CONCEPTES AVANÇATS DE SISTEMES OPERATIUS

Facultat d'Informàtica de Barcelona, Dept. d'Arquitectura de Computadors, curs 2018/2019 – 2Q

Pràctiques de laboratori System boot process

Establishing laboratory groups

CASO lab sessions are taken in groups of 2 people. Our first step in this session of laboratory is to build the laboratory groups:

Find the partner with whom you will share the laboratory sessions of CASO.

When you have a partner for the lab sessions, **find a free computer in the lab class**, and get it.

Start the computer and boot "**Càrrega Ubuntu 18.04 LTS 64 bits**".

At the login screen use

Username: alumne

Password: ********** <ask me>

and start this laboratory session.

Objectives

In this laboratory session, our main objective is to understand the various boot mechanisms that computers use nowadays. In order to do so, we will use the QEMU virtualizer with simpler and more complex loader and OS images to practice safely about all of them. During the session, you will practice with various components:

- QEMU virtualizer installation for x86_64 and arm64 architectures
- x86_64 legacy boot based on the traditional DOS partition table
- x86_64 UEFI boot based on the new GPT partition table
- arm64 boot based on u-boot

Additionally, this session will be also useful as a practice consisting to follow a pre-established system administrator task, like the following:

"Our company is interested in having a computer ready to test simple OS installations in a virtual environment"… a colleague of us has established and documented this procedure, and we'll follow it to achieve the same results. **Document any deviations you need to follow in case your environment is different.**

Preparation

Open a shell session (preferably a graphical X-Windows console) with the /bin/bash command line interpreter (it is used by default for user "alumne") to follow the following steps (the $ sign at the beginning of each line represents the BASH prompt).

During all the session, try **not to copy-paste** the text from this document, as **some characters** in the document (-, =, \…) **may have a different encoding** than those obtained from the keyboard, and copy-pasting them will make the commands fail with some strange errors.

Use your ${HOME} directory to start the session:

```
$ cd  ${HOME}
```

Download the latest stable version of QEMU in source form:

```
$ wget http://wiki.qemu-project.org/download/qemu-3.1.0.tar.bz2
```

Unpack the source code:

```
$ bunzip2 <qemu-3.1.0.tar.bz2 | tar xf -      # the minus sign is the argument to 'f', meaning
                                              # take the data from the standard input channel,
                                              # in this case, the channel is connected to the pipe
```

or, alternatively:

```
$ tar jxf qemu-3.1.0.tar.bz2          # using tar to implicitly uncompress the file
```

Create a directory to store the configuration and object files of the compilation:

```
$ mkdir obj
```

Enter the obj directory to perform a clean configuration and building:

```
$ cd obj
```

Configure QEMU (observe the two consecutive $ signs separated by spaces at the beginning of the line, which refer to i) BASH prompt, and ii) the command ${HOME}/configure to execute):

```
$ ${HOME}/qemu-3.1.0/configure --prefix=/usr/local \
  --target-list=aarch64-softmmu,x86_64-softmmu \
  --enable-kvm --enable-sdl --enable-curses
```

In the previous command, observe the back-slash characters (\) at the end of the two first lines. They mean that the command is not finished with the end of line, but it continues on the next line. This causes BASH to emit a secondary prompt, and allow the user to continue typing. Be careful that there should not be any character other than the end of line, after the back-slash.

You will encounter (probably) a configuration error:

ERROR: zlib check failed # this means that configure checks if zlib is installed, and
it is not!!!

So, install "libz-dev" # the name of the package is not intuitive, given the previous error

```
$ sudo apt-get install libz-dev       # use alumne's password for sudo to act as an administrator
```

Other packages missing:

```
SDL devel:     sudo apt-get install  libsdl-dev      # this installs most of the X-Windows devel
SDL2 devel:   sudo apt-get install libsdl2-dev          # just in case…
ncurses devel: sudo apt-get install libncurses5-dev libncursesw5-dev
autotools-dev: sudo apt-get install autotools-dev  # support for configure
automake:       sudo apt-get install automake        # autoreconf, automake, autoheader
libtool:          sudo apt-get install libtool           # libtool
```

And now the previous "configure… " command should succeed…

Compile QEMU

```
$ make
```

Install QEMU (you will need to install more packages)

```
$ make install
```

Now you should be able to run it correctly, because you have the PATH set to contain /usr/local/bin, where the two commands qemu-system-x86_64 and qemu-system-aarch64 have been installed.

