1 File system

1.1 Génération

Squelette de rootfs dans `workspace/nano/buildroot/system/skeleton`. Il est ensuite copié dans `buildroot/output/target` et les fichiers nécessaires y sont ensuite ajoutés.
Une fois que tous les fichiers sont ajoutés, une image `rootfs.xxx` est créé (xxx est ext4, squashfs, etc...)

1.2 Types de systèmes de fichier et leurs applications

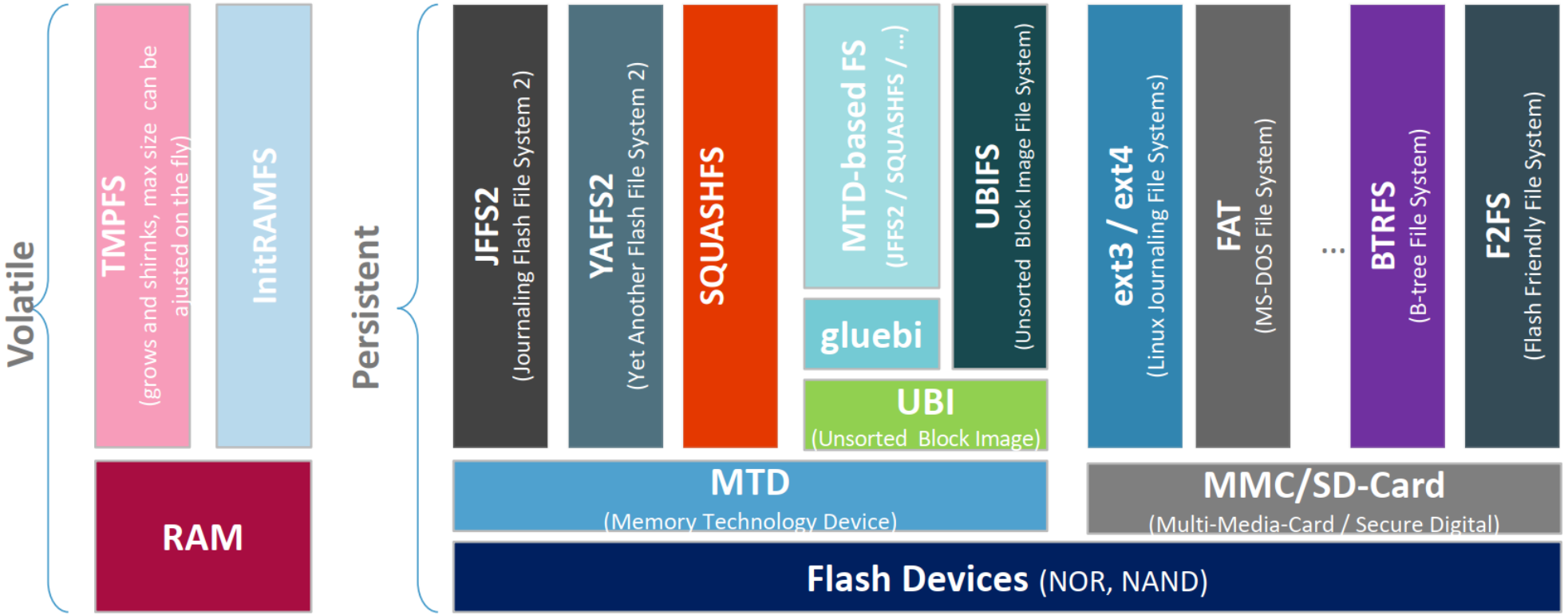
Pour les systèmes embarqués, il existe deux catégories de systèmes de fichiers :

1. Volatiles en RAM
2. Persitants sur des Flash (NOR et de plus en plus NAND)

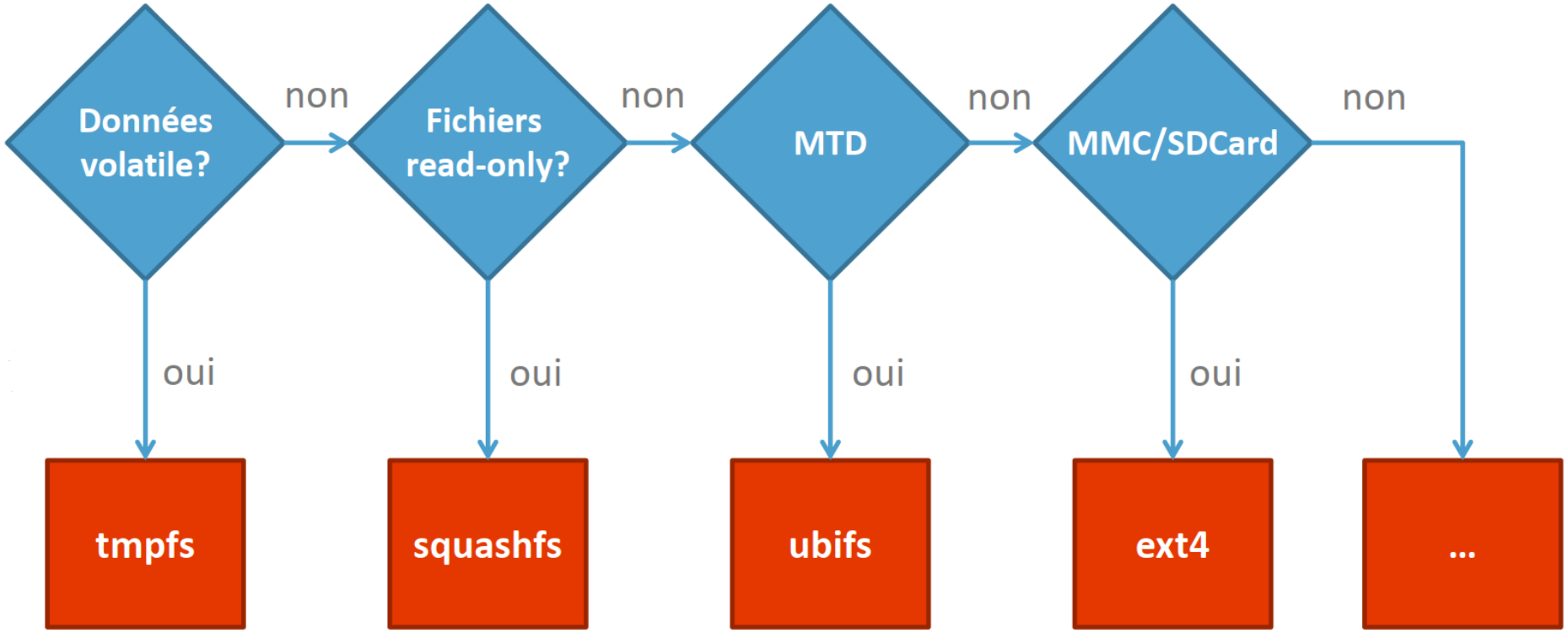
Deux technologies principales sont disponible sur les Flash :

- MTD (Memory Technology Device)
- MMC/SD-Card (Multi-Media-Card / Secure Digital Card)

1.2.1 FS types



1.2.2 Choix d'un FS



1.2.3 MMC technologies

MMC-eMMC-SD Card is composed by 3 elements

- MMC interface: handle communication with host
- FTL (Flash translation layer)
- Storage area: array of NAND chips

1.2.4 FTL

FTL is a small controller running a firmware. Its main purpose is to transform logical sector addressing into NAND addressing. It also handles:

- Bad block management
- Garbage collection.
- Wear levelling

1.3 Caractéristiques des filesystems ext2-3-4 et commande

"Filesystem considerations for embedded devices" is a good study about filesystems used on embedded systems.
This file system is very used in different Linux distribution.

