Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[Lab 02] Building the Kernel

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v1

Building the Kernel

Preparation

For building the kernel you not only need the gcc compiler but a set of build tools. Remember that specific versions of these tools can build specific versions of the Linux kernel. Before building the kernel have a look at the file `/Documentation/Changes`.

```
29 ====================== ===============  ========================================
30         Program         Minimal version       Command to check the version
31 ====================== ===============  ========================================
32 GNU C                  4.9              gcc --version
33 Clang/LLVM (optional)  10.0.1           clang --version
34 GNU make               3.81             make --version
35 binutils               2.23             ld -v
36 flex                   2.5.35           flex --version
37 bison                  2.0              bison --version
38 util-linux             2.10o            fdformat --version
39 kmod                   13               depmod -V
40 e2fsprogs              1.41.4           e2fsck -V
41 jfsutils               1.1.3            fsck.jfs -V
42 reiserfsprogs          3.6.3            reiserfsck -V
43 xfsprogs               2.6.0            xfs_db -V
44 squashfs-tools         4.0              mksquashfs -version
45 btrfs-progs            0.18             btrfsck
```

Example from source v5.11.2

Preparation

In Ubuntu 20, for installing most of the build tools for the kernel it suffices to install the package `build-essential` with the command

```
sudo apt install build-essential
```

Other tools that you need have to be installed manually, in our case (Ubuntu 20.10)

```
sudo apt install libncurses-dev flex bison libelf-dev libssl-dev dwarves
```

SUGGESTION: To speed-up next builds I highly suggest you to install ccache

```
sudo apt install ccache
```

and set a reasonable cache value, link 20G (make sure that you have at least 50GB free)

```
ccache -M 20G
```

For using ccache you need to symlink `gcc` to "`ccache /usr/bin/gcc`" otherwise when doing the make command you have to explicitly assign `CC="ccache /usr/bin/gcc"` see (L1 L2)

Build Commands

For building the kernel you need:

1. To generate or use an already available .config file

```
make menuconfig
```

2. To build the kernel itself and the modules (comprehends also drivers)

```
make -j <number of cores>
```

3. Copy compiled modules in `/lib/modules/<version>`

```
make modules_install
```

4. To generate an `initramfs`
5. To put the generated binaries and images in the /boot folder

```
sudo make install
```

6. To update the GRUB boot options

```
grub-mkconfig -o /boot/grub/grub.cfg
```

initrd / initramfs

Ramdisk

When kernel boots and the process init is spawned, before mounting physical disks, we need a temporary filesystem which contains all the auxiliary binaries and tools that are needed for the first process to run. This temporary filesystem is called ramdisk and when it is mounted, it is readonly and only exists in RAM it has not a physical counterpart and during the boot it is replaced by the real root filesystem ("/").

The ramdisk implementations are essentially two: initrd and initramfs. Ramdisk is generated as an image, a CPIO image with a particular tool, just after building the kernel binary image.

initrd vs initramfs

In the **`initrd`** scheme, the image may be a file system image (optionally compressed), which is made available in a special block device (/dev/ram) that is then mounted as the initial root file system ("/"). The driver for that file system must be compiled statically into the kernel. Once the initial root file system is up, the kernel executes `/linuxrc` as its first process; when it exits, the kernel assumes that the real root file system has been mounted and executes /sbin/init to begin the normal user-space boot process.

In the **`initramfs`** scheme (from kernel 2.6.13), the image may be a CPIO archive (optionally compressed). The archive is unpacked by the kernel into a special instance of a `tmpfs` that becomes the initial root file system. This scheme has the advantage of not requiring an intermediate file system or block drivers to be compiled into the kernel. In the initramfs scheme, the kernel executes `/init` as its first process that **is not expected to exit**. For some applications, initramfs can use the casper utility to create a writable environment using unionfs to overlay a persistence layer over a read-only root filesystem image.

initramfs

The latest distribution of Ubuntu for example uses initramfs (even if the image is called initrd), the image is created automatically when issuing `make install` with the script `update-initramfs` which in the end uses the `mkinitramfs` tool.

