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

File Systems, LUKS, InitRamFS

Youtube: https://youtu.be/UoA6rVIbUJA

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

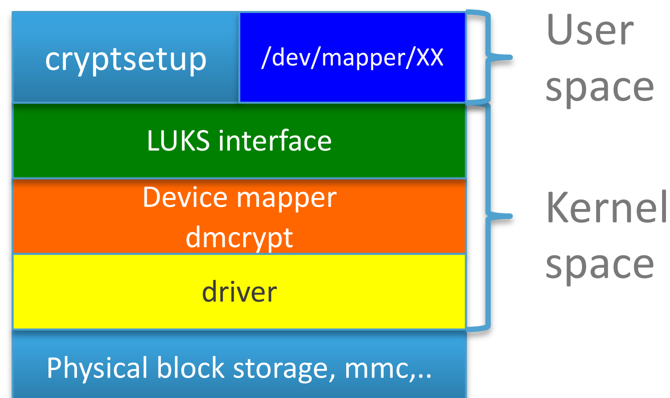
[9]: https://elinux.org/images/0/02/Filesystem_Considerations_for_Embedded_Devices.pdf

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 workspace/nano/buildroot`

  `make linux-xconfig or make linux-menuconfig`

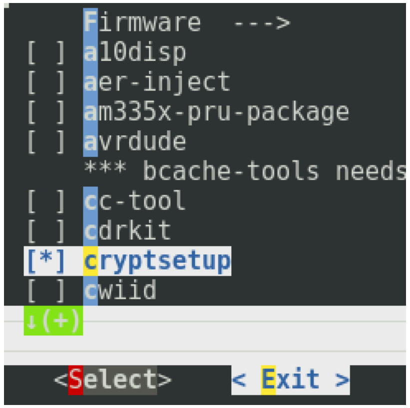
  `Go to: device driver → Multiple Devices drivers support (RAID and LVM) → Device mapper support → Crypt target support`

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

  `cd workspace/nano/buildroot`

  `make menuconfig`

  `Go to: Target Packages → Hardware handling : choose cryptsetup`



LUKS with NanoPi

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

Create LUKS partition

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`

SDCard

Create a LUKS partition

```
# sudo cryptsetup --debug --pbkdf-memory 256 luksFormat $DEVICE
    Be careful, type yes in UPPERCASE
    (--pbkdf-memory limits the amount of memory)
```

Dump the header information of a LUKS device

```
# sudo cryptsetup luksDump /dev/mmcblk0p3
```

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      254,    0 Jan   1 15:36 usrfs1
```

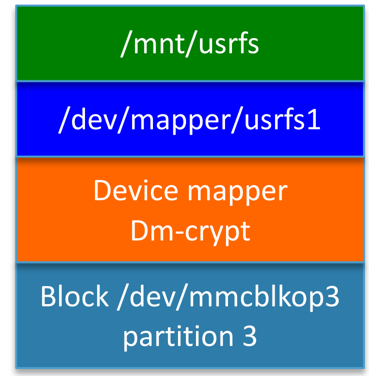
Create LUKS partition

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
```



Work with the LUKS partition

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

dmsetup: low level logical volume management

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

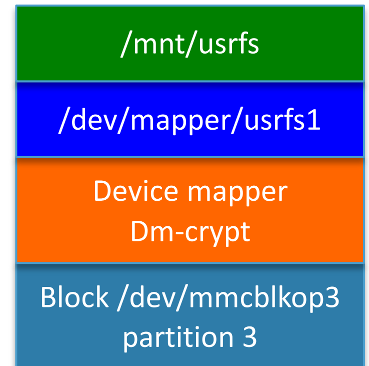
Use LUKS partition

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

```
# sudo cryptsetup --debug 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
```