- EXT filesystem was created in April 1992 and is a file system for the Linux kernel
  - Ext2 is not a journaled file system
  - Ext2 uses block mapping in order to reduce file fragmentation (it allocates several free blocks)
  - After an unexpected power failure or system crash (also called an unclean system shutdown), each mounted ext2 file system on the machine must be checked for consistency with the e2fsck program
- EXT2 replaced it in 1993
  - It was merged in the 2.4.15 kernel on November 2001
  - Ext3 is compatible with ext2
  - Ext3 is a journaled file system
  - The ext3 file system prevents loss of data integrity even when an unclean system shutdown occurs
- EXT4 arrived as a stable version in the Linux kernel in 2008
  - ext4 is backward compatible with ext3 and ext2, making it possible to mount ext3 and ext2 as ext4
  - Ext4 is included in the kernel 2.6.28
  - Ext4 supports Large file system:
    - Volume max: 2^60 bytes
    - File max: 2^40 bytes
  - Ext4 uses extents (as opposed to the traditional block mapping scheme used by ext2 and ext3), which improves performance when using large files and reduces metadata overhead for large files

1.3.1 Ext4 commands

```
# Create a partition (rootfs), start 64MB, length 256MB
sudo parted /dev/sdb mkpart primary ext4 131072s 655359s
# Format the partition with the volume label = rootfs
sudo mkfs.ext4 /dev/sdb1 -L rootfs
# Modify (on the fly) the ext4 configuration
sudo tune2fs <options> /dev/sdb1
# check the ext4 configuration
mount
sudo tune2fs -l /dev/sdb1
sudo dumpe2fs /dev/sdb1
# mount an ext4 file system
mount -t ext4 /dev/sdb1 /mnt/test // with default options
mount -t ext4 -o defaults,noatime,discard,nodiratime,data=writeback,acl,user_xattr
/dev/sdb1 /mnt/test
```

1.3.2 Ext4 mount options and MMC/SD-Card

- filesystem options can be activated with the mount command (or to the /etc/fstab file)
- These options can be modified with tune2fs command
- Journaling: the journaling guarantees the data consistency, but it reduces the file system performances
- MMC/SD-Card constraints: In order to improve the longevity of MMC/SDCard, it is necessary to reduce the unnecessary writes
- Mount options to reduce the unnecessary writes (man mount) :
  - noatime: Do not update inode access times on this filesystem (e.g., for faster access on the news spool to speed up news servers)
  - nodiratime: Do not update directory inode access times on this filesystem
  - relatime: this option can replace the noatime and nodiratime if an application needs the access time information (like mutt)

Mount options for the journaling (man ext4):

- Data=journal: All data is committed into the journal prior to being written into the main filesystem (It is the safest option in terms of data integrity and reliability, though maybe not so much for performance
- Data=ordered: This is the default mode. All data is forced directly out to the main file system before the metadata being committed to the journal
- Data=writeback: Data ordering is not preserved - data may be written into the main filesystem after its metadata has been committed to the journal. It guarantees internal filesystem integrity, however it can allow old data to appear in files after a crash and journal recovery.
- Discard: Use discard requests to inform the storage that a given range of blocks is no longer in use. A MMC/SD-Card can use this information to free up space internally, using the free blocks for wear-levelling.
- acl: Support POSIX Access Control Lists
- user_xattr: Support "user." extended attributes
- default : rw, suid, dev, exec, auto, nouser, and async
  - rw : read-write
  - suid : Allow set-user-identifier or set-group-identifier bits
  - dev : Interpret character or block special devices on the filesystem
  - exec : Permit execution of binaries
  - auto : Can be mounted with the -a option (mount –a)
  - nouser : Forbid an ordinary (i.e., non-root) user to mount the filesystem
  - async : All I/O to the filesystem should be done asynchronously

1.3.3 /etc/fstab file

File **/etc/fstab** contains descriptive information about the filesystems the system can mount

- file system : block special device or remote filesystem to be mounted
- mount pt : mount point for the filesystem
- type : the filesystem type
- options : mount options associated with the filesystem
- dump : used by the dump (backup filesystem) command to determine whichfilesystems need to be dumped (0 = no backup)

- pass : used by the fsck (8) program to determine the order in which filesystem checks are done at reboot time. The root filesystem should be specified with 1, and other filesystems should have a 2. if ¡pass¿ is not present or equal 0 -¿ fsck willassume that the filesystem is not checked.
- Field options: It contains at least the type of mount plus any additional options appropriate to the filesystem type.

Common for all types of file system are the options (man mount) :

- auto : Can be mounted with the -a option (mount -a)
- defaults : Use default options: rw, suid, dev, exec, auto, nouser, and async
- nosuid : Do not allow set-user-identifier or set-group-identifier bits to take effect
- noexec : Do not allow direct execution of any binaries on the mounted file system
- nodev : Do not interpret character or block special devices on the file system

1.4 Autre type de FS

1.4.1 BTRFS (B-Tree filesystem)

- BTRFS is a “new” file system compared to EXT. It is originally created by Oracle in 2007, it is a B-Tree filesystem
- It is considered stable since 2014
- Since 2015 BTRFS is the default rootfs for openSUSE
- BTRFS inspires from both Reiserfs and ZFS
- Theodore Ts’o (ext3-ext4 main developer) said that BTRFS has a better direction than ext4 because ”it offers improvements in scalability, reliability, and ease of management”

1.4.2 F2FS (Flash-Friendly File System)

It is a log filesystem. It can be tuned using many parameters to allow best handling on different supports.
F2FS features :

- Atomic operations
- Defragmentation
- TRIM support (reporting free blocks for reuse)

1.4.3 NILFS2 (New Implementation of a Log-structured File System)

- Developed by Nippon Telegraph and Telephone Corporation
- NILFS2 Merged in Linux kernel version 2.6.30
- NILFS2 is a log filesystem
- CoW for checkpoints and snapshots
- Userspace garbage collector

1.4.4 XFS (Flash-Friendly File System)

XFS was developed by SGI in 1993.

- Added to Linux kernel in 2001
- On disk format updated in Linux version 3.10
- XFS is a journaling filesystem
- Supports huge filesystems
- Designed for scalability
- Does not seem to be handling power loss (standby state) well

1.4.5 ZFS (Zettabyte ($10^2$1)File System)

ZFS is a combined file system and logical volume manager designed by Sun Microsystems.

- ZFS is a B-Tree file system
- Provides strong data integrity
- Supports huge filesystems
- Not intended for embedded systems (requires RAM)
- License not compatible with Linux

1.4.6 Conclusion

Performances:

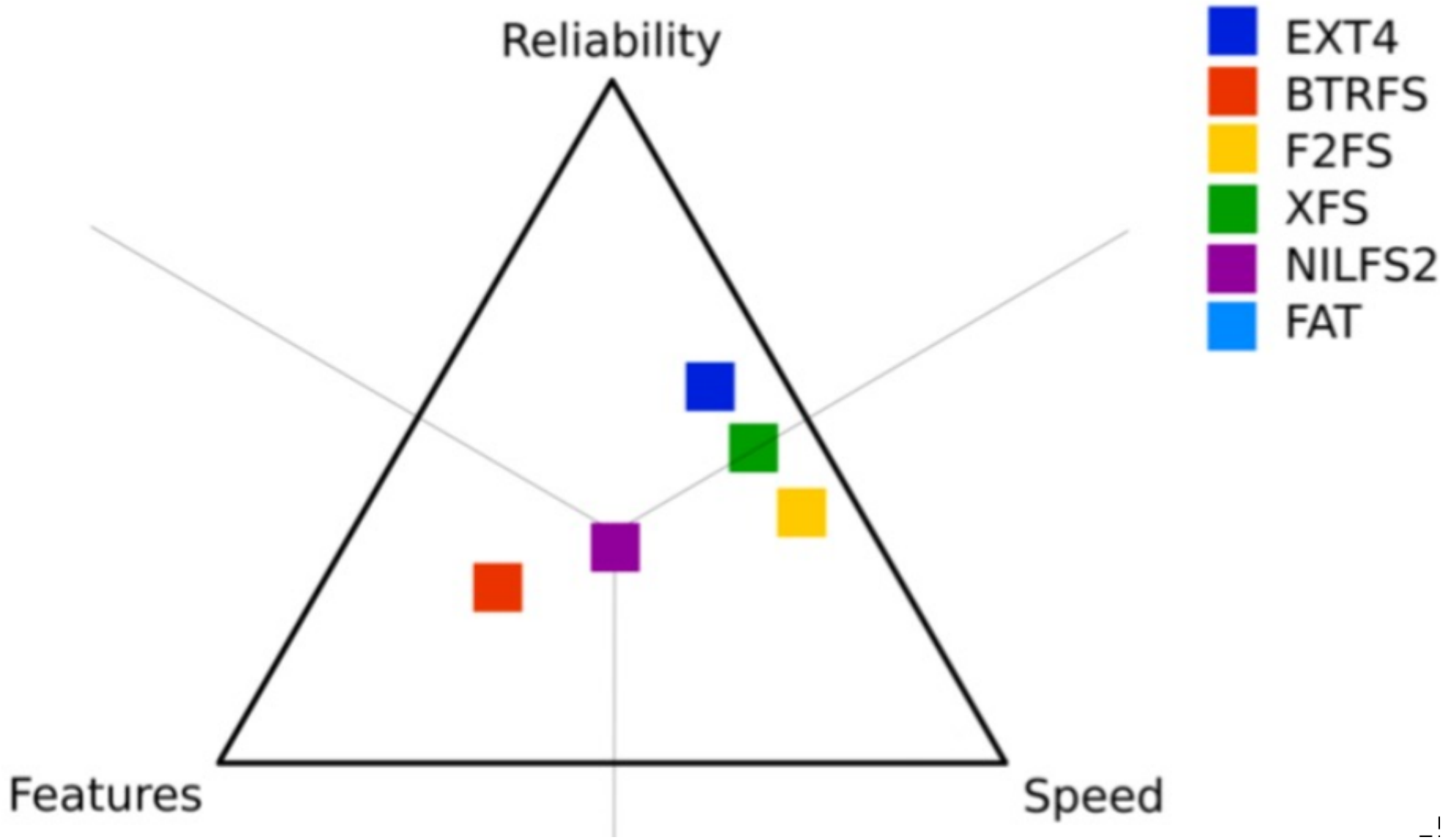
- EXT4 is currently the best solution for embedded systems using MMC
- F2FS and NILFS2 show impressive write performances

