

# Une approche synchrone à la conception de systèmes embarqués temps réel

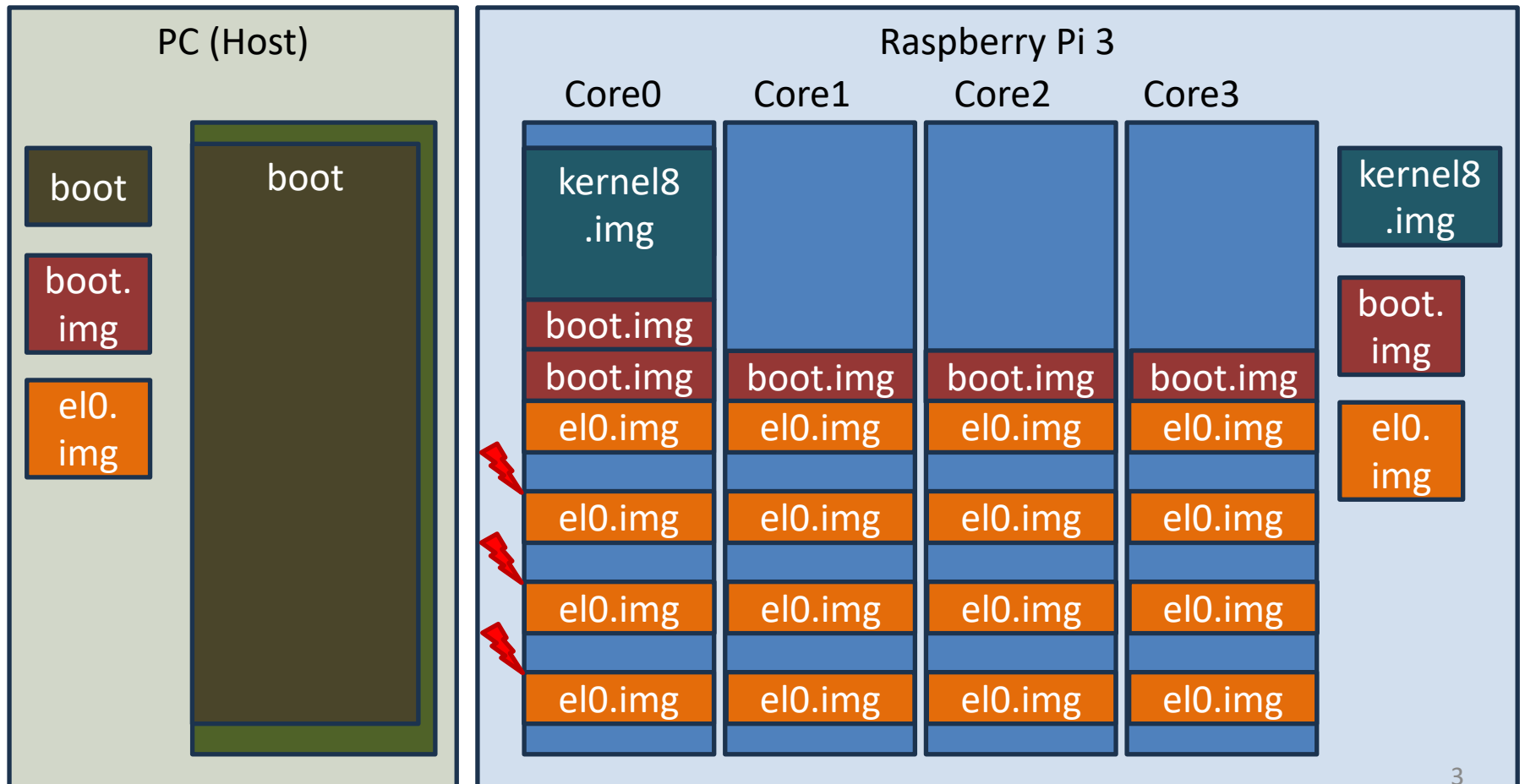
Dumitru Potop-Butucaru  
dumitru.potop@inria.fr  
cours EPITA, 2024, 6<sup>ème</sup> séance

# Contenu de ce cours

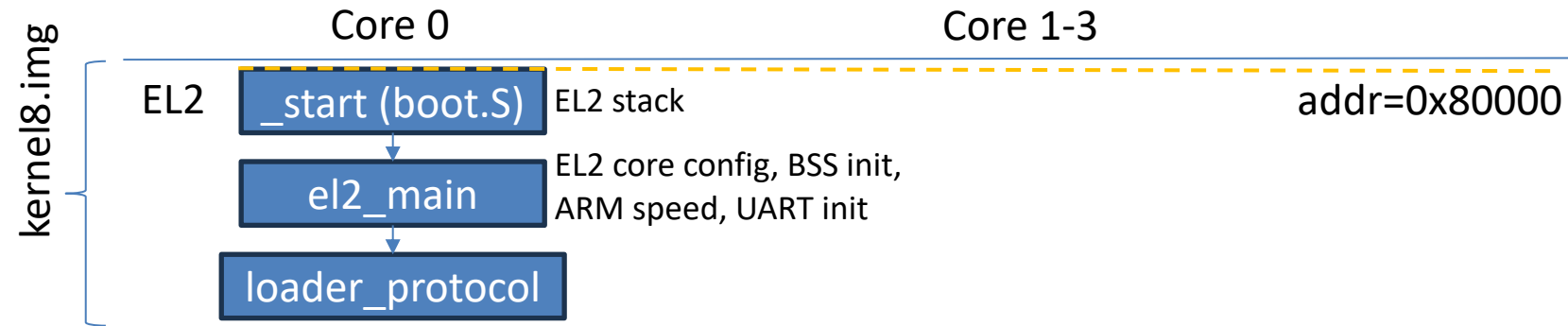
- Implantation multi-cœurs (2<sup>e</sup> partie)
  - Organisation d'un exécutif cyclique simple
  - Interruptions, niveaux de privilèges
  - Synchronisation
    - Avec le temps réel (timers)
    - Entre cœurs
      - Sémaphores
      - Accès mémoire à effet de barrière
- Préparation du TP

# Exécution multi-cœurs

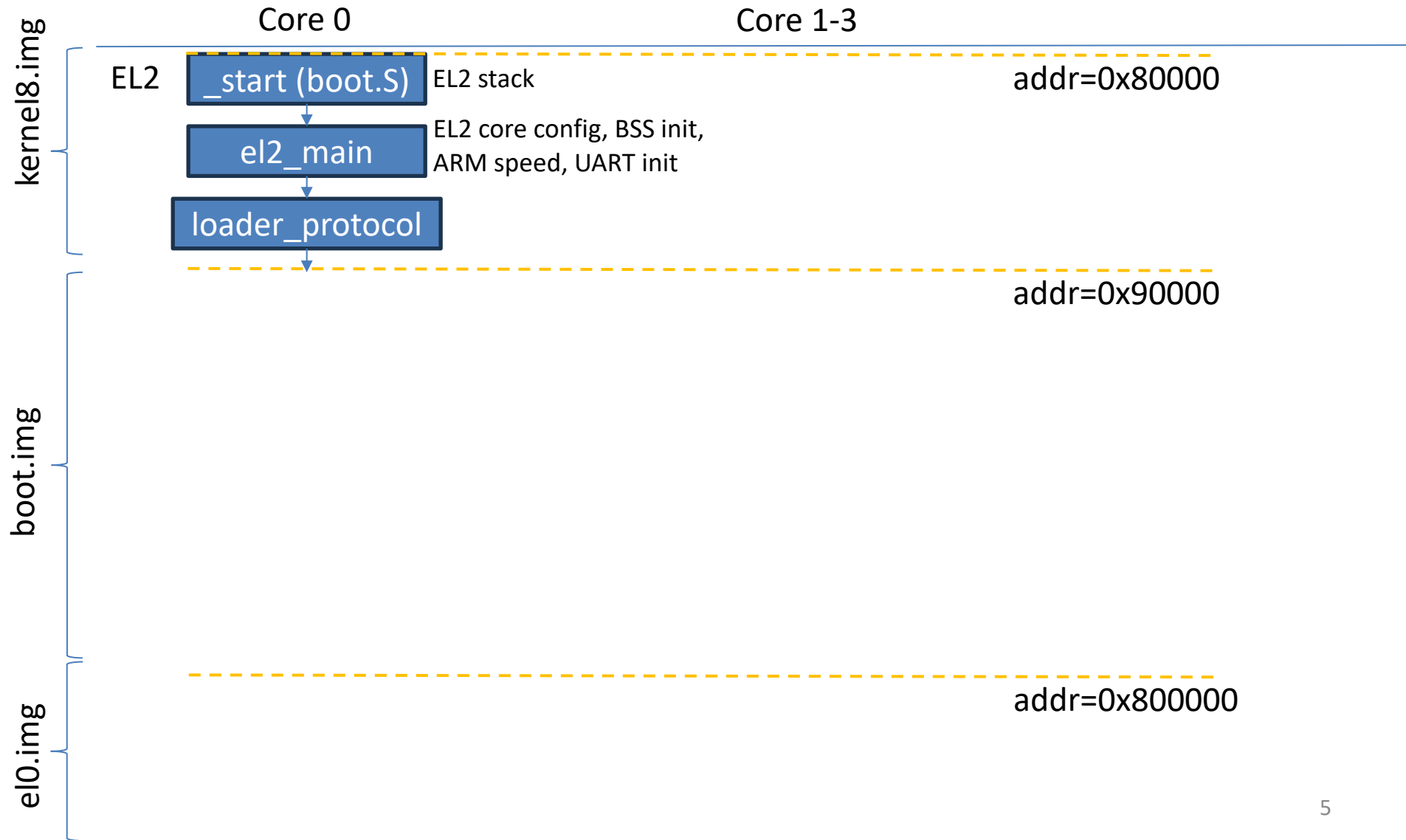
- Déclenchement timer (période = 1s)