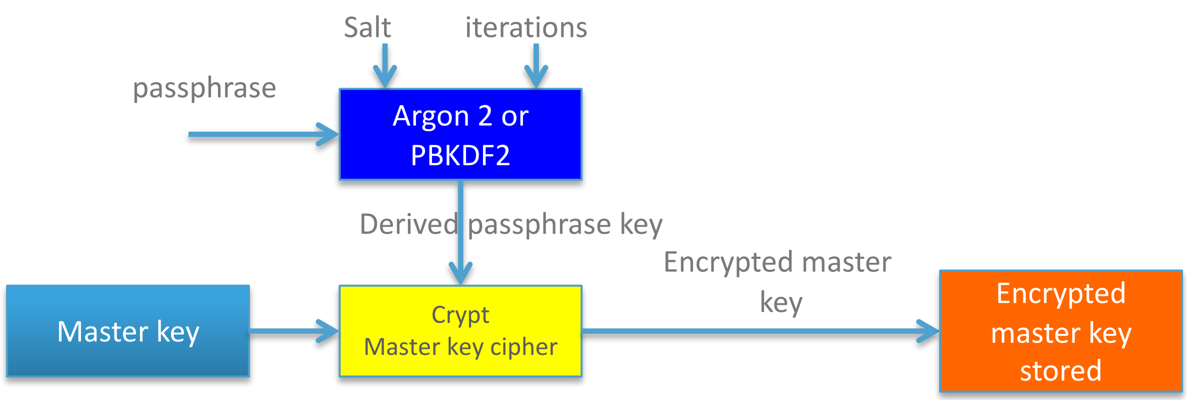
LUKS, key generation

- 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

```
# cryptsetup luksAddKey /dev/mmcblk0p3
```

Dump the header information of a LUKS device

```
# cryptsetup luksDump /dev/mmcblk0p3
```

Crypt the master key

```
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
```

Passphrase 1

```
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
```

Passphrase 2

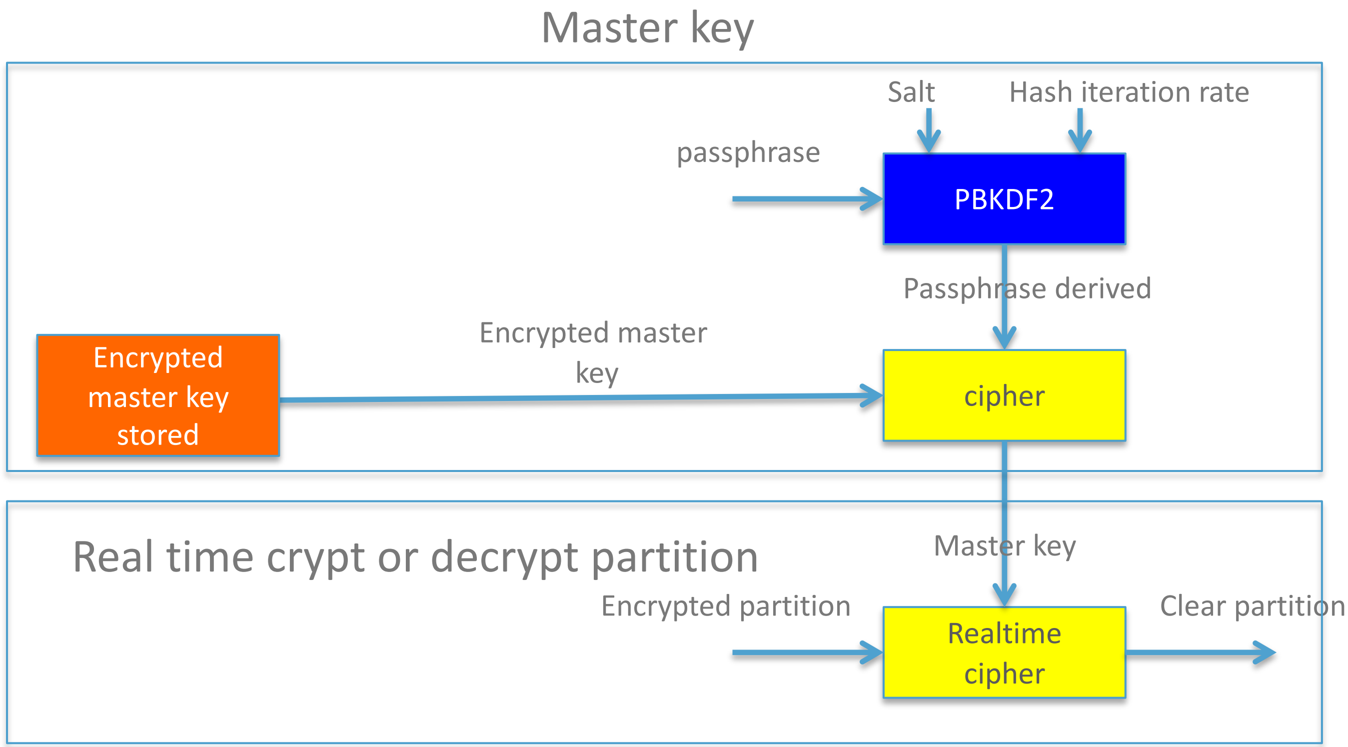
```
Key Slot 1: ENABLED
        Iterations:             81011
        Salt:                   fd 21 0f d6 39 c4 1c 79 b5 2b ec 4d 43 dd 66 e0
                                ff 41 76 2f 39 59 e2 00 9a 2c 42 6b 22 10 16 47
        Key material offset:    264
        AF stripes:             4000
```

