Teaching "Embedded Operating Systems" using Raspberry Pi and Virtual Machines

Diego R. Llanos
Dpto. de Informática
Universidad de Valladolid, Spain
diego@infor.uva.es

Abstract

Embedded systems design, implementation and programming is an important topic in many curricula of Computer Science and Computer Engineering. This paper describes the structure of a course on Embedded Operating Systems included in the Degree in Computer Science at the University of Valladolid. The course core elements are the use of virtual machines and Raspberry Pi kits. Our experience shows that the topics covered and the project-based methodology lead to excellent results in terms of student progression.

1 Introduction

Embedded systems design, implementation and programming is an important topic in many curricula of Computer Science and Computer Engineering. In the Degree in Computer Science offered by the University of Valladolid, study of embedded systems are divided in two optional courses. Both courses are credited with 6 ECTS credits (corresponding to sixty hours of lessons). The courses are Embedded Hardware and Embedded Operating Systems.

The former one, Embedded Hardware, covers the lower-level aspects of embedded devices, such as memory technologies, typical devices, buses architecture, transceivers, and electric specifications. This course is oriented to low-level design and programming, and it will not be covered in this paper.

In this paper we will focus in the last one, Embedded Operating Systems. This subject assumes the existence of an embedded hardware capable of running a tailored version of GNU/Linux. To do so, such a system should have at least a microprocessor with MMU management, 4 Mb of RAM and about 8 Mb of secondary storage, typically solid-state disks such as memory cards. Note that this was the standard configuration of a commodity PC in early 90s. However, miniaturization and cost reduction, together with the advent of low-powered Systems on Chip (SoC) have allowed the production of computers with these or better characteristics for less than forty euros. This is precisely the case of the Raspberry Pi microcomputer.

The Raspberry Pi [1] is a credit-card sized computer board composed of an ARM-like SoC and a board that includes all the I/O and storage interfaces

needed for practical use, including HDMI output, one or two USB ports, Ethernet port, slot for an SD memory card, and even a video output to connect the system to an analog TV. The Raspberry Pi was designed to teach Computer Science to teenagers, allowing an easy I/O access (including exposed GPIO pins) to facilitate the use of the system not only for programming, but also as the core of hardware projects of any size.

The existence of a computer such as the Raspberry Pi, capable of running GNU/Linux at a cost of some tens of euros, allows us to think that in the following years any embedded system will be able to do so, even systems with a cost of one or two euros. This fact would make the design of tailored embedded systems a comparatively expensive process, only useful when very special characteristics were needed. Besides this, the use of an operating system in embedded computing greatly simplifies the portability of embedded applications. These reasons justify the study of the development of tailored operating system for such devices.

We have found that the use of the Raspberry Pi is a good starting point to teach the difficulties of building an operating system for such devices, when memory and/or disk space constraints appear. Besides this, the ARM ISA of the Broadcomm processor integrated in the Raspberry Pi forces developers to use of cross-compiling tools to generate object code. Fortunately, as long as the Raspberry Pi is capable of running GNU/Linux, there exist many open-source tools available.

In the following sections we will first present the compromise solutions we have arrived to set up the needed hardware, and then we will describe the contents of the course.

2 Hardware and software requirements

Our course on Embedded Systems aims to teach the following concepts:

1. How to install and set-up a not-so-friendly GNU/Linux distribution in a personal computer (that we will call the *host* system).
2. How to compile the Linux kernel natively for the host system.
3. How to cross-compile the Linux kernel for a *target* system with a different ISA, in our case the Raspberry Pi.
4. How to compile the GNU operating system natively for the host system.
5. How to cross-compile the GNU operating system for the target system.

The development of specific applications for the target system is not a topic of this subject. The reason is that the knowledge about the cross-compilation process acquired in this course is the same that the one needed for application development. Besides this, the development of a C application for Linux is covered in depth in many other courses of our Grade in Computer Science.

To teach these concepts, two elements are needed:

- A personal computer for each student, that will act as the *host* system.
- An embedded-like system, to act as the *target* system.