Ensure that the two commands are the newly installed:

```
$ which qemu-system-x86_64
/usr/local/bin/qemu-system-x86_64

$ which qemu-system-aarch64
/usr/local/bin/qemu-system-aarch64
```

If any of the two paths returned by these ‘which’ commands is different, please do:

```
$ export PATH=/usr/local/bin:$PATH
```

to ensure that the directory /usr/local/bin is searched first while trying to find them.

(You can also display the contents of the PATH environment variable to ensure that it is correct:

```
$ echo $PATH
```

)

Starting on x86_64

Execute the command to boot a machine with the x86_64 architecture.

```
$ qemu-system-x86_64
```

It should open a graphical window. Try to understand the output displayed, until it stops.

1. Do you think this machine has a BIOS ROM memory associated?

2. Why do you think so?

At any time you can kill the execution with the window kill icon on the top-left corner.

If the mouse device gets trapped by the QEMU window, you can free it using **ctrl-alt**.

It tries to boot from the network, but it does not succeed, and it has no other devices defined to boot from.

Let’s look at the various machines emulated by QEMU:

```
$ qemu-system-x86_64 -machine help
pc              Standard PC (i440FX + PIIX, 1996) (alias of pc-i440fx-2.7)
…
q35             Standard PC (Q35 + ICH9, 2009) (alias of pc-q35-2.7)
isapc           ISA-only PC
none            empty machine
```

Let’s use from now on the modern machine “q35”.

```
$ qemu-system-x86_64 -machine q35
```

Check that it behaves exactly as the older emulated machine did before: it does not boot.

Booting on x86_64

Let's attach a disk to the machine. To do so, let's first create a file that will represent the virtual disk:

```
$ dd if=/dev/zero of=disk0.img bs=512 count=$((128*1024))
```

The "dd" command stands for "disk dump". It allows to copy data between files and devices and viceversa, in various forms. In the previous command, it reads zero (0) characters from /dev/zero, and writes them to the file disk0.img. Reading and writing is done in blocks of 512 characters at a time, and it copies 131072 blocks (= 128 x 1024). Check that the resulting disk0.img has 64 MBytes afterwards:

```
$ ls -l disk0.img

-rw-r--r--  1 xavim users    67108864    Sep  4   00:17   disk0.img

$ echo $((64*1024*1024))

67108864
```

An observation here: BASH allows to embed formulas in shell commands using the form of $ ((…)), as it is seen in the previous commands. This is a useful way to avoid getting the calculator to type the final command.

You can also check that the "virtual disk" created in such a file contains 64 Mbytes of characters, where all of them are zero (0):

```
$ od  -t  x1  disk0.img      #od stands for "octal dump"
                             # -t x1 means to print each character as an hexadecimal number

0000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      # the first 16 bytes are zero(0)
*                                                            # the next lines are all equal to the first
one
400000000                          # line after the end-of-file would start at this offset (in octal)
```

The offset displayed by "od" it the left hand side is printed in octal for historical reasons…

```
      # execute "bc" (basic calculator) taking as input the subsequent lines until +++

$ bc <<+++
obase=8         # setting output base to octal (8)
64*1024*1024
+++

 400000000     # result of 64*1024*1024 in octal
```

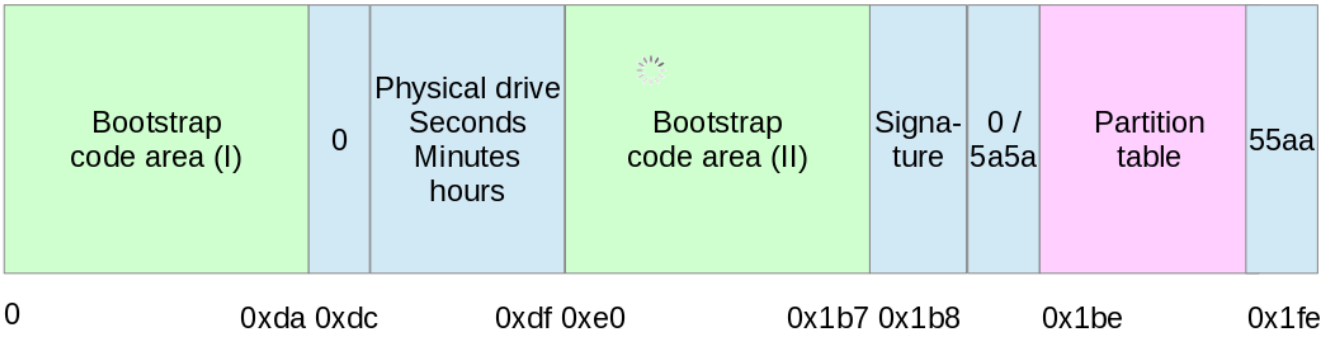
This is also a useful trick that can be used in shell scripts to execute commands with predefined input strings represented between the starting key "<<end-of-string" and the ending string "end-of-string".