LUKS, key generation

Dump the encrypted master key of a LUKS device

```
# cryptsetup luksDump –dump-master-key /dev/mmcblk0p3
    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
   …
```

U-boot-Linux boot-without initramfs 1/2

See u-boot course

Show boot.cmd file: `cat $HOME/workspace/nano/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
```

Linux kernel boot parameters

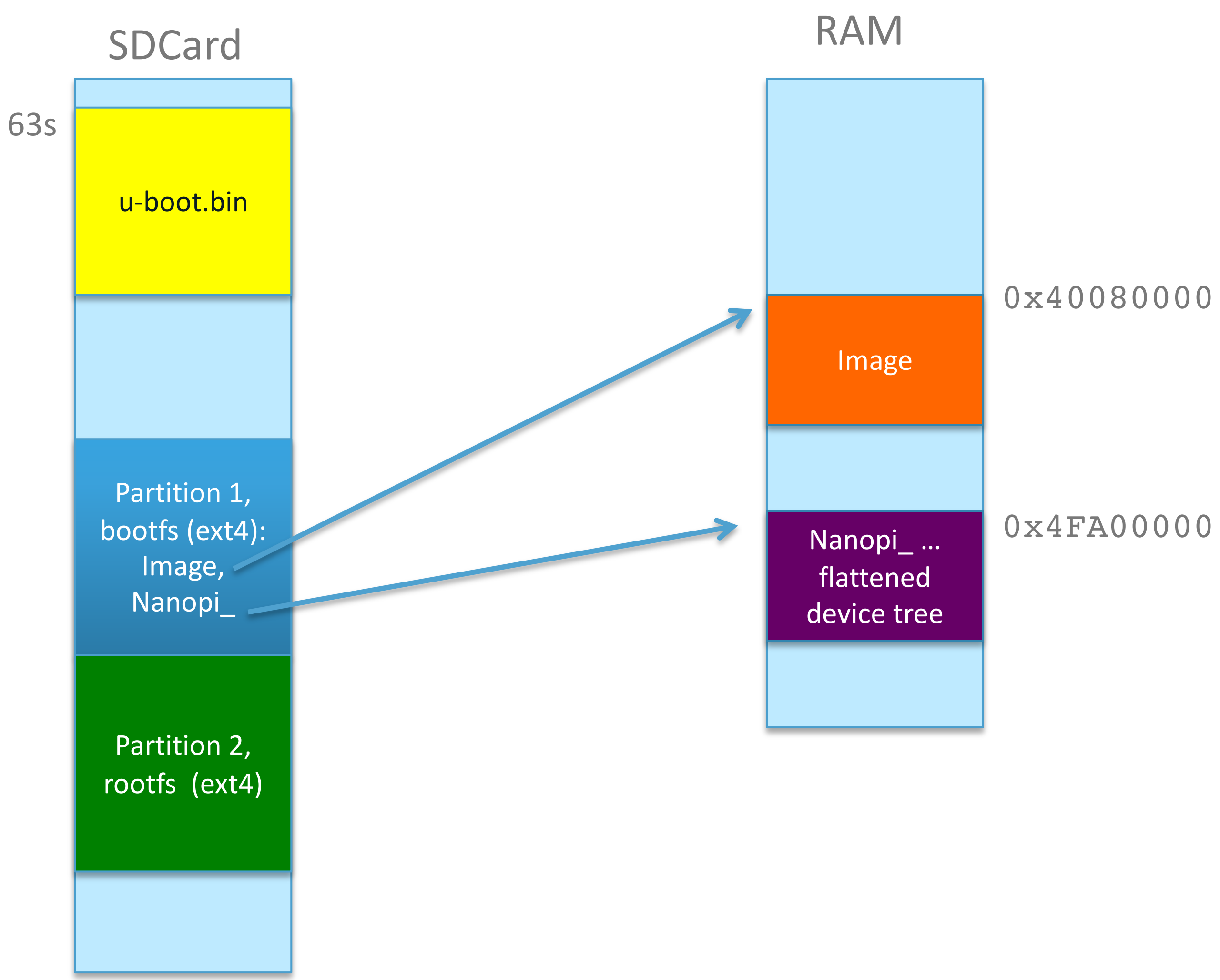
Start Linux

Load FDT

Load Image

mmc 0: SDCard 1st partition

U-boot-Linux boot-without initramfs 2/2



Initramfs

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-with initramfs

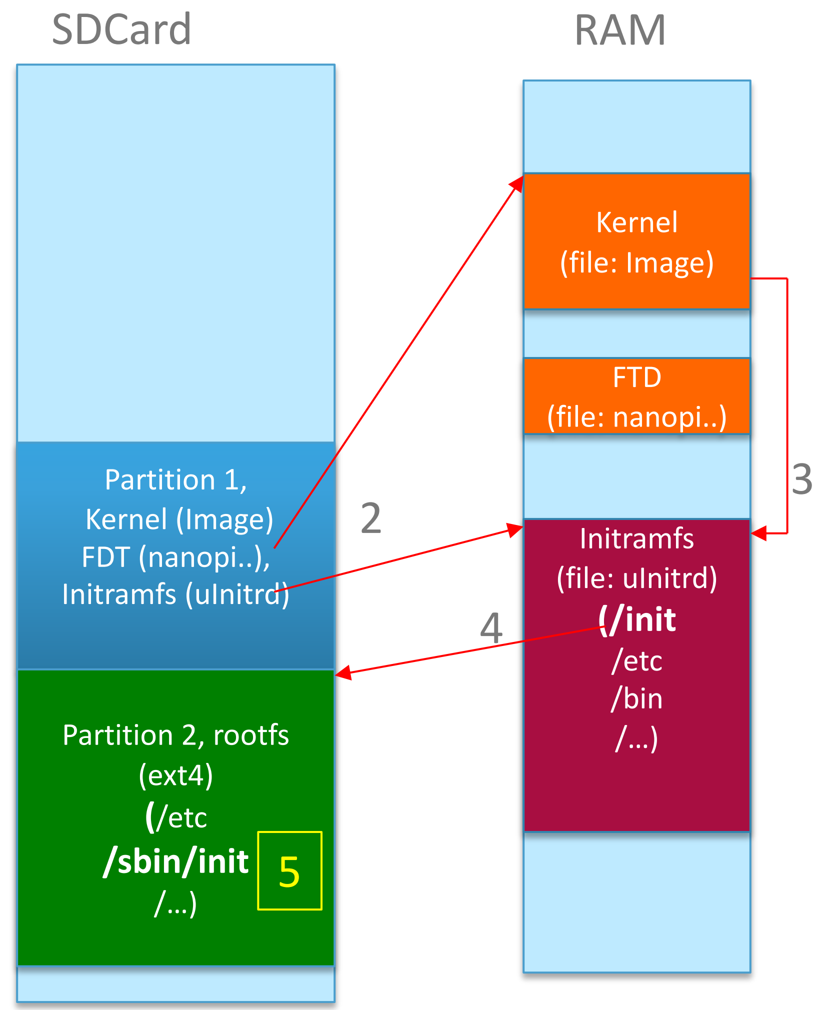