Features:

- BTRFS is a next generation filesystem
- NILFS2 provides simpler but similar features

Scalability:

- EXT4 clearly doesn't scale as well as BTRFS and F2FS



1.5 Architecture des FS

1.5.1 Journalized filesystem

A journalized filesystem keeps track of every modification in a journal in a dedicated area

- EXT3, EXT4, XFS, Reiser4
- Journal allows to restore a corrupted filesystem
- Modifications are first recorded in the journal

- Modifications are applied on the disk
- If a corruption occurs: The File System will either keep or drop the modifications
  - Journal is consistent : we replay the journal at mount time
  - Journal is not consistent : we drop the modifications

1.5.2 B-Tree filesystem

- ZFS, BTRFS, NILFS2
- B+ tree is a data structure that generalized binary trees
- CoW (Copy on Write) is used to ensure no corruption occurs at runtime :
  - The original storage is never modified. When a write request is made, data is written to a new storage area
  - Original storage is preserved until modifications are committed
  - If an interruption occurs during writing the new storage area, original storage can be used

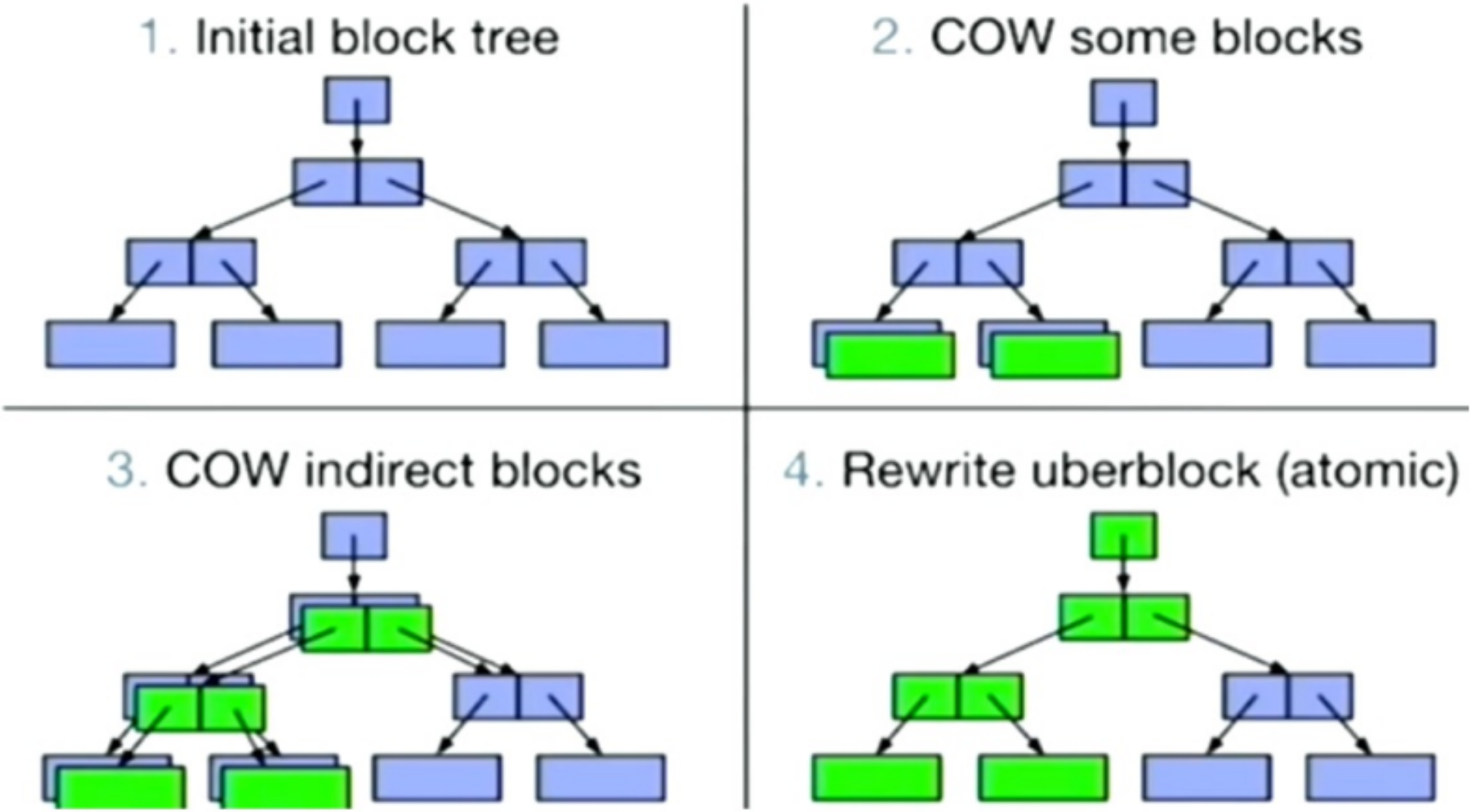
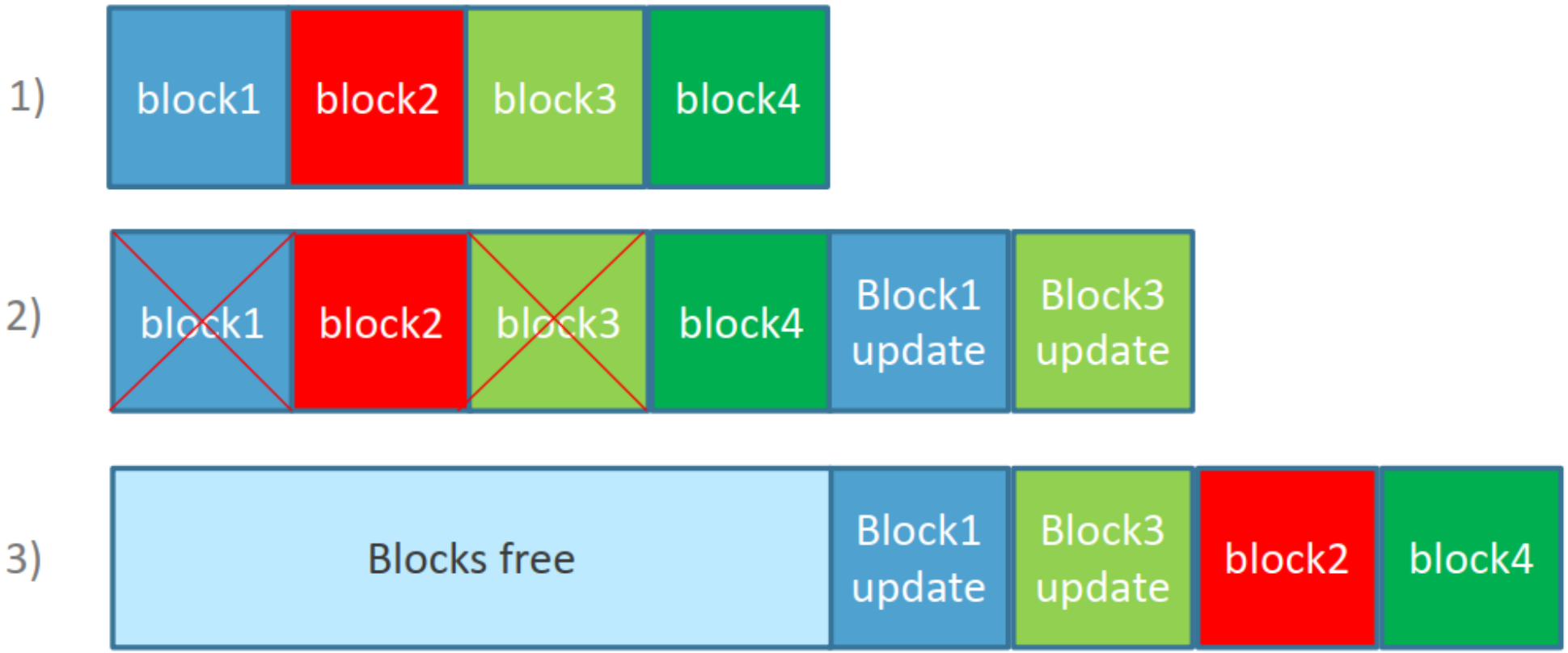