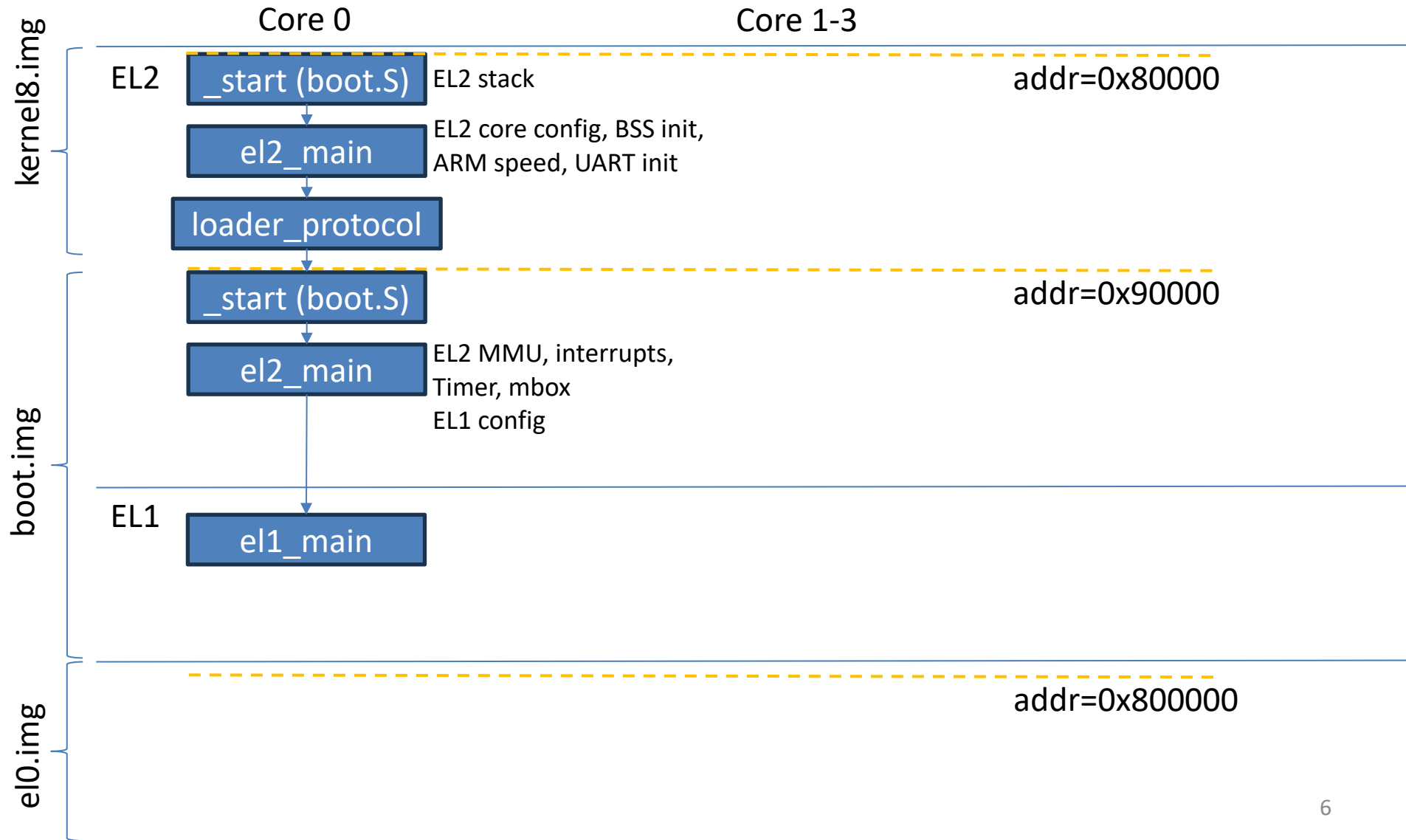
# Démarrage (détailé)



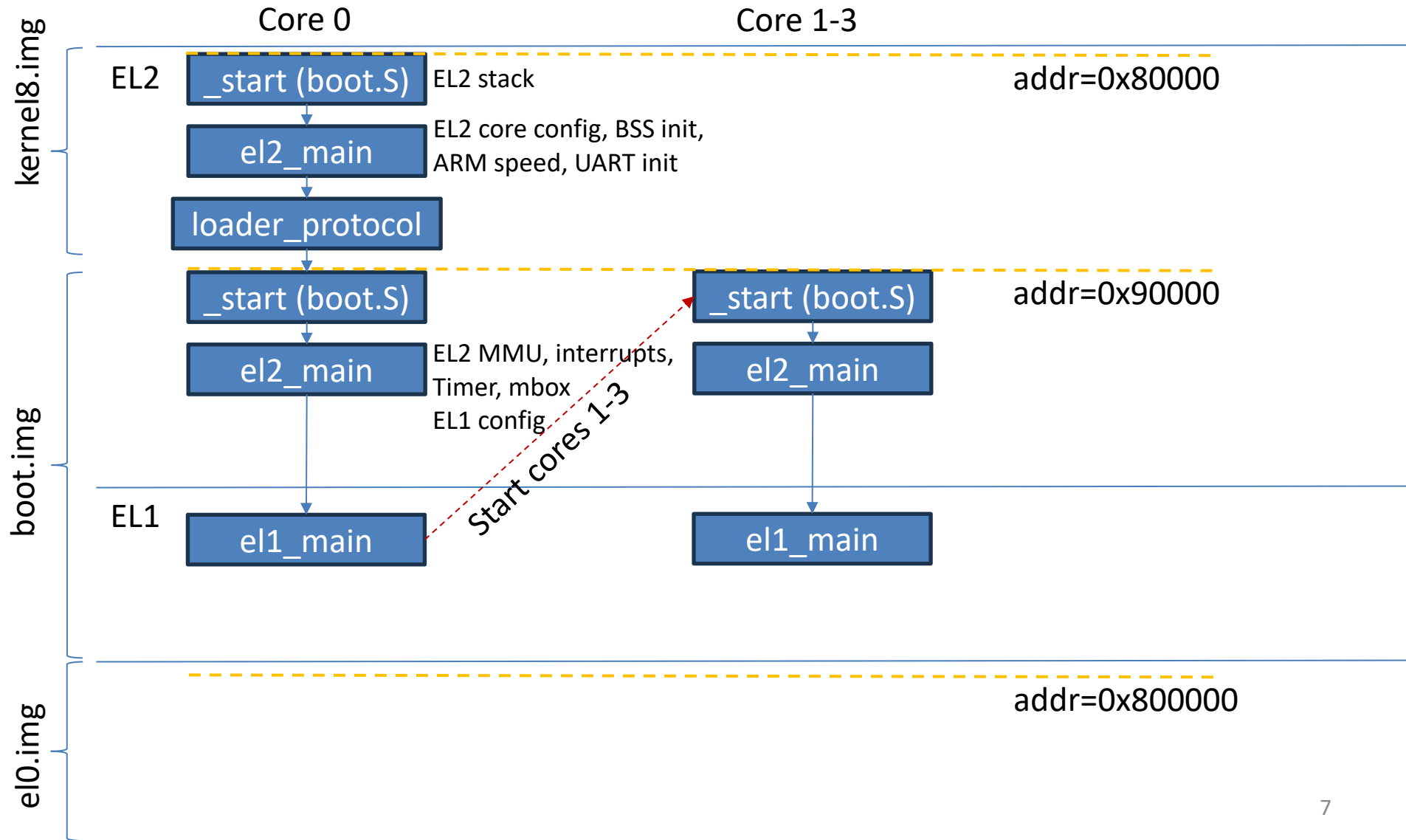
# Démarrage (détailé)



