Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

File Systems, LUKS, InitRamFS

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

References

[1]: <Linux Kernel sources>/Documentation/filesystems

[2]: http://www.tldp.org/HOWTO/html_single/SquashFS-HOWTO

[3]: http://squashfs.sourceforge.net

[4]: tree.celinuxforum.org/CelfPubWiki/ELCEurope2008Presentations?action=AttachFile&do=get&target=squashfs-elce.pdf

[5]: http://superuser.com/questions/228657/which-linux-filesystem-works-best-with-ssd //File for SSD card

[6]: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/index.html // very good site

[7]: https://code.google.com/p/cryptsetup/

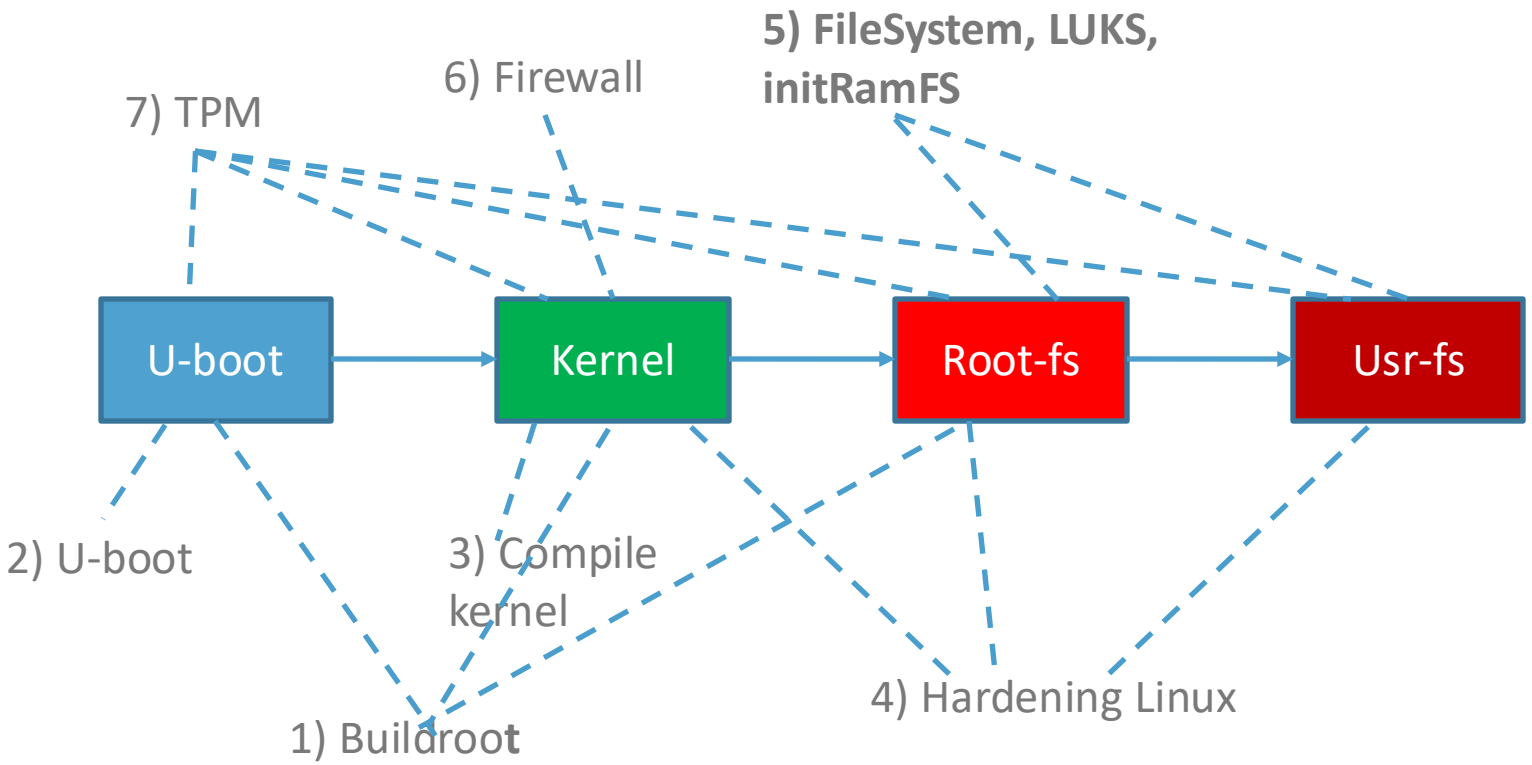
Power off embedded FS

[8|: http://stackoverflow.com/questions/14460091/embedded-file-system-and-power-off

[9]: https://elinux.org/images/0/02/Filesystem_Considerations_for_Embedded_Devices.pdf

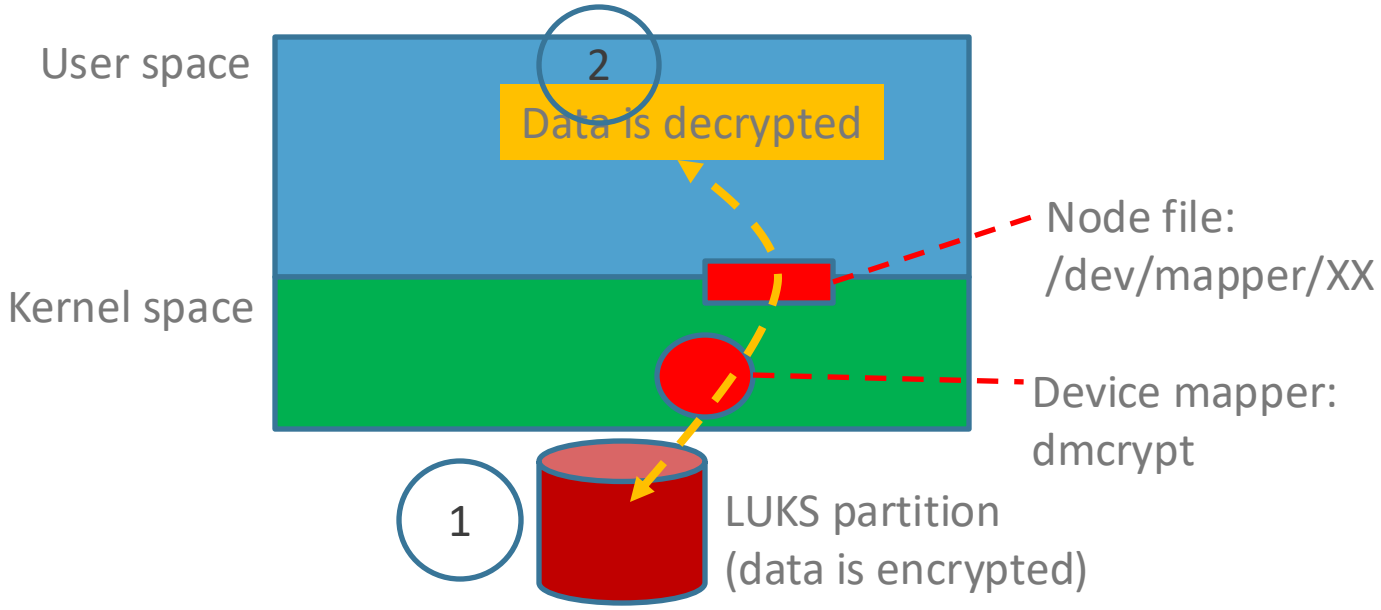
Course summary, main chapters

1) Buildroot
2) U-boot
3) Compile kernel
4) Hardening Linux
5) **FileSystem, LUKS, InitRamFS**
6) Firewall
7) TPM (Trusted Plarform Module)



LUKS principle

1. Data in the LUKS partition is encrypted
2. Data used in the user space is decrypted by dmcrypt

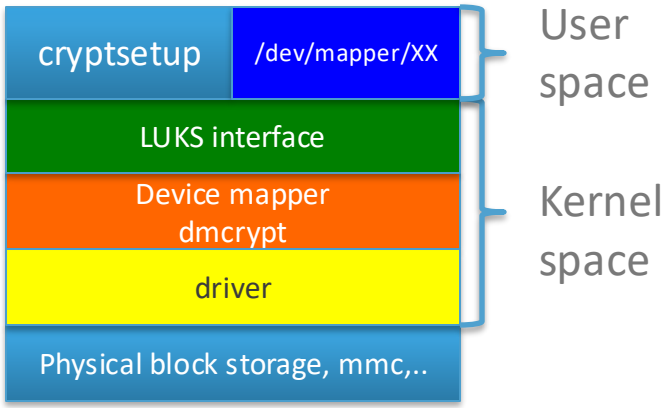


LUKS, cryptsetup, dmcrypt

- See [7]: https://code.google.com/p/cryptsetup/
- https://blog.tinned-software.net/automount-a-luks-encrypted-volume-on-system-start/

