Embedded Systems Programming
OS Linux - Kernel

Iwona Kochańska

Gdansk University of Technology
Kernel - component responsible for managing resources and interfacing with hardware

- usually tailored to particular hardware configuration
- device trees allow to create a generic kernel that is tailored to particular hardware by the contents of the device tree.
What does the kernel do?

- 1991 - Linus Torvalds started writing Linux OS for Intel 386 and 486-based personal computers.
- Inspired by the Minix OS written by Andrew S. Tanenbaum in 1987
- Minix and Linux main difference: a 32-bit virtual memory kernel and the open source code in Linux (later released under the GPL 2 license.
- Linus wrote a **kernel** of OS, using components from the GNU project (toolchain, C library, and basic command-line tools). That distinction remains today.
- Kernel can be combined:
  - with a GNU user space -> GNU/Linux for desktops and servers
  - with an Android user space -> mobile OS
  - with a small Busybox-based user space -> compact embedded system.
What does the kernel do?

The kernel has three main jobs:

- to manage resources,
- to interface with hardware,
- to provide an API that offers a useful level of abstraction to user space programs
What does the kernel do?

- Applications running in user space run at a low CPU privilege level. They can do very little other than make library calls.
- Primary interface between the user space and the kernel space is the C library, which translates user level functions (such as those defined by POSIX) into kernel system calls.
- The system call interface uses
  - an architecture-specific method such as a trap
  - a software interrupt

to switch the CPU from the low privilege user mode to the high privilege kernel mode, which allows access to all memory addresses and CPU registers.
What does the kernel do?

- The system call handler dispatches the call to the appropriate kernel subsystem:
  - scheduling calls to the scheduler,
  - filesystem calls to the filesystem code.
- Some of those calls require input from the underlying hardware and will be passed down to a device driver.
- In some cases, the hardware itself invokes a kernel function by raising an interrupt.
- Interrupts can only be handled in a device driver, never by a user space application.

All the useful things that the application does, it does them through the kernel.
Choosing a kernel

- Kernel development cycle - a new version released every 8 to 12 weeks.
- Version numbers:
  - before July 2011: a 3 number version scheme with version numbers like 2.6.39. The middle number indicated whether it was a developer (odd) or stable (even) release
  - July 2011: the change in numbering from 2.6.39 to 3.0. No middle number.
  - April 2015: version 4
Choosing a kernel

Following the development kernel tree:
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

- Currently, a full cycle of kernel development begins with a merge window of two weeks, during which Linus will accept patches for new features.
- At the end of the merge window, a stabilization phase begins, during which Linus will produce release candidates with version numbers ending in -rc1, -rc2, and so on, usually up to -rc7 or -rc8.
- During this time, people test the candidates and submit bug reports and fixes. When all significant bugs have been fixed, the kernel is released.
- Overview of each version: http://kernelnewbies.org/LinuxVersions.
Stable releases

- stable Linux kernel is maintained by Greg Kroah-Hartman.
  - After release, the kernel moves from being mainline to stable
  - Bug fix releases of the stable kernel are marked by a third number, 3.18.1, 3.18.2, and so on. Before version 3, there were four release numbers, 2.6.29.1, 2.6.39.2, and so on.

- To cater for those users who would like updates for a longer period of time and be assured that any bugs will be found and fixed, some kernels are labeled long term and maintained for two or more years.
- There is at least one long term kernel each year (4.1, 3.18, 3.14, 3.12, 3.10, 3.4, 3.2, 2.6.32).
- 2.6.32 has been maintained for five years and is at version 2.6.32.68.
Vendor support

- mainline Linux has solid support for only a small subset of the many devices that can run Linux.
- support from independent open source projects: Linaro, Yocto Project
- support from companies providing third party support for embedded Linux,
- in many cases vendor of the SoC provides a working kernel.
The Linux source code is licensed under GPL v2

The actual text of the license for the kernel is in the file COPYING. It begins with an addendum written by Linus that states that code calling the kernel from user space via the system call interface is not considered a derivative work of the kernel and so is not covered by the license. Hence, there is no problem with proprietary applications running on top of Linux.
Licensing