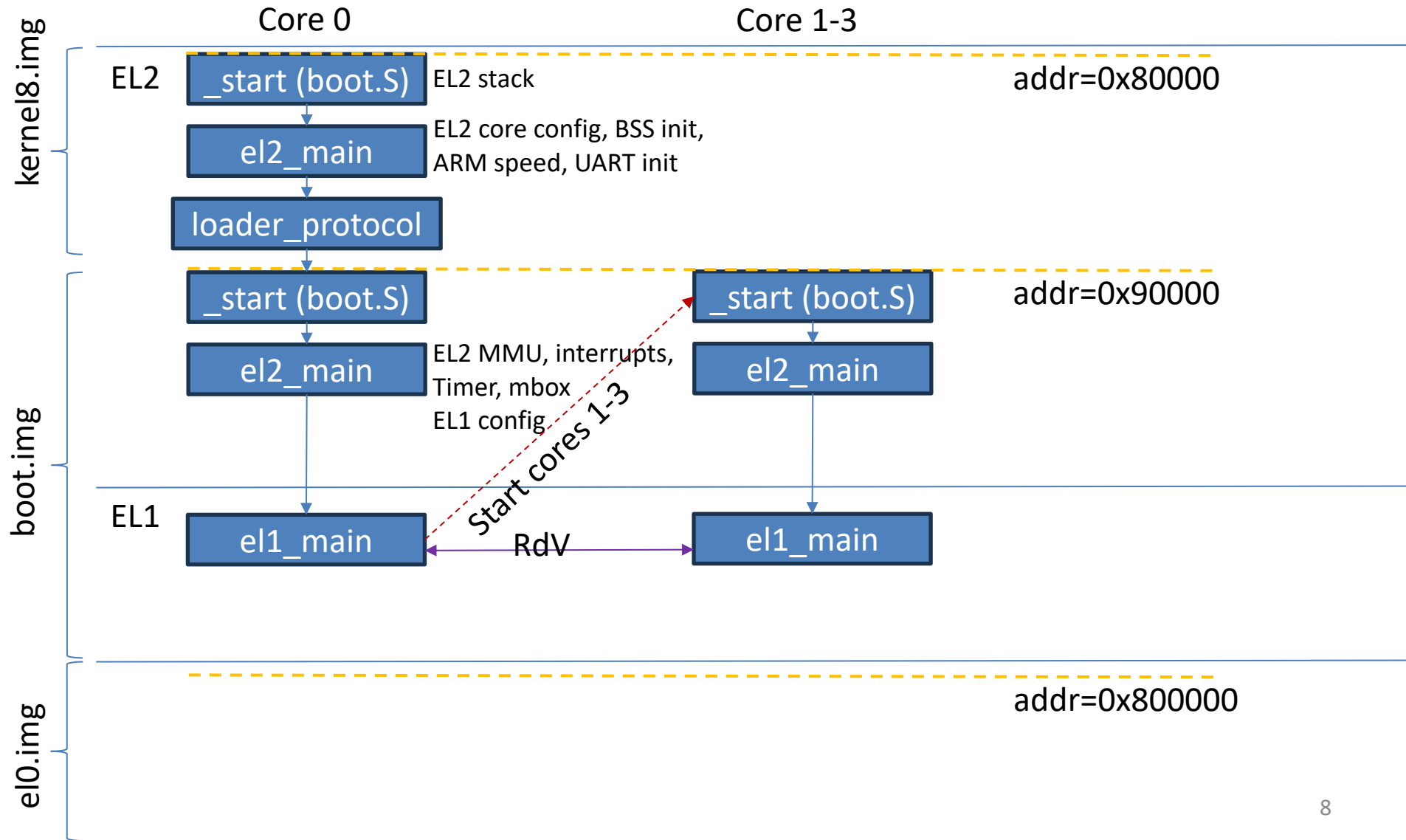
# Démarrage (détailé)