2.1 Choice of a host system

The first choice we faced was to select an appropriate host system. Since the first objective listed in the previous section is to install a Linux on it, we needed complete (administrator-level) access to the host chosen. The use of the PCs that belong to the University laboratories for this purpose was quickly discarded because of several reasons. First, these machines are needed for other courses (and students) as well, so their configurations should not be modified by students. Second, to allow students to have root access to systems that are wired to the University network implies a security risk. Third, to assign a computer to each student would force the student to use the same physical terminal during the entire course, not allowing students to work after hours or at home.

To solve all these problems, we decided to use virtual machines, assigning one to each student at the beginning of the term. Virtual machines are processes running in a single server. Each process emulates the behavior of a PC architecture. The console of each virtual machine can be accessed remotely via a Java-enabled, web-based interface, that shows the console as a window, with buttons that allow to turn the virtual machine on and off, or to reset it, among other functionalities. The system administrator of the virtual environments can define the basic characteristics of each virtual machine, including processor type, amount of memory, hard disks types and sizes, number and types of network interfaces, and attach ISO images as DVD devices.

In our case, we created three dozens of virtual systems, one per student, configured as a single Pentium-compatible processor, 2 Gb of RAM, a 10 GB virtual hard disk, and a virtual DVD drive with an image of the GNU/Linux distribution chosen.

Besides granting unlimited access to their virtual machines, this solution has the additional advantage of being ubiquitous: Students can stop working at any time, closing the window that represents the console, and resuming their work later from a different location, just by accessing to the virtual server website with their credentials, and opening their console again. Note that to close the console window does not imply to turn off the virtual machine: When the student resumes his/her work, the console will show the exact contents the student may expect from a real machine that has been left out for several hours.

2.2 The target system

Regarding the target system, we have decided to use the Raspberry Pi type B microcomputer. Once chosen the architecture, we had two options: To ask students to buy their own, or to lend one to each student at the beginning of the semester. We have chosen the second option for a couple of reasons. First, to obtain a Raspberry Pi is not straightforward. The interested user should buy it online, and depending on the stocks available, his/her request may take several weeks to be served. Unless all students buy their systems well in advance, they may not have them ready for use at the beginning of the academic term. Second, to own a Raspberry Pi is not enough. Several additional components are needed, including:

- An USB keyboard.
- An USB mouse.

- An 1000 mA charger.
- An SD card.
- A video cable.

These items combined, together with the Raspberry Pi type B, cost around 80 euros. We wanted to avoid such a cost to our students whenever possible.

What we did is to buy enough Raspberry Pi kits as the one described above, plus a small, tablet-size bag to transport everything —except the keyboard[1]. The cost of the entire kit is around 100 euros. We have bought them with the support of the University of Valladolid. Regarding the video cable, as long as the Raspberry Pi comes with an HDMI output, we have chosen to buy HDMI-to-DVI cables, because the monitors of our laboratory have both VGA and DVI interfaces, and in this way we can use the monitor to work concurrently with the Raspberry Pi (selecting the DVI input) and with the laboratory PC that allows the student to use their virtual machines (using the VGA input). Moreover, right now it is more likely that the students have an DVI-capable monitor at home than a monitor that supports HDMI, although this situation will likely change in a near future. At that moment, to upgrade our kits with a pure HDMI cable will be straightforward.

3 Course structure and contents

The course has been divided into eight personal projects, designed to cover everything from the basic installation of a GNU/Linux system in a host machine to the cross-compilation of an entire GNU/Linux system from scratch. Two weeks (eight laboratory hours plus homework when needed) are devoted to each project, except Project #2, that only requires one week (four lab hours). We will briefly describe their contents in the following paragraphs.

Project #1: Installing a GNU/Linux system

The first task that our students face is to install a GNU/Linux distribution in the virtual machine that will be later used as host system, using an ancient, text-mode installation procedure. The time devoted to this first project (eight lab hours) may seem excessive, but two out of ten students fails when installing the operating system and should repeat the process, even several times.

Instead of using a mainstream distribution such as Ubuntu, we have chosen the Slackware GNU/Linux distribution. Slackware has a "package" classification that is easy to understand, allowing to install just the packages needed. Besides this, the distribution has a much simpler script structure, with very few dependences among packages. This is not the case with Ubuntu, which, for example, requires the installation of a web browser to run a barebone GIMP desktop manager (!). The advantages of using Slackware will become apparent in the following projects. One disadvantage is that Slackware is less "friendly", for example requiring the use of `fdisk` for disk partitioning. The project guidelines include all the intermediate steps that should be carried out.