Figure 1: B-tree type FS execution

1.5.3 Log filesystem

Log-structured filesystems use the storage medium as circular buffer and new blocks are always written to the end.

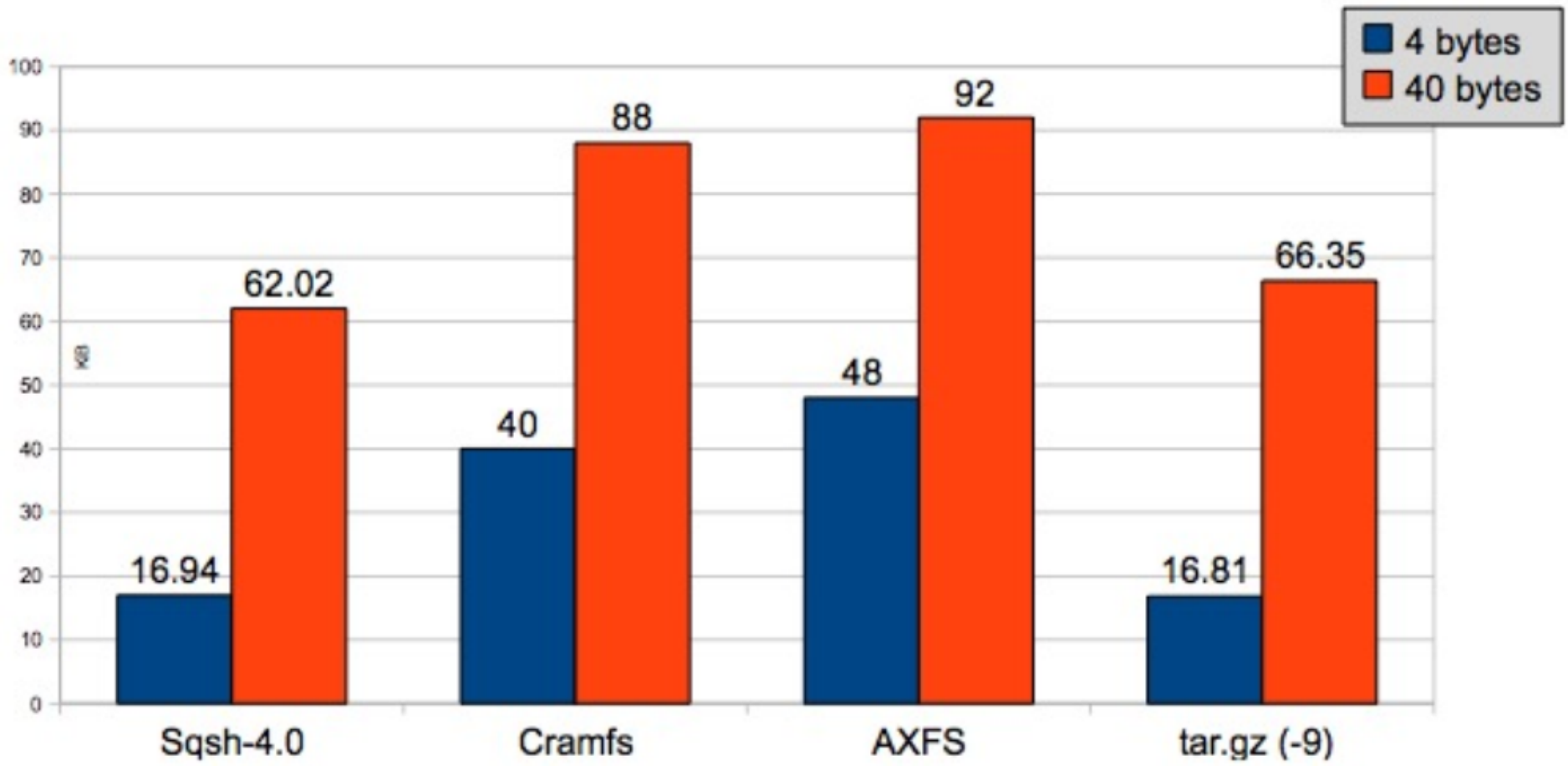
- F2FS, NILFS2, JFFS2, UBIFS
- Log-structured filesystems are often used for flash media since they will naturally perform wear-levelling
- The log-structured approach is a specific form of copy-on-write behavior



1. Initial state
2. Block 1-3 are updated, old blocks 1-3 are not used
3. Garbage copies block2 and 4, and frees old block1-2-3-4

1.6 Caractéristiques de Squashfs et commande

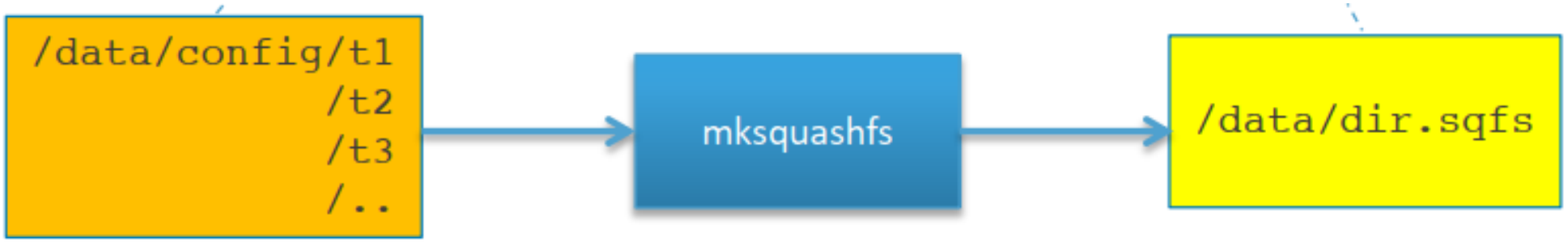
- Squashfs is a compressed read-only filesystem for Linux
- Squashfs versions
  - Squashfs 2.0 and squashfs 2.1: 2004, kernel 2.2
  - Squashfs 3.0: 2006, kernel 2.6.12
  - Squashfs 4.2: 2011, kernel 2.6.29
- It uses gzip, lzma, lzo, lz4 and xz compression to compress files, inodes and directories
- SquashFS 4.0 supports 64 bit filesystems and files (larger than 4GB), full uid/gid information, hard links and timestamps
- Squashfs is intended for general read-only filesystem use, for archival use, and in embedded systems with small processors where low overhead is needed