Let's try to boot from the newly created disk:

$ qemu-system-x86_64 -machine q35 -drive file=disk0.img,format=raw

You should get the same result as the disk is not bootable. Let's remind that it contains only zeros (0).

To make a bootable disk, we need to write some startup code on its boot sector. The boot sector of the disk is the first 512-character block. It is also referenced as Master Boot Record (MBR). It has the following format. It has been a de-facto standard since long time ago, with various upgrades and patches:



As it can be observed, there are two areas labeled as "Bootstrap code area". They contain the first executable code that your machine is going to execute when the BIOS reads this boot sector, and executes it starting on the code at offset zero (0), the Bootstrap code area (I).

In order to comply with the conditions to become a boot sector, the last two bytes of the sector (bytes 510 and 511) should be 0x55 and 0xaa (or, in binary: 01010101 and 10101010).

We have developed a tool to write such a boot sector with a very simple program to display characters in the text screen of the machine. Get the following files from the CASO website (http://docencia.ac.upc.edu/FIB/grau/CASO/lab0/):

Makefile

bootcode.s

genboot.c

Look at the **bootcode.s** and **genboot.c** source files, which describe themselves. This is a brief description:

- bootcode.s contains a simple program in Intel 16-bit instruction format to indefinitely write the sequence of characters " !"#$%&...0123...ABCD…abcd..." onto the terminal, using standard PC BIOS services. When assembled with as86, it generates a listing of the instructions with the binary opcodes on which they are translated.
- genboot.c writes the binary codes of the program obtained from the assembly of bootcode.s to the 512 first characters of the disk/file provided as argument. DO NOT USE IT WITH YOUR OWN DISK AS TARGET, AS YOU WILL LOOSE THE FULL CONTENTS OF THE DISK...

Compile the files using make:

$ make

Execute genboot on the virtual disk:

```
$ ./genboot disk0.img
```

Now, when booting, you should see the characters !"#$%&/… written into the terminal, approximately one per second:

```
$ qemu-system-x86_64 -machine q35 -drive file=disk0.img,format=raw
```

It would be good to do the same experiment on an actual disk, but as this would be dangerous because of the loose of data, we are going to use a RAMDISK (e.g. /dev/ram0). Ram-disks are initially configured at Linux boot, and they appear in /dev[1]:

```
$ ls -l /dev/ram*

…
brw-rw---- 1 root disk 1,  7 Sep  4 14:52 /dev/ram7
brw-rw---- 1 root disk 1,  6 Sep  4 14:52 /dev/ram6
brw-rw---- 1 root disk 1,  5 Sep  4 14:52 /dev/ram5
brw-rw---- 1 root disk 1,  9 Sep  4 14:52 /dev/ram9
brw-rw---- 1 root disk 1,  0 Sep  4 17:50 /dev/ram0
```

In order to use them as a regular user (non-root), we are going to grant permissions ourselves:

```
$ chmod go+rw /dev/ram0 ;    ls -l /dev/ram0
brw-rw-rw- 1 root disk 1,  0 Sep  4 17:50 /dev/ram0
```

Then we can use genboot to set the boot sector of the ram-disk:

```
$ ./genboot /dev/ram0
Apparently trying to write onto an actual DISK!!!
ALL DATA ON  WILL BE LOST!!! (confirm: yes/NO)

yes                                    # type yes here (if using /dev/ram0 only!!)
```

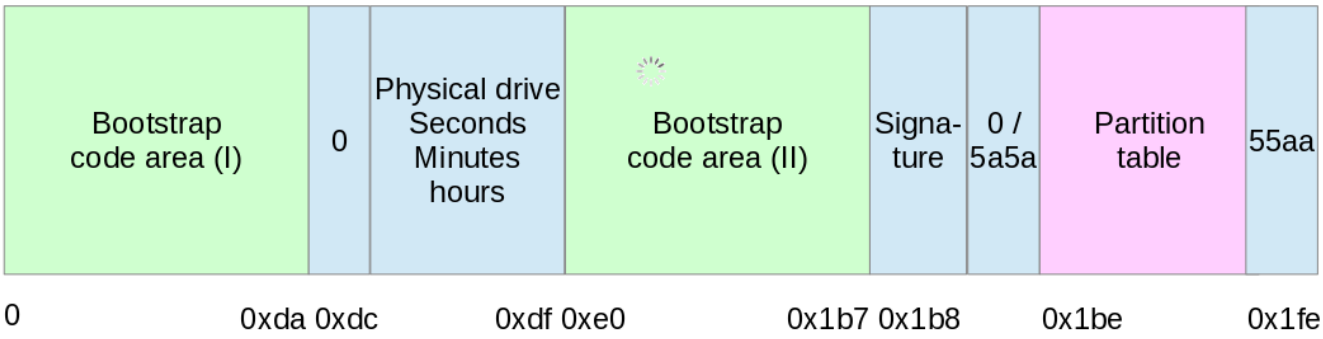
and execute the QEMU emulator on this disk:

```
$ qemu-system-x86_64 -machine q35 -drive file=/dev/ram0,format=raw
```