1) Kernel (Image), initramfs (uInitrd) and flattened device tree (Sun50i...) files are located in the partition 1 of the SDCard

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 1/2

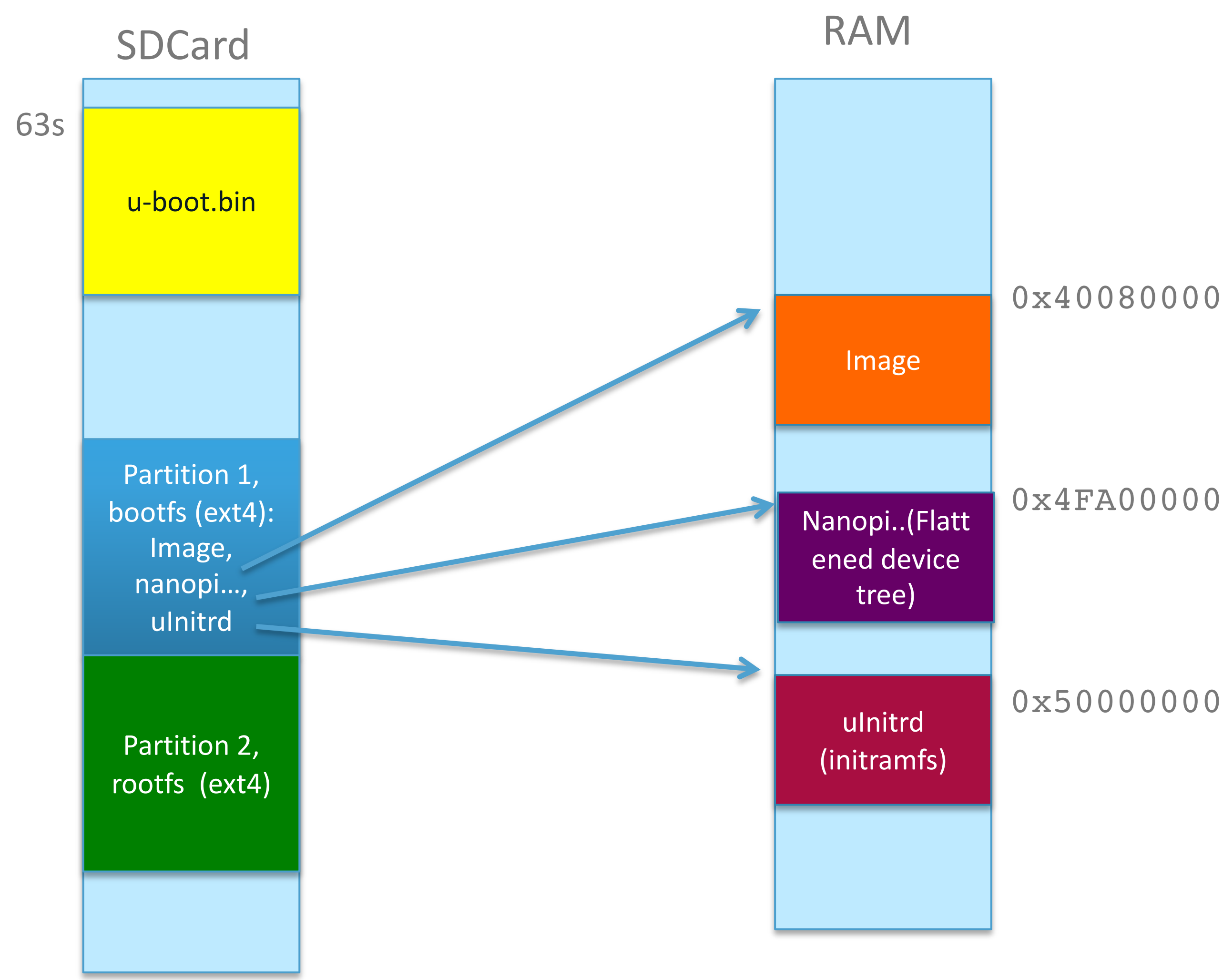
Show boot.cmd file: `cat $HOME/workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd`