- **LUKS** (Linux Unified Key Setup) is the standard for Linux hard disk encryption.
- By providing a standard on-disk-format, it does not only facilitate compatibility among distributions but also provides secure management of multiple user passwords.
- In contrast to existing solution, LUKS stores all necessary setup information in the partition header, enabling the user to transport or migrate his data seamlessly.
- LUKS – **`dmcrypt`** crypts an entire partition

- LUKS features
  - compatibility via standardization
  - secure against attacks
  - support for multiple keys
  - effective passphrase revocation
  - free



- `cryptsetup` is a utility used to configure dmcrypt
- `cryptsetup` uses the /dev/random and /dev/urandom node file

LUKS, cryptsetup, dmcrypt

- See : [7]: https://code.google.com/p/cryptsetup/wiki/DMCrypt
- `dmcrypt` (Device-mapper) crypts target and provides transparent encryption of block devices using the kernel crypto API (kernel configuration, Cryptographic API)

- Device-mapper is included in the Linux 2.6 and 3.x kernel that provides a generic way to create virtual layers of block devices. It is required by LVM2 (Logical Volume Management).

- The user can basically specify one of the symmetric ciphers, an encryption mode, a key (of any allowed size), an iv-generation mode and then the user can create a new block device node file in /dev/mapper.

- All data written to this device will be encrypted

and

- All data read from this device will be decrypted.

LUKS, cryptsetup, NanoPi, buildroot, kernel

- In order to enable dmcrypt, it is necessary to configure the kernel:
- `cd /buildroot`

  `make linux-xconfig or make linux-menuconfig`

  Go to: device driver → Multiple Devices drivers support (RAID and LVM) → Device mapper support → Crypt target support

  Important: choose <*> not <M>

```
<*>   Device mapper support
[ ]     Device mapper debugging support
< >     Unstriped target
<*>     Crypt target support
```

- In order to use “cryptsetup”, it is required to add a new package in buildroot

  `cd /buildroot`

  `make menuconfig`

  Go to: Target Packages → Hardware handling : choose cryptsetup

```
[ ] brltty
[ ] cc-tool
[ ] cdrkit
[ ] cpuburn-arm
[*] cryptsetup
[ ] cwiid
[ ] dhadi-linux
```

LUKS, cryptsetup, NanoPi, buildroot, kernel

For crypto userspace API

```
Cryptographic API --->
   [*] User-space interface for hash algorithms
   [*] User-space interface for symmetric key cipher algorithms
```

For device mapper and dm-crypt:

```
Device Drivers --->
   [*] Multiple devices driver support (RAID and LVM) --->
         <*> Device mapper support
         <*> Crypt target support
```

For crypto algorithms :

```
Cryptographic API --->
   <*> XTS support
   <*> SHA224 and SHA256 digest algorithm
   <*> AES cipher algorithms
```

```
CONFIG_CRYPTO_USER_API=y
CONFIG_CRYPTO_USER_API_HASH=y
CONFIG_CRYPTO_USER_API_SKCIPHER=y
CONFIG_DM_CRYPT=y
CONFIG_CRYPTO_AES=y
CONFIG_CRYPTO_XTS=y
CONFIG_CRYPTO_SHA256=y
CONFIG_CRYPTO_SHA1=y
CONFIG_CRYPTO=y
CONFIG_BLK_DEV_DM=y
```

- These modules must be integrated to the kernel in order to give the possibility to use the userland API

LUKS with NanoPi

On the SD Card, create a third partition (with fdisk or parted)

Create LUKS partition 1/2

Initialize a LUKS partition, be careful all data will be lost. A passphrase generates the encryption key (--debug is optional)

On the NanoPi: `$DEVICE = /dev/mmcblk0p3` On PC: `$DEVICE = /dev/sdc3`

Creation: `dd if=/dev/zero of=file.luks bs=1024 count=400000; $DEVICE=file.luks`

Create a LUKS partition for a system with few memory (nanopi) (pbkdf2 algo)

```
# sudo cryptsetup --debug --pbkdf pbkdf2 luksFormat $DEVICE
    Be careful, type yes in UPPERCASE
```

Or create a LUKS partition for the system with enough memory

```
# sudo cryptsetup --debug luksFormat $DEVICE
```

Dump the header information of a LUKS device

```
# sudo cryptsetup luksDump $DEVICE
```

Create a mapping `/dev/mapper/usrfs1` and ask the passphrase

```
# sudo cryptsetup --debug open --type luks $DEVICE usrfs1
```

Show the node file

```
# ls /dev/mapper/
  brw-------    1 root     root      253,   0 Jan  1 15:36 usrfs1
```

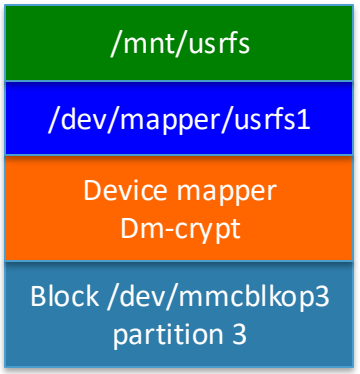
Create LUKS partition 2/2

Format the LUKS partition as ext4 partition

```
# sudo mkfs.ext4 /dev/mapper/usrfs1
```

Mount the LUKS partition to /mnt/usrfs

```
# sudo mkdir /mnt/usrfs
# sudo mount /dev/mapper/usrfs1 /mnt/usrfs
```



Use the LUKS partition

```
# ls /mnt/usrfs
# copy files to /mnt/usrfs
```

Unmount the LUKS partition

```
# sudo umount /dev/mapper/usrfs1
```

Removes the existing mapping `usrfs1` and wipes the key from kernel memory

```
# sudo cryptsetup close usrfs1
```

dmsetup: low level logical volume management

```
# dmsetup info -C
# dmsetup remove -f usrfs1
```

Use LUKS partition

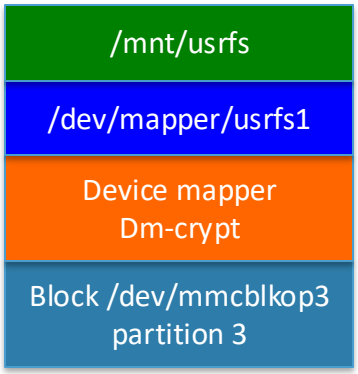
Remark: The LUKS partition is already created

Create a mapping `/dev/mapper/usrfs1` and ask the passphrase

```
# sudo cryptsetup --debug open --type luks $DEVICE usrfs1
Or
# sudo cryptsetup --debug --key-file=passphrase open --type luks $DEVICE usrfs1
```

Mount the LUKS partition to /mnt/usrfs

```
# sudo mount /dev/mapper/usrfs1 /mnt/usrfs
```



Unmount the LUKS partition

```
# umount /dev/mapper/usrfs1
```

Removes the existing mapping `usrfs1` and wipes the key from kernel memory

```
# cryptosetup close usrfs1
```

It is possible to manage a luks partition with:

```
# dmsetup info -c
# dmsetup remove -f usrfs1
```

