REDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

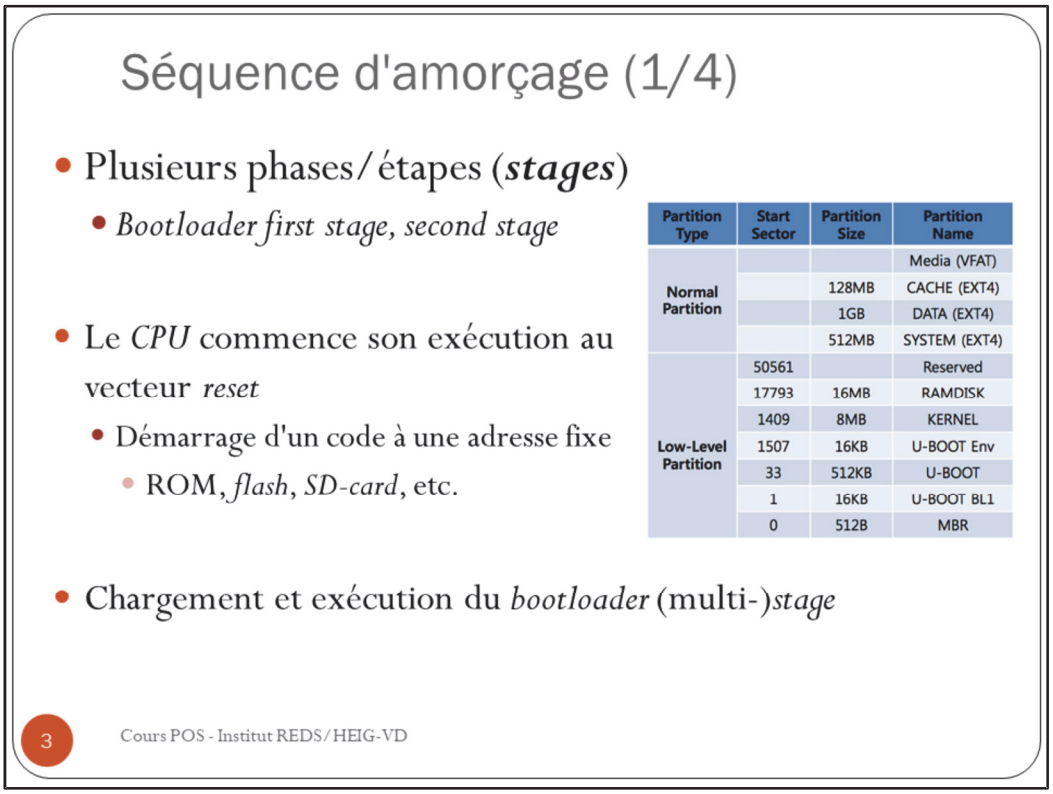
Portage de systèmes d'exploitation (POS)
Code d'amorçage

Prof. Daniel Rossier

Version 1.3 (2017-2018)

Plan

- Séquence d'amorçage
- Amorçage *U-boot*
- Amorçage *Linux*
- SO3



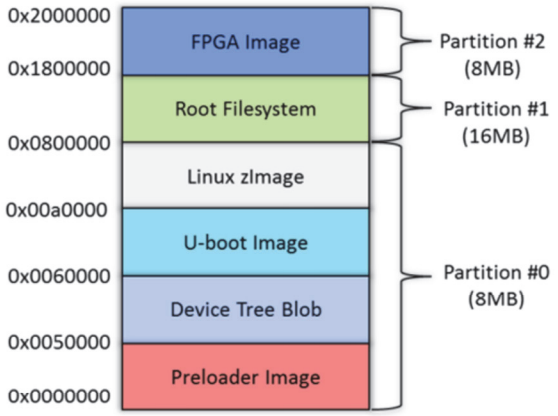
Chaque SoC présente des modes de démarrage différenst. La plupart offre la possibilité de démarrer le moniteur (*bootloader*) stocké en mémoire *flash*, sur un support de stockage de type SD/MMC, ou encore directement en mémoire moyennant l'utilisation d'un code spécial situé dans une ROM interne au SoC (*romcode, bootrom, etc.*).

Dans le cas d'un démarrage sur un support de stockage de type MMC, le microcode recherche le *bootloader* à un secteur précis.

Séquence d'amorçage (2/4)