```
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

booti $kernel_addr_r 0x50000000 $fdt_addr_r
```

initramfs address

U-boot-Linux boot-with initramfs 2/2

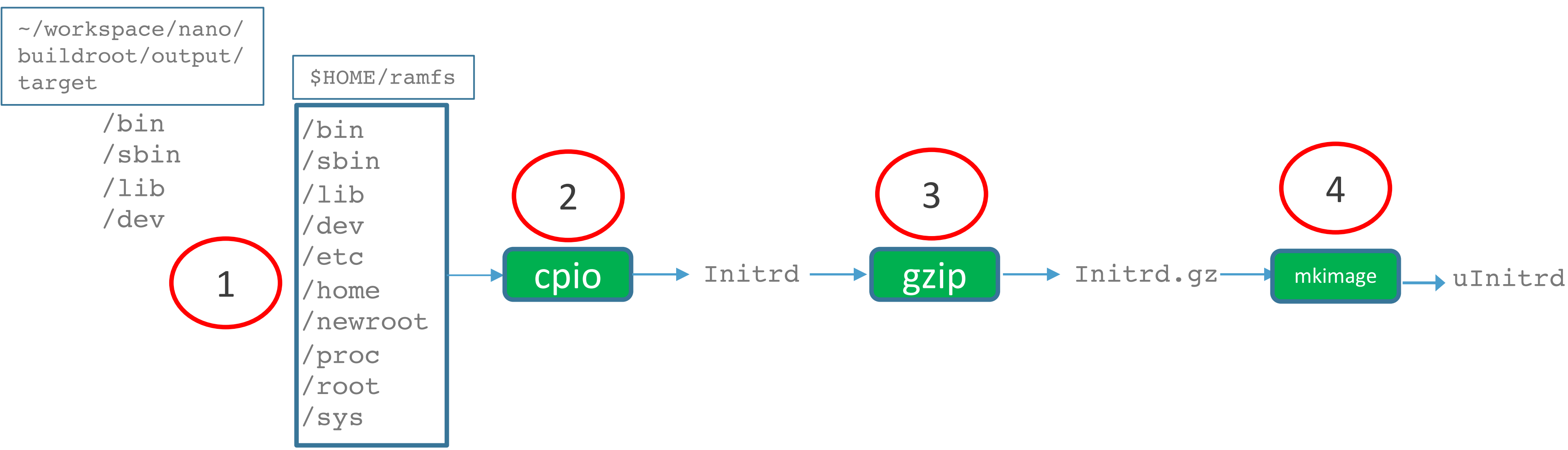


How to build Initramfs

On PC, NanoPi rootfs is in this directory: `$HOME/workspace/nano/buildroot/output/target/`

Principle to build an initramfs:

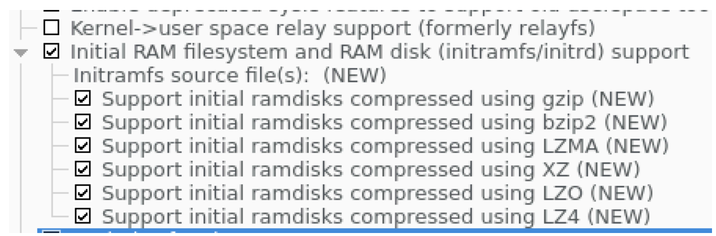
1. to copy the right files into a directory (`$HOME/ramfs`),
2. to copy these files in a cpio archive file,
3. to compress this file
4. To add the uboot header



Initramfs: kernel configuration

Kernel configuration:

```
CONFIG_BLK_DEV_INITRD=y
   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
```

(Generally this option is not used)

Embedding the initramfs into the kernel:

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

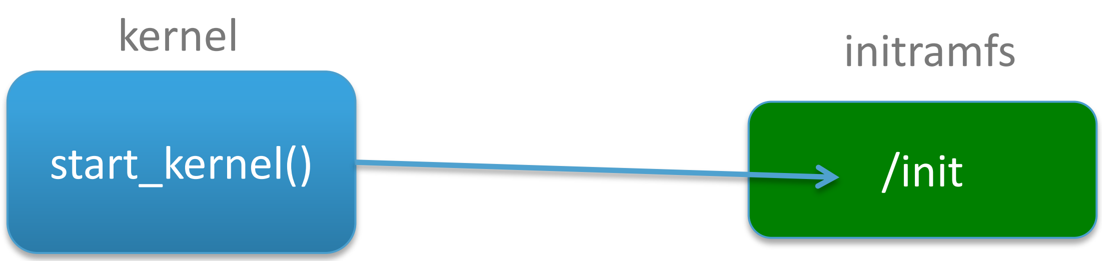
Initramfs manual generation 1/11

- initramfs is a cpio archive file. 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 2/11

```
#!/bin/busybox sh
# Init script in the initRamFS
mount -t proc none /proc
mount -t sysfs none /sys

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

Mount the rootfs (2nd partition) 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 3/11

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

```
ldd ls
      linux-vdso.so.1 =>  (0x00007fff46198000)
      libselinux.so.1 => /lib64/libselinux.so.1 (0x00000032fca00000)
      libcap.so.2 => /lib64/libcap.so.2 (0x0000003dd9c00000)
      libacl.so.1 => /lib64/libacl.so.1 (0x0000003dd9800000)
      libc.so.6 => /lib64/libc.so.6 (0x0000003dc0800000)
      libdl.so.2 => /lib64/libdl.so.2 (0x0000003dc0c00000)
      libpcre.so.1 => /lib64/libpcre.so.1 (0x00000032fc600000)
      libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003dc1000000)
      /lib64/ld-linux-x86-64.so.2 (0x0000003dc0400000)
      libattr.so.1 => /lib64/libattr.so.1 (0x0000003dd7800000)
```

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

```
strings ls | grep lib
```

Shared library dependency 4/11

- 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 program cryptsetup on 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
```

Shared library dependency 5/11

- In order to limit the shared libraries, busybox program can be used (see course nanopi.pdf)

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

```
Linux$ cd ~/workspace/nano/buildroot/output/target
Linux$ cd bin
Linux$ strings busybox | grep lib
          libc.so.6
          libresolv.so.2
          /lib/ld-linux-aarch64.so.1
```

Important: busybox needs these three libraries:

```
libc.so.6
libresolv.so.2
/lib/ld-linux-aarch64.so.1
```

It is necessary to know in which directories are these three libraries

Shared library dependency 5/11

- strace shows used libraries and the path where these libraries are

- Example: ls program on NanoPi (ls is a symbol link to busybox)

```
strace –f ls
openat(AT_FDCWD, "/lib64/libresolv.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

We can see that `ls` (link to busybox) uses:

- `/lib64/libresolv.so.2` and
- `/lib64/libc.so.6` libraries

Shared library dependency 5/11

```
Linux$ cd ~/workspace/nano/buildroot/output/target
```

libc library

```
Linux$ cd lib64
-rwxr-xr-x. 1 root root 1414752 Nov  4 09:12 libc-2.30.so
lrwxrwxrwx. 1 root root      12 Sep 14 16:23 libc.so.6 -> libc-2.30.so
```

libresolv library

```
Linux$ cd lib64
-rwxr-xr-x    1 root root 80392 Nov  4  2019 libresolv-2.30.so
lrwxrwxrwx    1 root root    17 Sep 14  2019 libresolv.so.2 -> libresolv-2.30.so
```

Shared library dependency 5/11

```
Linux$ cd ~/workspace/nano/buildroot/output/target
```

ld-linux-aarch64 library

```
Linux$ cd lib64
lrwxrwxrwx. 1 root root 19 Sep 14 16:23 ld-linux-aarch64.so.1 -> ../lib64/ld-2.30.so
-rwxr-xr-x. 1 root root 159384 Nov  4 09:12 ld-2.30.so
```

```
Linux$ cd lib
-rwxr-xr-x. 1 root root 159384 Nov  4 09:12 ld-2.30.so
lrwxrwxrwx. 1 root root     19 Sep 14 16:23 ld-linux-aarch64.so.1 -> ../lib64/ld-2.30.so
```