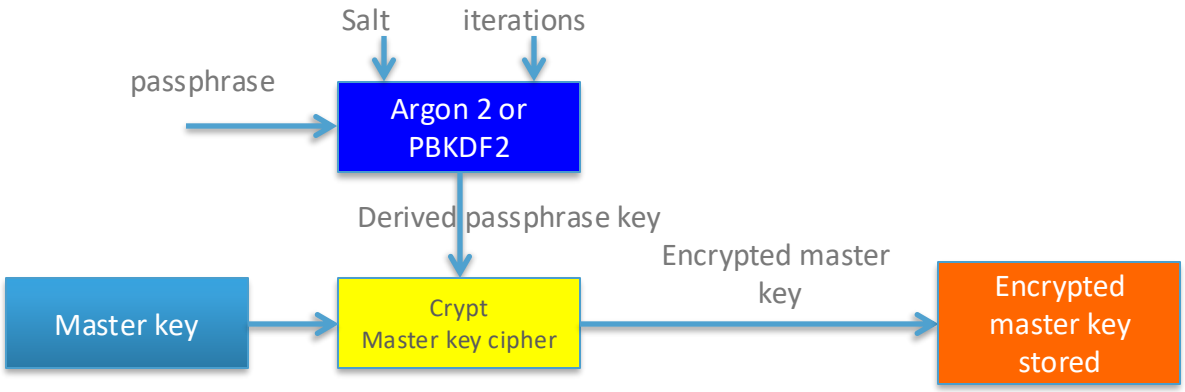
LUKS, key generation

- LUKS uses the TKS1 template in order to generate secure key.
- The key is derived from a passphrase
- LUKS supports multiple keys/passphrases
- TKS1 uses Argon2 or PBKDF2 (Password-Based Key Derivation Function 2) method in order to provide a better resistance against brute force attacks based on entropy weak user passphrase.
- TKS1 uses two level hierarchy of cryptographic keys to provide the ability to change passphrases
- See also: http://clemens.endorphin.org/cryptography

LUKS, key generation

The system initialization is straight forward:

- A master key is generated
- Passphrase, Salt, iterations and other values are inputs of the functions Argon2 or PBKDF2
- A derived passphrase key is computed by Argon2 or PBKDF2
- The master key is encrypted by the derived passphrase key.
- The encrypted master key, the iteration rate and the salt are stored



LUKS, key generation

Add a new passphrase to the LUKS partition (`$DEVICE=/dev/mmcblk0p3 (nanopi) or /dev/sdc3(PC))`

```
# cryptsetup luksAddKey $DEVICE
```

Dump the header information of a LUKS device

```
# cryptsetup luksDump $DEVICE
```

```
Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha1
Payload offset: 4096
MK bits:        256
MK digest:      6a ef 4e be 5d e5 90 80 48 fa a9 b0 21 cd cf be 9b cf 40 0e
MK salt:        d0 12 4d a2 52 80 72 fc 14 d2 f2 16 02 c5 e0 1d
                9c 59 c4 fc 4e 9f 7b e7 be f6 b3 34 aa 09 ce 9c
MK iterations:  20125
UUID:           d04071bc-d7e8-45d7-a950-a46c4e90d122
Key Slot 0: ENABLED
        Iterations:             80000
        Salt:                   bb e5 b8 ef 1d b4 03 5a f7 e5 1e 8e e0 70 d4 48
                                31 0c 31 52 b0 a4 2f 55 55 be 83 f2 ad c5 97 32
        Key material offset:    8
        AF stripes:             4000
Key Slot 1: ENABLED
        Iterations:             81011
        Salt:                   fd 21 0f d6 39 c4 1c 79 b5 2b ec 4d 43 dd 66 e0
                                ff 41 76 2f 39 59 e2 00 9a 2c 42 6b 22 10 16 47
        Key material offset:    264
        AF stripes:             4000
```

Crypt the master key

Passphrase 1

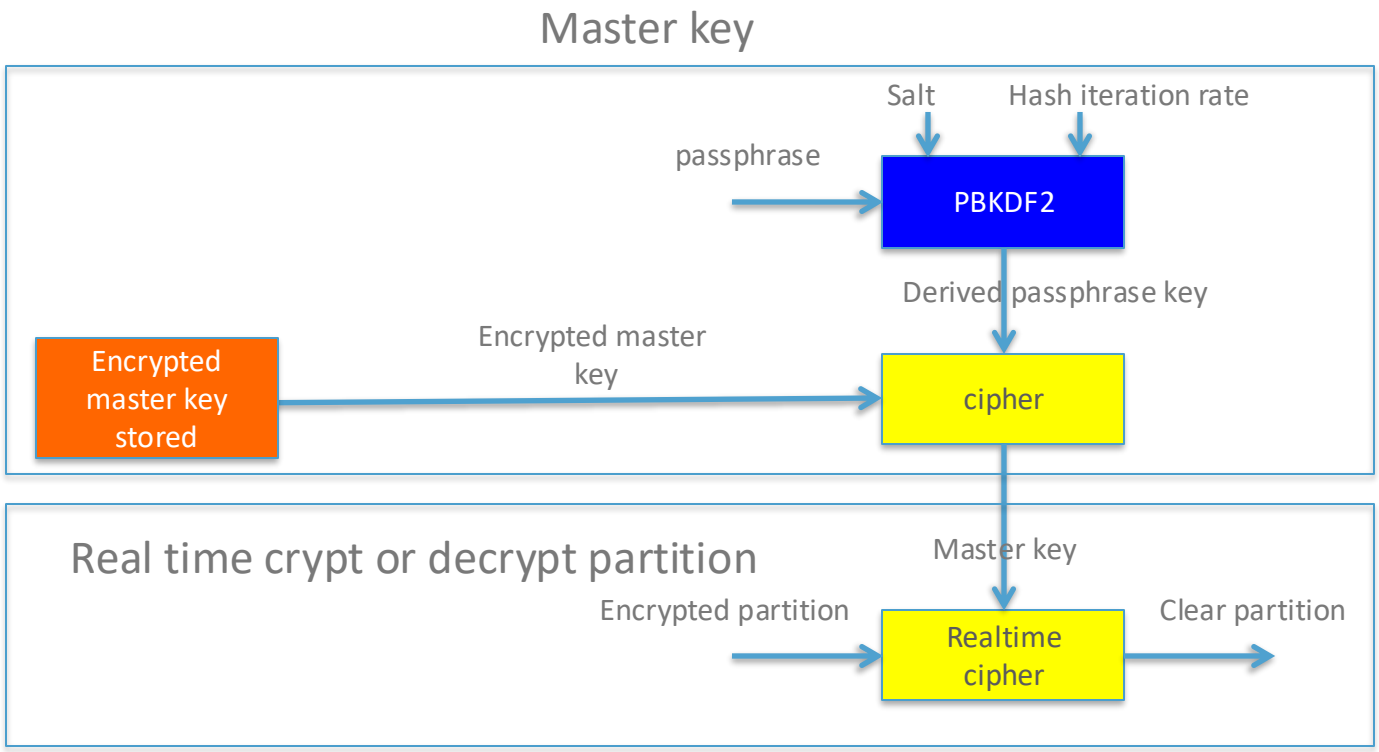
Passphrase 2

LUKS, key generation

Dump the encrypted master key of a LUKS device

```
# cryptsetup luksDump -dump-master-key $DEVICE
    LUKS header information for /dev/mmcblk0p3
    Cipher name:    aes
    Cipher mode:    xts-plain64
    Payload offset: 4096
    UUID:           d04071bc-d7e8-45d7-a950-a46c4e90d122
    MK bits:        256
    MK dump:        1e e2 d8 02 12 1a ce a4 74 66 20 3e 00 21 a7 c1
                    1b 92 88 76 d7 c1 d8 fd 1b 6e 42 fd ac 91 20 52
```

LUKS, crypt partition



LUKS, cryptsetup, dmcrypt

- Check **/proc/crypto** which contains supported ciphers and modes (but note it contains only currently loaded crypto API modules).