It should boot from the disk and execute the same simple program printing the characters…

[1] If ram* files do not exist, create them with these command lines:
$ mknod -m 660 /dev/ram b 1 1
$ chown root.disk /dev/ram
$ dd if=/dev/zero of=/dev/ram bs=1k count=4k

More on the PC boot sector



If the bootstrap code is longer than 0xda (decimal 218) bytes, it can jump onto the Bootstrap code area (II) and continue.

The bootstrap code is usually used to load the OS boot loader (SYSLINUX, LILO, GRUB…).

The boot loader

Let's use the SYSLINUX boot loader on our virtual file. The following command writes the boot loader onto the disk:

```
$ syslinux  --install disk0.img
```

```
syslinux: invalid media signature (not an FAT/NTFS volume?)
```

But it returns an error, why? Because our disk has neither FAT nor NTFS filesystem on it. Let's create a FAT filesystem on our virtual disk:

```
$ /sbin/mkfs.vfat  disk0.img
```

```
mkfs.fat 3.0.28 (2015-05-16)
```

This command has created a FAT filesystem on our disk. The "file" command can be used to confirm the creation of the file system:

```
$ file disk0.img
```

```
disk0.img: DOS/MBR boot sector, code offset 0x3c+2, OEM-ID "mkfs.fat", sectors/cluster 4, root
entries 512, Media descriptor 0xf8, sectors/FAT 128, sectors/track 32, heads 64, sectors 131072
(volumes > 32 MB) , serial number 0x24a6ae5c, unlabeled, FAT (16 bit)
```

Now, we can re-issue the syslinux command onto the virtual disk:

```
$ syslinux –install disk0.img
```

and boot syslinux:

```
$ qemu-system-x86_64 -machine q35 -drive file=disk0.img,format=raw
```

when the machine boots, in the display you should see a boot: prompt after the text:

```
SYSLINUX 4.07 EDD 2013-07-25 Copyright © 1994-2013 H. Peter Anvin et al
ERROR: No configuration file found
```

No DEFAULT or UI configuration directive found!

boot:

As we have no installation of a linux kernel on this disk, we will not be able to proceed further with the booting :(…

We cannot do an equivalent experiment with LILO or GRUB as these tools check that the boot disk is actually bootable in the current real machine, which is not the case for the file disk0.img or the /dev/ram0 device :(

To see SYSLINUX in action, you can boot our QEMU machine with the image provided "slackware14.2-boot-disk.img". It contains an install boot disk of the Slackware Linux distribution. When you get the boot prompt, you can type the TAB key, to display the list of kernels available to boot:

boot: <TAB>

huge.s kms.s speakup.s memtest

But where these OS boot loaders are located? They are usually on a "hidden" area between the bootsector and the starting of the first partition of the disk.

You can try to boot the "memtest" utility to check the memory used by QEMU :)

Disk partitions

The boot sector also contains the Partition Table (see previous figure), at offset 0x1be, and till the end of the sector. This table traditionally has been able to store 4 partitions. In Linux/UNIX, partitions on a disk are managed by utilities like fdisk, cfdisk…. As it will be observed, those tools need to modify the partition table WITHOUT changing any of the rest of the boot sector, specifically, they cannot modify the bootstrap code. Let's create a partition on the virtual disk and check that the machine still boots our simple program.

```
$ ./genboot disk0.img
$ /sbin/fdisk disk0.img
```

Create a partition with **n** (new), **p** (primary), **1** (partition number), and accept the default values for starting/ending sectors (2048 and 131071).

Issue **t** (type) and change the type of the partition to W95 FAT32 (type **b**). Issue **a** to make it bootable (active). As a result you should obtain (use **p** to print the partition table):

```
Device          Boot   Start      End    Sectors     Size   Id    Type
disk0.img1      *       2048   131071   129024     63M    b     W95 FAT32
```

To finish, write the partition table back to disk: issue **w**

```
Command (m for help): w
The partition table has been altered.
```

Syncing disks.