- Exemple d'une organisation interne d'une *SD-card*

Spécification	Système de fichier
SD	FAT16 (et FAT12)
SDHC	FAT32
SDXC	exFAT



4 Cours POS - Institut REDS/HEIG-VD

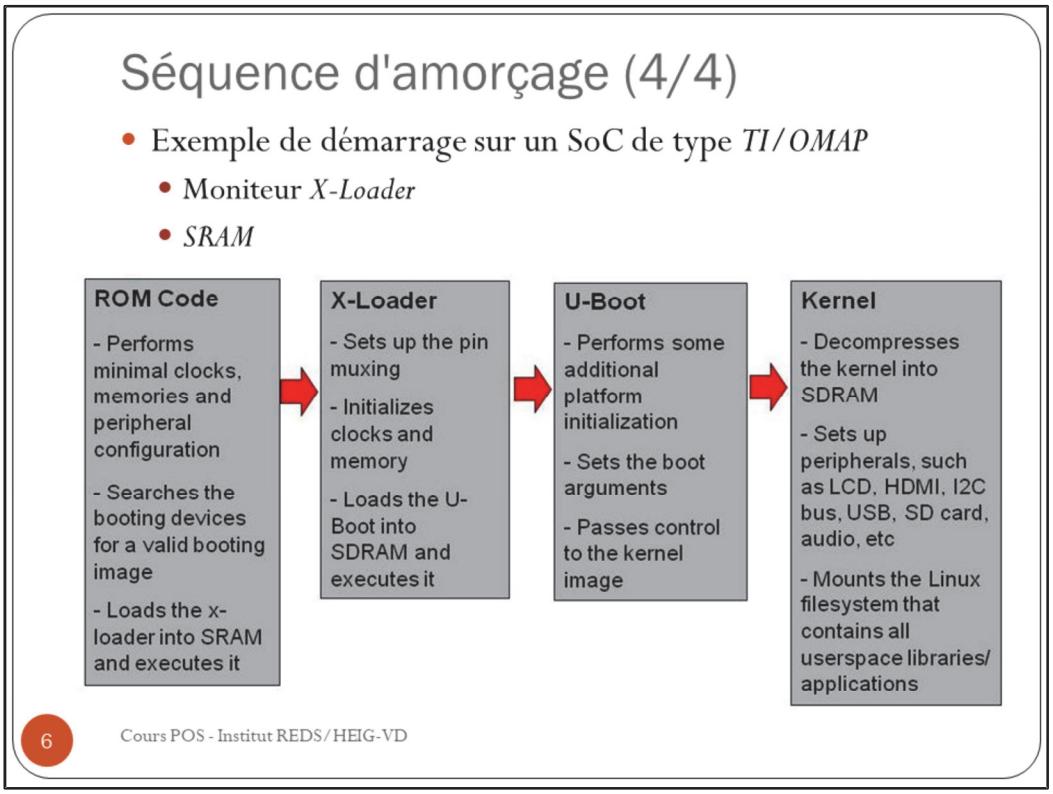
La technologie de *SD card* repose sur celui du support de type *NAND* (mémoire *flash*). Une carte SD dispose d'un microcontrôleur effectuant la translation entre un bloc logique et un bloc de type *flash*.

Les systèmes de fichiers supportés par les standards SD sont énumérés dans le tableau ci-dessus. D'autres formats peuvent être utilisés mais nécessitent la prise en charge du dispositif utilisant la *SD-card*.

La présence d'un *MBR* comme sur un disque dur permet le stockage de la table des partitions. C'est toujours le premier secteur de la carte SD (secteur 0) qui constitue par conséquent une zone critique.

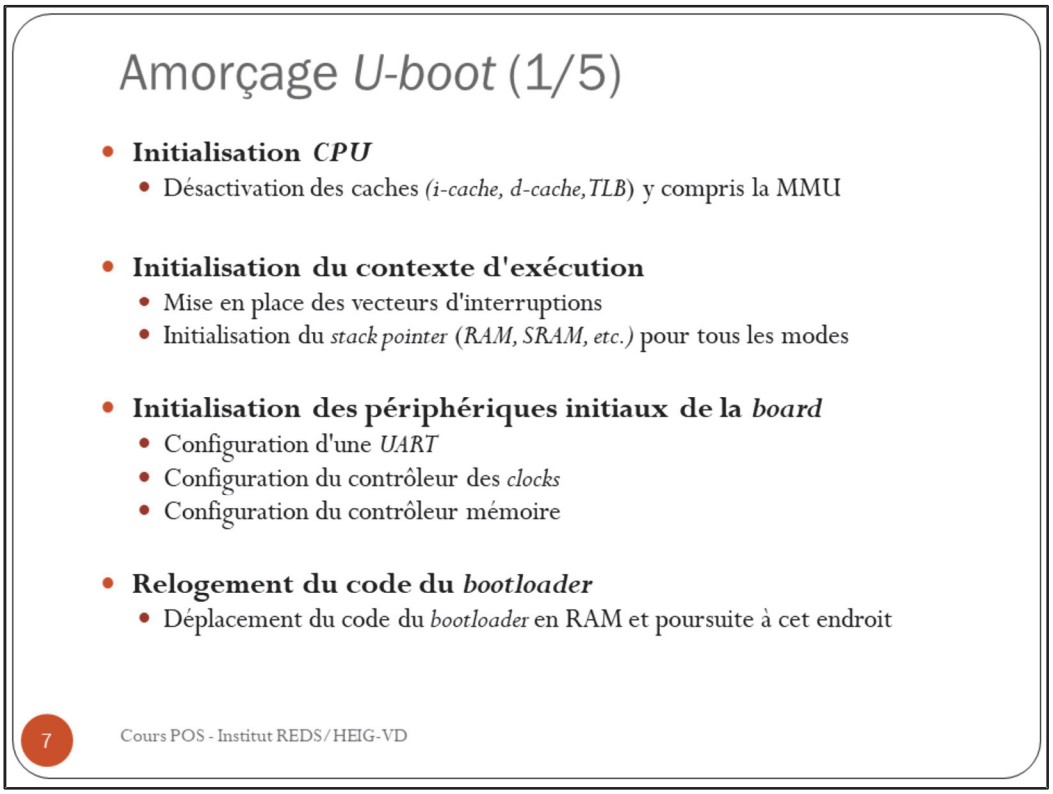
Séquence d'amorçage (3/4)

- Mode d'exécution
  - Au démarrage, mode *supervisor / kernel*
  - Passage au mode *user* à la fin du démarrage
- Piles (*stacks*)
  - Mise en place de la pile *système* (mode *SVC*)
  - Piles en fonction des modes (*ARM*)
- *Memory Management Unit (MMU)*
  - Initialisation des tables de page systèmes
  - Mappage linéaire initial
- Section *BSS*



La plupart des SoCs dispose d'une mémoire *SRAM* interne au microcontrôleur d'une taille d'env. de 64 KB. Cette mémoire peut être utilisée au démarrage du processeur afin d'y stocker un *bootloader* de première étape (*first stage*).

