Un Sistema Linux Empotrado Sencillo

Se propone al alumno la lectura cuidadosa del siguiente artículo y la realización de actividades descritas en él. Con ello el alumno experimentará en primera persona construcción de un sistema Linux empotrado que, aunque extremadamente sencillo, es capaz de hacer arrancar un PC virtual, para, después, hacerlo desde un disco USB.

Como colofón, conteste a las siguientes preguntas. Conteste **con sus propias palabras**, citando convenientemente las fuentes de información y **no sea breve**, extiéndase todo lo necesario para explicarse clara y suficientemente.

1. **a)** ¿Para qué se realiza el mandato `usermod`,
   b) ¿Por qué sólo de ha de realizar una vez?
   c) ¿Por qué se necesita hacerlo bajo el mandato `sudo` ?

2. **a)** ¿Qué es el archivo `vmlinuz` y cuál es su formato?
   b) ¿Qué es el archivo `simple.igz` y cuál es su formato?
   c) ¿Se está utilizando un sistema de archivos `initramfs` ?

3. **a)** ¿Por qué el paquete que se instala es `busybox-static` y no `busybox` ?
   b) ¿Qué hace la invocación `/bin/busybox --install –s` ?
   (OJO! no ejecute este mandato en su HOST, podría corromper su instalación).
   c) Experimentalmente, esto es, interactuando con el sistema mediante mandatos, obtenga un listado de todas las herramientas a las que equivale dicho `busybox`.

4. **a)** ¿Pará que sirve el mandato `/init` ?
   b) ¿Pará que sirve el mandato `/sbin/init` ?

5. Tras el arranque desde USB:
   a) ¿Cuántos terminales virtuales se despliegan y para qué (un *shell* u otra cosa)?
   b) ¿Cómo hay que modificar la inicialización para que estos terminales queden configurados para el teclado nativo de la máquina (español)?

6. Tras el arranque desde USB:
   a) ¿Puede retirarse el *pendrive* o se está usando de algún modo?
   b) ¿Qué impide montar el *pendrive* para poder acceder a su contenido?
   c) ¿Cómo habría que modificar la instalación para poder resolver esto?

Envíe sus respuestas por correo electrónico a Francisco Rosales frosal@fi.upm.es, con asunto “SEMU USLES respuestas” identificándose claramente como grupo.

AVISO: Si al utilizar o instalar ubuntu para hacer esta práctica, el archivo `/vmlinuz` resulta no estar, mire en `/boot/` y si no, proceda a reinstalar el paquete que lo contiene:

Archivo:	`/boot/vmlinuz-2.6.XX-YY-generic`
Paquete:	`linux-image-2.6.XX-YY-generic`
Mandato:	`sudo apt-get –-reinstall install linux-image-2.6.XX-YY-generic`

---FIN---

by Vincent Sanders and Daniel Silverstone

Introduction

Constructing an embedded system with Linux is often seen as a complex undertaking. This article will show the fundamental aspects of constructing such systems and enable the reader to apply this knowledge to their specific situation.

This article covers the construction of the most basic system possible, which will provide a command shell on the console. It assumes a basic understanding of a Linux-based operating system. While discussing concepts and general approaches, these concepts are demonstrated with extensive practical examples. All the practical examples are based upon a Debian- or Ubuntu-based distribution.

What is an embedded system?

The term "Embedded System" has been applied to such a large number of systems that its meaning has become somewhat ill-defined. The term has been applied to everything from 4-bit microcontroller systems to huge industrial control systems.

The context in which we are using the term here is to refer to systems where the user is limited to a specific range of interaction with a limited number of applications (typically one). Thus, from the whole spectrum of applications which a general purpose computer can run, a very narrow selection is made by the creator of the embedded system software.

It should be realized that the limits of interaction with a system may involve hardware as well as software. For example, if a system is limited to a keypad with only the digits 0 to 9, user interaction will be more constrained than if the user had access to a full 102-key keyboard.

In addition to the limiting of user interaction, there may also be limits on the system resources available. Such limits are typically imposed by a system's cost, size, or environment. However, wherever possible, these limits should be arrived at with as much knowledge of the system requirements as possible. Many projects fail unnecessarily because an arbitrary limit has been set which makes a workable solution unachievable. An example of this would be the selection of a system's main memory size before the application's memory requirements have been determined.