Fdisk informs you that the partition table has been altered, and the kernel has been informed of the change.

Now booting should still work properly:

```
$ qemu-system-x86_64 -machine q35 -drive file=disk0.img,format=raw
```

Do the same procedure to create a partition on the Ram-disk /dev/ram0. You will need to use:

```
$ sudo /sbin/fdisk /dev/ram0
```

This is needed because the fdisk will need to inform the kernel about the change in the partitions of /dev/ram0, so that /dev/ram0p1 gets created. Providing such information requires fdisk to run as supervisor (root):

After creating the partition, check that the output of the two following commands is the same as the one provided here:

```
$ ls -ltr /dev/ram0*
brw-rw---- 1 root disk 259, 0 Sep  4 18:27 /dev/ram0p1
brw-rw---- 1 root disk   1, 0 Sep  4 18:27 /dev/ram0          # The disk has been protected again!
```

```
$ sudo /sbin/fdisk  -l  /dev/ram0
passwd:
Disk /dev/ram0: 16 MiB, 16777216 bytes, 32768 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disklabel type: dos
Disk identifier: 0x00000000

Device      Boot Start   End Sectors Size Id Type
/dev/ram0p1 *     2048 32767   30720  15M  b W95 FAT32
```

Linux has created automatically the special file /dev/ram0p1 to identify the new partition.

Let’s unprotect the disk special files, in order to be able to access the disk and the partition as a regular user:

```
$ sudo chmod go+rw   /dev/ram0  /dev/ram0p1
```

Now, QEMU on the ram0 disk should work properly, displaying the characters:

```
$ qemu-system-x86_64 -machine q35 -drive file=/dev/ram0,format=raw
```

Now, let’s create a VFAT filesystem on /dev/ram0p1:

```
$ /sbin/mkfs.vfat /dev/ram0p1

mkfs.fat 3.0.28 (2015-05-16)
```

unable to get drive geometry, using default 255/63

And let’s mount the new partition onto the /mnt/point directory to see that it is correct:

$ sudo mkdir /mnt/point

$ sudo mount /dev/ram0p1 /mnt/point

$ df /mnt/point

Filesystem 1K-blocks Used Available Use% Mounted on

/dev/ram0p1 15310 0 15310 0% /mnt/point

It has been correctly mounted.

Unmount the partition:

$ sudo umount /mnt/point

Verify that the simple program still works on the bootsector of the Ram-disk:

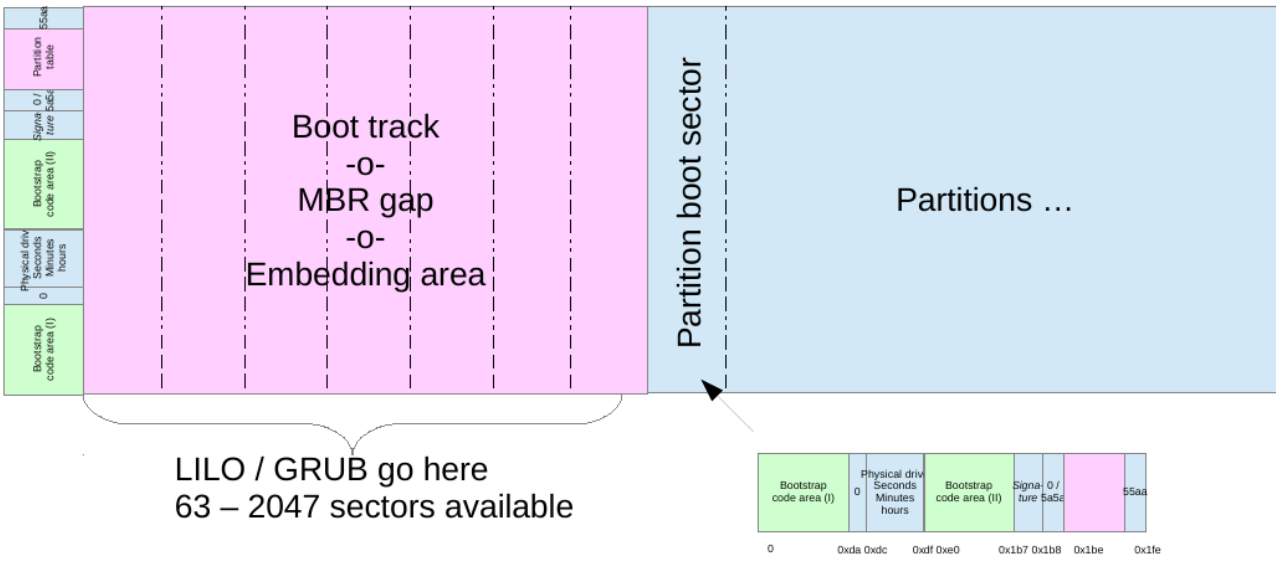
$ qemu-system-x86_64 -machine q35 -drive file=/dev/ram0,format=raw