1.6.1 Create and use squashed file systems

1. Create the squashed file system dir.sqsh for the regular directory /data/config/ :

```
bash# mksquashfs /data/config/ /data/dir.sqsh
```



2. The mount command is used with a loopback device in order to read the squashed file system dir.sqsh

```
bash# mkdir /mnt/dir
bash# mount -o loop -t squashfs /data/dir.sqsh /mnt/dir
bash# ls /mnt/dir
```

3. It is possible to copy the dir.sqsh to an unmounted partition (e.g. /dev/sdb2) with the dd command and next to mount the partition as squashfs file system

```
bash# umount /dev/sdb2
bash# dd if=dir.sqsh of=/dev/sdb2
bash# mount /dev/sdb2 /mnt/dir -t squashfs
bash# ls /mnt/dir
```

1.7 Caractéristiques de tmpfs et commandes

- Tmpfs is a file system which keeps all files in virtual memory
- Everything in tmpfs is temporary in the sense that no files will be created on your hard drive. If you unmount a tmpfs instance, everything stored therein is lost.
- tmpfs puts everything into the kernel internal caches and grows and shrinks to accommodate the files it contains and is able to swap unneeded pages out to swap space. It has maximum size limits which can be adjusted on the fly via 'mount -o remount ...'
- If you compare it to ramfs you gain swapping and limit checking. Another similar thing is the RAM disk (/dev/ram*), which simulates a fixed size hard disk in physical RAM, where you have to create an ordinary filesystem on top. Ramdisks cannot swap and you do not have the possibility to resize them

1.7.1 Devtmpfs

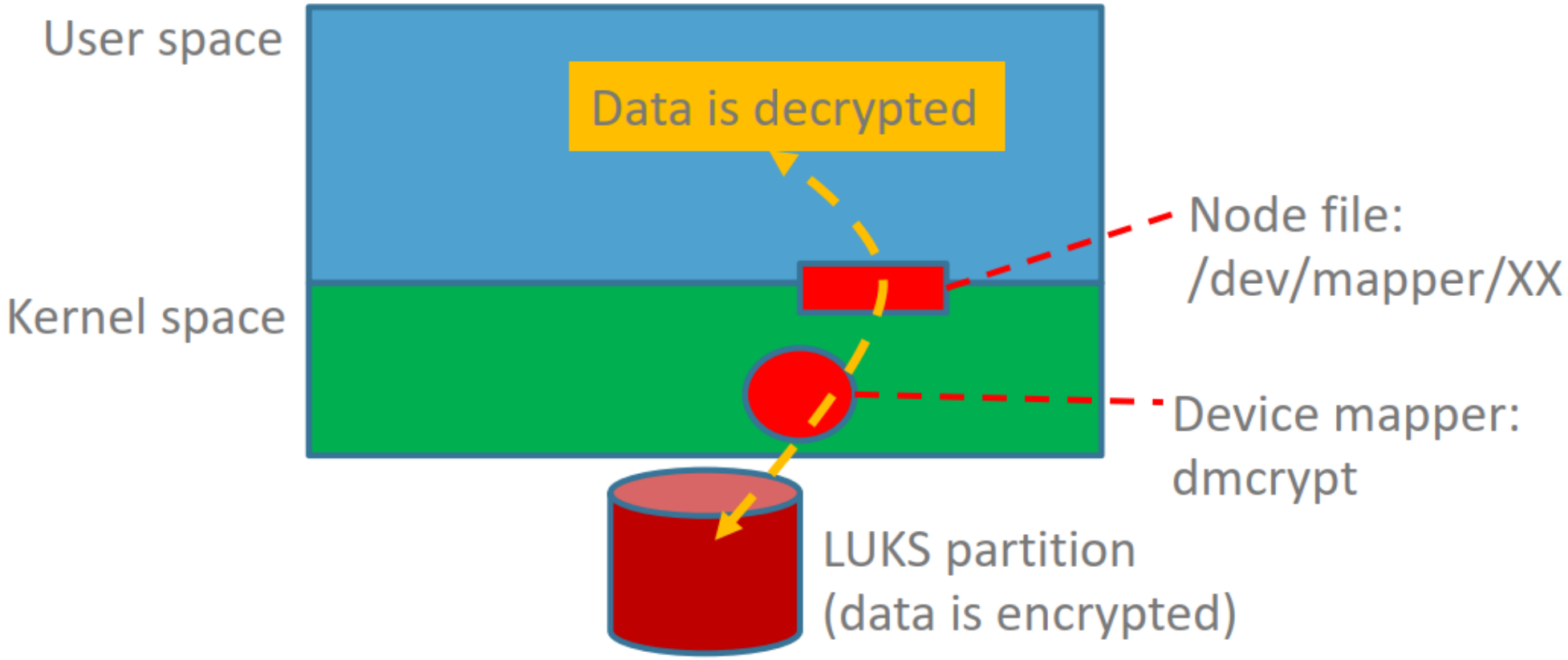
- devtmpfs is a file system with automatically populates nodes files (/dev/...) known by the kernel.
- This means, it is not necessary to have udev running nor to create a static /dev layout with additional, unneeded and not present device nodes.
- Instead the kernel populates the appropriate information based on the known devices.
- The kernel executes this command : mount -n -t devtmpfs devtmpfs /dev
- /dev is automatically populated by the kernel with its known devices

```
# ls /dev
autofs ptypf tty47
btrfs-control random tty48
bus rtc0 tty49
```

```
console shm tty5
cpu_dma_latency snapshot tty50
```

1.8 Caractéristiques de LUKSfs et commandes

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

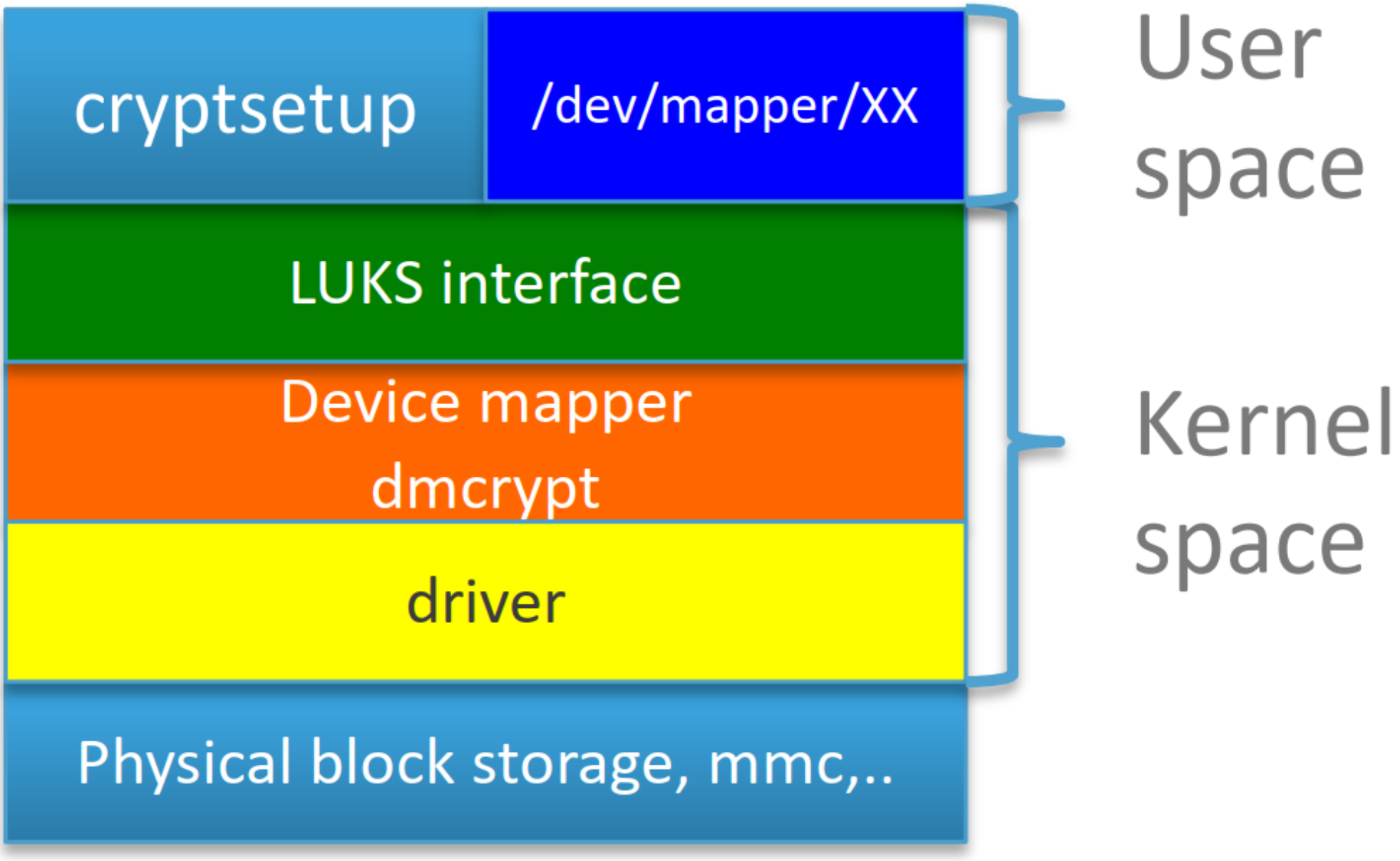


-

- 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
- 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