C'est le cas par exemple du moniteur *X-loader* dans les SoCs de type TI, qui est chargé en SRAM (depuis une *flash* ou carte *MMC*) permettant ainsi l'initialisation du contrôleur mémoire (DDRAM par ex.) et le chargement et l'exécution du moniteur de seconde étape.



L'initialisation des périphériques est minimale dans *U-boot*. Les horloges (*clocks*), le contrôleur mémoire et le contrôleur UART constituent l'ensemble minimal des périphériques à initialiser afin de disposer rapidement de messages de *logs* ainsi qu'une mémoire RAM permettant le chargement complet du *bootloader* puis d'un système d'exploitation ou des applications.

Le *bootloader U-boot* n'utilise pas la MMU. Toutes les adresses sont donc des adresses physiques.

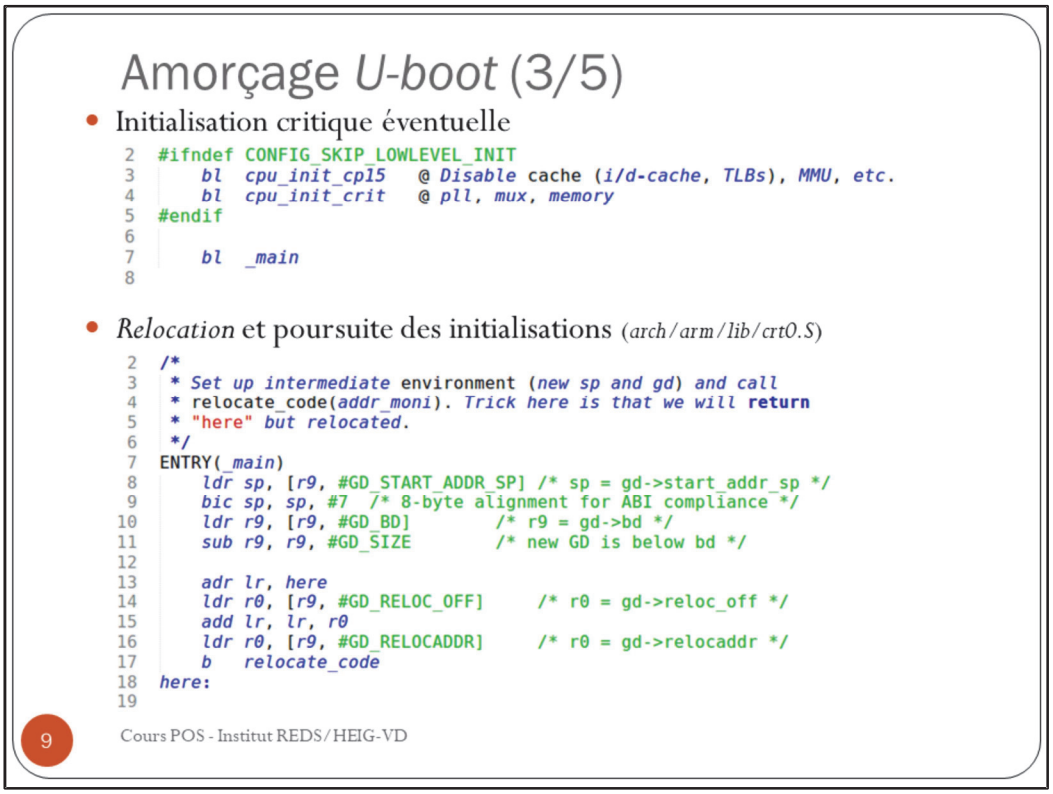
Amorçage *U-boot* (2/5)

- Démarrage de *U-boot* *(arch/arm/cpu/armv7/start.S)*

```
2  .globl _start
3  _start:
4    b      reset
5    ldr    pc, _undefined_instruction
6    ldr    pc, _software_interrupt
7    ldr    pc, _prefetch_abort
8    ldr    pc, _data_abort
9    ldr    pc, _not_used
10   ldr    pc, _irq
11   ldr    pc, _fiq
12
```

- Fonction associée au vecteur *reset*

```
2  reset:
3      bl  save_boot_params
4      /*
5       * disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
6       * except if in HYP mode already
7       */
8      mrs r0, cpsr
9      and r1, r0, #0x1f       @ mask mode bits
10     teq r1, #0x1a           @ test for HYP mode
11     bicne   r0, r0, #0x1f   @ clear all mode bits
12     orrne   r0, r0, #0x13   @ set SVC mode
13     orr r0, r0, #0xc0       @ disable FIQ and IRQ
14     msr cpsr,r0
15
```



Le relogement *(relocation)* du moniteur permet de le faire tourner en RAM et d'éviter des accès sur le support d'origine (mémoire *flash* par exemple).

Un exemple d'un tel code est montré ci-dessous :

```
3  ENTRY(relocate_code)
4      ldr r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
5      subs    r4, r0, r1          /* r4 <- relocation offset */
6      beq relocate_done           /* skip relocation */
7      ldr r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
8
9  copy_loop:
10     ldmia   r1!, {r10-r11}      /* copy from source address [r1]    */
11     stmia   r0!, {r10-r11}      /* copy to   target address [r0]    */
12     cmp r1, r2                  /* until source end address [r2]    */
13     blo copy_loop
```