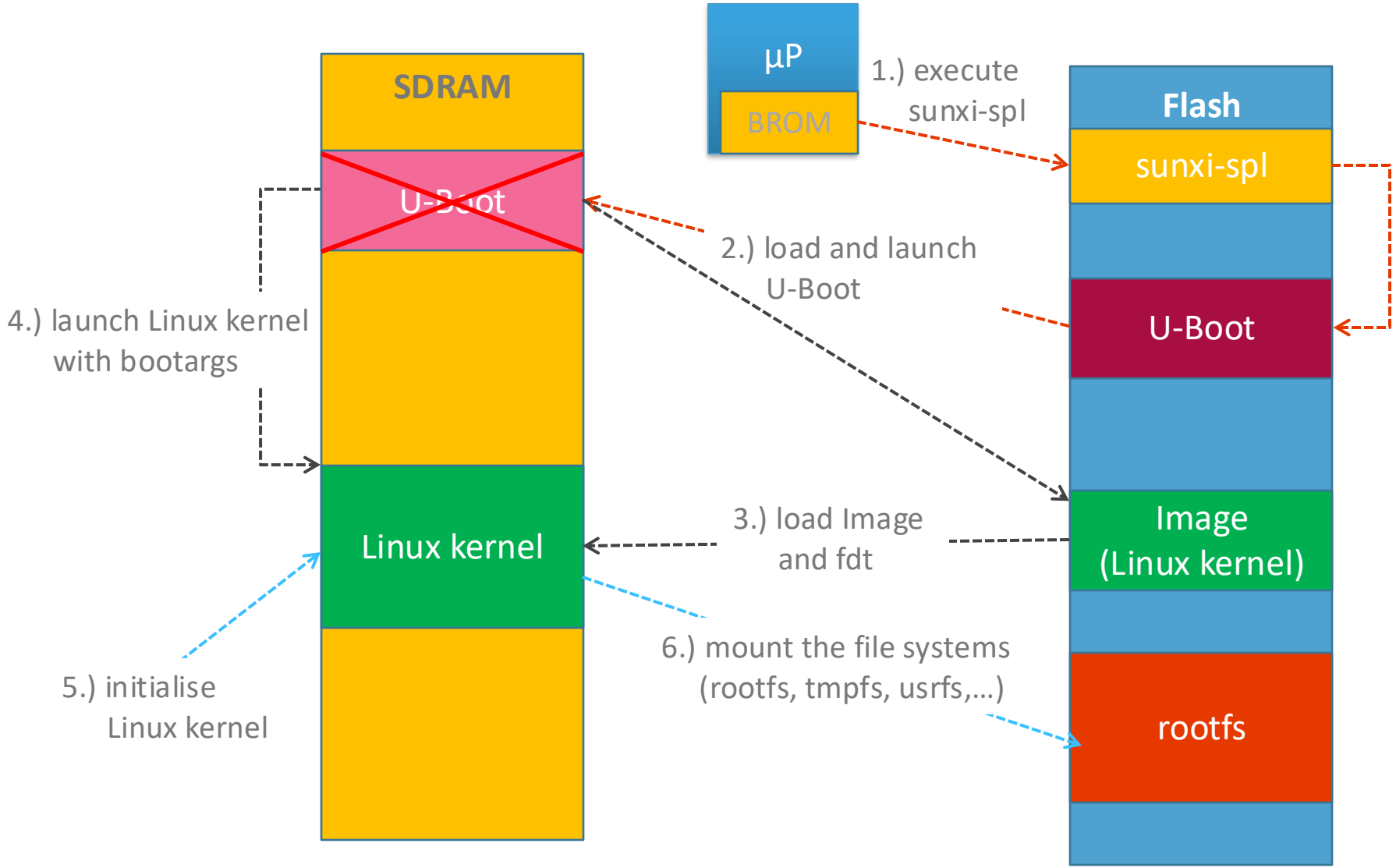
```
Bash# cat /proc/crypto
   …
   name         : aes
   driver       : aes-generic
   module       : kernel
   priority     : 100
   refcnt       : 1
   selftest     : passed
   type         : cipher
   blocksize    : 16
   min keysize  : 16
   max keysize  : 32
   …
```

Boot sequence, without initramfs 1/4

- **The NanoPi NEO Plus2 startup process consists of 6 phases:**

- When the µP is powered up, the code stored in its BROM will load the "sunxi-spl" firmware stored in sector 16 of the SD card/eMMC into its 32KiB of internal SRAM and execute it.
- The "sunxi-spl" (Secondary Program Loader) firmware initializes the lower layers of the µP, then loads U-Boot into the µP's RAM before launching it.
- U-Boot will perform the necessary hardware initializations (clocks, controllers, etc.) before loading the uncompressed Linux kernel image into RAM, the "Image" file, and the FDT (flattened device tree) configuration file.
- U-Boot will launch the Linux kernel by passing it the boot arguments (bootargs).
- The Linux kernel will initialize itself based on the bootargs and configuration elements contained in the FDT file (sun50i-h5-nanopi-neo-plus2.dtb).
- The Linux kernel will attach the file systems (rootfs, usrfs, etc.) and continue running.