1.8.1 Create LUKS partition

```
# 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
Be careful, type yes in UPPERCASE
```

```
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/
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
# sudo umount /dev/mapper/usrfs1
Removes the existing mapping usrfs1 and wipes the key from kernel memory
# sudo cryptosetup close usrfs1
dmsetup: low level logical volume management
# dmsetup info -C
# dmsetup remove -f usrfs1
```

1.8.2 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
```

1.9 Gestion des clés de LUKS 42

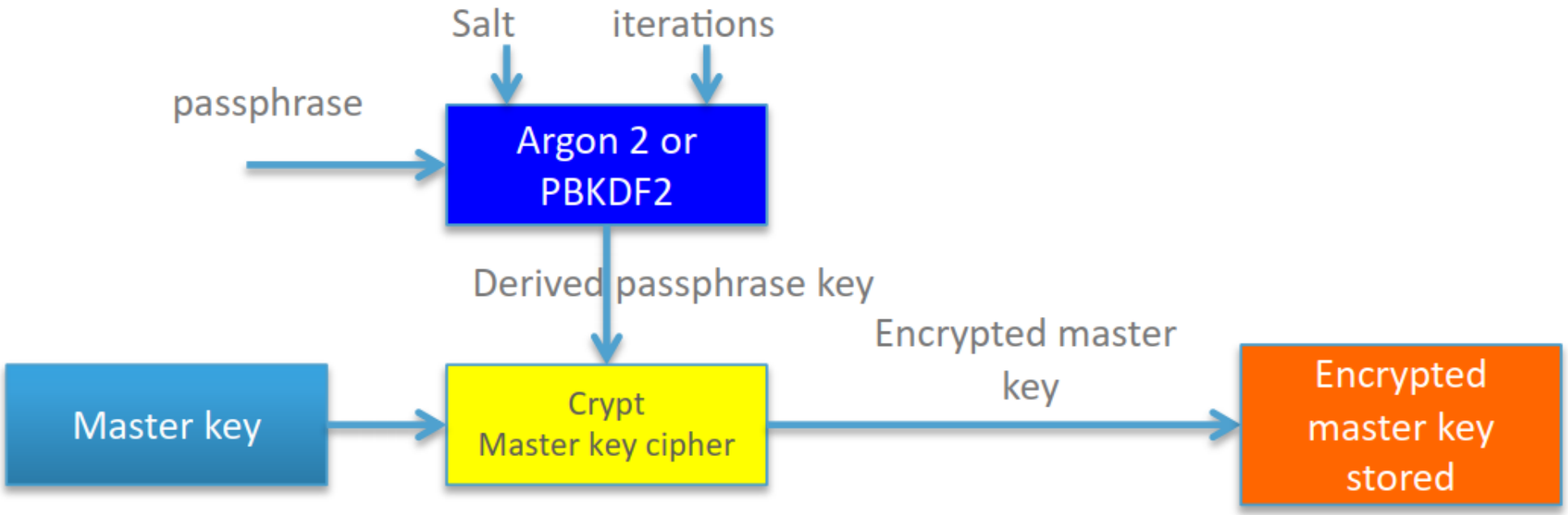
1.9.1 LUKS - key generation

- 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

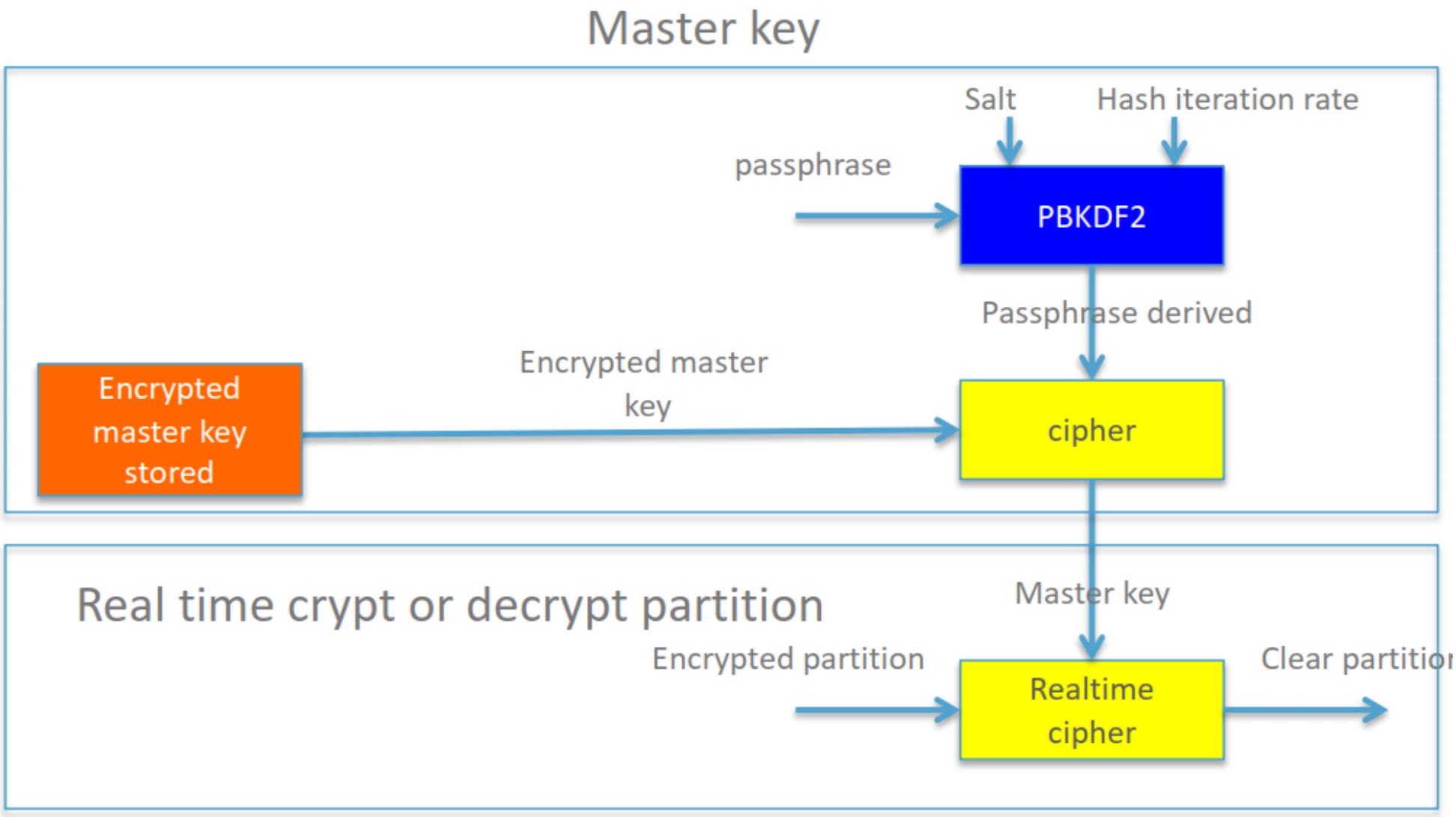
The system initialization is straight forward :

- 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