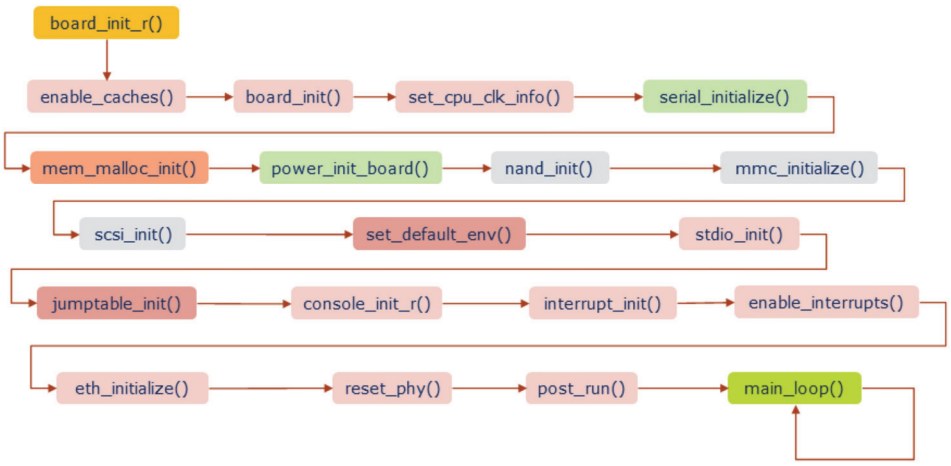
Amorçage *U-boot* (4/5)

- Préparation finale avant le code C

```
here:

  /* Set up final (full) environment */

  bl    c_runtime_cpu_setup /* we still call old routine here */

  ldr   r0, =__bss_start    /* this is auto-relocated! */
  ldr   r1, =__bss_end      /* this is auto-relocated! */

  mov   r2, #0x00000000 /* prepare zero to clear BSS */

clbss_l:
  cmp   r0, r1              /* while not at end of BSS */
  strlo r2, [r0]            /* clear 32-bit BSS word */
  addlo r0, r0, #4          /* move to next */
  blo   clbss_l

  bl coloured_LED_init
  bl red_led_on

  /* call board_init_r(gd_t *id, ulong dest_addr) */
  mov     r0, r9            /* gd_t */
  ldr   r1, [r9, #GD_RELOCADDR] /* dest_addr */

  /* call board_init_r */
  ldr   pc, =board_init_r   /* this is auto-relocated! */

  /* we should not return here. */
```

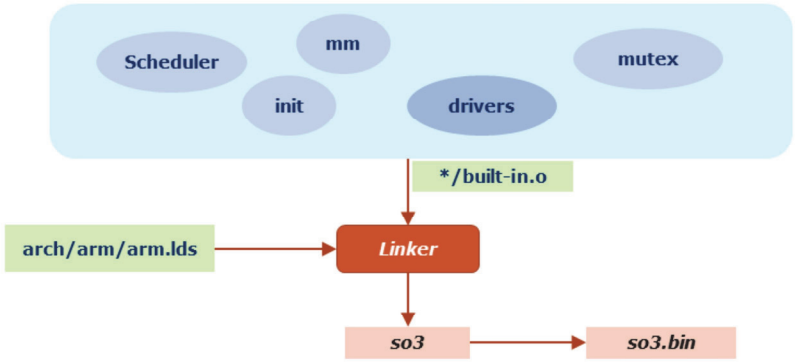
Amorçage *U-boot* (5/5)

- Initialisation avancée et finale de l'environnement (*soft* & *hard*) dans la fonction *board_init_r (r: RAM)* *(arch / arm / lib / board.c)*



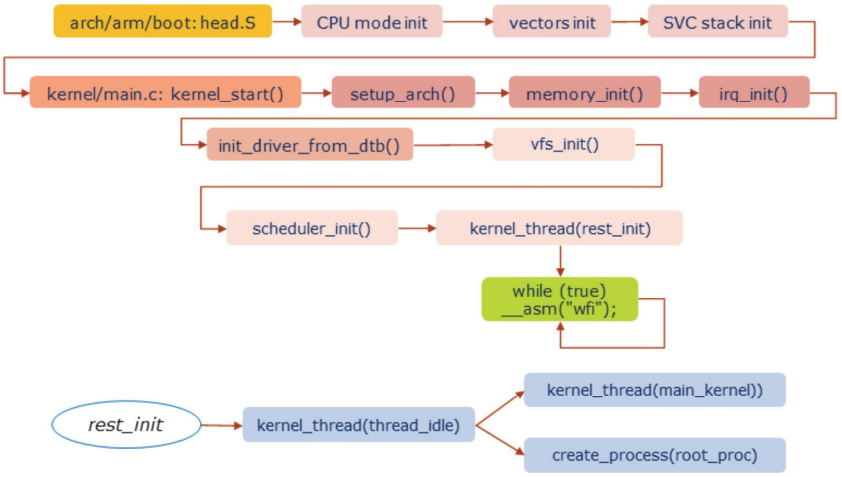
S03 (1/2)