# Démarrage (détailé)

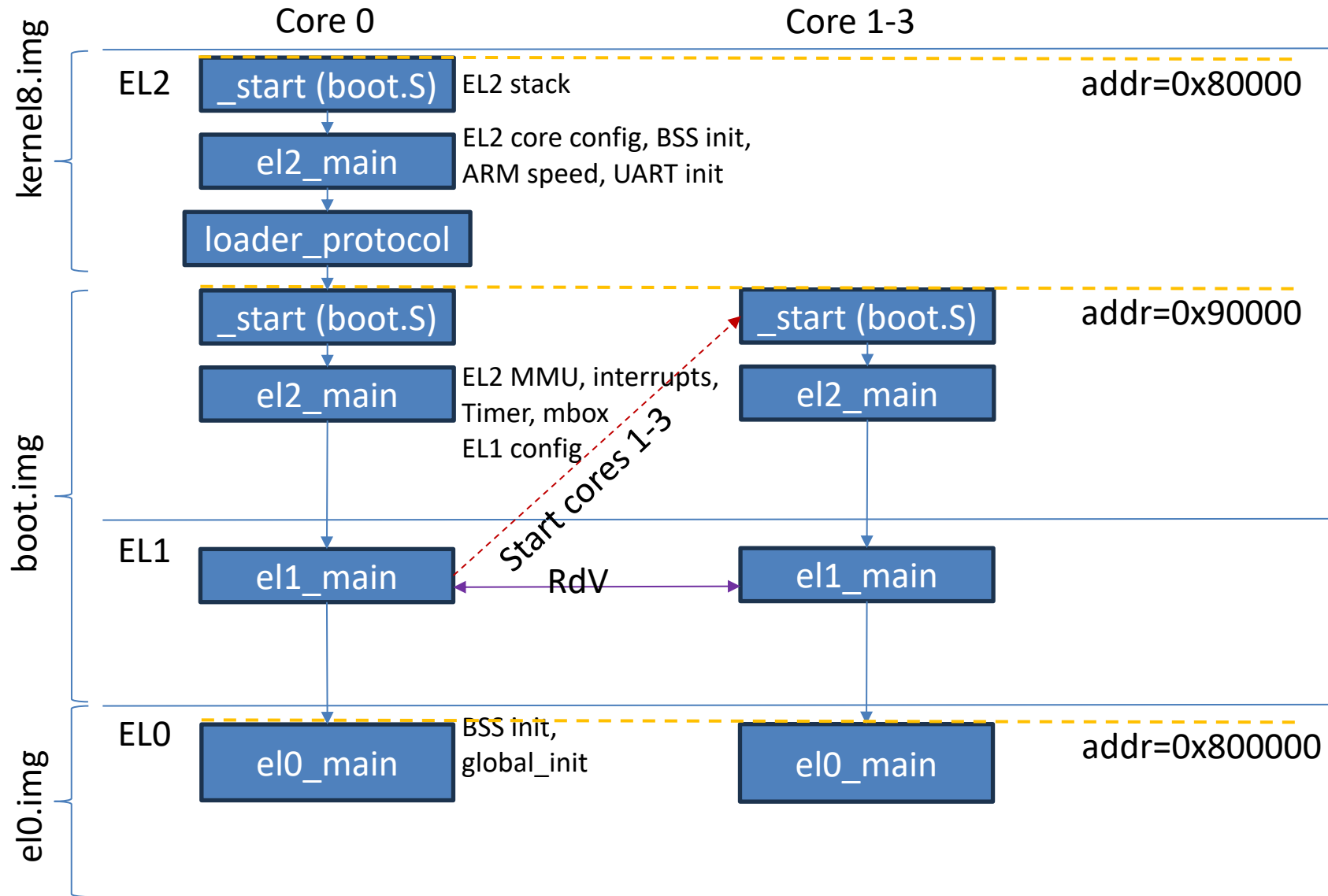


# Démarrage (détailé)

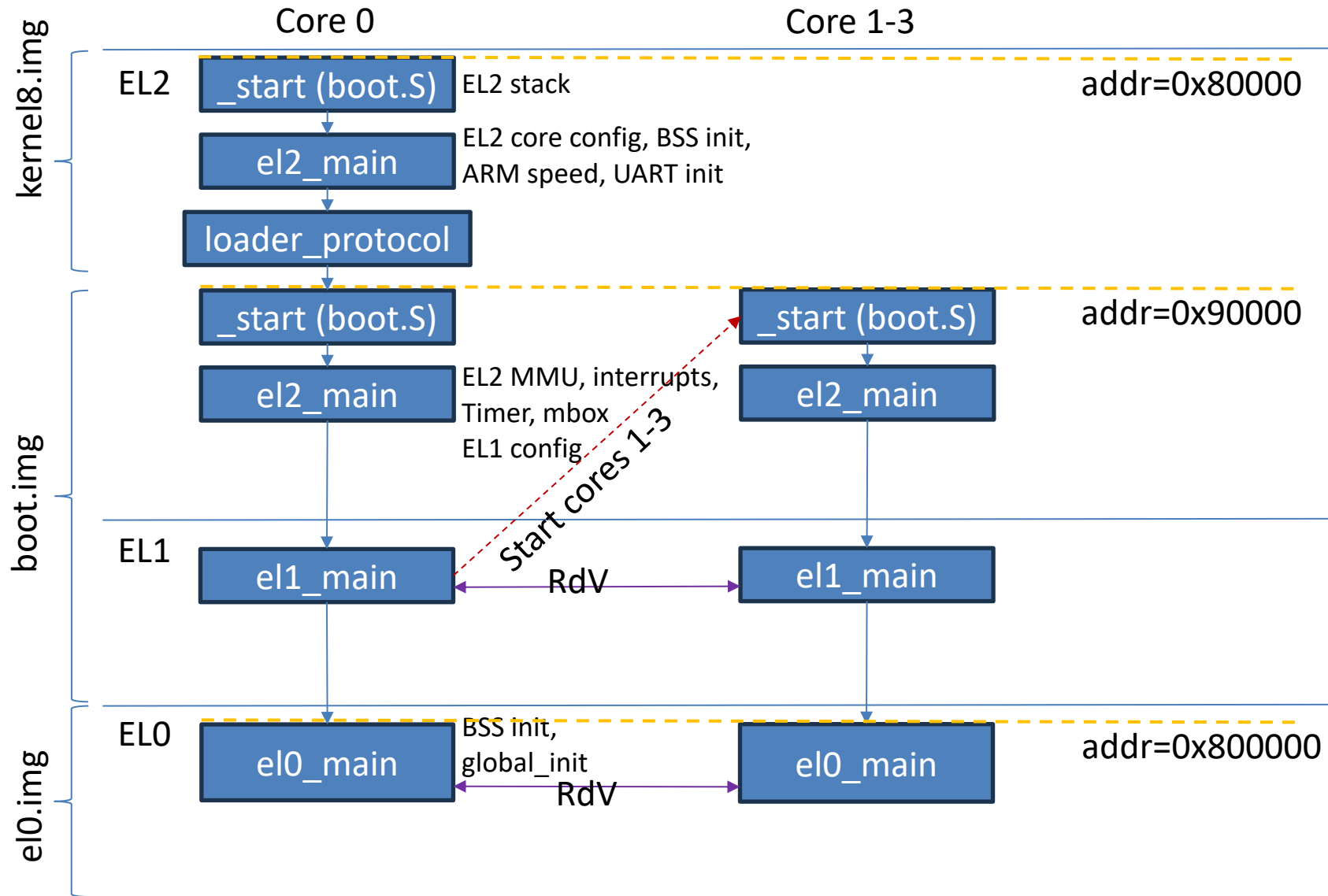




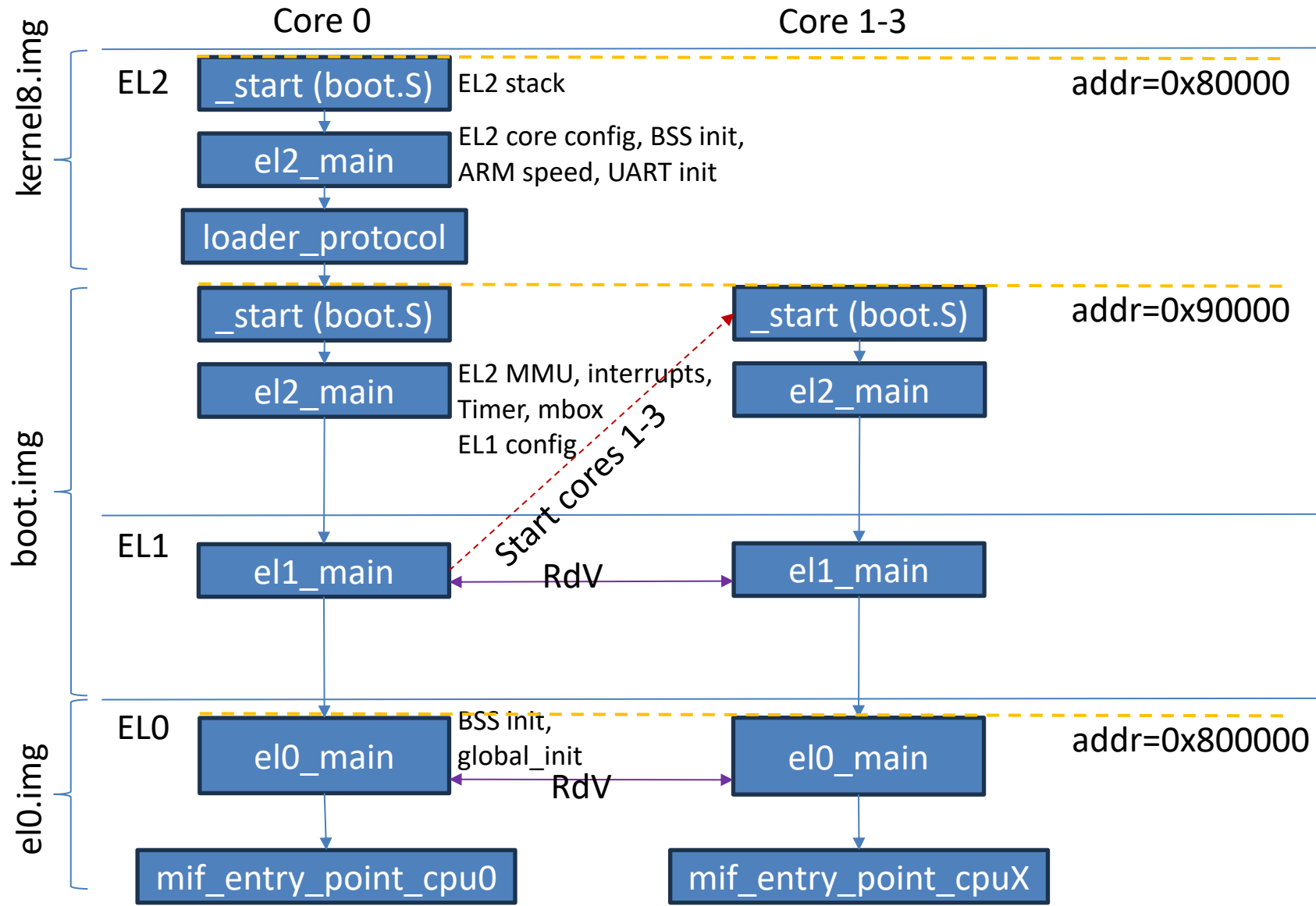
# Démarrage (détailé)



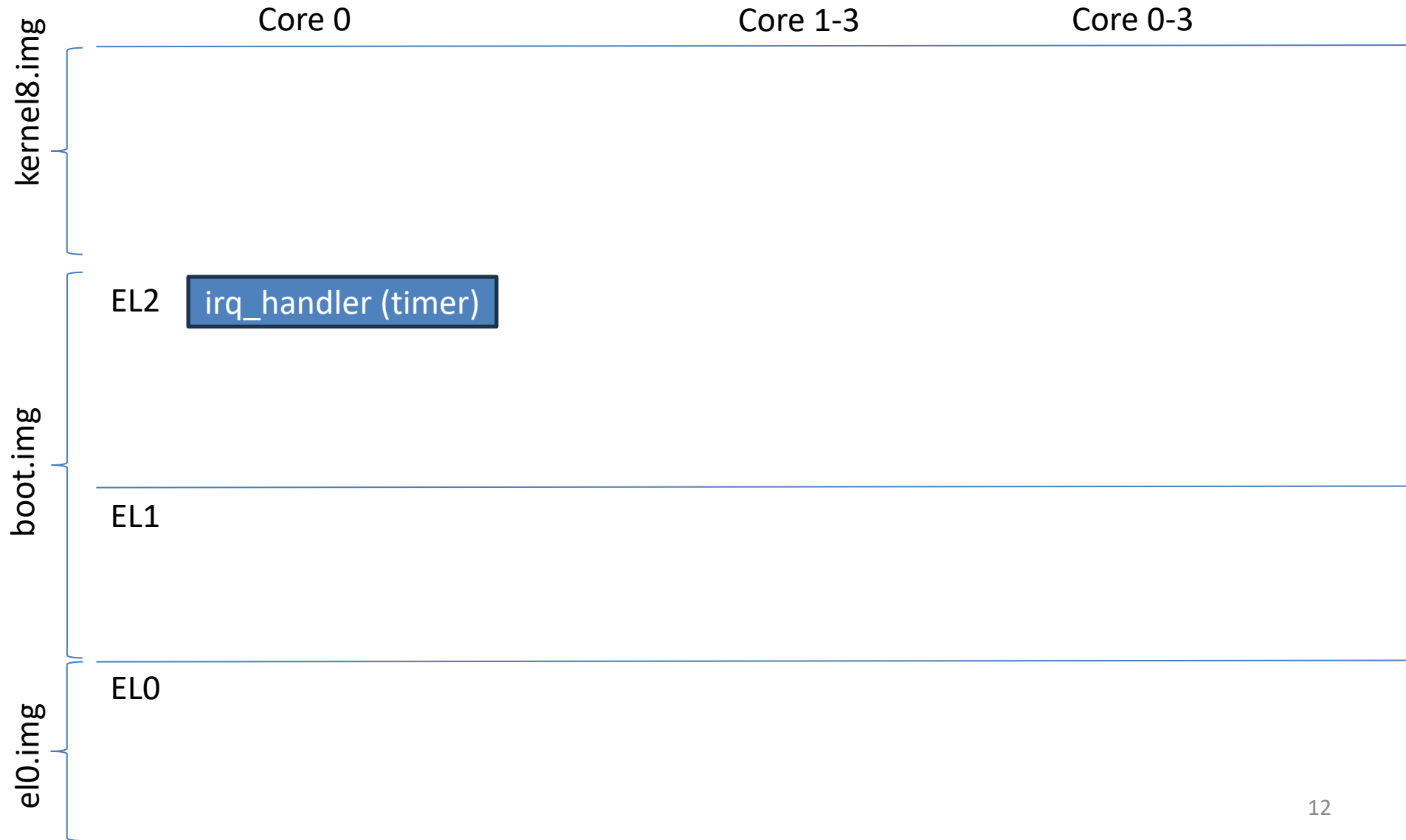
# Démarrage (détailé)