```
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
Dump the encrypted master key of a LUKS device
# cryptsetup luksDump -dump-master-key $DEVICE
Check /proc/crypto which contains supported ciphers and modes (but note it contains only currently loaded crypto API modules).
# cat /proc/crypto
```



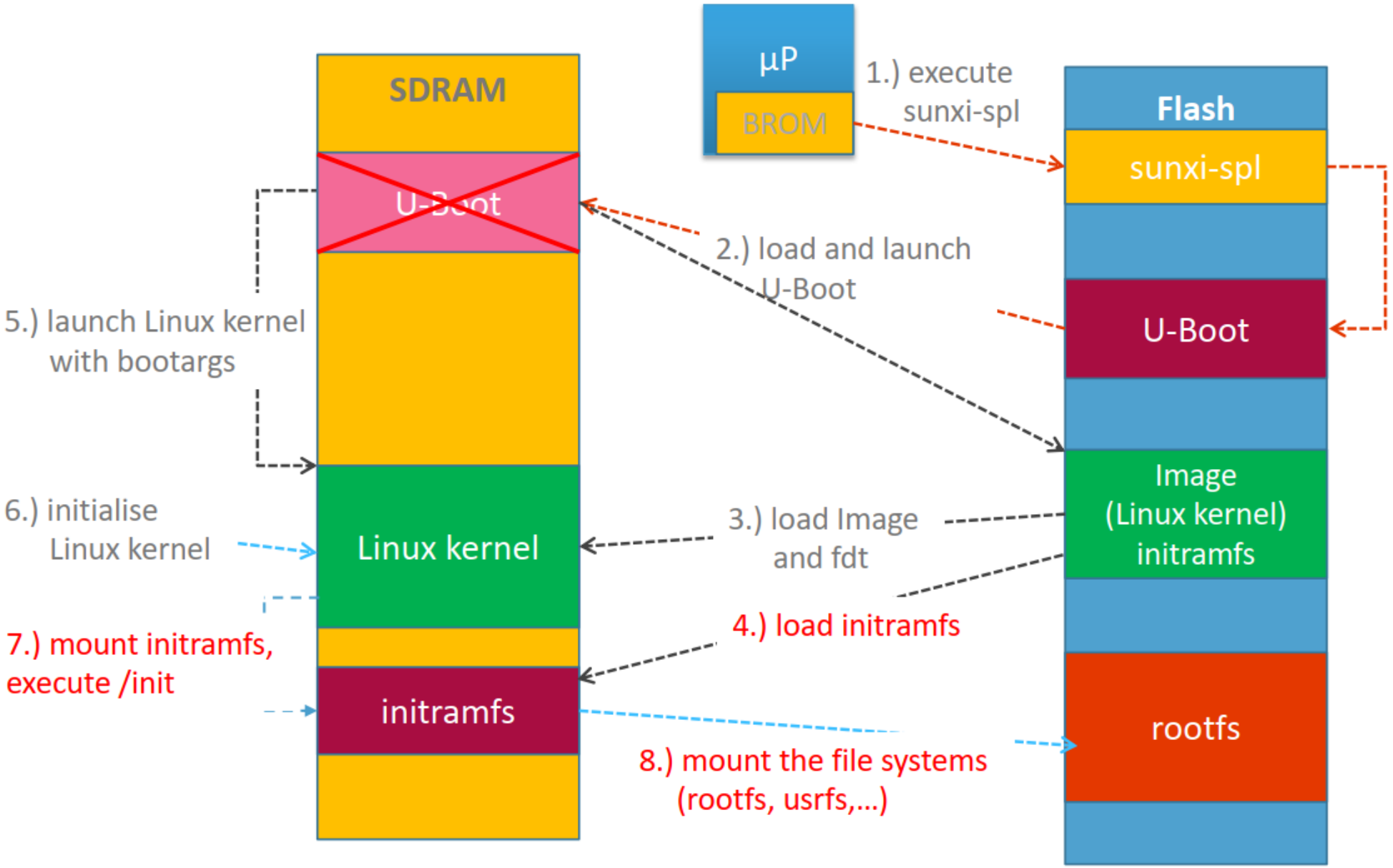
1.10 Caractéristiques de InitramFS et commandes

- 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.

1.10.1 Boot sequence, with initramfs



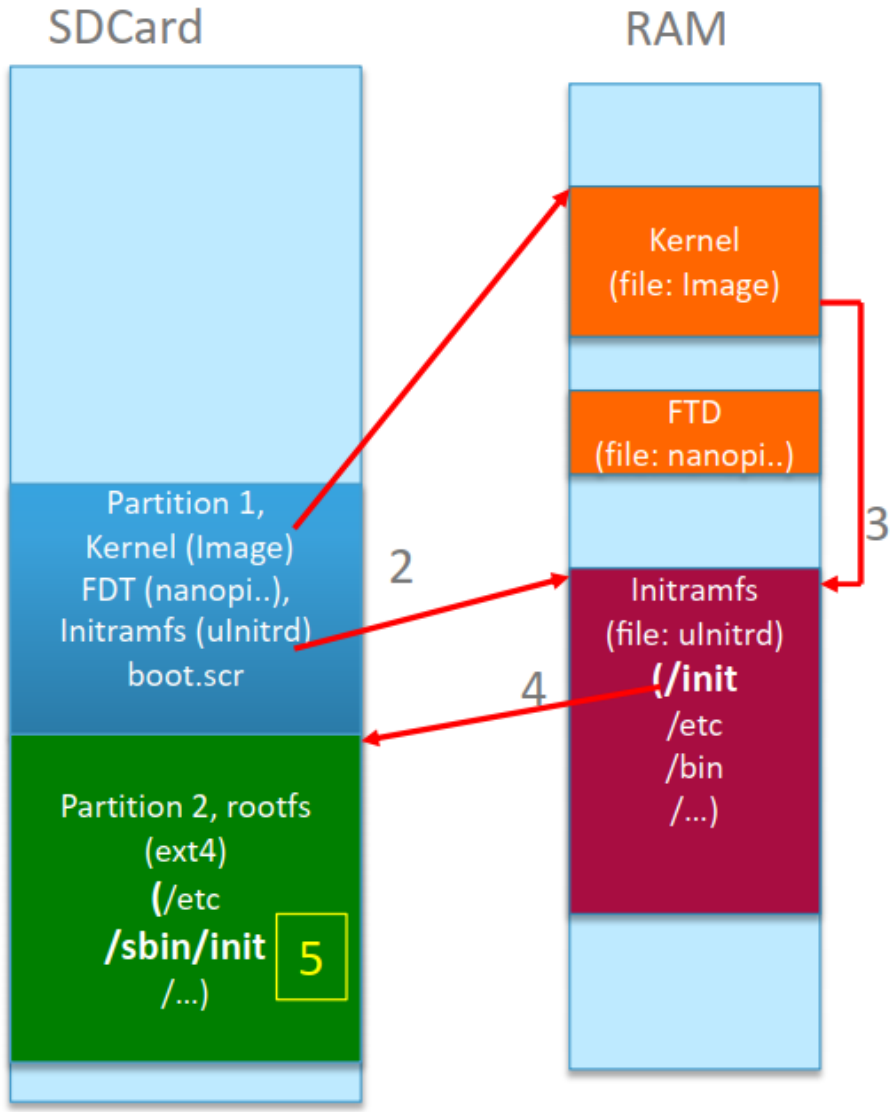
1) Kernel (Image), initramfs (uInitrd), flattened device tree (Sun50i…) and boot.scr files are located in the partition 1 of the SDCard

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

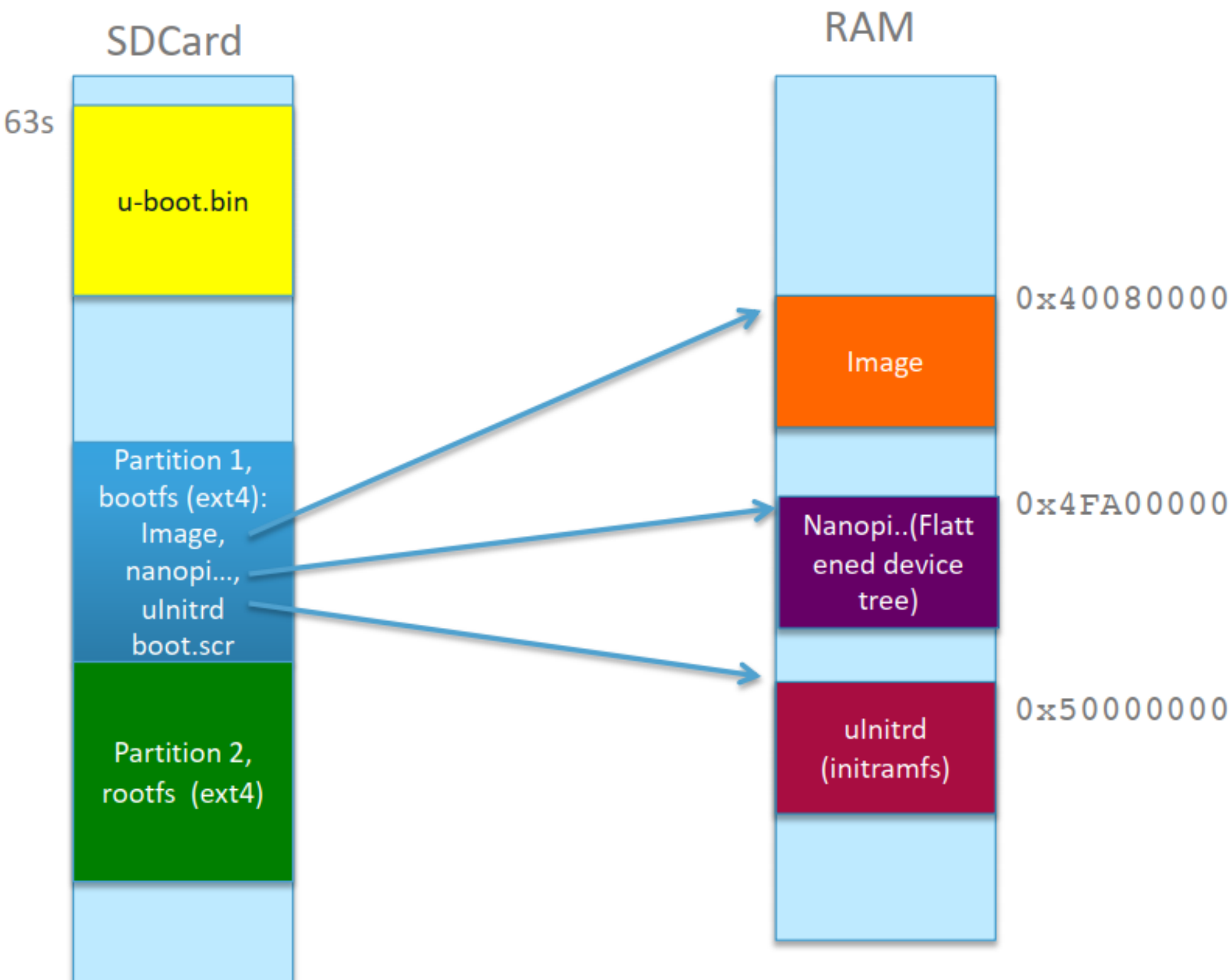


Show boot.cmd file： `cat $HOME/workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd`