- There is one area of Linux licensing that causes confusion: **kernel modules**.
- A kernel module - a piece of code that is dynamically linked with the kernel at runtime, extending the functionality of the kernel.
  - The GPL makes no distinction between static and dynamic linking, but is the source for kernel modules covered by the GPL?
  - It is accepted practice that the GPL does not necessarily apply to kernel modules. This is codified by the kernel **MODULE_LICENSE** macro, which may take the value **Proprietary** to indicate that it is not released under the GPL.
- The GPL should be considered a good thing because it guarantees that embedded system developers can always get the source code for the kernel.
Building the kernel - getting the source

- Assuming that a board is supported in mainline, the source code can be downloaded through git or as a tarball.
- Example: clone the stable tree and check out the version tag 4.1.10:

  ```
  $ cd linux $ git checkout v4.1.10
  ```

- The main directories:
  - `arch`: architecture-specific files.
  - **Documentation**: kernel documentation.
  - `drivers`: device drivers
  - `fs`: filesystem code.
  - `include`: kernel header files, including those required when building the toolchain.
  - `init`: kernel start-up code.
  - `kernel`: core functions, including scheduling, locking, timers, power management, and debug/trace code.
  - `mm`: memory management.
Kernel configuration

- Kernel can be configured to suit different jobs, from a small dedicated device such as a smart thermostat to a complex mobile handset.
- In current versions there are many thousands of configuration options.
- **Kconfig** - the configuration mechanism
- **Kbuild** - the build system
- Both are documented in *Documentation/kbuild/*.
- Kconfig/Kbuild is used in a number of other projects as well as the kernel, including crosstool-NG, U-Boot, Barebox, and BusyBox.
Kernel configuration

- The configuration options are declared in a hierarchy of files named Kconfig using a syntax described in Documentation/kbuild/kconfig-language.txt.

- Top level Kconfig in Linux:

```
mainmenu "Linux/$ARCH $KERNELVERSION Kernel Configuration"
config SRCARCH
  string
  option env="SRCARCH"
  source "arch/$SRCARCH/Kconfig"
```

The last line includes the architecture-dependent configuration file which sources other Kconfig files depending on which options are enabled.
Kernel configuration

- specify an architecture by setting ARCH=[architecture], otherwise it will default to the local machine architecture,
- the layout of the top level menu is different for each architecture.
- the value of ARCH - one of the subdirectories in directory arch (ARCH=i386 and ARCH=x86_64 both have the source arch/x86/Kconfig).
Kernel configuration

The Kconfig files consist largely of menus, delineated by keywords:
- menu,
- menu title,
- endmenu
and menu items marked by config.
Kernel configuration

- Configuration items are stored in a file named `.config` (note that the leading dot ‘.’ means that it is a hidden file that will not be shown by the `ls` command unless you type `ls -a` to show all files).
- The variable names stored in `.config` are prefixed with `CONFIG_`, so if DEVMEM is enabled, the line reads: `CONFIG_DEVMEM=y`
Kernel configuration

Data types:

- **bool**: either *y* or *not defined*.
- **tristate**: used where a feature can be built as a kernel module or built into the main kernel image. The values are *m* for a module, *y* to be built in, and *not defined* if the feature is not enabled.
- **int**: integer value written using decimal notation.
- **hex**: unsigned integer value written using hexadecimal notation.
- **string**: string value.
Kernel configuration

- Dependencies between items are expressed by the `depends on` phrase:

```c
config MTD_CMDLINE_PARTS
  tristate "Command line partition table parsing"
  depends on MTD
```

If `CONFIG_MTD` has not been enabled elsewhere, this menu option is not shown and so cannot be selected.

- Reverse dependencies: the `select` keyword enables other options if this one is enabled (enable features specific to the architecture)

```c
config ARM
  bool
  default y
  select ARCH_HAS_ATOMIC64_DEC_IF_POSITIVE
  select ARCH_HAS_ELF_RANDOMIZE
  [...]
```
Configuration utilities that can read the Kconfig files and produce a .config file:

- Menuconfig
- xconfig
- gconfig.