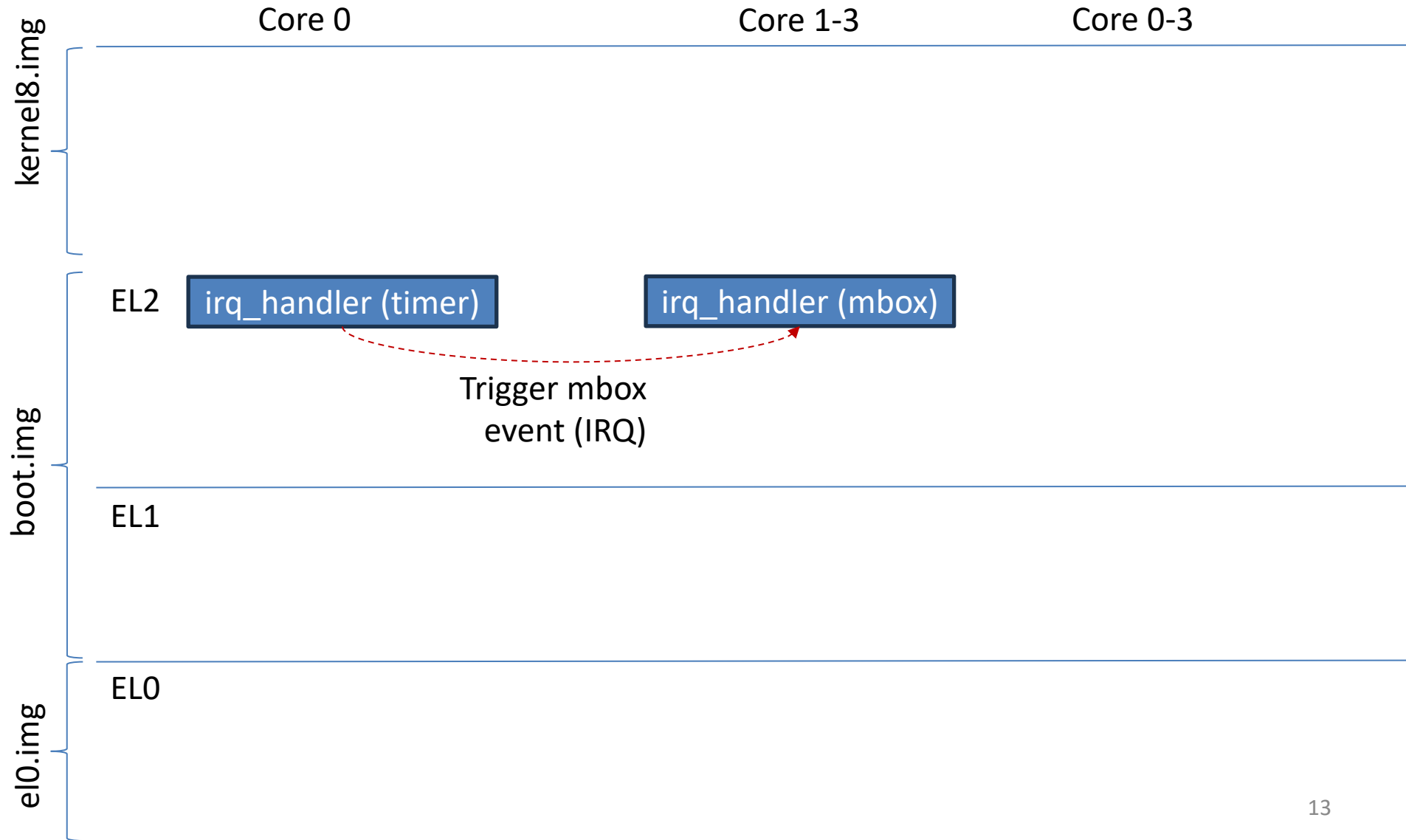
# Démarrage (détailé)