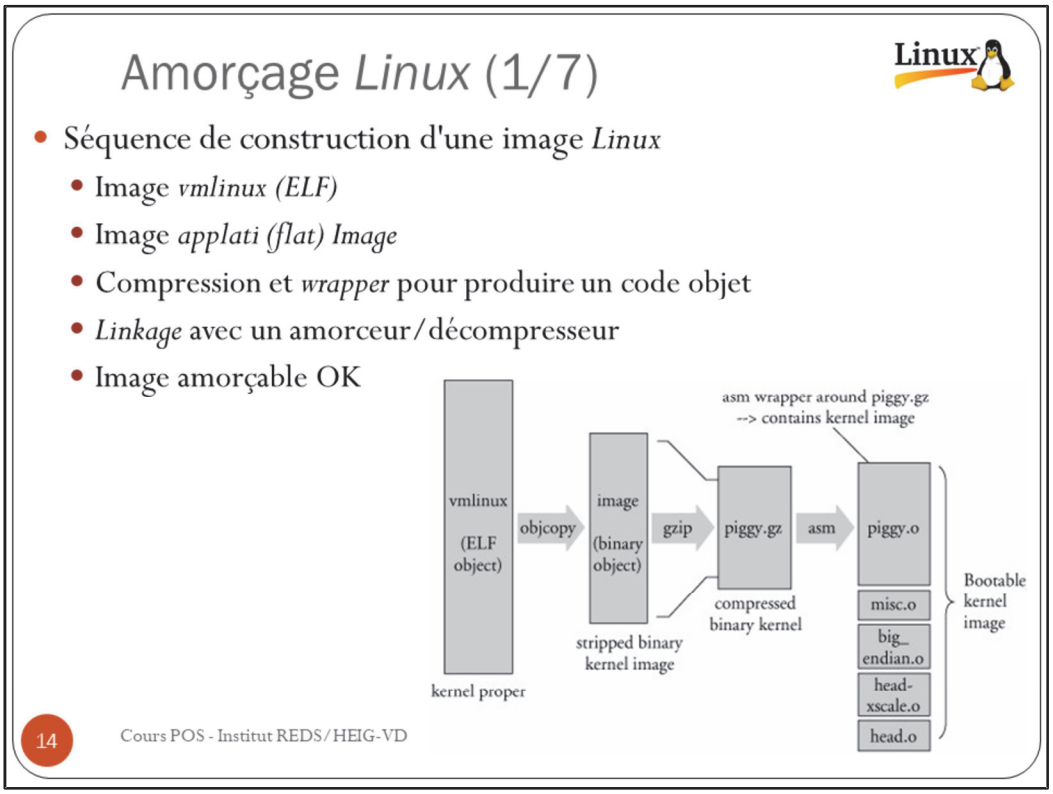
- SOO® Operating System
- Version libre du système d'exploitation
- *Build system* basé sur celui de *Linux*



S03 (2/2)

- Séquence de démarrage dans *S03*



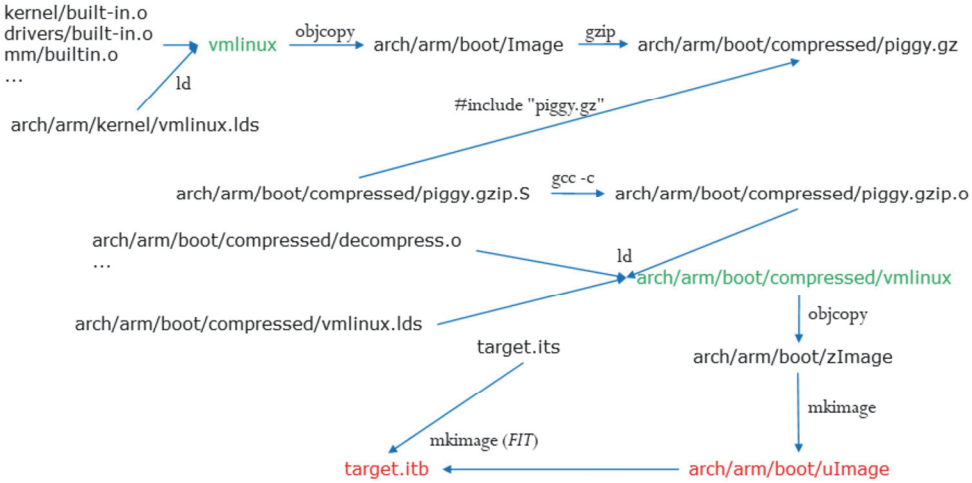


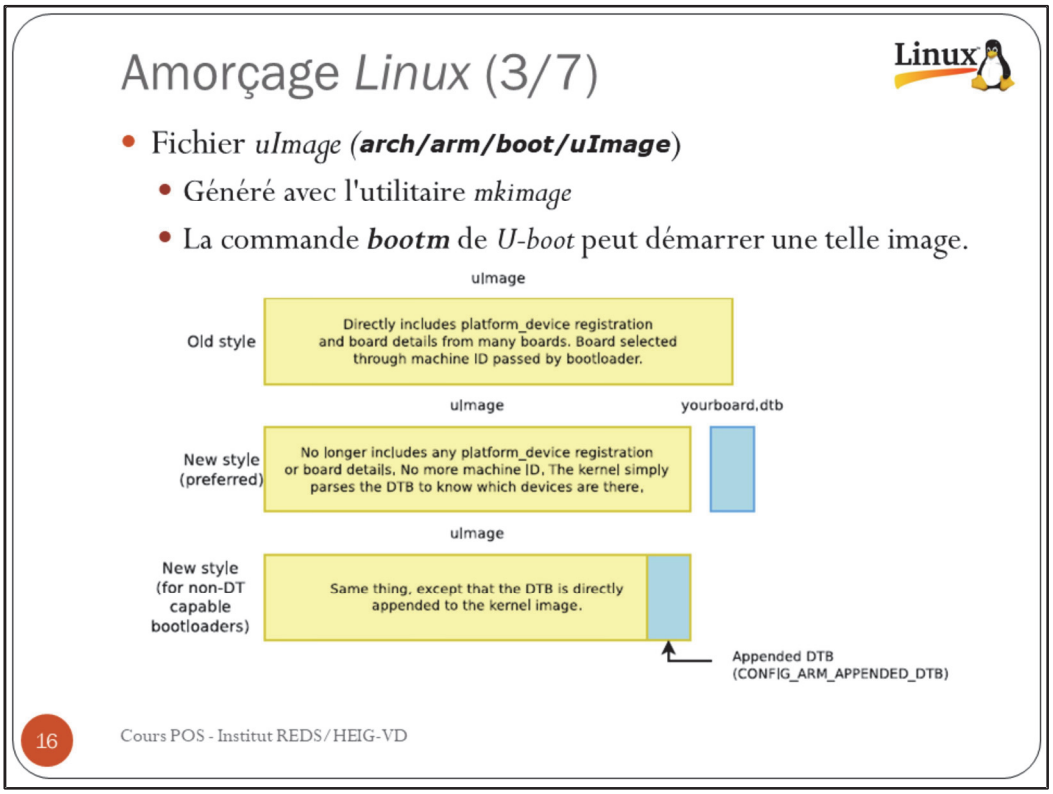
La construction d'une image de noyau *Linux* commence par la compilation de l'ensemble des fichiers sources et le *linkage* des fichiers de type code objet produits. Il en résulte une image binaire en format *ELF* qui se trouve à la racine de l'arborescence du noyau. Puis cet image est aplati (avec *objcopy*) afin d'avoir un code non relogeable aux adresses définitifs.