The SYSLINUX, LILO, GRUB, boot loaders live in the hidden area between the boot sector, and the start of the first partition, in our disk:

$ /sbin/fdisk -l /dev/ram0

Device	Boot	Start	End	Sectors	Size	Id	Type
/dev/ram0p1	*	2048	32767	30720	15M	b	W95 FAT32

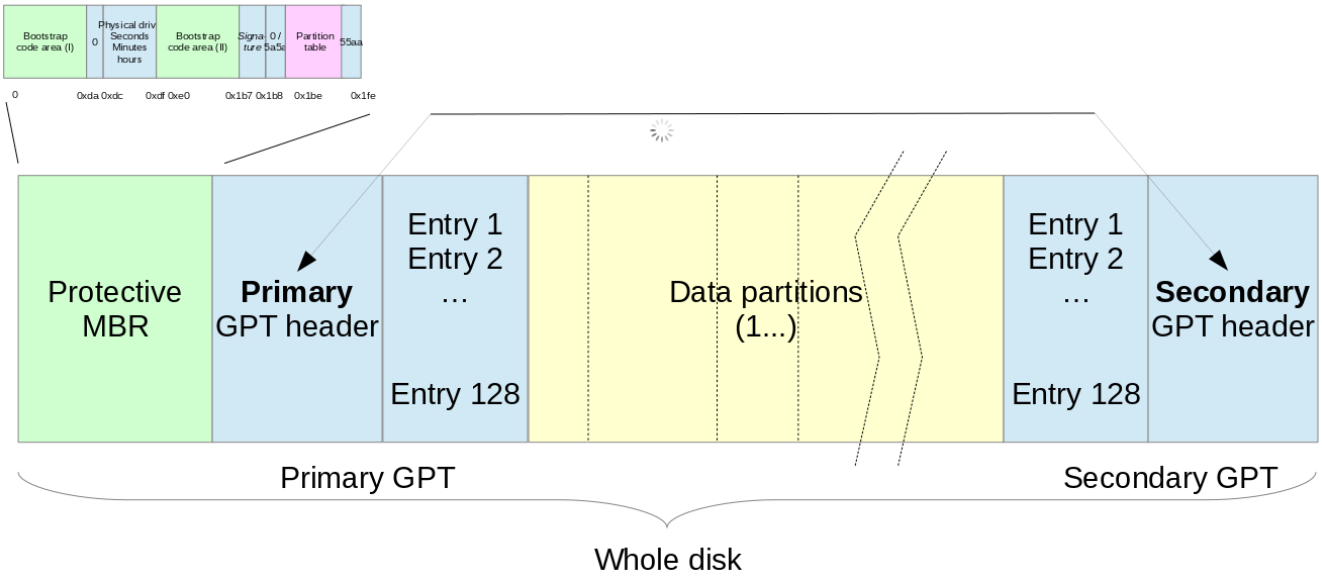
In this example, there is some “empty space” between sector 1 and Start (2048). This space is not protected in any way, and the starting point of the first partition can be wrongly selected by the user when creating the partitions. Traditionally it was 63 sectors, and it was later increased to 2048, which is the current value. Graphically, the next figure represents the location of the Boot Loader (see Embedding area).



http://www.gnu.org/software/grub/manual/grub.html#BIOS-installation

The Unified Extensible Firmware Interface (http://uefi.org)

UEFI (Unified Extensible Firmware Interface) came to solve the issue about where to store the boot loader in our system. UEFI defines a new schema for the partition table, the GPT (GUID Partition Table, or Global Unified Identifiers Partition Table). The structure of a UEFI disk is as follows:



In the usual case, there is a protection for the structure of the full disk, consisting on the Protective MBR. This is a special format of the boot sector, that includes a single partition from sector 1 to the end of the disk. This is an example of the output of fdisk on the protective MBR:

```
$ fdisk  -l /dev/sda

WARNING: GPT (GUID PartitionTable) detected on '/dev/sda'!

Disk /dev/sda: 500.1 GB, 500107862016 bytes
255 heads, 63 sectors/track, 60801 cyls, total 976773168 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disk identifier: 0x4916a240

Device     Boot      Start         End      Blocks   Id  System
/dev/sda1               1   976773167   488386583+  ee  GPT
Partition 1 does not start on physical sector boundary.
```

The second sector of the disk is the Primary GPT header, which is mirrored in the very last sector of the disk (Secondary GPT header). The header (and its copy) describe the GPT entries. Each entry describes a partition of the disk. Observe that the secondary entries are listed on the sectors before the Secondary header. Each GPT entry is 128-bytes long, so there are 4 GPT entries per sector. GPT header and 128 GPT entries get up to 16896 bytes at the beginning of the disk.