What do you want to achieve?

A project must have a clearly defined goal.

This may be viewed as a statement of the obvious, but it bears repeating as for some unfortunately inexplicable reason, embedded systems seem to suffer from poorly-defined goals.

An "embedded" project, like any other, should have a clear statement of what must be achieved to be declared a success. The project brief must contain all the requirements, as well as a list of "desirable properties." It is essential that the two should not be confused; e.g., if the product must fit inside a 100mm by 80mm enclosure, that is a requirement. However, a statement that the lowest cost should

be achieved is a desirable item, whereas a fixed upper cost would be a requirement.

If information necessary to formulate a requirement is not known, then it should be kept as a "desirable item" couched in terms of that unknown information. It may be possible that once that information is determined, a requirement can be added.

It is, again, self-evident that any project plan must be flexible enough to cope with changes to requirements, but it must be appreciated that such changes may have a huge impact on the whole project and, indeed, may invalidate central decisions which have already been made.

General IT project management is outside the scope of this article. Fortunately there exist many good references on this topic.

Requirements which might be added to a project brief based on the assumptions of this article are:

- The system software will be based upon a Linux kernel.
- The system software will use standard Unix-like tools and layout.

The implications of these statements mean the chosen hardware should have a Linux kernel port available, and must have sufficient resources to run the chosen programs.

Another important consideration is what kind of OS the project warrants. For example, if you have a project requirement of in-field updates, then you may want to use a full OS with package management, such as Debian GNU/Linux or Fedora. Such a requirement would, however, imply a need for a non-flash-based storage medium such as a hard disc for storing the OS, as these kinds of systems are typically very large (even in minimal installations), and not designed with the constraints of flash-based storage in mind. However, given that additional wrinkle, using an extant operating system can reduce software development costs significantly.

Anatomy of a Linux-based system

Much has been written on how Linux-based systems are put together; however a brief review is in order, to ensure that basic concepts are understood.

To be strictly correct the term "Linux" refers only to the kernel. Various arguments have been made as to whether the kernel constitutes an operating system (OS) in its entirety, or whether the term should refer to the whole assemblage of software that makes up the system. We use the latter interpretation here.

The general steps when any modern computer is turned on or reset is:

- The CPU (or designated boot CPU on multi-core/processor systems) initializes its internal hardware state, loads microcode etc.
- The CPU commences execution of the initial boot code, e.g., the BIOS on x86 or the boot-loader on ARM.
- The boot code loads and executes the kernel. However, it is worth noting that x86 systems generally use the BIOS to load an intermediate loader such as GRUB or syslinux, which then fetches and starts the kernel.
- The kernel configures the hardware and executes the init process.

- The init process executes other processes to get all the required software running.

The kernel's role in the system is to provide a generic interface to programs, and arbitrate access to resources. Each program running on the system is called a process. Each operates as if it were the only process running. The kernel completely insulates a program from the implementation details of physical memory layout, peripheral access, networking, etc.

The first process executed is special in that it is not expected to exit, and is expected to perform some basic housekeeping tasks to keep a system running. Except in very specific circumstances, this process is provided by a program named /sbin/init. The init process typically starts a shell script at boot to execute additional programs.

Some projects have chosen to run their primary application as the init process. While this is possible, it is not recommended, as such a program is exceptionally difficult to debug and control. A programming bug in the application halts the system, and there is no way to debug the issue.

One feature of almost all Unix-like systems is the shell, an interactive command parser. Most common shells have the Bourne shell syntax.

A simple beginning

We shall now consider creating a minimal system. The approach taken here requires no additional hardware beyond the host PC, and the absolute minimum of additional software.

As already mentioned, these examples assume a Debian or Ubuntu host system. To use the QEMU emulator for testing, the host system must be supported by QEMU as a target. An example where this might not be the case is where the target system is x86-64, which QEMU does not support.

To ease construction of the examples, we will use the kernel's *initramfs* support. An initramfs is a gzip-compressed cpio archive of a file system. It is unpacked into a RAM disk at kernel initialization. A slight difference to normal system start-up is that while the first process executed must still be called init, it must be in the root of the file system. We will use the /init script to create some symbolic links and device nodes before executing the more-typical /sbin/init program.