Cette image produite est ensuite compressée (*gzip*) et contenu dans un *wrapper* (fonction contenant simplement l'image compressé) de telle sorte qu'un code de décompresseur et d'implantation puisse être exécuté au démarrage; l'astuce étant de *linker* ce *wrapper* avec ce code d'amorçage primaire afin d'avoir un binaire exécutable (qui pourra à son tour être applati ou non selon que le *bootloader* puisse interpréter du code *ELF*).

Amorçage *Linux* (2/7)

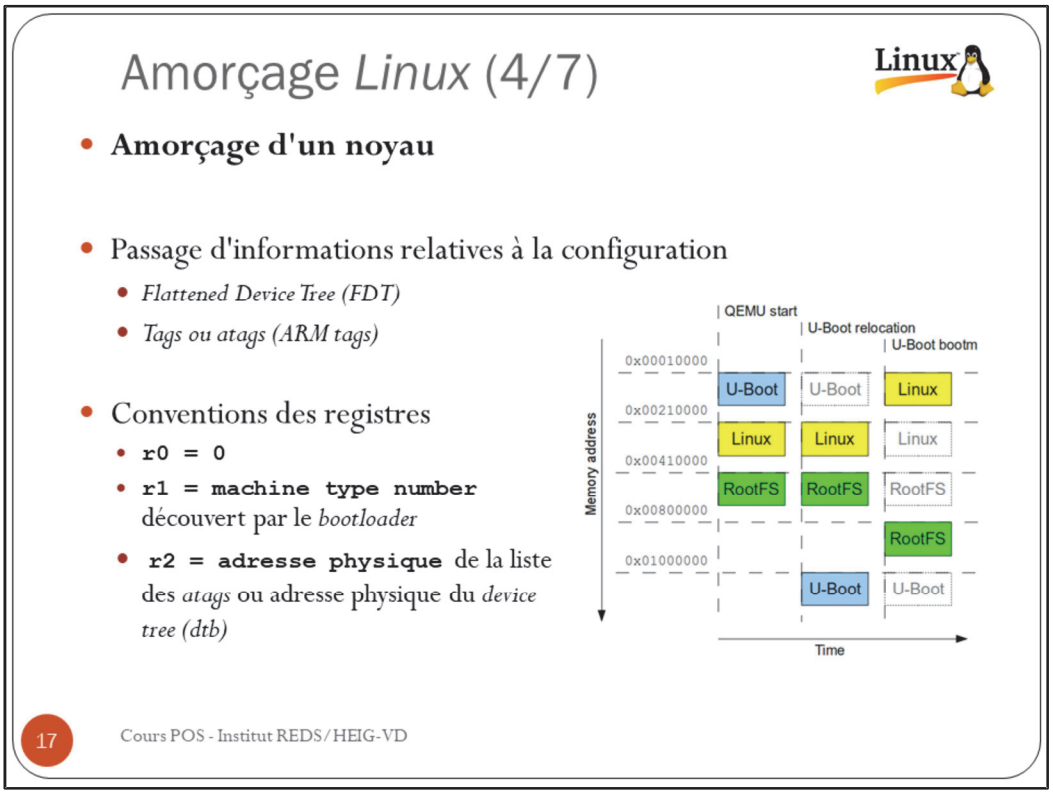
- Processus de génération de l'image binaire





Avec l'apparition du *device tree* dans *Linux*, *U-boot* peut amorcer le démarrage du noyau selon différentes méthodes en tenant compte la séparation entre l'image noyau et le *blob*, comme illustré ci-dessus.