Launch each one via make. In case of the kernel supply an architecture:

```
$ make ARCH=arm menuconfig
```
Kernel configuration

menuconfig with the DEVMEM config option highlighted:

Search function: forward slash key, /
Kernel configuration

- A set of known working configuration files: arch/$ARCH/configs, each containing suitable configuration values for a single SoC or a group of SoCs.
- When building the kernel, a header file, include/generated/autoconf.h, is generated which contains a #define for each configuration value so that it can be included in the kernel source.
Kernel identification

- discover the kernel version that you have built using the `make kernelversion` target:

  ```
  $ make kernelversion
  4.1.10
  ```

  This is reported at runtime through the `uname` command and is also used in naming the directory where kernel modules are stored.

- Changing the configuration from the default - append the version information, by setting `CONFIG_LOCALVERSION` (General setup configuration menu). The information is seen after:

  ```
  $ make kernelrelease
  ```

  and printed at the beginning of the kernel log.
Kernel modules

- Desktop Linux distributions
  - use kernel modules extensively so that the correct device and kernel functions can be loaded at runtime depending on the hardware detected and features required.
  - without them, every single driver and feature would have to be statically linked in to the kernel, making it very large.

- Embedded devices
  - the hardware and kernel configuration is usually known at the time the kernel is built so modules are not so useful.
  - modules create a version dependency between the kernel and the root filesystem which can cause boot failures if one is updated but not the other.
  - It is quite common for embedded kernels to be built without any modules at all.
Kernel modules

When the kernel modules are good idea for embedded system?

- For proprietary modules, for the licensing reasons given in the preceding section.
- To reduce boot time by deferring the loading of non-essential drivers.
- When there are a number of drivers that could be loaded and it would take up too much memory to compile them statically. For example: USB interface to support a range of devices.
Compiling the kernel

**Kbuild** - kernel build system. A set of make scripts that:

- take the configuration information from the `.config` file,
- work out the dependencies
- compile everything that is necessary to produce a kernel image containing all the statically linked components (dtb, kernel modules)
Compiling the kernel

The **dependencies** are expressed in the **makefiles** that are in each directory with buildable components. Example:

```plaintext
obj-y += mem.o random.o
obj-$(CONFIG_TTY_PRINTK) += ttyprintfk.o
```

- **obj-y** - unconditionally compiles a file to produce the target
- **obj-$(CONFIG_PARAM)** - dependent on a configuration parameter (y, m or not defined).
Compiling the kernel image

What the bootloader expects?

- **U-Boot**: traditionally a *uImage*, but newer versions can load a *zImage* file using the *bootz* command
- **x86 targets**: a *bzImage* file
- Most other bootloaders: a *zImage* file

Example of building a *zImage* file:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- zImage
```

The -j 4 option tells make how many jobs to run in parallel
Compiling the kernel image

ARM case:

- Multi-platform support for ARM was introduced in Linux 3.7. It allows a single kernel binary to run on multiple platforms.
- The kernel selects the correct platform by reading the machine number or the device tree passed to it by the bootloader.
- Location of physical memory might be different for each platform (and so the relocation address for the kernel).
- LOADADDR = load address read from mach-[your SoC]/Makefile.boot, value of zreladdr-y.

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- LOADADDR=0x80008000 uImage
```
Compiling the kernel image

A kernel build generates two files in the top level directory:

- **vmlinux** - kernel as an ELF binary. If kernel was compiled with debug enabled (CONFIG_DEBUG_INFO=y), it will contain debug symbols which can be used with debuggers like **kgdb**.
- **System.map** - contains the symbol table in human readable form.
Compiling the kernel image

- Most bootloaders cannot handle ELF code directly.
- In further stage of processing:
  - vmlinux is converted to raw binary.
  - uImage: zImage plus a 64-byte U-Boot header.
Compiling the device tree