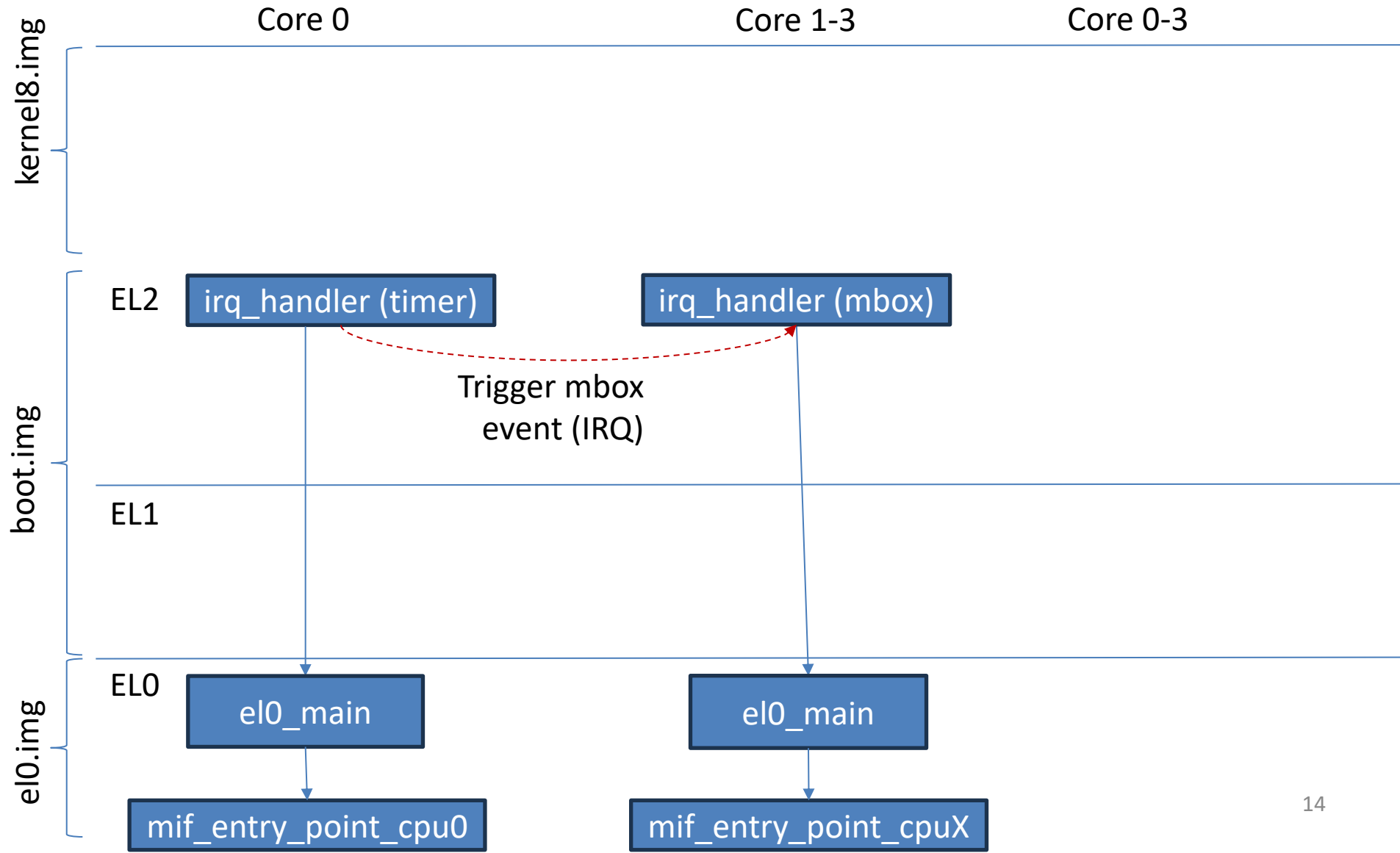
# Chaque cycle après démarrage



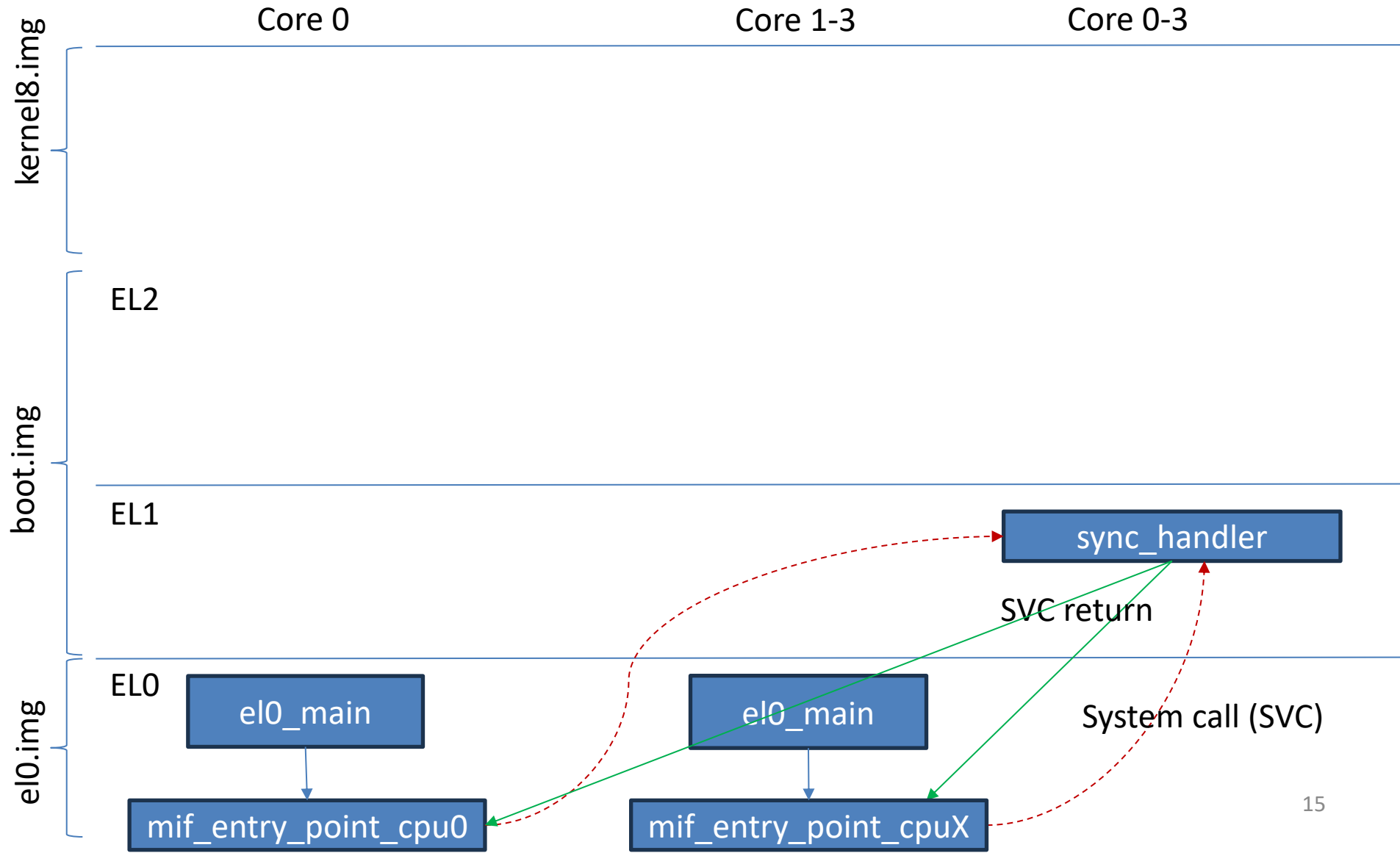
# Chaque cycle après démarrage



# Chaque cycle après démarrage



# Services système (à chaque cycle)



# Configuration matérielle

- MMU et caches activés
  - Cohérence de caches
  - Nécessaire pour performance et synchronisation
- Timers
  - IRQ sur coeur 0, EL2
- Mailboxes (synchronisation entre coeurs)
  - IRQ sur coeurs 1-3, EL2
  - Synchronisation très précise
    - Inversions entre f et g durant l'exécution
- SVC = services système (interruptions synchrones)



# Synchronisation

## 1. Avec le temps réel : timers

- Utilisation 1 : Compteurs qui peuvent être lus
  - E.g. fonction `get_cpuid` (de `svc.h`)
- Utilisation 2 : Alarmes configurables
- Plusieurs timers disponibles : (SoC timer, multicore timer, core timers, GPU timer, USB timer...)
  - Le code fourni utilise le timer ARM (« local timer »)
    - Base à 38.4MHz (1s = 38400000 impulsions, cf. lignes 65-69 de `arm-config.c`)
    - Dérivé de la fréquence GPU et CPU, donc il faut fixer celles-ci (ajouts dans `config.txt` sur la carte SD)
    - <https://datasheets.raspberrypi.com/bcm2836/bcm2836-peripherals.pdf>
  - Routage vers Core 0 (ligne 74 de `arm-config.c`)
  - Configuration par `arm_timer_config_el2()` (`arm-config.c`)

# Synchronisation

## 2. Entre coeurs

### a. Mailboxes

- Nécessaires en l'absence de cohérence de caches
- Plutôt utilisés pour communiquer entre processeurs (e.g. multi-core ARM <-> GPU)
- Peu portable sur autres ISA
- Configuration par `arm_timer_config_el2()` (`arm-config.c`)

### b. Mémoire partagée

# Synchronisation en mémoire partagée

- Vision matérielle – plus portable que les mbox
  - Cohérence des caches requise
  - Plusieurs protocoles matériels – instructions spécifiques
    - Load/Store-exclusive (e.g. ldxr/stxr)
    - Load-acquire/store-release (e.g. ldar/stlr)
    - Barrières (e.g. dmb)
- Vision logicielle – très portable
  - Sémaphores binaires (mutexes)
  - Horloges vectorielles

# Sémaphores binaires

- Construction fondamentale en algorithmique parallèle et concurrente (cf. Edsger Dijkstra)
  - pthread\_mutex
  - Permet l'implantation de sections critiques
- Un type, deux opérations
  - type semaphore\_t à deux valeurs (0=disponible, 1=pris)
  - get(s)
    - Si le sémaphore « s » vaut 0, une des opérations « get(s) » active va obtenir le sémaphore, le faisant passer à 1. Les autres « get(s) » en attente restent en attente. Une fois lancé, « get(s) » se termine quand il obtient « s ».
  - release(s)
    - Met « s » à 0. Si « s » vaut 0, l'opération n'a aucun effet.
  - Initialement, tout sémaphore vaut 0

# Sémaphores binaires

- Implémentation sous Aarch64 (semaphore.S)

```
// void semaphore_get(semaphore_t s)
semaphore_get:
    mov    w2, 1    // registre w2 := 1
.lockloop:
    // Lire en w1 la location mémoire d'adresse x0, et y marquer l'accès exclusif.
    ldaxr  w1, [x0]
    // Si la location [x0] est toujours en accès exclusif, y stocker 1.
    // Si succès, w3 vaudra 0.
    stxr   w3, w2, [x0]
    cbnz   w3, .lockloop // si stxr a échoué (un autre accès sépare ldaxr et stxr)
    cbnz   w1, .lockloop // si la valeur initiale de [x0] n'est pas 0
    ret

// void semaphore_release(semaphore_t s)
semaphore_release:
    stlrb  wzr, [x0] // store release 0
    ret
```

– Détails: <https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Synchronization>