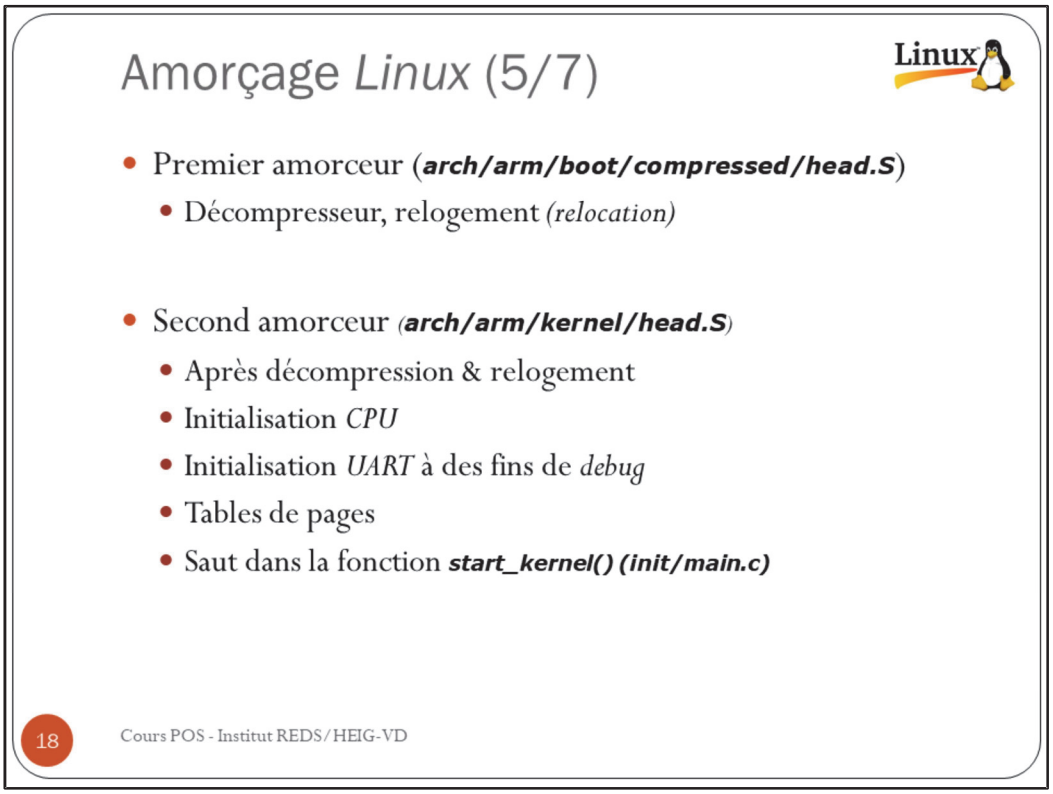
Si aucun moniteur n'est présent, ou que la version de *U-boot* ne permet pas de gérer le *device tree* de manière séparée, le noyau *Linux* peut toujours être compilé avec une option particulière permettant *d'appondre* le *blob* à l'image noyau. Le noyau sera capable ensuite de localiser le *blob* et d'y accéder.



Lorsqu'un noyau est démarré par *U-boot*, celui-ci peut lui transmettre des arguments liés à la configuration machine/noyau. Les méthodes de passage d'arguments varient d'un OS à l'autre, et il est bien entendu crucial de savoir comment récupérer ces arguments lorsqu'on effectue un portage de noyau. Avec *Linux*, il existe principalement deux méthodes pour passer des arguments au noyau : ce sont les ***device tree*** et les ***atags***.

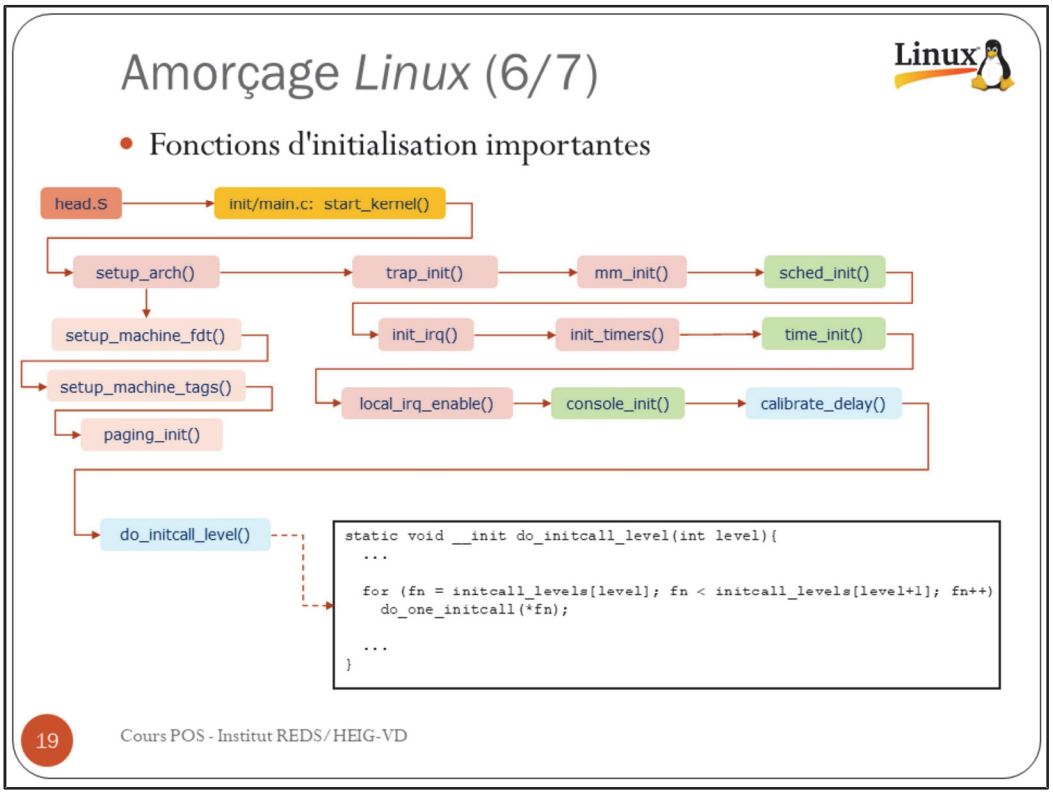
Les *atags* sont des paramètres passés par le *bootloader* (ou *qemu* si aucun *bootloader* n'est utilisé) au noyau *Linux*. Les *atags* sont gérés typiquement à l'aide d'un structure semblable à celle-ci :

```
struct tag {
  struct tag_header hdr;
  union {
    struct tag_core         core;
    struct tag_mem32        mem;
    struct tag_videotext    videotext;
    struct tag_ramdisk      ramdisk;
    struct tag_initrd       initrd;
    struct tag_serialnr     serialnr;
    struct tag_revision     revision;
    struct tag_videolfb     videolfb;
    struct tag_cmdline      cmdline;
    struct tag_custom custom;
  } u;
};
```



L'initialisation de l'*UART* peut s'effectuer très tôt au démarrage du noyau grâce à des macros et du code assembleur présents dans le fichier *arch/arm/kernel/debug.S*
Cette initialisation permet d'obtenir rapidement des messages de *log* lors du *bootstrap* du noyau. Ce code est spécifique au microcontrôleur et effectue un mappage rapide (virtuel/physique) des registres de l'UART.