Boot sequence, without initramfs 2/4

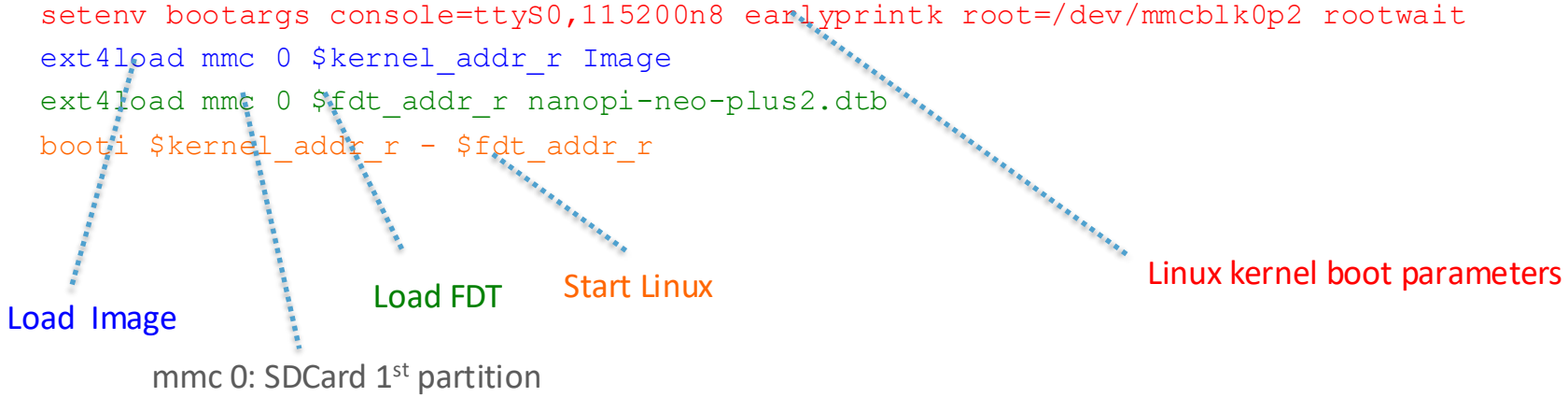


U-boot-Linux boot-without initramfs 3/4

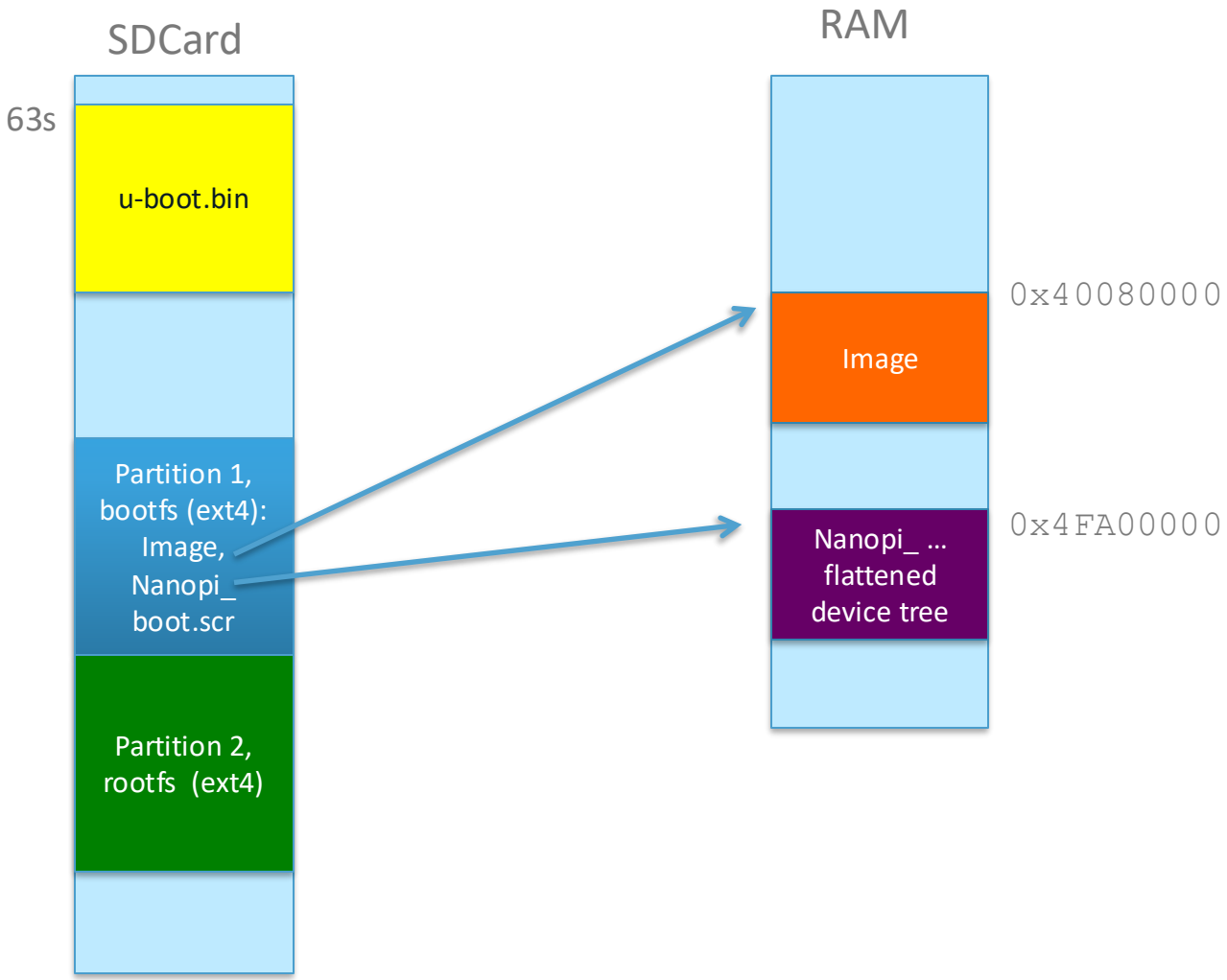
See u-boot course

Show boot.cmd file： `cat /buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd`

```
setenv bootargs console=ttyS0,115200n8 earlyprintk root=/dev/mmcblk0p2 rootwait
ext4load mmc 0 $kernel_addr_r Image
ext4load mmc 0 $fdt_addr_r nanopi-neo-plus2.dtb
booti $kernel_addr_r - $fdt_addr_r
```



U-boot-Linux boot-without initramfs 4/4



Initramfs 1/6

https://wiki.gentoo.org/wiki/Custom_Initramfs : good reference

- initramfs is a root filesystem that is loaded at an early stage of the boot process.
- It is the successor of initrd.
- It provides early userspace commands which lets the system do things that the kernel cannot easily do by itself during the boot process.
- **Using initramfs is optional**.

- Boot without initramfs:
  - By default, the kernel initializes hardware using built-in drivers, mounts the specified root partition, loads the rootfs and starts the init scritps
  - Init scripts can load additional modules and starts services until it eventually allows users to log in. This is a good default behavior and sufficient for many users.
- An initramfs is generally used for advanced requirements; for users who need to perform certain tasks as early as possible, even before the rootfs is mounted.

Boot sequence, with initramfs 2/6

The points:

1) execute sunxi-spl

2) load and launch U-Boot

3) load Image and fdt

are the same with or without initramfs

4) u-boot copies the initramfs to RAM

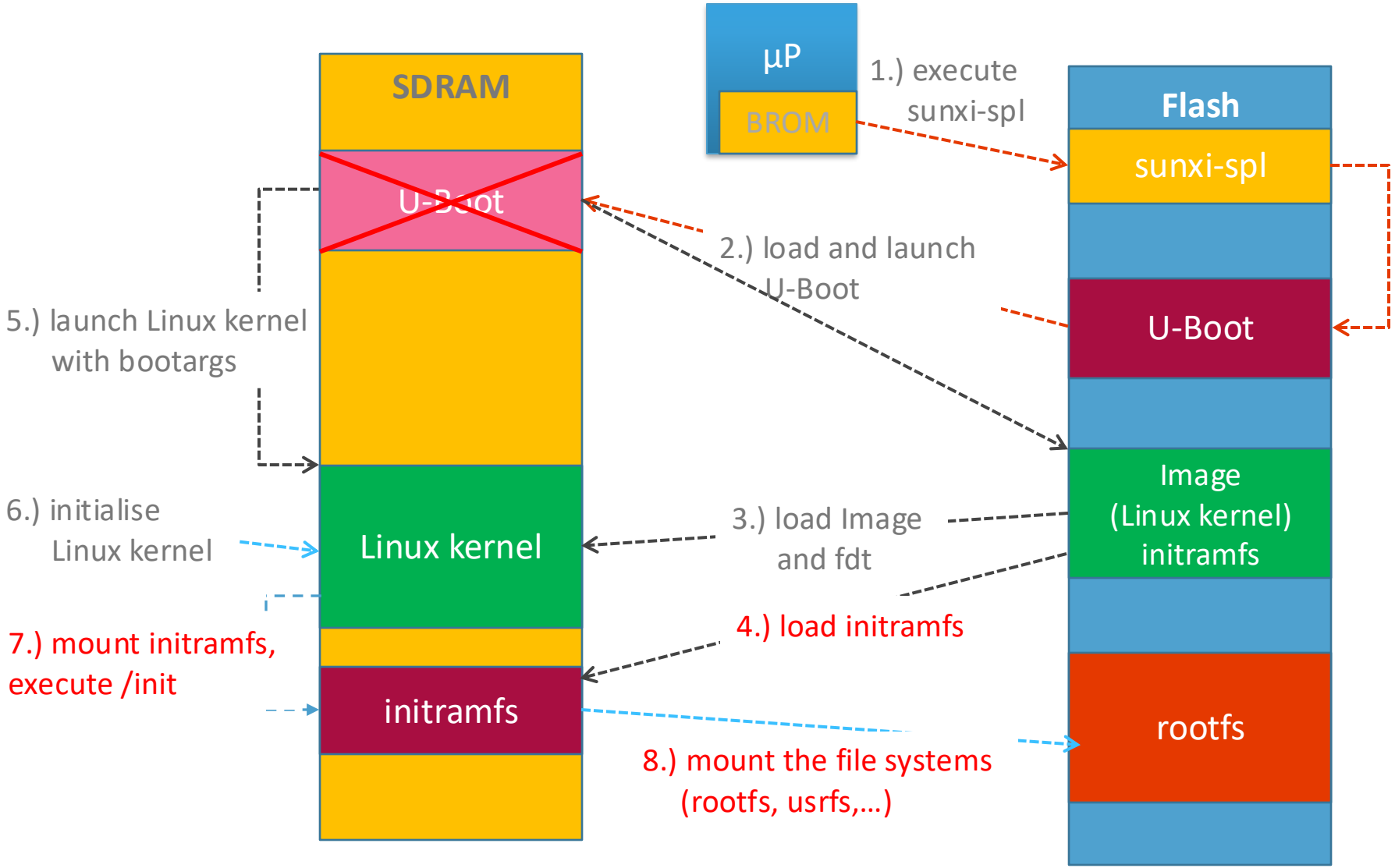
5) u-boot starts the Linux kernel

6) the Linux kernel initializes

7) the Linux kernel mounts the initramfs and executes the `/init` script, which may contain various commands, for example: decrypting the rootfs that is in a LUKS partition.

8) the `/init` script activates the rootfs.