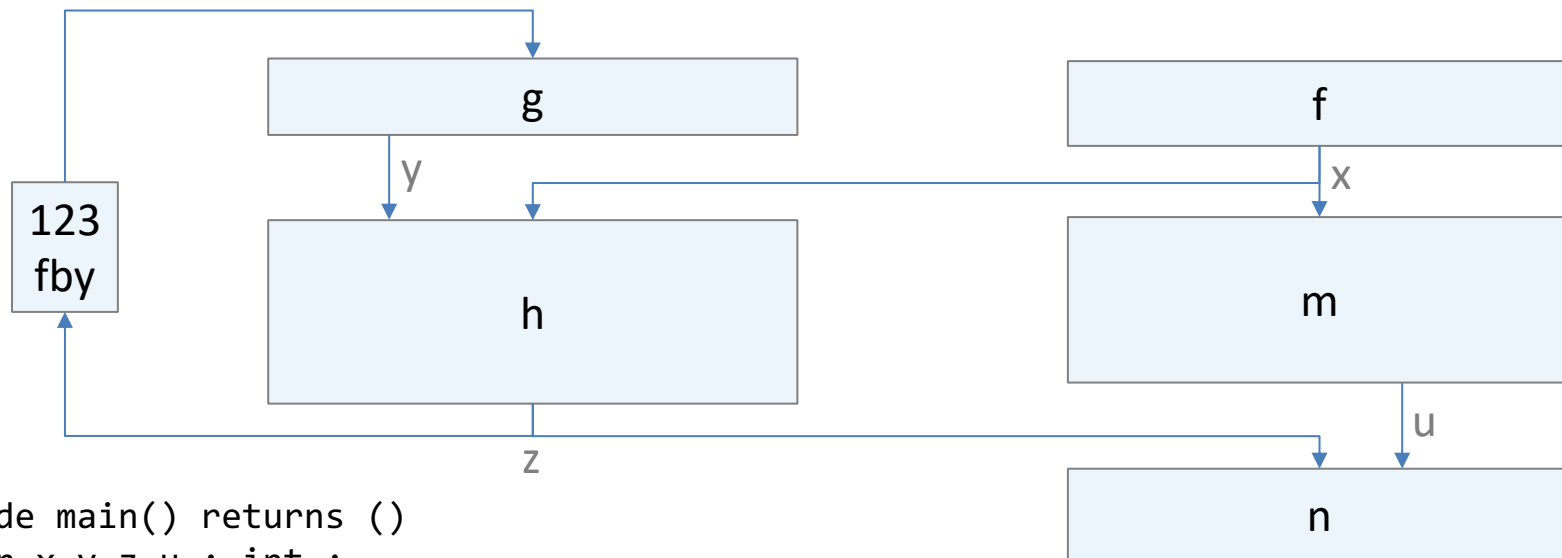
# Sémaphores binaires

- Permet l'implantation de sections critiques
  - Exemple: accès au terminal pour l'impression
    - Un sémaphore « console\_sem ». Pour imprimer un string « str », un cœur doit obtenir ce sémaphore

```
// svc.c, lignes 10, 18-39
semaphore_get(&console_sem);
miniuart_puts(str) ;
semaphore_release(&console_sem) ;
```

# Horloges vectorielles

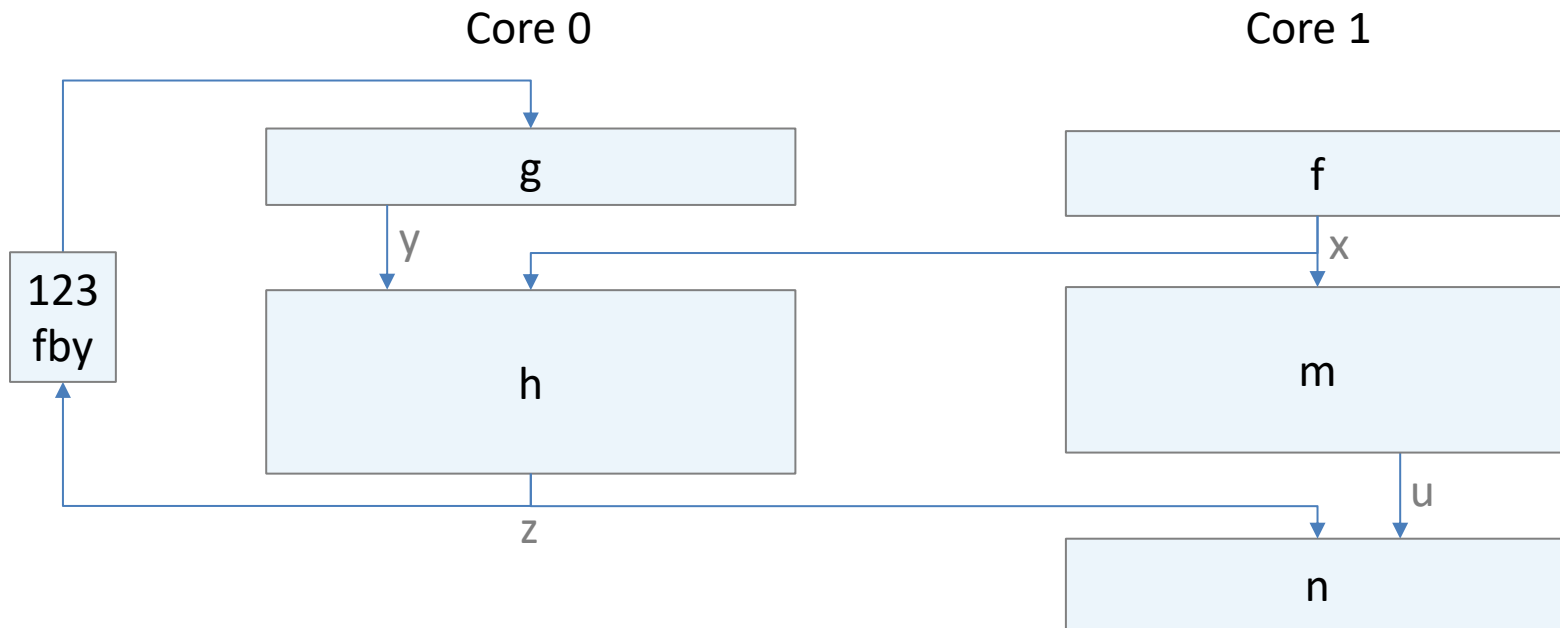
- Prenons un nouvel exemple de spécification fonctionnelle Heptagon



```
node main() returns ()  
var x,y,z,u : int ;  
let  
  y = g(123 fby z) ;  
  x = f() ;  
  z = h(x,y) ;  
  () = n(z,u) ;  
tel
```

# Horloges vectorielles

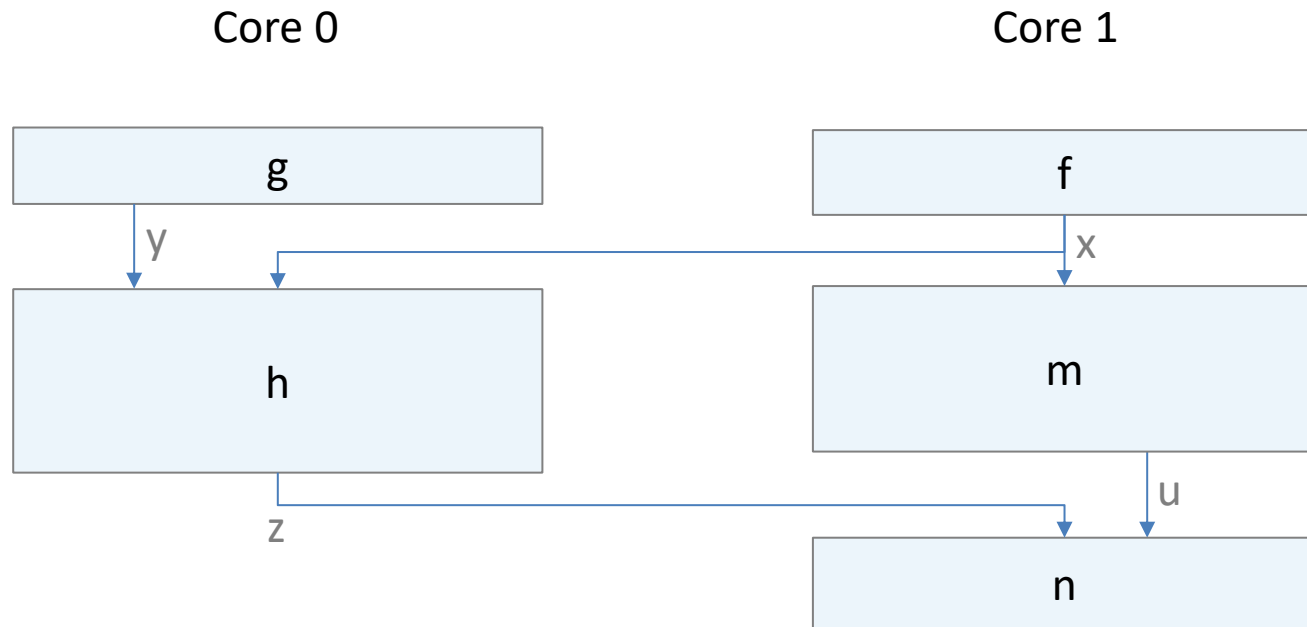
- Allocation & ordonnancement sur 2 cœurs





# Horloges vectorielles

- Synthèse des variables de communication C



```
int x,y,u ;  
int z = 123;
```

