METU Department of Computer Engineering

Contents

Kernel Compilation and Development Howto

Introduction

This tutorial guides you toward kernel programming assignments in CEng 536 Advanced Unix. You need to install dependencies before the compilation.

1. Retrieve Kernel

Kernel sources can be retrieved from various resources:

- Github Kernel Repo: https://github.com/torvalds/linux/
  Pick the correct version from the "Tags" 6.11 version is accessible from:
  https://github.com/torvalds/linux/tree/98f7e32f20d28ec452afb208f9cffc08448a2652
- linux-source packages of distributions `apt install linux-source-6.11`
  Newever versions can be found from backports:
  http://ftp.tr.debian.org/debian/pool/main/l/linux/linux-source-6.11_6.11.5-1~bpo12+1_all.deb

2. Configuration

Kernel source code can be configured using `make menuconfig` or `make xconfig`. It is a long task. If you need to compile a similar kernel with your current distribution, `cp /boot/config-$(uname -r) .config` command will copy the distribution kernels configuration.

If you like your kernel to be compatible with the distributions modules (in `initramfs` and modules under `/lib/modules/$(uname -r)`) you want to make the kernel names compatible. In order to do that, edit first lines of `Makefile`.

```
VERSION = 6
PATCHLEVEL = 1
SUBLEVEL = 119
EXTRAVERSION =
NAME = Curry Ramen
```

If your kernel is like `6.1.0-27-amd64`, set `SUBLEVEL = 0` and `EXTRAVERSION = 27-amd64`. The distributions also use signed binaries. You may need to import their keys as well

When you add a new system call as it is in the second assignment, distribution kernel modules will have a modversion calculation mismatch and complain (number of system calls is a component of kernel module version hash at compilation). In order to deal with that we build a small sample kernel that can boot a virtual machine without modules.

The following is a set of variables you need to change (mostly from module to yes, builtin)

```
CONFIG_ACPI_BUTTON=y
CONFIG_ACPI_CONFIGFS=y
```

```
CONFIG_MODULE_SIG_ALL=y
CONFIG_BLK_DEV_INTEGRITY_T10=y
CONFIG_NETFILTER_XTABLES=y
CONFIG_IP_NF_IPTABLES=y
CONFIG_FW_CFG_SYSFS=y
CONFIG_EFI_VARS_PSTORE=y
CONFIG_PARPORT=y
CONFIG_PARPORT_PC=y
# CONFIG_BLK_DEV_FD is not set
CONFIG_CDROM=y
CONFIG_BLK_DEV_LOOP=y
CONFIG_SCSI_MOD=y
CONFIG_RAID_ATTRS=y
CONFIG_SCSI_COMMON=y
CONFIG_SCSI=y
CONFIG_BLK_DEV_SD=y
CONFIG_CHR_DEV_ST=y
CONFIG_BLK_DEV_SR=y
CONFIG_CHR_DEV_SG=y
CONFIG_CHR_DEV_SCH=y
CONFIG_ATA=y
CONFIG_SATA_AHCI=y
CONFIG_SATA_AHCI_PLATFORM=y
CONFIG_ATA_PIIX=y
CONFIG_PATA_MPIIX=y
CONFIG_PATA_ACPI=y
CONFIG_ATA_GENERIC=y
CONFIG_BLK_DEV_MD=y
CONFIG_MD_AUTODETECT=y
CONFIG_MD_RAID0=y
CONFIG_MD_RAID1=y
CONFIG_MD_RAID10=y
CONFIG_MD_RAID456=y
CONFIG_BLK_DEV_DM=y
CONFIG_DM_BUFIO=y
CONFIG_DM_RAID=y
# CONFIG_DM_INIT is not set
CONFIG_DM_INTEGRITY=y
CONFIG_TARGET_CORE=y
CONFIG_E1000=y
CONFIG_E1000E=y
CONFIG_INPUT_JOYDEV=y
CONFIG_INPUT_EVDEV=y
CONFIG_MOUSE_PS2=y
CONFIG_INPUT_PCSPKR=y
CONFIG_SERIO_RAW=y
CONFIG_PPDEV=y
CONFIG_I2C_ALGOBIT=y
CONFIG_I2C_PIIX4=y
CONFIG_DRM=y
# CONFIG_DRM_DEBUG_MM is not set
CONFIG_DRM_KMS_HELPER=y
CONFIG_DRM_TTM=y
CONFIG_DRM_VRAM_HELPER=y
```

```
CONFIG_DRM_TTM_HELPER=y
CONFIG_DRM_GEM_SHMEM_HELPER=y
CONFIG_DRM_BOCHS=y
CONFIG_DRM_CIRRUS_QEMU=y
# CONFIG_USB_CONFIGFS is not set
CONFIG_EXT2_FS=y
# CONFIG_EXT2_FS_XATTR is not set
CONFIG_EXT3_FS=y
# CONFIG_EXT3_FS_POSIX_ACL is not set
# CONFIG_EXT3_FS_SECURITY is not set
CONFIG_EXT4_FS=y
CONFIG_JBD2=y
CONFIG_FS_MBCACHE=y
CONFIG_FS_ENCRYPTION_ALGS=y
CONFIG_AUTOFS4_FS=y
CONFIG_AUTOFS_FS=y
CONFIG_FUSE_FS=y
CONFIG_FAT_FS=y
CONFIG_MSDOS_FS=y
CONFIG_VFAT_FS=y
CONFIG_CONFIGFS_FS=y
CONFIG_NLS_CODEPAGE_437=y
CONFIG_NLS_ASCII=y
CONFIG_NLS_UTF8=y
CONFIG_XOR_BLOCKS=y
CONFIG_ASYNC_CORE=y
CONFIG_ASYNC_MEMCPY=y
CONFIG_ASYNC_XOR=y
CONFIG_ASYNC_PQ=y
CONFIG_ASYNC_RAID6_RECOV=y
CONFIG_CRYPTO_CRC32C=y
CONFIG_CRYPTO_CRCT10DIF=y
CONFIG_CRYPTO_CRC64_ROCKSOFT=y
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"
CONFIG_SYSTEM_TRUSTED_KEYS=""
CONFIG_RAID6_PQ=y
CONFIG_CRC16=y
CONFIG_CRC_T10DIF=y
CONFIG_CRC64_ROCKSOFT=y
CONFIG_CRC64=y
CONFIG_LIBCRC32C=y
```