Boot sequence, with initramfs 3/6



Boot-with initramfs 4/6

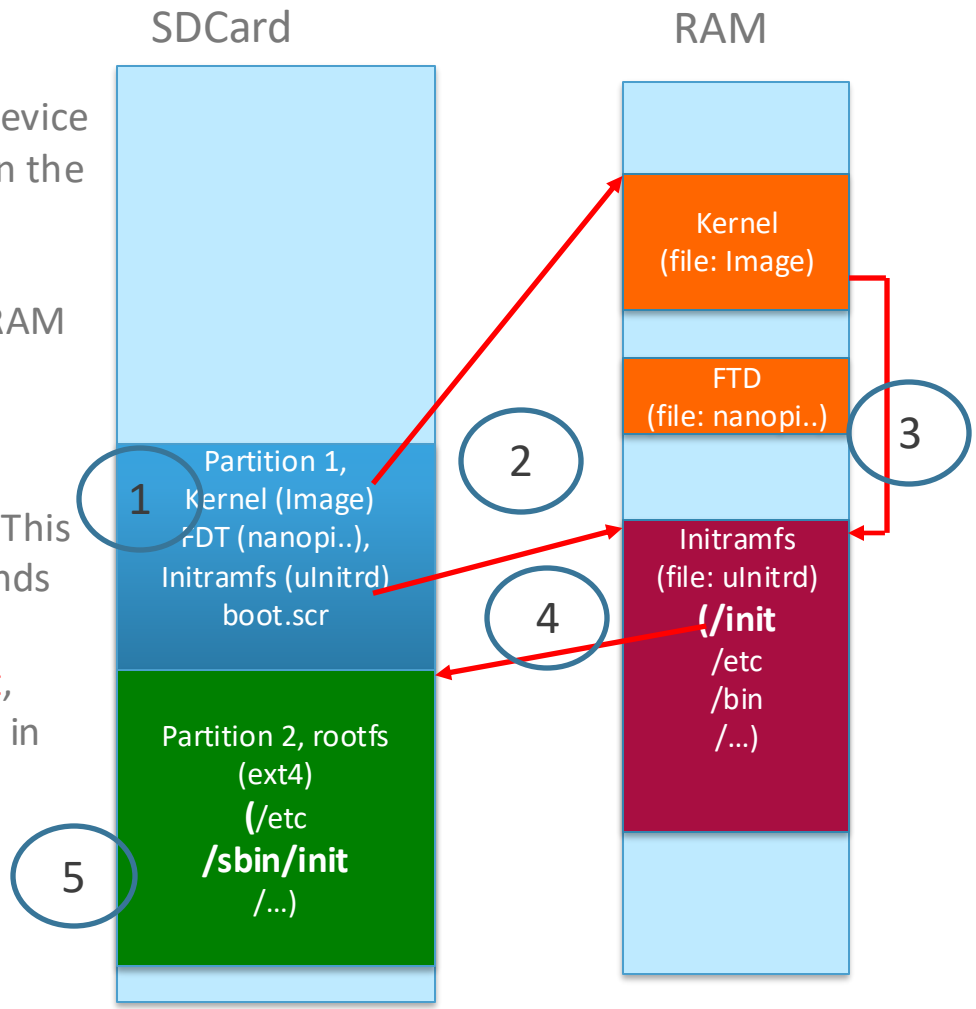
1) Kernel (Image), initramfs (uInitrd), flattened device tree (Sun50i…) and boot.scr files are located in the partition 1 of the SDCard

2) Kernel, initramfs, Sun50i.. are copied to the RAM

3) Kernel mounts initramfs (uInitrd file)

4) Kernel executes init script stored in initramfs. This init script can execute early different commands

5) Init script executes the command switch_root, which switches to the standard rootfs located in the partition 2 and executes the /sbin/init command



U-boot-Linux boot-with initramfs 5/6

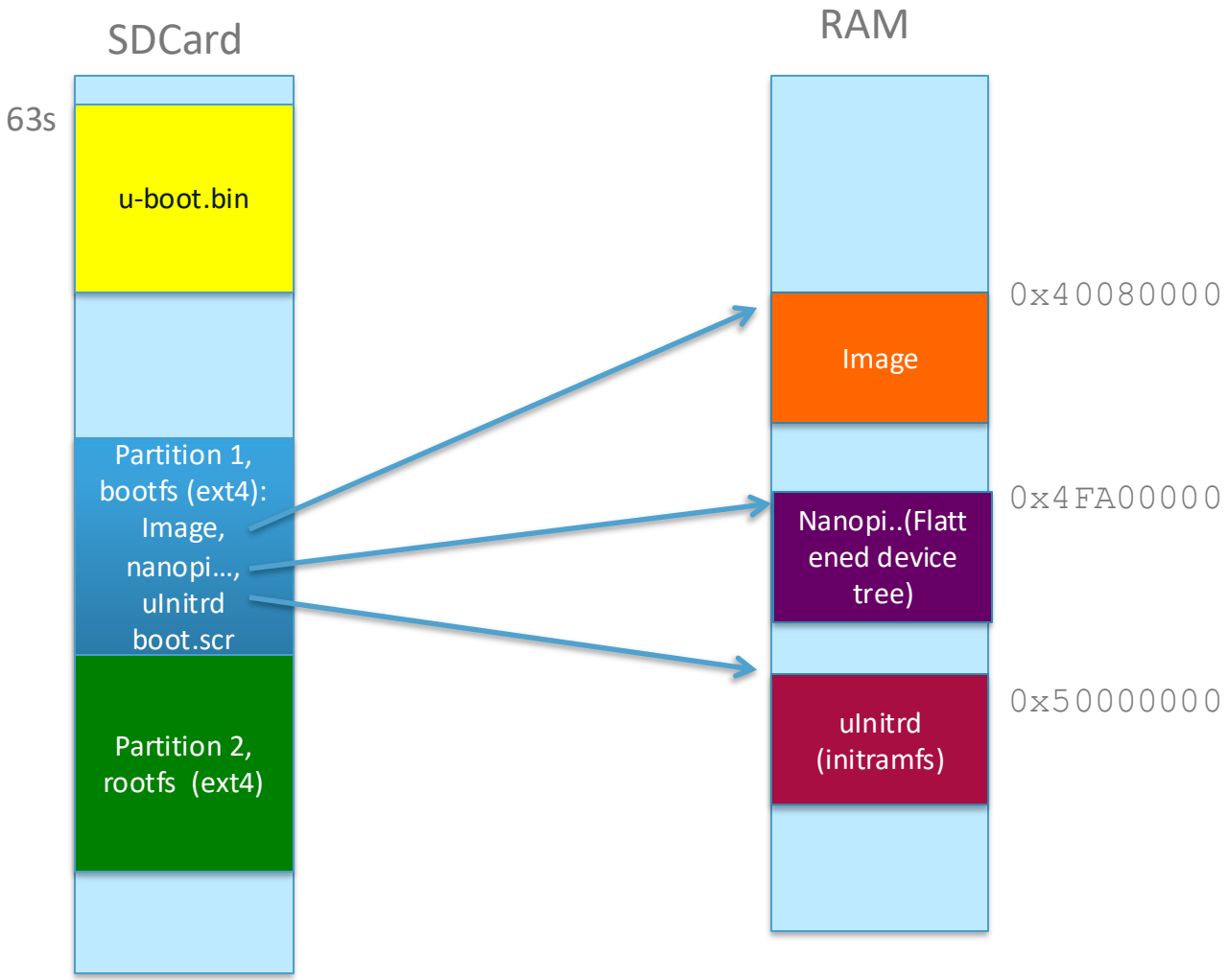
Show boot.cmd file：

```
cat /buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd
     setenv bootargs console=ttyS0,115200n8 earlyprintk root=/dev/mmcblk0p2 rootwait
     ext4load mmc 0 $kernel_addr_r Image
     ext4load mmc 0 $fdt_addr_r nanopi-neo-plus2.dtb
     ext4load mmc 0 0x50000000 uInitrd        // Load initramfs: uInitrd is the initramfs

     booti $kernel_addr_r 0x50000000 $fdt_addr_r
```

