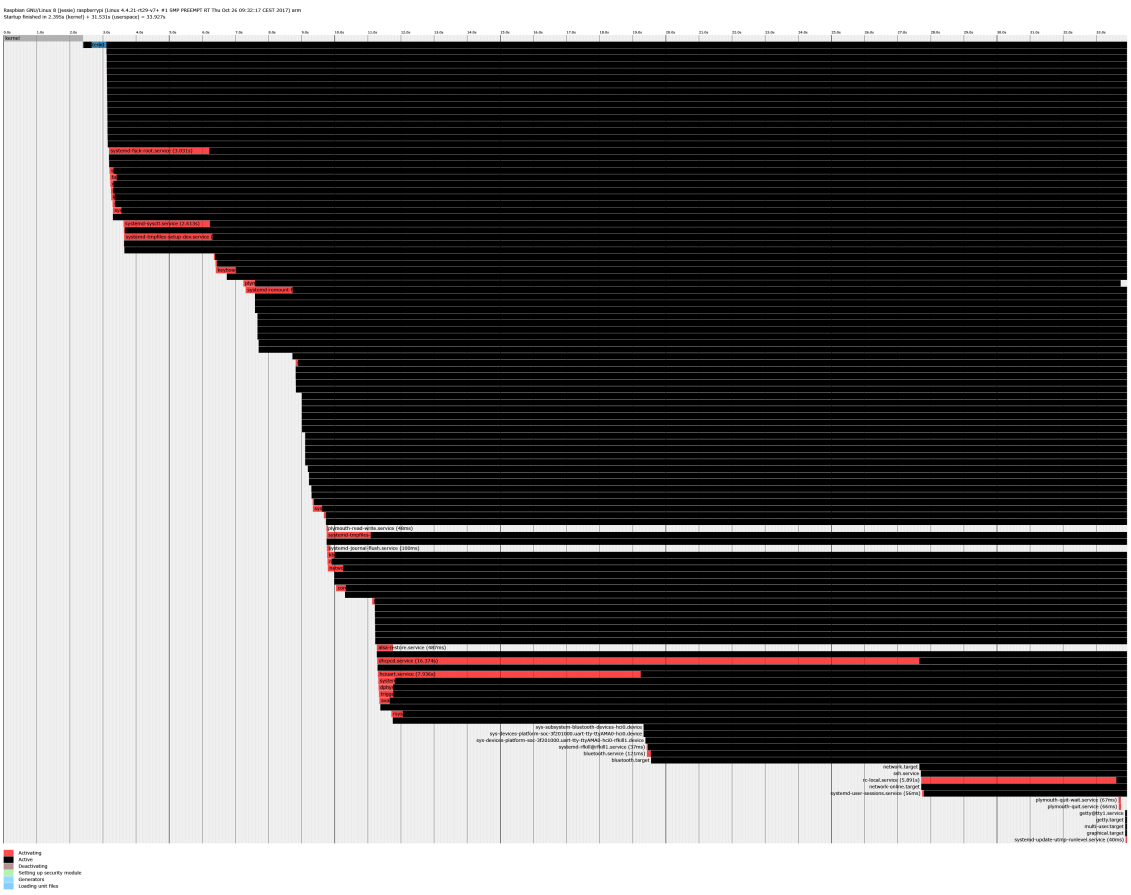

Figure 2: Boot sequence with dynamic ip

```
#google's dns server adress
static domain_name_servers=8.8.8.8.8.8.4.4
```

We didn't configure wlan0 because, in case this failed we would have to manually alter the config files from the sd card.
After reboot, we re-tested our boot time and it had significanly got shorter down to 20.993 seconds and 18.684 for userspace. Our dhcp service now only used up 6.907 seconds of boot time. If we wanted to ge further we could start dissabling services like bluetooth or hciuart that take up boot time and that we currently have no use for.