[1] We discourage the use of "portable" keyboards: their keys distribution is awful!

We ask the students not to install the X interface, since all the projects should be completed through text-only interfaces. Only a subset of Slackware packages will be needed in this course, namely A, AP, D, F, K, L, N, and TCL.

Project #2: Installing a Raspberry Pi GNU/Linux distribution

This task is very simple. It consists in downloading an image of an Raspberry Pi GNU/Linux image from www.raspberrypi.org, transferring it to the Raspberry Pi SD card, and boot up and configure the operating system. As it can be seen, it is not related with the previous one, but it allows the students to face the use of the (potentially dangerous) dd command.

Project #3: Native compilation of a Linux kernel in the host

It consists in downloading a tarball with a recent Linux kernel source to the host system, uncompressing it in a working directory, configuring and compiling it natively, and installing it in our virtual machine. Since Slackware uses LiLo as the boot loader, the installation of the new kernel requires changes in the /etc/lilo.conf file. The use of LiLo is far better than using GRUB for the same purpose, for a couple of reasons. First GRUB uses a name convention for disk drives and partitions that is different from the classical names that can be found in /dev. Second, with LiLo it is always clear when we are performing physical changes in the disk drive and when we are not, a situation not so clear with GRUB.

After a first, successful installation, the students are challenged to tweak the kernel configuration (using make menuconfig) to arrive to the smallest possible kernels. Students receive six points out of ten for completing the installation of a fresh kernel. The four remaining points are proportionally assigned to the students depending on how successful they were reducing the space needed by the kernel and the associated kernel modules.

Project #4: Cross compilation of a Linux kernel for Raspberry Pi in the host

To compile the Linux kernel in a Raspberry Pi may take several hours to finish. To speed up the process, in this project we use cross-compilation tools. Therefore, the first task is to download and install the crosstool-NG cross-compiling toolchain[2].

Having a cross compiler, it may seem feasible to start with a standard, platform-independent kernel tarball and configure it to run on a Raspberry Pi. However, this is not the case. Kernel configuration for a specific platform requires an in-depth knowledge of both the kernel and the target system, and this task would require considerably more than the eight hours devoted to this project. Instead, we have chosen to instruct our students to download a certain kernel source code package that is known to work with Raspberry Pi[3]. With the help of a .config file obtained from a Raspberry Pi running kernel, that gives a

[2] Available at https://github.com/raspberrypi/tools.
[3] Available at https://github.com/raspberrypi/linux.

basic set of options, the students should make oldconfig to use this basic configuration as a starting point. Then they should cross-compile the kernel in the host system and transfer the kernel image, together with the corresponding modules tree, to the Raspberry Pi.

A final remark: Root access to the host system is not strictly necessary to cross-compile the Linux kernel. fakeroot can be used instead.

Project #5: Creating a *naïve* root filesystem (RFS) for the host system

It consists in the manually creation, in the host system, of the structure of a root filesystem (RFS). This RFS will be stored into a special partition created in Project #1 as part of the Slackware GNU/Linux installation. This project makes students to face the problem of how to create an entire RFS by re-using components already available in a Linux distribution. Let us remark that this task only involves the development of a RFS to be used natively by the host system: It has nothing to do with the Raspberry Pi (yet).

The students should manually create a RFS structure, using *mkdir* to create directories, chmod to change their permissions, and populating them with some user commands (namely bash, cat, date, elvis, grep, ldd, lddlibc4, loadkeys, login, ls, mkdir, mknod, more, mount, passwd, ping, pwd, rmdir, setterm, sh, sleep, ssh, stty, sync, umount, uname, vi, which, who, whoami) and some system commands (namely agetty, fsck, halt, ifconfig, init, insmod, klogd, kmod, ldconfig, lsmod, modprobe, mount, pwconv, pwunconv, reboot, rmmod, route, shutdown, sulogin, syslogd, telinit, umount.). This process implies not only copying the commands to their final locations into the RFS (preserving their ownership by using cp -a), but also to copy the dynamic libraries needed by them (using ldd to get the library names and manually copying them too). This allows the student to understand the role of dynamic libraries in a Linux system. The result is a RFS that will be converted in the following project as a bootable partition. Other options for creating a RFS, such as the use of files formatted as partitions and mounted in *loopback* filesystems, and the use of *ramdisks* for the same purpose, are discussed but not used.