initramfs address

U-boot-Linux boot-with initramfs 6/6

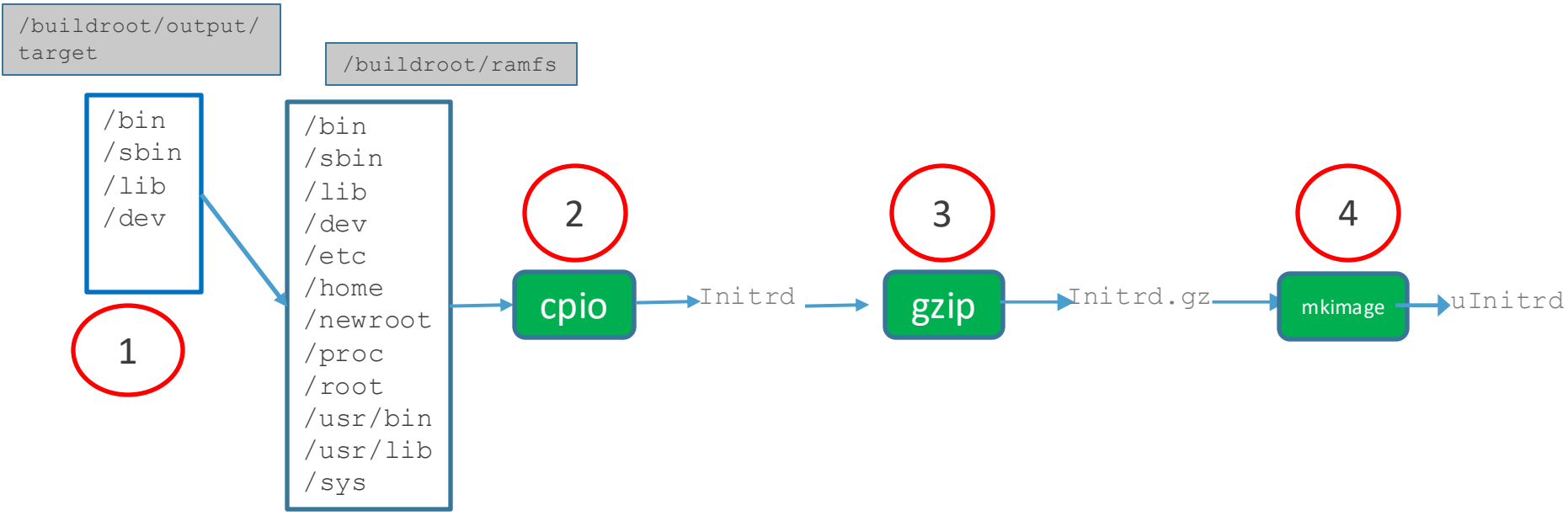


How to build Initramfs

On PC, NanoPi rootfs is in this directory: `/buildroot/output/target/`

Principle to build an initramfs:

1. to copy the right files into a directory (`/buildroot/ramfs`),
2. to copy these files in a cpio archive file,
3. to compress this file
4. To add the uboot header

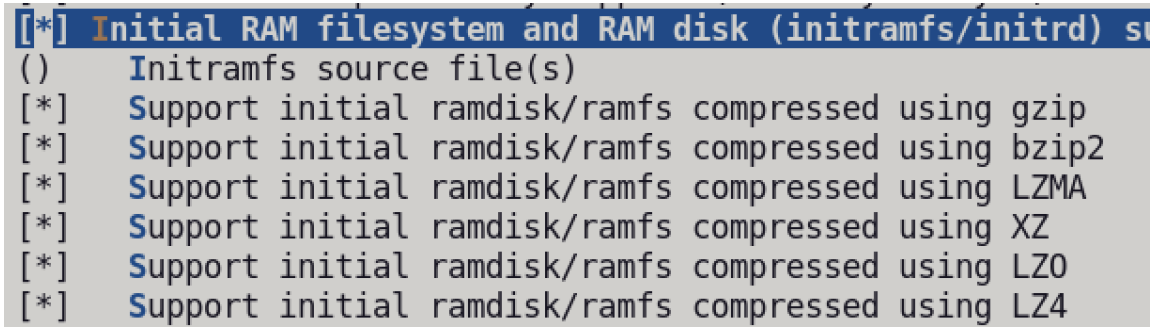


Initramfs: kernel configuration

```
cd /buildroot
make linux-menuconfig
```

Kernel configuration:

```
CONFIG_BLK_DEV_INITRD=y
   General setup ---> [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
```



(Generally, this option is not used)

Add the initramfs into the kernel:

```
CONFIG_INITRAMFS_SOURCE="/usr/src/initramfs”
   General setup ---> [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
```

Automount a devtmpfs and initiate the /dev:

```
Device Drivers → Generic Drivers options →
                        [*] Maintain a devtmpfs filesystem to mount at /dev
                        [*] Automount a devtmpfs at /dev, after the kernel mounted the rootfs
```

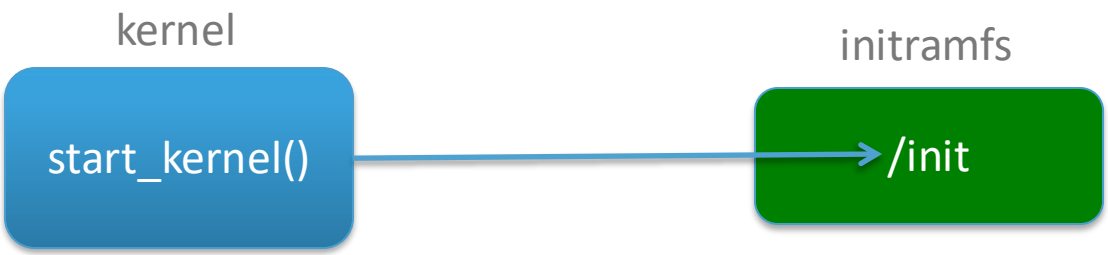
Initramfs manual generation

- initramfs is a cpio archive file.
- Remark: It can be generated automatically with genkernel or dracut commands.

- initramfs can be manually generated.

- An *initramfs* contains at *least* one file called **/init**.

- kernel function start_kernel( ) (<Linux sources>/init.main.c) searches and executes the /init program or script



/init script

```
#!/bin/busybox sh
# Init script in the initRamFS

mount -t proc none /proc
mount -t sysfs none /sys

mount -n -t devtmpfs devtmpfs /dev
mount -t ext4 /dev/mmcblk0p2 /newroot


mount -n -t devtmpfs devtmpfs /newroot/dev


exec switch_root /newroot /sbin/init
```

Mount the /proc and /sys pseudo-filesystem

Populate /dev

Mount the rootfs (2nd partition of the sdcard) to /newroot

Populate /newroot/dev

Switch to the rootfs on partition 2 and execute the /sbin/init command

```
man switch_root: switch to another filesystem as the root of the mount tree
```

Shared library dependency (1/2)

- Generally, programs are dynamically linked (other possibility: statically linked)
- A dynamically linked program must have all necessary libraries.
- Example (on PC) for the `/bin/ls` program:

```
readelf -d /bin/ls | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libselinux.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

- `ls` program needs the `libselinux.so.1, libc.so.1` libraries
- These libraries are in the `/lib or /lib64` directories.
- It is possible to use ldd or strings command in order to find the library dependency

```
strings /bin/ls | grep lib
```

Shared library dependency (2/2)

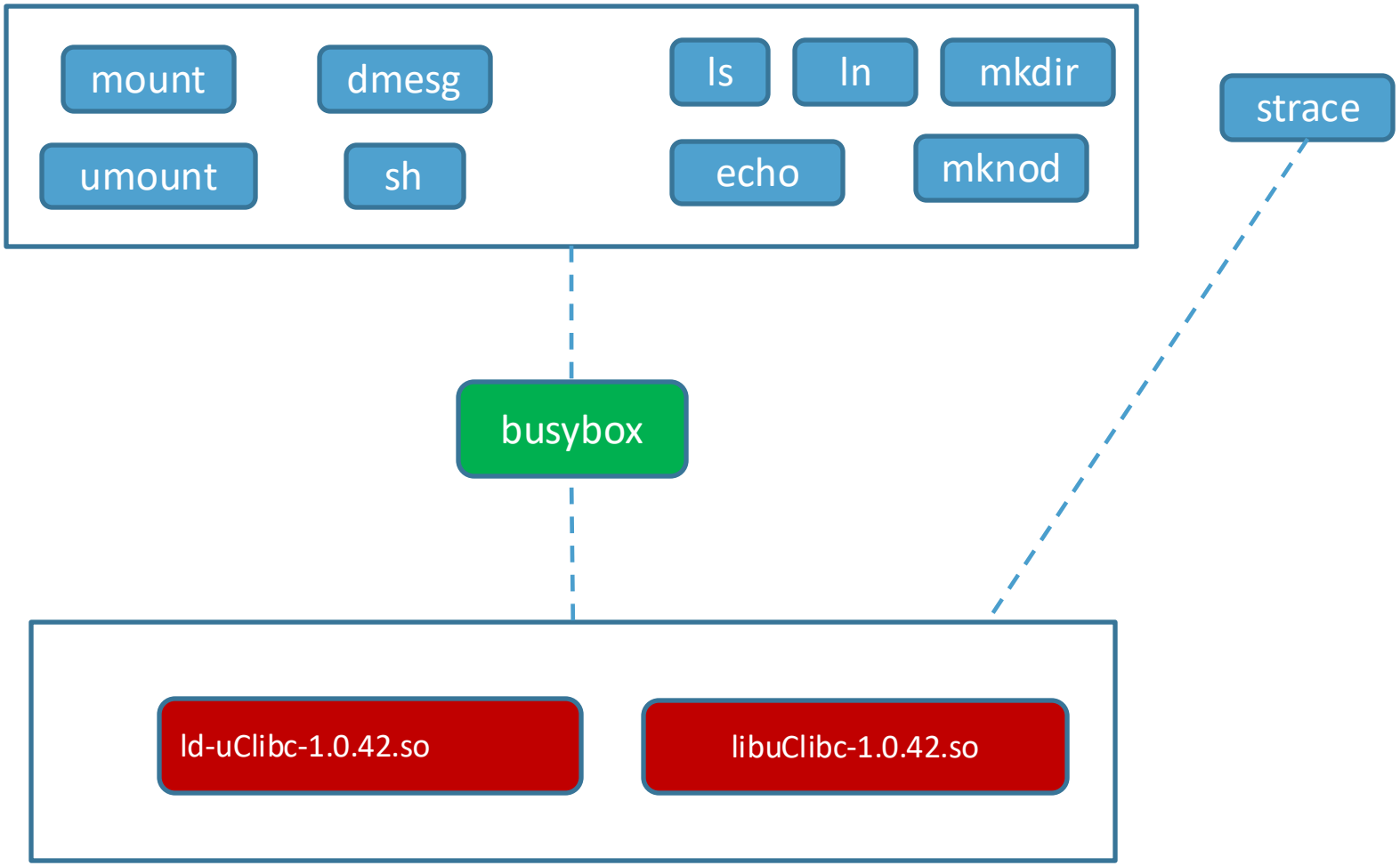
- `strace` command is another possibility to find dynamic libraries used by a program.
- `strace` shows used libraries and the path where these libraries are

- Example with the cryptsetup program executed on the NanoPi