In order to install a boot loader into the UEFI system, there is a special partition type (EFI System Partition). This partition should be formatted using the FAT filesystem, and it will contain the boot

loader (in a BOOT directory), and the different OS boot loaders, usually with a directory dedicated to each one.

UEFI-based systems are a little more complicated than the traditional legacy boot systems. For this reason, we have prepared a pair of pre-compiled images, with the UEFI boot, and a basic Linux installation. Additionally, we will practice with the ARM64 architecture, using a UEFI boot.

You can download these images from:

```
$ wget http://docencia.ac.upc.edu/FIB/grau/CASO/lab0/QEMU_EFI.fd
```

```
$ wget http://docencia.ac.upc.edu/FIB/grau/CASO/lab0/vexpress64-openembedded_minimal-armv8-gcc-4.9_20140823-686.tar.xz
```

Let's execute first the ARM64 machine with the UEFI boot:

```
$ qemu-system-aarch64  -m 1024  -cpu cortex-a57  -M virt  \
        -bios  QEMU_EFI.fd  -serial stdio
```

Booting takes a pair of minutes…

After booting, you will get a command Shell>. Type "help" to see the available commands.

Now let's create a new virtual disk to use the EFI boot loader to boot from a GPT disk on the ARM64 architecture.

In order to do so, we need to create the 3 areas that we know about: boot-area, where the GPT will reside, the EFI partition, and the Linux partition. Let's follow these steps:

create boot area of disk, assuming the first partition will start in sector 2048

```
$ dd if=/dev/zero of=boot-area.img bs=512 count=2048   # (corrected v2017.2q.2)
```

create first partition for EFI, of approx. size 100MB, or 200*1024 = 204800 sectors, and build a VFAT filesystem on it:

```
$ dd if=/dev/zero of=efi-partition.img bs=512 count=$((200*1024))
```

```
$ /sbin/mkfs.vfat  efi-partition.img
```

create second partition, of approx. size 250MB, or 500*1024 = 512000 sectors

```
$ dd if=/dev/zero of=linux-partition.img bs=512 count=$((500*1024))
```

```
$ /sbin/mkfs.ext4  linux-partition.img
```

Populate EFI partition with the kernel Image and startup EFI script

```
$ sudo mount -o loop efi-partition.img /mnt/point
```

```
# extract the kernel Image from the vexpress filesystem
```

```
$ unxz < vexpress64-openembedded_minimal-armv8-gcc-4.9_20140823-686.tar.xz | \
        tar xf - ./boot/Image-3.16.0-1-linaro-vexpress64
```

```
$ sudo cp ./boot/Image-3.16.0-1-linaro-vexpress64  /mnt/point/Image
```

```
$ sudo mkdir -p /mnt/point/EFI/BOOT
```

```
$ echo "Image root=/dev/vda2 console=ttyAMA0,38400n8 earlycon=pl011,0x9000000" | \
        sudo dd of=/mnt/point/EFI/BOOT/startup.nsh

$ sudo umount /mnt/point


# Populate Linux partition with a minimal root filesystem for ARM64

$ sudo mount -o loop linux-partition.img /mnt/point

$ sudo tar -C /mnt/point/ -Jxf    \
    vexpress64-openembedded_minimal-armv8-gcc-4.9_20140823-686.tar.xz

$ sudo umount /mnt/point


# now we concatenate the three "partitions" onto a virtual disk file

$ cat boot-area.img efi-partition.img linux-partition.img >disk1.img


# now we make the virtual disk a little larger to accomodate the Secondary
# GPT and GPT entries (remember that those are located at the very end of the disk)
#  observe the use of conv=notrunc, in order to avoid dd to truncate the output
#     file disk1.img
#  and the use of oflag=append, in order to add the new data at the end of
#     the file

$ dd if=/dev/zero of=disk1.img conv=notrunc oflag=append bs=512 count=2048


# now we need to create the GPT into the boot area of the newly created virtual disk

$ /sbin/fdisk disk1.img

# the following are the commands for fdisk…
#       do not type the explanations written after the # symbol
#       text shown with > is an approximate output after the command is executed

Command (m for help):  g                        # create a new empty GPT table

> Created a new GPT disklabel (GUID: ED476572-2ED2-4059-A1A3-802FB99C0A52).

Command (m for help):  n                        # Create a new partition

Partition number (1-128, default 1): 1          # Select partition number 1

First sector (2048-718814, default 2048): 2048  # Use 2048, end of boot-area

Last sector, +sectors... default 718814): +204799  # Use size of part#1, 204800-1

> Created a new partition 1 of type 'Linux filesystem' and of size 100 MiB.

Command (m for help): t                     # change the type of the new partition #1
```