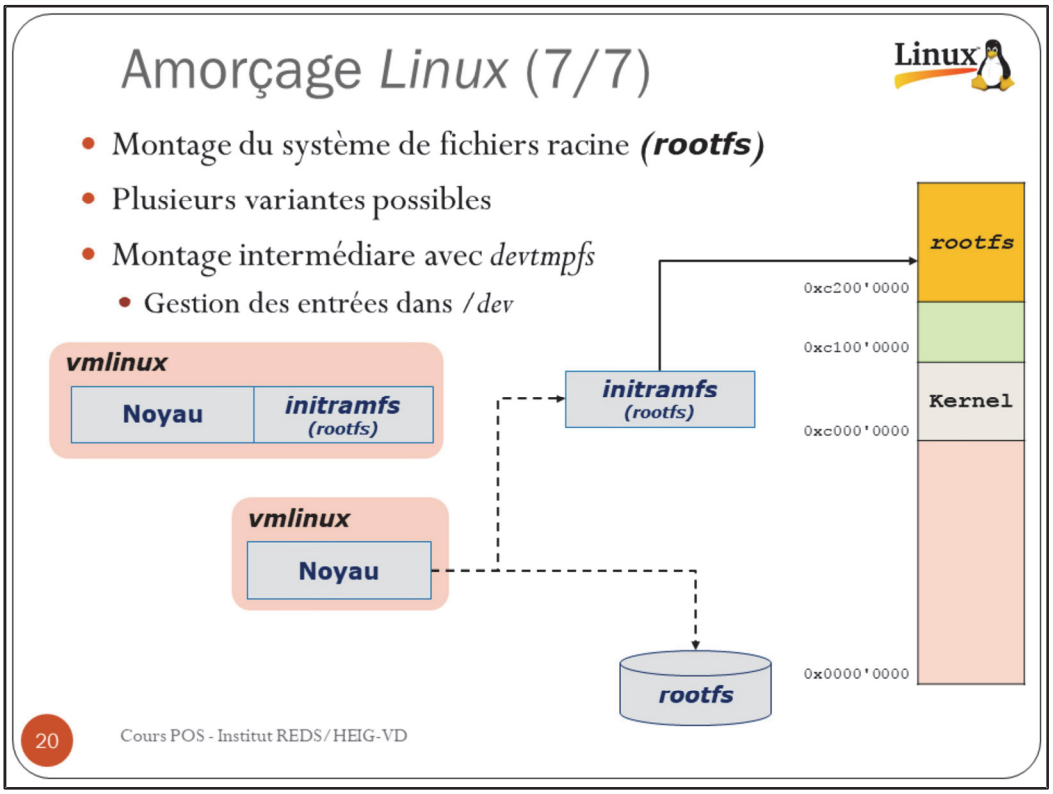
Par la suite, une première initialisation des tables de page permet au noyau d'activer rapidement la MMU et de poursuivre avec le *bootstrap.* Sur ARM typiquement, les premiers méga-octets de la mémoire RAM dans laquelle se trouve le code du noyau sont mappés sous forme de section de 1 MB (en principe les quatre premiers mégas sont mappés permettant de poursuivre l'exécution du code d'initialisation).



Les *initcalls* sont des fonctions qui ont été *linkées* statiquement avec le noyau et qui peuvent se trouver dans n'importe quel sous-système y compris les *drivers*. Ces fonctions effectuent des initialisations diverses et sont simplement déclarées en tant que telle dans le noyau à l'aide de macros prédéfinies (*device_initcall()* par exemple).

Il existe plusieurs types de telles macros permettant de définir quand la fonction de type *initcall* devra être appelée durant le processus de *bootstrap*. L'ensemble des fonctions correspondantes à un type d'*initcall* sont enregistrées à partir de leur adresse (pointeur sur fonction) **dans une section** réservée à cet effet dans le binaire.

L'ensemble des fonctions d'un type particulier seront ensuite appelées sous forme de *callback* durant le processus de démarrage à des instants différents.



Plusieurs scénarios de montage du *rootfs* (système de fichiers racine) existe dans *Linux.*

Il existe principalement trois variantes :

a) Montage en mémoire RAM avec un *rootfs* intégré dans l'image du noyau lors de sa génération. Le format du *rootfs* peut être de type *cpio*.

b) Montage en mémoire RAM avec un *rootfs* externe à l'image du noyau. Il peut également être dans un format de type *cpio* mais stocké dans une mémoire secondaire (*flash, SD,* etc.)

c) Montage du *rootfs* se trouvant dans une mémoire de stockage secondaire

Dans le cas des scénarios a) et b), le *rootfs* est aussi appelé *initrd* et peut constituer un premier système de fichier racine limité en taille. Ces scénarios nécessitent un système de fichiers intermédiaire de type *devtmpfs* afin que le noyau puisse créer des entrées dans */dev* requises lors de l'utilisation de *drivers*. Ces variantes permettent l'utilisation d'applications et/ou de modules nécessaire au montage d'un *rootfs* plus conséquent sur un support de stockage externe.

Références

- Pierre Ficheux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)
- Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. http://free-electrons.com