```
# strace -f cryptsetup luksFormat /dev/mmcblk1p3
 openat(AT_FDCWD, "/lib64/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/usr/lib64/libcryptsetup.so.12", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/usr/lib64/libpopt.so.0", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/lib64/libuuid.so.1", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/lib64/libblkid.so.1", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/lib64/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/usr/lib64/libdevmapper.so.1.02", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/usr/lib64/libssl.so.1.1", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/usr/lib64/libcrypto.so.1.1", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/usr/lib64/libjson-c.so.4", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/lib64/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
 openat(AT_FDCWD, "/lib64/libatomic.so.1", O_RDONLY|O_CLOEXEC) = 3
```

Summary of programs and shared libraries

Here is a list of programs used in an initramfs: `mount, umount, dmesg, sh, ls, ln, mkdir, mknod, sleep, echo`



Shared library dependency (1/3)

- On PC, the directory `/buildroot/output/target` contains an image of the rootfs

```
cd /buildroot/output/target
ls -al bin/mount
   lrwxrwxrwx 1 root root 7 Jul 21 10:23 bin/mount -> bin/busybox
ls -al bin/ls
   lrwxrwxrwx 1 root root 20 Jul 21 10:14 bin/ls -> bin/busybox
ls -al sbin/switch_root
   lrwxrwxrwx 1 root root 20 Jul 21 10:14 sbin/switch_root -> bin/busybox

ls -al usr/bin/strace
-rwxr-xr-x 1 root root 1312984 Oct 29 07:07 usr/bin/strace
```

- `mount, ls` and the others are symbolic link to `bin/busybox`
- `strace` is an executable program

Shared library dependency (2/3)

Find the shared libraries:

```
cd /buildroot/output/target/bin
readelf -d busybox | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.0]

cd /buildroot/output/target/usr/bin
readelf -d strace | grep NEEDED
0x0000000000000001 (NEEDED)             Shared library: [libc.so.0]
```

The shared librarie (libc.so.0) is in this directory:

```
/buildroot/output/target/lib
```

```
cd /buildroot/output/target/lib
 lrwxrwxrwx  1 root root   libc.so.0 -> libuClibc-1.0.42.so
 lrwxrwxrwx  1 root root   libc.so.1 -> libuClibc-1.0.42.so

readelf -d libuClibc-1.0.42.so | grep NEEDED
    0x0000000000000001 (NEEDED)             Shared library: [ld-uClibc.so.1]
readelf -d ld-uClibc.so.1 | grep NEEDED
    no dependency
```

Shared library dependency (3/3)

```
cd /buildroot/output/target/lib
ls -al
lrwxrwxrwx  1 root root     19 Aug 17 09:32 libc.so.0 -> libuClibc-1.0.42.so
lrwxrwxrwx  1 root root     19 Aug 17 09:32 libc.so.1 -> libuClibc-1.0.42.so
-rwxr-xr-x  1 root root 514736 Oct  3 08:07 libuClibc-1.0.42.so

lrwxrwxrwx  1 root root     19 Aug 17 09:27 ld-uClibc.so.0 -> ld-uClibc.so.1
lrwxrwxrwx  1 root root     19 Aug 17 09:27 ld-uClibc.so.1 -> ld-uClibc-1.0.42.so
-rwxr-xr-x  1 root root  33864 Oct  3 08:07 ld-uClibc-1.0.42.so
```

Initramfs summary

The initramfs contains these files and directories:

```
/bin
/bin/mount
/bin/umount
/bin/sh
/bin/dmesg
/bin/echo
/bin/ls
/bin/ln
/bin/mkdir
/bin/mknod
/bin/sleep
/bin/busybox
/sbin
/sbin/switch_root
/usr/bin/strace

/proc

/lib/ld-uClibc-1.0.42.so
/lib/libuClibc-1.0.42.so
/lib/ld-uClibc.so.0
/lib/ld-uClibc.so.1
/lib/libc.so.0
/lib/libc.so.1
```

Busybox and symbolic links

Shared libraries
And symbolic links

```
/dev
/etc
/home

/init

/sys
/newroot
/root
```

/init script (p 38)

Build initramfs (1/4)

This script builds the initramfs in the directory RAMFS=/buildroot/ramfs

```
#!/bin/bash
BR_TARGET=/buildroot/buildroot3/output/target
RAMFS=/buildroot/ramfs

RAMFS2=$RAMFS/t
mkdir $RAMFS
mkdir $RAMFS2
cd $RAMFS

mkdir -p $RAMFS2/{bin,dev,etc,home,lib,newroot,proc,root,usr/bin,usr/lib,sbin,sys}

cd $RAMFS2
cd bin
cp ${BR_TARGET}/bin/busybox .

ln -s busybox mount
ln -s busybox umount
ln -s busybox sh
ln -s busybox dmesg
ln -s busybox echo
ln -s busybox ls
ln -s busybox ln
ln -s busybox mkdir
ln -s busybox mknod
ln -s busybox sleep

cd $RAMFS2
```

/bin: symbolic links to busybox

Build initramfs (2/4)

```
cd  usr/bin
cp ${BR_TARGET}/usr/bin/strace .
cd $RAMFS2

cd sbin
ln -s ../bin/busybox switch_root
cd $RAMFS2

cd lib
cp ${BR_TARGET}/lib/ld-uClibc-1.0.42.so .
cp ${BR_TARGET}/lib/libuClibc-1.0.42.so .
ln -s ld-uClibc.so.1      ld-uClibc.so.0
ln -s ld-uClibc-1.0.42.so ld-uClibc.so.1
ln -s libuClibc-1.0.42.so libc.so.0
ln -s libuClibc-1.0.42.so libc.so.1
cd $RAMFS2
```

Shared libraries

Build initramfs (3/4)

```
######initialize the /init file
cat > init << endofinput
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys

mount -n -t devtmpfs devtmpfs /dev
mount -t ext4 /dev/mmcblk0p2 /newroot
mount -n -t devtmpfs devtmpfs /newroot/dev

exec switch_root /newroot /sbin/init
endofinput
######
chmod 755 init
```

Init script

Build initramfs (4/4)

```
cd $RAMFS2

find . | cpio --quiet -o -H newc > ../Initrd

cd $RAMFS

gzip -9 -c Initrd > Initrd.gz


mkimage -A arm -T ramdisk -C none -d Initrd.gz uInitrd

cp ${RAMFS}/uInitrd ${RAMFS}/../output/images/.
cd /buildroot
rm -rf $RAMFS
```

It is important to be in the RAMFS2 directory

Compress the initramfs

Add the u-boot header

uInitrd is the initramfs