> Selected partition 1

Hex code (type L to list all codes): 1 # to EFI System (id 1)

> Changed type of partition 'Linux filesystem' to 'EFI System'.

Command (m for help): n # Create a new partition
Partition number (2-128, default 2): 2 # Select partition number 2
First sector (206848-720862, default 206848): 206848 # initial sector
Last sector, +sectors ... -720862, default 720862): +511999 # actual part#2 size -1

> Created a new partition 2 of type 'Linux filesystem' and of size 249 MiB.

Command (m for help): w # Commit the new GPT to disk
> The partition table has been altered.
> Syncing disks.

Let's check if the disk seems to have a valid partition table, compare this output, with the output you get from /sbin/fdisk -l disk1.img:

```
$ /sbin/fdisk -l disk1.img

Disk disk1.img: 352 MiB, 369098752 bytes, 720896 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 81CD9435-F0FA-41C8-90B5-BBA74763B7FB

Device          Start    End   Sectors  Size Type
disk1.img1       2048 206847    204800  100M EFI System
disk1.img2     206848 718847    512000  250M Linux filesystem
```

Now we have a new disk ready to boot, let's start qemu on it:

```
$ qemu-system-aarch64    -m 1024    -cpu cortex-a57    -M virt  \
    -drive if=none,file=disk1.img,id=hd0 -device virtio-blk-device,drive=hd0 \
    -bios QEMU_EFI.fd -nographic
```

Booting takes again a pair of minutes….

Booting the EFI bios will show a message like:

```
UEFI Interactive Shell v2.1
EDK II
UEFI v2.60 (EDK II, 0x00010000)
Mapping table
    FS0: Alias(s):HD1b:;BLK2:
        VenHw(837DCA9E-E874-4D82-B29A-
23FE0E23D1E2,003E000A00000000)/HD(1,GPT,62F22C00-ACB7-48A0-A43B-
4A85BD6E5D27,0x800,0x32000)
    BLK4: Alias(s):
```

```
        VenHw(F9B94AE2-8BA6-409B-9D56-B9B417F53CB3)
    BLK0: Alias(s):
        VenHw(8047DB4B-7E9C-4C0C-8EBC-DFBBAACACE8F)
    BLK1: Alias(s):
        VenHw(837DCA9E-E874-4D82-B29A-23FE0E23D1E2,003E000A00000000)
    BLK3: Alias(s):
        VenHw(837DCA9E-E874-4D82-B29A-        23FE0E23D1E2,003E000A00000000)/
HD(2,GPT,64F1AA2D-98D9-4F7B-88BE-  864FCA755AF5,0x32800,0x7D000)
    Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
```

And after 5 seconds the EFI bios should boot Linux:

```
    Shell> Image root=/dev/vda2 console=ttyAMA0,38400n8 earlycon=pl011,0x9000000
    EFI stub: Booting Linux Kernel...
[    0.000000] Linux version 3.16.0-1-linaro-vexpress64 (buildslave@x86-64-07) (gcc version 4.8.3
20140401 (prerelease) (crosstool-NG linaro-1.13.1-4.8-2014.04 - Linaro GCC 4.8-2014.04) )
#1ubuntu1~ci+140819081106 SMP PREEMPT Tue Aug 19 08:11:52 UTC 20
…….
```

And end up giving a shell:

```
…
Starting auto-serial-console: done
Stopping Bootlog daemon: bootlogd.
INIT: no more processes left in this runlevel

Last login: Sat Aug 23 14:07:57 UTC 2014 on tty1
root@genericarmv8:~#
```

Now you can use some of the typical Linux commands to observe the system:

```
$ more /proc/cpuinfo        # can you start qemu in such a way that the system has 4 cores?
$ top                       # how can you display the last cpu used by each process?
```

What to deliver for this lab session

For the delivery of this lab session, check the associated practice created in Racó, and answer the questions there. Each group can deliver his/her report once, and please, indicate clearly the names of the participants.