This example system will use a program called *Busybox*, which provides a large number of utilities in a single executable, including a shell and an init process. Busybox is used extensively to build embedded systems of many types.

The *busybox-static* package is required to obtain pre-built copy of the Busybox binary and the *qemu* (or the new *qemu-system*) package is required to test the constructed images. These may be obtained by executing:

```
$ sudo apt-get install busybox-static qemu-system
```

As mentioned, our initramfs-based approach requires a small /init script. This configures some basic device nodes and directories, mounts the special /sys and /proc file systems, and starts the processing of hotplug events using mdev.

```
#!/bin/sh
#
/bin/busybox --install -s
#
[ -d /dev  ] || mkdir -m 0755 /dev
[ -d /root ] || mkdir --mode=0700 /root
[ -d /sys  ] || mkdir /sys
[ -d /proc ] || mkdir /proc
[ -d /tmp  ] || mkdir /tmp
mkdir -p /var/lock
#
mount -t sysfs none /sys -onodev,noexec,nosuid
mount -t proc none /proc -onodev,noexec,nosuid
#
mknod /dev/zero c 1 5
mknod /dev/null c 1 3
mknod /dev/tty c 5 0
mknod /dev/console c 5 1
mknod /dev/ptmx c 5 2
mknod /dev/tty0 c 4 0
mknod /dev/tty1 c 4 1
echo "/sbin/mdev" > /proc/sys/kernel/hotplug
#
/sbin/mdev -s
exec /sbin/init
```

To construct the cpio archive, the following commands should be executed in a shell. Note, however, that INITSCRIPT should be replaced with the location of the above script.

```
$ cp /vmlinuz vmlinuz
$ mkdir simple
$ cd simple
$ mkdir -p bin sbin usr/bin usr/sbin
$ cp /bin/busybox bin/busybox
$ ln -s busybox bin/sh
$ cp ../INITSCRIPT init
$ chmod a+x init
$ find . | cpio --quiet -o -H newc | gzip >../simple.igz
$ cd ..
```

To test the constructed image use a command like:

```
$ qemu-system -kernel vmlinuz -initrd simple.igz -append "root=/dev/ram" /dev/zero
```

This should present a QEMU window where the OS you just constructed boots and displays the message "Please press Enter to activate this console." Press enter and you should be presented with an interactive shell from which you can experiment with the commands Busybox provides. This environment is executing entirely from a RAM disc and is completely volatile. As such, any changes you make will not persist when the emulator is stopped.

Booting a real system

Starting the image under emulation proves the image ought to work on a real system, but there is no substitute for testing on real hardware. The *syslinux* package allows us to construct bootable systems for standard PCs on DOS-formatted storage.

```
$ sudo apt-get install syslinux
```

A suitable medium should be chosen to boot from, e.g., a DOS-formatted floppy disk or a DOS-formatted USB flash drive. A meaningfully labeled device (i.e. BOOTLINUX) will simplify the following procedure. The DOS partition of the USB flash drive must be marked bootable. Some USB flash drives might need repartitioning and reformatting with the Linux tools in order to work correctly.

The *syslinux* program should be run on the device /dev/fd0 for a floppy disk, or something similar to */dev/sdx1* for a USB flash drive. Care must be taken, as selecting the wrong device name might overwrite your host system's hard drive. The syslinux loader can be configured using a file called syslinux.cfg which would look something like:

```
Default simple
timeout 100
prompt 1
label simple
  kernel vmlinuz
  append initrd=simple.igz root=/dev/ram
```

The target device should then be manually mounted if it is not automounted by the system, and the kernel and the simple.igz file, copied on.

The complete command sequence to perform these actions, substituting file locations as appropriate, is:

```
$ sudo usermod -G disk -a $(whoami)    # Do this only once.
$ syslinux /dev/sdX1
$ #   # Mount hand if not magicaly mounted when connected.
$ #   sudo mount -t vfat -o shortname=mixed /dev/sdX1 /mnt/
$ #   # And then work with /mnt/ instead of /media/BOOTLINUX/
$ cp vmlinuz simple.igz syslinux.cfg /media/BOOTLINUX/
$ umount /media/BOOTLINUX/
```

The device may now be removed and booted on an appropriate PC. The PC should boot the image and present a prompt exactly the same way the emulator did.