So you can copy stock kernels config and edit `.config` to match does values or simply copy the provided configuration.

3. Compilation

`make -j 10 bzImage` command will compile the kernel in 10-20 minutes into `arch/x86/book/bzImage` which is a compressed kernel image. It as the same format with the original kernel at `/boot/vmlinuz-$(uname -r)`.

Another important file is `vmlinuz` in the kernel source top level. This is the uncompressed, unstripped version of the same image. You can use this to debug your kernel with `gdb`.

4. Virtual Machine Image

Even though this kernel can boot your host machine, it is not a bad idea since it lacks many drivers and it cannot load the modules in the distribution kernel. Also your kernel crash (it happens frequently during development) cause your host computer to crash with a stack trace, which is called a *kernel panic*. By using a virtual machine, we make our virtual machine process crash instead. Also host computer can debug the guest using `gdb`.

In order to create an image you need `qemu-img` tool from `qemu-utils` package. It creates a file as a virtual disk. It uses kernel network block device `nbd`. Load the module if not isntalled

```
sudo modprobe nbd
```

The next step will be installing Linux on this image. Running a virtual machine and installing from a ISO image is time consuming so a better solution is to use `debootstrap` utility.

The script provided in github combines all steps:
https://github.com/onursehitoglu/qemu-debian-create-image.git

Checkout this script with `git` command and execute with:

```
export IMGSIZE=30G
sudo ./qemu-debian-create-image yourimagefile.qcow2 yourhostname bookwork
```

Replace `yourimagefile` and `yourhostname` with image file name and your host name. This command depends debian like Linux distributions.

QCOW2 format is a dynamic format, it grows as you use your image in the virtual machine. Initially it will be small. 30G above is an upper limit. If you like, you can extend it.

After the command successfully complete you will have a linux in `yourimagefile.qcow2`.

5. Booting Virtual Machine

`qemu-system` package provides the command line virtual machine starter. In order to run a VM as fast as the host, you need maching architectures and `kvm` module. `kvm` module can be loaded as:

```
modprobe kvm
```

Your user needs to be in `kvm` group to use kvm directly. Otherwise you may need `sudo`. A typical VM start command line is:

```
qemu-system -enable-kvm -m size=1G -hda yourimagefile.qcow2
```

This will boot your image. `root` password is `root` in `debootstrap` images.

6. Booting Your Compiled Kernel

Your image contains the kernel image and an initial ram disk to boot your stock debian kernel. If you like to boot it with your compiled kernel, you need to pass `-kernel` parameter. In order to complete the boot and mount root file system you need to pass `-append "ro root=/dev/sda3 nokaslr"` parameters.

```
qemu-system -enable-kvm -m size=1G -hda yourimagefile.qcow2 -kernel arch/x86/boot/bzImage -append "ro
root=/dev/sda3 nokaslr" -s
```

The last `nokaslr` and `-s` are usefull for debugging.

7. Debugging

In order to debug your implementation. You can use `gdb` in the host machine. Boot your VM with `-s` parameter that makes it listen `localhost:1234` for debugging. Also `nokaslr` makes kernel symbols to be accessible. Then run the debugger with:

```
gdb /usr/src/linux-6.11/vmlinuz
```

This will load the uncompressed kernel image in the debugger. But debugger is not attached to the VM yet. Use

```
target remote localhost:1234
```

To attach it. This will stop the VM and give the current stack position on debugger console. You can run `cont` to resume kernel execution. In order to put a breakpoint on a function, run:

```
break __x64_sys_check_addr
```

on debugger console. This will put a breakpoint on your system call defined as `sys_check_addr` in 64bit Intel architecture. You can than run `cont`, execute your function in the guest and debug it.