The `dtbs` target builds device trees according to the rules in `arch/$ARCH/boot/dts/Makefile` using the device tree source files in that directory:

```
$ make ARCH=arm dtbs
...
DTC arch/arm/boot/dts/omap2420-h4.dtb
DTC arch/arm/boot/dts/omap2420-n800.dtb
DTC arch/arm/boot/dts/omap2420-n810.dtb
DTC arch/arm/boot/dts/omap2420-n810-wimax.dtb
DTC arch/arm/boot/dts/omap2430-sdp.dtb
...
```

The `.dtb` files are generated in the same directory as the sources.
Compiling modules

Modules can build separately using the modules target:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- modules
```

- Compiled modules have a .ko suffix and are generated in the same directory as the source code,
- Use the `modules_install` make target to install them in the right place.
- The default location is `/lib/modules`. Provide a different path using `INSTALL_MOD_PATH`
Cleaning kernel sources

Make targets for cleaning the kernel source tree:

- **clean**: removes object files and most intermediates.
- **mrproper**: removes all intermediate files, including the .config file. Return the source tree to the state it was in immediately after cloning or extracting the source code.
- **distclean**: the same as mrproper but also deletes editor backup files, patch leftover files, and other artifacts of software development.
Booting the kernel

Booting is highly device-dependent.
Example - U-Boot commands to boot Linux on a BeagleBone Black:

```
U-Boot# fatload mmc 0:1 0x80200000 zImage
reading zImage
4606360 bytes read in 254 ms (17.3 MiB/s)
U-Boot# fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
reading am335x-boneblack.dtb
29478 bytes read in 9 ms (3.1 MiB/s)
U-Boot# setenv bootargs console=tty00,115200
U-Boot# bootz 0x80200000 - 0x80f00000
Kernel image @ 0x80200000 [ 0x000000 - 0x464998 ]
## Flattened Device Tree blob at 80f00000
  Booting using the fdt blob at 0x80f00000
  Loading Device Tree to 8fff5000, end 8ffff325 ... OK
Starting kernel ...
[ 0.000000] Booting Linux on physical CPU 0x0
...```
Booting the kernel - kernel panic

**Kernel panic** occurs when the kernel encounters an unrecoverable error.

- by default, it will print out a message to the console and then halt.
- You can set the panic command line parameter to allow a few seconds before it reboots following a panic.

Example:

```
[ 1.886379] Kernel panic — not syncing:
VFS: Unable to mount root fs on unknown–block (0,0)
[ 1.895105] ——[ end Kernel panic — not syncing:
VFS: Unable to mount root fs on unknown–block (0, 0)
no root filesystem!
```
A user space can be supplied by providing a root filesystem either as a ramdisk or on a mountable mass storage device.

```bash
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
fatload mmc 0:1 0x81000000 uRamdisk
setenv bootargs console=tty00,115200 rdinit=/bin/sh
cbootz 0x80200000 0x81000000 0x80f00000
```
In order to transition from kernel initialization to user space, the kernel has to mount a root filesystem and execute a program in that root filesystem.

This can be via a ramdisk or by mounting a real filesystem on a block device.

The code for all of this is in `init/main.c`, starting with the function `rest_init()` which creates the first thread with PID 1 and runs the code in `kernel_init()`. If there is a ramdisk, it will try to execute the program `/init`, which will take on the task of setting up the user space.
Booting the kernel - early user space

- If it fails to find and run /init, it tries to mount a filesystem by calling the function `prepare_namespace()` in `init/do_mounts.c`. This requires a `root=` command line to give the name of the block device to use for mounting, usually in the form:
  - `root=/dev/<disk name><partition number>`
  - `root=/dev/<disk name>p<partition number>`

- For example, for the first partition on an SD card, that would be `root=/dev/mmcblk0p1`. If the mount succeeds, it will try to execute `/sbin/init`, followed by `/etc/init`, `/bin/init`, and then `/bin/sh`, stopping at the first one that works.
Kernel messages

Useful information can be printed using `printk()`. The messages are categorized according to importance, 0 being the highest:

<table>
<thead>
<tr>
<th>Level</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KERN_EMERG</td>
<td>0</td>
<td>The system is unusable</td>
</tr>
<tr>
<td>KERN_ALERT</td>
<td>1</td>
<td>Action must be taken immediately</td>
</tr>
<tr>
<td>KERN_CRIT</td>
<td>2</td>
<td>Critical conditions</td>
</tr>
<tr>
<td>KERN_ERR</td>
<td>3</td>
<td>Error conditions</td>
</tr>
<tr>
<td>KERN_WARNING</td>
<td>4</td>
<td>Warning conditions</td>
</tr>
</tbody>
</table>
Kernel messages

<table>
<thead>
<tr>
<th>Level</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KERN_NOTICE</td>
<td>5</td>
<td>Normal but significant conditions</td>
</tr>
<tr>
<td>KERN_INFO</td>
<td>6</td>
<td>Informational</td>
</tr>
<tr>
<td>KERN_DEBUG</td>
<td>7</td>
<td>Debug-level messages</td>
</tr>
</tbody>
</table>
Kernel messages

- Messages are first written to a buffer, \texttt{\_log\_buf}, the size of which is two to the power of \texttt{CONFIG\_LOG\_BUF\_SHIFT}.
- Dump the entire buffer using the command \texttt{dmesg}.
- If the level of a message is less than the console log level, it is displayed on the console as well as being placed in \texttt{\_log\_buf}.
- The default console log level is 7, meaning that messages of level 6 and lower are displayed, filtering out KERN\_DEBUG which is level 7.
- Change the console log level: using the kernel parameter \texttt{loglevel=<level>} or the command \texttt{dmesg -n <level>}. 
Kernel command line

Kernel command line - string passed to the kernel:
- by the bootloader (bootargs variable in the case of U-Boot)
- defined in the device tree,
- set as part of the kernel configuration in CONFIG_CMDLINE.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>Sets the console log level to the highest level, eight, to ensure that you see all the kernel messages on the console.</td>
</tr>
<tr>
<td>init=</td>
<td>The <strong>init</strong> program to run from a mounted root filesystem, which defaults to /sbin/init.</td>
</tr>
<tr>
<td>lpj=</td>
<td>Sets the <strong>loops_per_jiffy</strong> to a given constant</td>
</tr>
<tr>
<td>panic=</td>
<td>Behavior when the kernel panics: if it is greater than zero, it gives the number of seconds before rebooting; if it is zero, it waits forever (this is the default); or if it is less than zero, it reboots without any delay.</td>
</tr>
</tbody>
</table>
### Kernel command line

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quite</td>
<td>Sets the console log level to one, suppressing all but emergency messages. Since most devices have a serial console, it takes time to output all those strings. Consequently, reducing the number of messages using this option reduces boot time.</td>
</tr>
<tr>
<td>rdinit=</td>
<td>The <em>init</em> program to run from a ramdisk, it defaults to <code>/init</code>.</td>
</tr>
<tr>
<td>ro</td>
<td>Mounts the root device as read-only. Has no effect on a ramdisk which is always read/write.</td>
</tr>
<tr>
<td>root=</td>
<td>Device to mount the root filesystem.</td>
</tr>
</tbody>
</table>
## Kernel command line

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootdelay=</td>
<td>The number of seconds to wait before trying to mount the root device, defaults to zero. Useful if the device takes time to probe the hardware, but also see rootwait.</td>
</tr>
<tr>
<td>rootfstype=</td>
<td>The filesystem type for the root device. In many cases, it is auto-detected during mount, but it is required for jffs2 filesystems.</td>
</tr>
<tr>
<td>rootwait</td>
<td>Waits indefinitely for the root device to be detected. Usually necessary with mmc devices.</td>
</tr>
<tr>
<td>rw</td>
<td>Mounts the root device as read-write (default).</td>
</tr>
</tbody>
</table>
Futher reading

- Ch. Simmonds, Mastering Embedded Linux Programming, PACKT 2015
- Linux Kernel Newbies, kernelnewbies.org