Initramfs summary 7/11

The initramfs contains these files and directories:

/bin /bin/mknod /bin/mount /bin/ln /bin/sleep /bin/umount /bin/ls /bin/mkdir /bin/sh /bin/busybox /sbin /sbin/switch_root /proc — Busybox and symbolic links /lib64/libc.so.6 /lib64/libc-2.30.so /lib64/libresolv.so.2 /lib64/libresolv-2.30.so /lib64/ld-linux-aarch64.so.1 /lib64/ld-2.30.so /lib/ld-linux-aarch64.so.1 /lib/ld-2.30.so — Shared libraries And symbolic links	/dev /dev/null /dev/console /dev/tty /dev/random /dev/urandom /dev/ttyS0 /dev/ttyS1 /dev/ttyS2 /dev/ttyS3 /dev/mmcblk0p /dev/mmcblk0p1 /dev/mmcblk0p2 /dev/mmcblk0p3 /dev/mmcblk0p4 — Nodes files ./etc ./home ./init — /init script (p 23) ./sys ./newroot ./root

Build initramfs 8/11

This script builds the initramfs in the directory ROOTFSLOC=ramfs

```
#!/bin/bash
ROOTFSLOC=ramfs
cd $HOME
mkdir $ROOTFSLOC
mkdir -p $ROOTFSLOC/{bin,dev,etc,home,lib,lib64,newroot,proc,root,sbin,sys}

cd $ROOTFSLOC/dev
sudo mknod null    c 1 3
sudo mknod tty     c 5 0
sudo mknod console c 5 1
sudo mknod random  c 1 8
sudo mknod urandom c 1 9

sudo mknod mmcblk0p  b 179 0
sudo mknod mmcblk0p1 b 179 1
sudo mknod mmcblk0p2 b 179 2
sudo mknod mmcblk0p3 b 179 3
sudo mknod mmcblk0p4 b 179 4

sudo mknod ttyS0 c 4 64
sudo mknod ttyS1 c 4 65
sudo mknod ttyS2 c 4 66
sudo mknod ttyS3 c 4 67
```

Nodes files

Build initramfs 9/11

```
cd ../bin
cp ~/workspace/nano/buildroot/output/target/bin/busybox .

ln -s busybox ls
ln -s busybox mkdir
ln -s busybox ln
ln -s busybox mknod
ln -s busybox mount
ln -s busybox umount
ln -s busybox sh
ln -s busybox sleep
ln -s busybox dmesg
cp ~/workspace/nano/buildroot/output/target/usr/bin/strace .
```

/bin
Busybox and symbolic links
strace

```
cd ../sbin
ln -s ../bin/busybox switch_root
```

/sbin

Build initramfs 10/11

```
cd ../lib64
cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.30.so .
cp ~/workspace/nano/buildroot/output/target/lib64/libresolv-2.30.so .
cp ~/workspace/nano/buildroot/output/target/lib64/libc-2.30.so .
ln -s libresolv-2.30.so libresolv.so.2
ln -s libc-2.30.so libc.so.6
ln -s ../lib64/ld-2.30.so ld-linux-aarch64.so.1

cd ../lib
cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.30.so .
ln -s ../lib64/ld-2.30.so ld-linux-aarch64.so.1
```

Shared libraries

Build initramfs 10/11

```
cd ..
cat > init << endofinput
#!/bin/busybox sh

mount -t proc none /proc
mount -t sysfs none /sys
mount -t ext4 /dev/mmcblk0p2 /newroot
mount -n -t devtmpfs devtmpfs /newroot/dev

#exec sh
exec switch_root /newroot /sbin/init

endofinput
######
chmod 755 init
```

/init

```
cd ..
sudo chown -R 0:0 $ROOTFSLOC
```

Change owner to root:root (0:0)

Build the initramfs cpio archive 11/11

```
cd $ROOTFSLOC

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

cd ..

gzip -9 -c Initrd > Initrd.gz

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

It is mandatory to be in the ROOTFSLOC directory

Compress the initramfs

Add the u-boot header