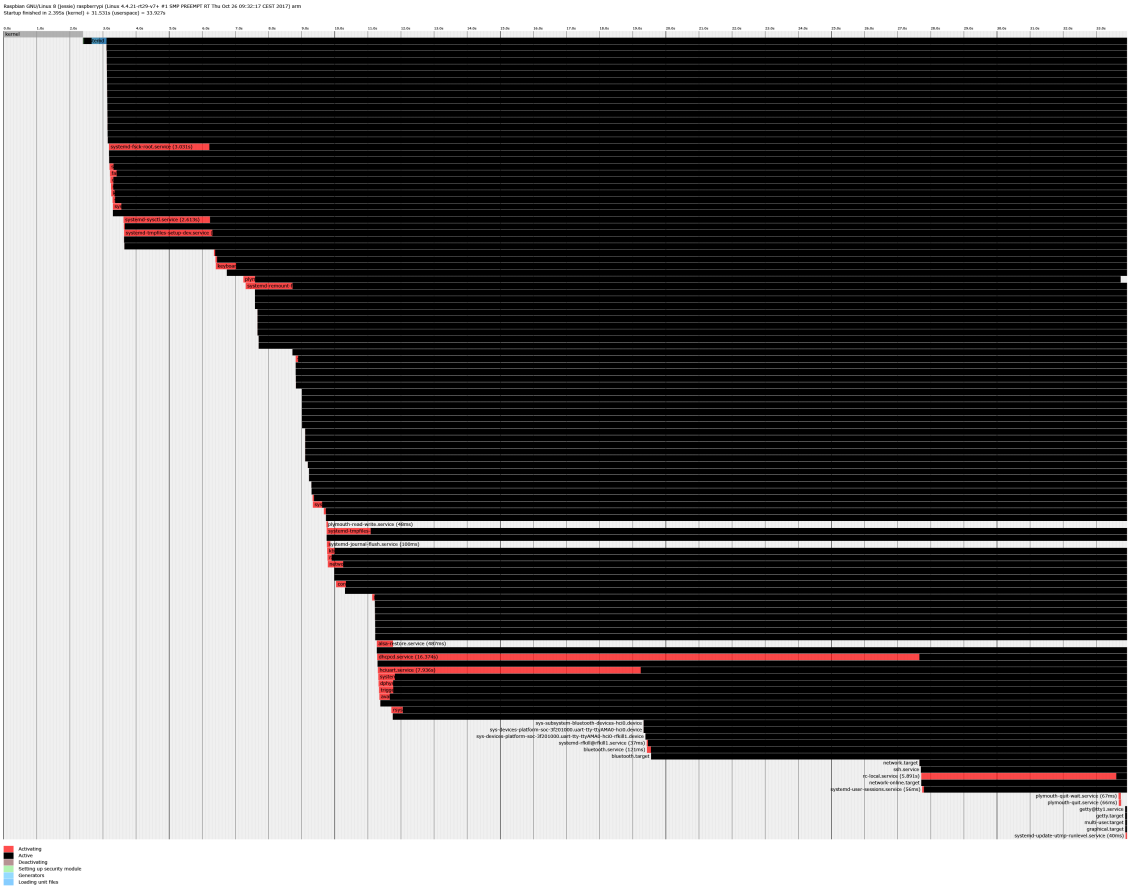

Figure 3: Boot sequence with dynamic ip

8 Kernel modules

8.1 Create and compiling modules

Code for hello.c:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_AUTHOR("Julia Desmazes");
MODULE_LICENSE("GPLv3");
MODULE_DESCRIPTION("A short and winey hello world");

static int __init hello_hummans(void)
{
printk(KERN_INFO"Loading hello module...\n");
printk(KERN_INFO"Hello world I am a module and I am loaded, chears and cookies \n");
return 0;
}

```

```
static void __exit hello_end(void)
{
printk(KERN_INFO "Goodbye cruel world, oh why don't you love me?\n");
printk(KERN_INFO"Module has been murdered...\n");
}

module_init(hello_hummans);
module_exit(hello_end);
```

Accompaning makefile:

```
obj-m += hello.o
all:
  make ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) -C $(DIR_TO_KERNEL) M=$(PWD) modules

clean:
  make -C $(DIR_TO_KERNEL) M=$(PWD) clean
```

We hade to perform a few changes to our environmental bash variables , first we changed the value of CROSS_COMPILE to account for the fact that we here compiling form folder `extra`. Then we created `DIR TO KERNEL` that is the absolute path to our kernel compilation.

```
#done from extre
export CROSS_COMPILE=$PWD/../tools/arm-bcm2708/arm-bcm2708-linux-gnueabi/bin/arm-bcm2708-
    linux-gnueabi-
export DIR_TO_KERNEL=$PWD/../linux/
```

8.2 Loading/Unloading modules

After having copied the modules to the pi with the `scp` protocol we then use `sudo insmod hello.ko` to load the module. We then checked to see if the module had been loaded by printing out a list of all the loaded modules with `lsmod | less`. Then to unload it we would use `sudo rmmod hello`.

```
Module                  Size  Used by
hello                    944  0
bnep                   10668  2
hci_uart               19492  1
btbcm                   6245  1 hci_uart
bluetooth             338111  22 bnep,btbcm,hci_uart
```

So we know our module was loaded but we were not getting any messages, we supposed this came from the fact that we where in as root under run level 3. To be shure the messages there we checked with `dmesg`.

```
[ 7236.713120] Loading hello module...
[ 7236.713133] Hello world I am a module and I am loaded, chears and cookies
[ 7345.372593] Goodbye cruel world, oh why don't you love me?
[ 7345.372616] Module has been murdered...
```

And, we confirmed our module was unloaded:

```
odule                   Size  Used by
bnep                   10668  2
hci_uart               19492  1
btbcm                   6245  1 hci_uart
bluetooth             338111  22 bnep,btbcm,hci_uart
```