The rationale behind this project (and the following one) is to make students to really understand why this method for building a distribution is not a good idea: It is a prone-error task, and the resulting RFS takes comparatively a lot of disk space.

Project #6: Adding support for standalone boot to our host-based RFS

This project consists in completing the transformation of our new RFS in a bootable one. This implies to copy a kernel image and its corresponding modules, add bootup scripts, adjust the runlevels that will be offered, install the bootloader, reboot and fix all the issues that may appear in the host.

Projects #5 and #6 helps the student to understand what is needed to successfully boot a small (in terms of storage footprint) version of GNU/Linux. However, to further reduce the disk space needed, several actions can be taken, from stripping executables to build an entire system from scratch, with tailored

flags for our particular system architecture. This topics will be covered in the next project.

Project #7: Space reduction: The strip command, BusyBox, and Buildroot

In this project the students will try different solutions to reduce the disk space needed for their RFS. The first one is to reduce the size of executables and library files using the strip command. The second one is to compile and install BusyBox, a single executable that is capable of acting as many different Unix commands, depending on the name used in its invocation. The last one is to compile a Buildroot distribution, an entire package that combines BusyBox with a small GlibC-like library called μlibC, and generates an entire RFS ready to be installed in the system.

Project #8: Cross-compiling a RFS for the Raspberry Pi

The last project is to use the accumulated knowledge of the previous projects to cross-compile and install an entire RFS from scratch for the Raspberry Pi system. The RFS will then be augmented with the cross-compiled kernel and modules generated in Project #2.

Scores

Finally, a note about the scores of this course. The final score has two parts, "practice" and "theory", with weights of 65% and 35% respectively. Projects #1, #2, #4 and #5 account for 10% of the "practice" part of the final score, while the remaining projects account for 15% each.

The remaining 35% of the final score (the "theory" part) is evenly divided into three tests, covering the projects developed, with the aim of testing the gain in terms of knowledge of each student. The first test covers projects #1 to #4; the second one, projects #5 and #6, and the third one projects #7 and #8. Since practice is the best way to learn, students usually pass these tests with little effort.

4 Conclusions

This course on Embedded Operating Systems has been taught during two consecutive years. Our experience allows us to draw the following observations.

- **The Raspberry Pi are tough.** We have had around 20 students in the first year, and 35 in the following year. The Raspberry Pi kits given to our students were returned in perfect conditions. This is somewhat surprising, since they are given to the students without a protecting case, just with the cheap plastic boxes that are part of the basic kit. The only exception was one that was received from the distributor with a condenser completely desoldered. The student that the Raspberry was assigned to discovered the situation, asked us about the polarity of the condenser, soldered himself and the Raspberry Pi booted with no further issues.

- **The course structure is scalable (to a certain extent).** The use of virtual machines allows the student to work at home. A side effect of this flexibility is that the number of students that effectively assist to the lab classes are about one third of the enrolled students, potentially allowing larger groups. This flexibility is perceived by the students as an advantage: The feedback received shows that students prefer to work in this way.
- **The Raspberry Pi is a versatile platform.** Besides being inexpensive, the Raspberry Pi allows not only to be used as a target platform for embedded operating systems, but as a valid testbed for any further development, from any software project to hardware I/O, thanks to their exposed GPIO ports. This fact adds value to the investment in Raspberry Pi kits as part of universities' equipment. This is particularly true for universities with tight budget restrictions.

Our future work includes to merge Projects #5 and #6, making room in the term schedule for a final project consisting in (a) building a simple hardware device with some leds and buttons, (b) attach it to the Raspberry Pi GPIO interface, and (c) write a small application to control it.

In summary, we are particularly happy with the results obtained so far with this experience. Any comment or question is welcome: Please contact the author for any further inquiry.

5 Acknowledgements

The author would like to thank the anonymous reviewers for their suggestions, and the Teaching Activities Support Program of the Universidad de Valladolid for its support to the purchase of the Raspberry Pi kits needed in this course.

References

[1] Raspberry Pi User Guide, Eben Upton and Gareth Halfacree, Wiley, 2nd edition, ISBN-13 978-1118795484.