```
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
```

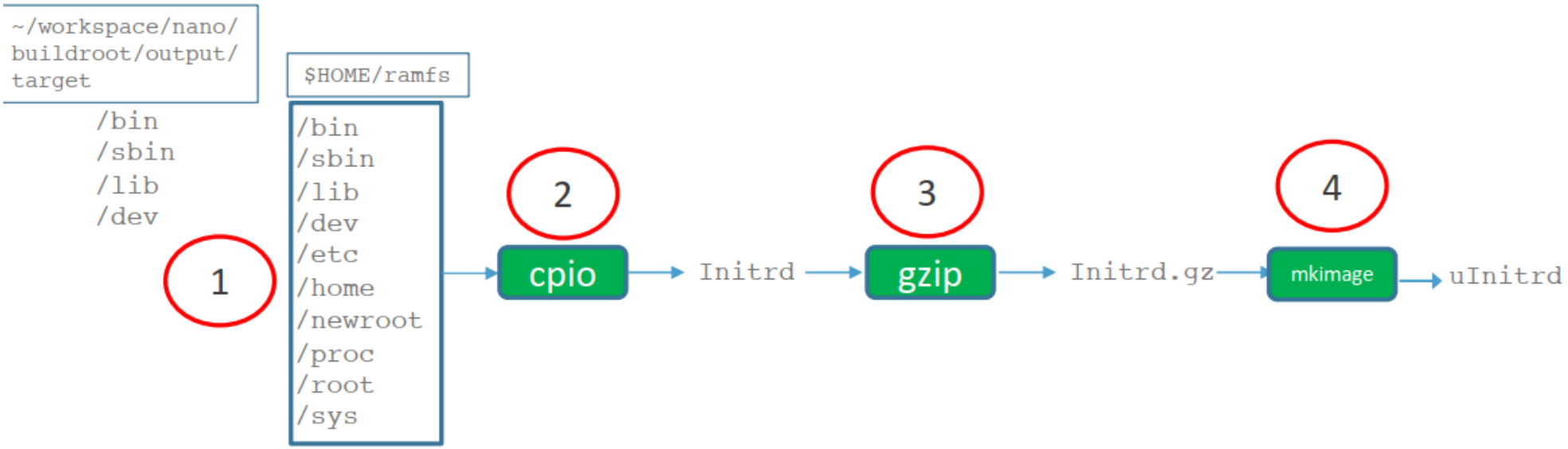
initramfs address


SDCard
RAM
63s
u-boot.bin
Partition 1,
bootfs (ext4):
Image,
nanopi...,
ulnitrd
boot.scr
Partition 2,
rootfs (ext4)
Image
Nanopi..(Flatt
ened device
tree)
ulnitrd
(initramfs)
0x40080000
0x4FA00000
0x50000000

1.11 Création d'un initramFS

On PC, NanoPi rootfs is in this directory: HOME/workspace/nano/buildroot/output/target/
Principle to build an initramfs :

1. Copy the right files into a directory ($HOME/ramfs)
2. Copy these files in a cpio archive file
3. Compress this file
4. Add the uboot header



1.12 Init script

```
#!/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
```

1.12.1 Shared library dependency

- 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**

```
#!/bin/sh
ROOTFSLOC=ramfs
HOME=/home/lmi
echo "---------------------- Begin ----------------------"
cd $HOME
mkdir $ROOTFSLOC
mkdir -p $ROOTFSLOC/{bin,dev,etc,home,lib,lib64,newroot,proc,root,sbin,sys}
echo "--------------------- Cpy /dev --------------------"
cd $ROOTFSLOC/dev
sudo mknod null c 1 3
...
sudo mknod mmcblk0p b 179 0
...
sudo mknod ttyS0 c 4 64
...
sudo mknod ttyS3 c 4 67
echo "--------------------- Cpy /bin --------------------"
cd ../bin
cp ~/workspace/nano/buildroot/output/target/bin/busybox .
ln -s busybox ls
...
ln -s busybox dmesg
cp ~/workspace/nano/buildroot/output/target/usr/bin/strace .
echo "------------------- Cpy /sbin -------------------"
cd ../sbin
ln -s ../bin/busybox switch_root
echo "--------------------- Cpy /lib64 ----------------"
cd ../lib64
...
cp ~/workspace/nano/buildroot/output/target/lib64/libc-2.31.so .
ln -s libresolv-2.31.so libresolv.so.2
ln -s libc-2.31.so libc.so.6
ln -s ../lib64/ld-2.31.so ld-linux-aarch64.so.1
echo "--------------------- Cpy /lib ----------------"
cd ../lib
cp ~/workspace/nano/buildroot/output/target/lib64/ld-2.31.so .
ln -s ../lib64/ld-2.31.so ld-linux-aarch64.so.1
echo "----------------- Create /init ----------------"
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
cd ..
sudo chown -R 0:0 $ROOTFSLOC
echo "-------------- cpio / gzip / mkimage ---------"
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
```

```
echo "----------------- DONE ----------------------"
```