This on the right is the content of the `initramfs`, you can unpack it by using `unmkinitramfs` command. In the early folder you can find the microcode used for Intel and AMD, and in the main folder there is real filesystem that is mounted before launching `init`.

```
├── early
│   └── kernel
├── early2
│   └── kernel
└── main
    ├── bin -> usr/bin
    ├── conf
    ├── etc
    ├── init
    ├── lib -> usr/lib
    ├── lib32 -> usr/lib32
    ├── lib64 -> usr/lib64
    ├── libx32 -> usr/libx32
    ├── run
    ├── sbin -> usr/sbin
    ├── scripts
    ├── usr
    └── var
```

init

The init file in the root of the initramfs is a script for the `/bin/sh` interpreter that contains all the necessary instructions for initializing the true root filesystem by also parsing the kernel cmdline, namely the arguments passed by GRUB when launching the kernel.

In the end the script calls `exec` command by passing the program `run-init`,

```
exec run-init ${drop_caps} "${rootmnt}" "${init}" "$@" <"${rootmnt}/dev/console"
                           >"${rootmnt}/dev/console" 2>&1
```

The run-init executable:

1. Delete all files in the initramfs;
2. Remounts /real-root onto the root filesystem;
3. Drops comma-separated list of capabilities;
4. Chroots;
5. Opens /dev/console;
6. Spawns the specified `init` program (with arguments.)

https://git.kernel.org/pub/scm/libs/klibc/klibc.git/tree/usr/kinit/run-init/run-init.c

Kernel Build System

Build files

V5.11

The kernel Makefile is split in 5 parts:

- **Makefile**: the top Makefile.
- **.config**: the kernel configuration file, stores each config symbol's selected value. You can edit this file manually or use one of the many make configuration targets, such as menuconfig and xconfig, that call specialized programs to build a tree-like menu and automatically update (and create) the .config file for you by reading **KConfig** files
- **arch/$(SRCARCH)/Makefile**: the arch Makefile
- **scripts/Makefile.***: common rules etc. for all kbuild Makefiles.
- **kbuild Makefiles:** exist in every subdirectory and instruct how to build subsystems

Full source at https://www.kernel.org/doc/html/v5.11/kbuild/makefiles.html

Build files

KConfig Example for Block devices

```
# SPDX-License-Identifier: GPL-2.0
#
# Block device driver configuration
#

menuconfig BLK_DEV
        bool "Block devices"
        depends on BLOCK
        default y
        help
          Say Y here to get to see options for various different block device
          drivers. This option alone does not add any kernel code.

          If you say N, all options in this submenu will be skipped and disabled;
          only do this if you know what you are doing.

if BLK_DEV

source "drivers/block/null_blk/Kconfig"

config BLK_DEV_FD
        tristate "Normal floppy disk support"
        depends on ARCH_MAY_HAVE_PC_FDC
        help
          If you want to use the floppy disk drive(s) of your PC under Linux,
          say Y. Information about this driver, especially important for IBM
          Thinkpad users, is contained in
          <file:Documentation/admin-guide/blockdev/floppy.rst>.
          That file also contains the location of the Floppy driver FAQ as
          well as location of the fdutils package used to configure additional
          parameters of the driver at run time.

          To compile this driver as a module, choose M here: the
          module will be called floppy.
```

https://elixir.bootlin.com/linux/v5.11.2/source/drivers/block/Kconfig

Build files

Kbuild Example for Block devices

A KBuild file can be named `KBuild` or just `Makefile`. A KBuild file define the files to be built, any special compilation options, and any subdirectories to be entered recursively.

```
1  # SPDX-License-Identifier: GPL-2.0
2  #
3  # Makefile for the kernel block device drivers.
4  #
5  # 12 June 2000, Christoph Hellwig <hch@infradead.org>
6  # Rewritten to use lists instead of if-statements.
7  #
8
9  # needed for trace events
10 ccflags-y                              += -I$(src)
11
12 obj-$(CONFIG_MAC_FLOPPY)              += swim3.o
13 obj-$(CONFIG_BLK_DEV_SWIM)            += swim_mod.o
14 obj-$(CONFIG_BLK_DEV_FD)              += floppy.o
15 obj-$(CONFIG_AMIGA_FLOPPY)            += amiflop.o
16 obj-$(CONFIG_PS3_DISK)                += ps3disk.o
17 obj-$(CONFIG_PS3_VRAM)                += ps3vram.o
```