# Horloges vectorielles

- Synthèse de code C (sauf synchronisation)

```
void mif_entry_point_cpu0(void){
```

```
  g_step(z, &y);
```

```
  h_step(x, y, &z);
```

z

```
void mif_entry_point_cpu1(void){
```

```
  f_step(&x) ;
```

```
  m_step(x, &u);
```

```
  n_step(u, z);
```

```
int x,y,u ; }  
int z = 123;
```

```
}
```

# Horloges vectorielles

- Synthèse de code C (synchro entre nœuds)
  - En début de cycle:  $\text{loc\_pc\_X} = -1$

```
void mif_entry_point_cpu0(void){
```

```
    g_step(z, &y);
```

```
    while(loc_pc_1 < 1);  
    h_step(x, y, &z);  
    loc_pc_0 = 1;
```

z

```
}
```

```
void mif_entry_point_cpu1(void){
```

```
    f_step(&x) ;
```

```
    loc_pc_1 = 1;
```

```
    m_step(x, &u);
```

```
    while(loc_pc_0 < 1);  
    n_step(u, z);
```

```
}
```

# Horloges vectorielles

- Synthèse de code C (remise à zéro)
  - En début de cycle:  $\text{loc\_pc\_X} = -1$

```
void mif_entry_point_cpu0(void){
```

```
    g_step(z, &y);
```

```
    while(loc_pc_1 < 1);  
    h_step(x, y, &z);  
    loc_pc_0 = 1;
```

z

```
    while(loc_pc_1 < 2);  
    loc_pc_1 = -1;  
    loc_pc_0 = -1;
```

```
}
```

```
void mif_entry_point_cpu1(void){
```

```
    f_step(&x) ;
```

```
    loc_pc_1 = 1;
```

```
    m_step(x, &u);
```

```
    while(loc_pc_0 < 1);  
    n_step(u, z);
```

```
    loc_pc_1 = 2;
```


```
}
```

# Horloges vectorielles

- Mais ce code est incorrect sur un CPU moderne

```
void mif_entry_point_cpu0(void){  
    g_step(z, &y);
```

```
void mif_entry_point_cpu1(void){  
    f_step(&x) ;  
    loc_pc_1 = 1;
```



```
    while(loc_pc_1 < 1);  
    h_step(x, y, &z);  
    loc_pc_0 = 1;
```


- Caches L1 séparés => loc\_pc\_1 peut ne pas changer sur core 0 (si caches non-cohérents)
- Out-of-order execution => affectation de loc\_pc\_1 exécutée avant f => h peut utiliser une ancienne valeur de x

# Horloges vectorielles

- AARCH64: accès mémoire à effet de barrière
  - Load-acquire (ldar), store-release (stlr)

```
void mif_entry_point_cpu0(void){  
    g_step(z, &y);
```

```
void mif_entry_point_cpu1(void){  
    f_step(&x) ;  
    loc_pc_1 = 1; //STLR
```



```
    while(loc_pc_1<1); //LDAR  
    h_step(x, y, &z);  
    loc_pc_0 = 1; //STLR
```


- STLR: toutes les écritures le précédant dans le programme sont réalisées avant
- LDAR: toutes les lectures le suivant dans le programme sont réalisées après

# Horloges vectorielles

- AARCH64: accès mémoire à effet de barrière
  - Load-acquire (ldar), store-release (stlr)

```
void mif_entry_point_cpu0(void){  
    g_step(z, &y);
```

```
void mif_entry_point_cpu1(void){  
    f_step(&x) ;  
    UPDATE_CPU(loc_pc_1,1);
```

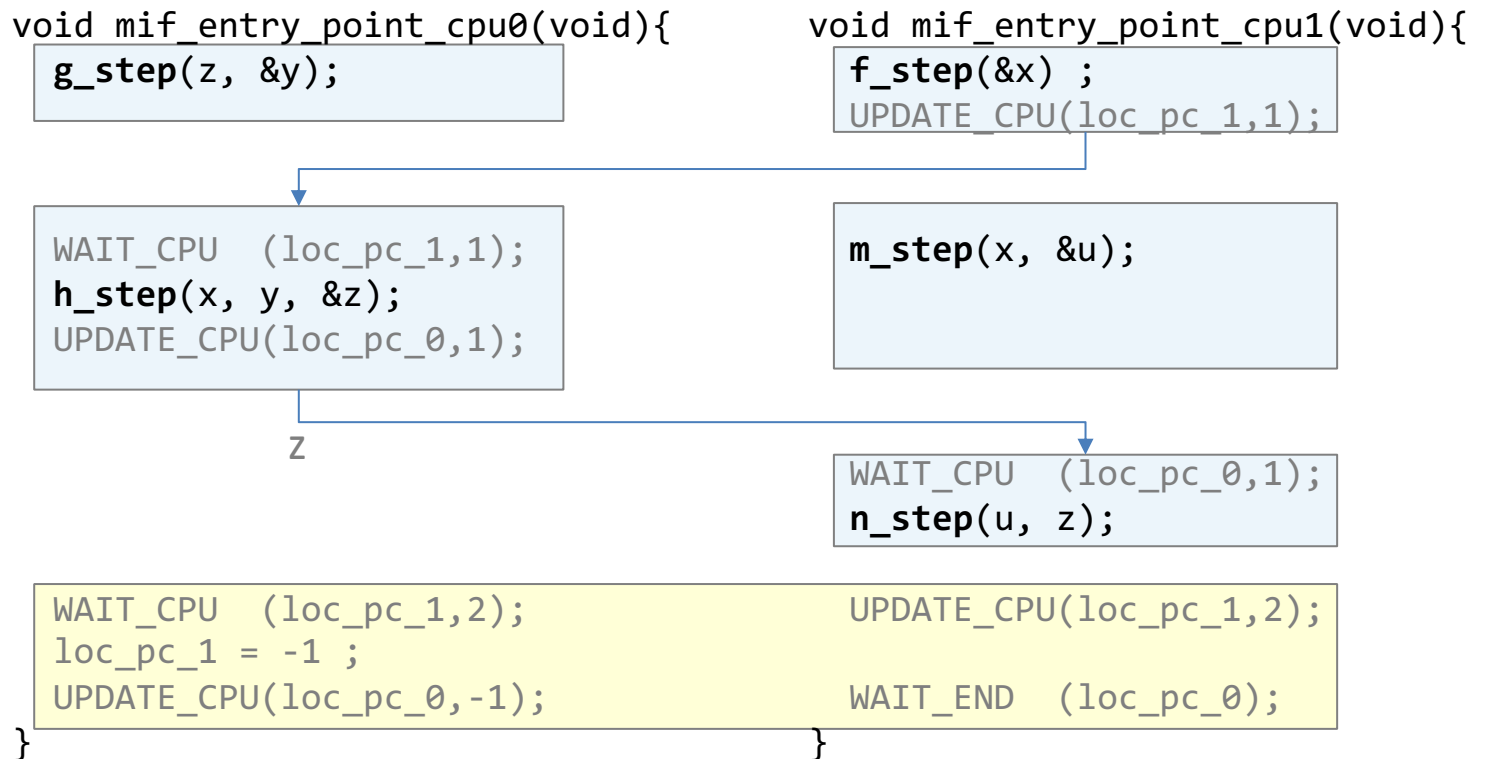


```
    WAIT_CPU (loc_pc_1,1);  
    h_step(x, y, &z);  
    UPDATE_CPU(loc_pc_0,1);
```

- STLR: toutes les écritures le précédant dans le programme sont réalisées avant
- LDAR: toutes les lectures le suivant dans le programme sont réalisées après

# Horloges vectorielles

- Synthèse de code C (y compris remise à zéro)





# Préparation du TP

- Objectif 1: mise en œuvre de l'exemple du transparent 32
  - Partir de l'exemple de l'archive demo-tp6.tar.gz
  - Ajouter a définition des fonctions m,n
    - demo-tp6/aarch64-cortexA53-rpi3-runtime/application/nodes/nodes.c
    - Suivre le modèle de f,g,h (avec  $m(x) = x-17$ )
  - Ajouter la declaration de m,n
    - demo-tp6/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/nodes.h
  - Modifier la définition des variables C de communication
    - demo-tp6/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/variables.[ch]

# Préparation du TP

- Modifier les threads
  - `demo-tp6/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/threads/thread_cpuX.c`
  - Ne garder que les fonctions `mif_entry_point_cpuX` et une fonction `global_init` vide
- Modifier la définition des variables de synchronisation `loc_pc_X`
  - `demo-tp6/aarch64-cortexA53-rpi3-runtime/application/gen-t1042/syncvars.[ch]`
  - Initialiser les variables `loc_pc_X` à -1
- Compilation de `boot.img`, `el0.img`
  - Pas besoin de changer le fichier sur la carte SD
- Exécution sur carte Raspberry Pi