Eventual gcc flags

This **`$(...)`** can be y or `m`:

- **`obj-y`** the object will be built-in in the kernel image
- **`obj-m`** the object will be compiled as a module
- otherwise it will be not compiled or linked

This the dependency

Linking

V5.11

KBuild and `obj-y`

Kbuild compiles all the `$(obj-y)` files. It then calls "`$(AR) rcSTP`" to merge these files into one `built-in.a` file. This is a thin archive without a symbol table. It will be later linked into vmlinux by `scripts/link-vmlinux.sh`

The order of files in `$(obj-y)` is significant. Duplicates in the lists are allowed: the first instance will be linked into `built-in.a` and succeeding instances will be ignored.

Link order is significant, because certain functions (`module_init()` / `__initcall`) will be called during boot in the order they appear. So keep in mind that changing the link order may e.g. change the order in which your SCSI controllers are detected, and thus your disks are renumbered.

Final Kernel linking

The final linking is carried out via the link-vmlinux.sh script.

```
#!/bin/sh
# SPDX-License-Identifier: GPL-2.0
#
# link vmlinux
#
# vmlinux is linked from the objects selected by $(KBUILD_VMLINUX_OBJS) and
# $(KBUILD_VMLINUX_LIBS). Most are built-in.a files from top-level directories
# in the kernel tree, others are specified in arch/$(ARCH)/Makefile.
# $(KBUILD_VMLINUX_LIBS) are archives which are linked conditionally
# (not within --whole-archive), and do not require symbol indexes added.
#
# vmlinux
#   ^
#   |
#   +--< $(KBUILD_VMLINUX_OBJS)
#   |    +--< init/built-in.a drivers/built-in.a mm/built-in.a + more
#   |
#   +--< $(KBUILD_VMLINUX_LIBS)
#   |    +--< lib/lib.a + more
#   |
#   +-< ${kallsymso} (see description in KALLSYMS section)
#
# vmlinux version (uname -v) cannot be updated during normal
# descending-into-subdirs phase since we do not yet know if we need to
# update vmlinux.
# Therefore this step is delayed until just before final link of vmlinux.
#
# System.map is generated to document addresses of all kernel symbols
```

Symbols table subsystem

https://elixir.bootlin.com/linux/v5.11.2/source/scripts/link-vmlinux.sh

Kernel Symbols Tables

There are two ways for the kernel of maintaining the symbols table, i.e. the mapping between symbols and virtual memory locations:

- **System.map** is a file generated during linking time and contains only static symbols that regards, steady-state Kernel functions (steady-state ones), kernel data structures. Symbols are associated with 'storage class':
  - `T`: global (non-static but not necessarily exported) function
  - `t`: a function local to the compilation unit (i.e. static)
  - `D`: global data;
  - `d`: data local to the compilation unit
  - `R/r`: same as `D/d`, but for read-only data
  - `B/b`: uninitialized data section (.bss)
- **kallsyms** is a specific subsystem that maintains a list of the kernel symbols table that are both deriving from static code, namely the ones that are extracted during the kernel linking phase, as well as run-time ones, for example given by loaded modules for example. The table is available at location `/proc/kallsyms`

Kernel Symbols Tables

Why maintaining a symbols table?

- Kernel debugging, for kernel oops
- Kernel run-time hacking, some driver may require to search symbols in the table

System.map example

```
ffffffff8339ad60 D x86_cpu_to_node_map_early_map
ffffffff833a2d60 D numa_nodes_parsed
ffffffff833a2de0 d kaslr_regions
ffffffff833a4000 d sme_cmdline_off
ffffffff83a00000 B __end_of_kernel_reserve
ffffffff83a10000 b .brk.dmi_alloc
ffffffff83a2c000 B __brk_limit
ffffffff83a2c000 B _end
```

Advanced Operating Systems and Virtualization

[Lab 02] Building the Kernel

LECTURER

Gabriele **Proietti Mattia**

gpm.name · proiettimattia@diag.uniroma1.it

DIAG