Une approche synchrone à la conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru
dumitru.potop@inria.fr

cours EIDD, 2025

Contenu du cours

- Concurrence: haut niveau vs. bas niveau
- Implantation multi-cœurs, 3^{e} partie
  - Déjà fait :
    - Principes et programmation d'un ordonnanceur en ligne préemptif
    - Organisation d'un exécutif multi-cœurs simple
    - Synchronisation entre cœurs
  - Aujourd'hui : ordonnancement en ligne non-préemptif
    - RM/EDF
  - Préparation du TP

Concurrence: haut niveau vs. bas niveau

- Haut niveau – spécification fonctionnelle

```
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
```

- Bas niveau – implantation bi-coeur

Exécutif/OS
- Minimaliste vs POSIX
- Configuration (Cœurs, MMU, interruptions)
- Vecteurs d'interruptions
  - IRQ - déclenchement des threads…
  - SVC – services système (printf)

```
void mif_entry_point_cpu0(){
  g_step(z, &y);
  WAIT_CPU  (loc_pc_1,1);
  h_step(x, y, &z);
  UPDATE_CPU(loc_pc_0,1);

  WAIT_CPU  (loc_pc_1,2);
  loc_pc_1 = -1 ;
  UPDATE_CPU(loc_pc_0,-1);
}
```

```
void mif_entry_point_cpu1(){
  f_step(&x) ;
  UPDATE_CPU(loc_pc_1,1);
  m_step(x, &u);
  WAIT_CPU  (loc_pc_0,1);
  n_step(u, z);
  UPDATE_CPU(loc_pc_1,2);

  WAIT_END  (loc_pc_0);
}
```

Concurrence: haut niveau vs. bas niveau

- Haut niveau – spécification fonctionnelle

```
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
```

- Bas niveau – implantation bi-coeur

Peut-on ne pas avoir affaire à tous ces détails ?

```
void mif_entry_point_cpu0(){
  g_step(z, &y);
  WAIT_CPU  (loc_pc_1,1);
  h_step(x, y, &z);
  UPDATE_CPU(loc_pc_0,1);

  WAIT_CPU  (loc_pc_1,2);
  loc_pc_1 = -1 ;
  UPDATE_CPU(loc_pc_0,-1);
}
```

```
void mif_entry_point_cpu1(){
  f_step(&x) ;
  UPDATE_CPU(loc_pc_1,1);
  m_step(x, &u);
  WAIT_CPU  (loc_pc_0,1);
  n_step(u, z);
  UPDATE_CPU(loc_pc_1,2);

  WAIT_END  (loc_pc_0);
}
```

Concurrence: haut niveau vs. bas niveau

- Haut niveau – spécification fonctionnelle
  - Lustre/Heptagon
  - Forte abstraction - modèle très simple
    - Déterminisme
    - Fortes contraintes d'isolation
      - Fonctions sans état ou effets de bords
      - Variables non-rémanentes…
    - Proche de Simulink, TensorFlow…
- Bas niveau – implantation bi-coeur
  - **Sous-ensemble** de C11 (C+pthreads)
    - **Threads généraux trop expressifs**
      - Déterminisme difficile à assurer
  - Dépendant de la plate-forme
    - Contraintes de ressources
  - Exigences d'efficacité
    - **Allocation des ressources**
    - **Génération de code**

```
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
```

```
void mif_entry_point_cpu1(){
  f_step(&x) ;
  UPDATE_CPU(loc_pc_1,1);
  m_step(x, &u);
  WAIT_CPU  (loc_pc_0,1);
  n_step(u, z);
  UPDATE_CPU(loc_pc_1,2);

  WAIT_END  (loc_pc_0);
}
```

Concurrence: haut niveau vs. bas niveau

- Les threads généraux sont peu utilisés dans l'embarqué critique
  - A haut niveau : non-déterminisme et format différent de celui des langages de spécification dédiés
  - A bas niveau : trop expressifs

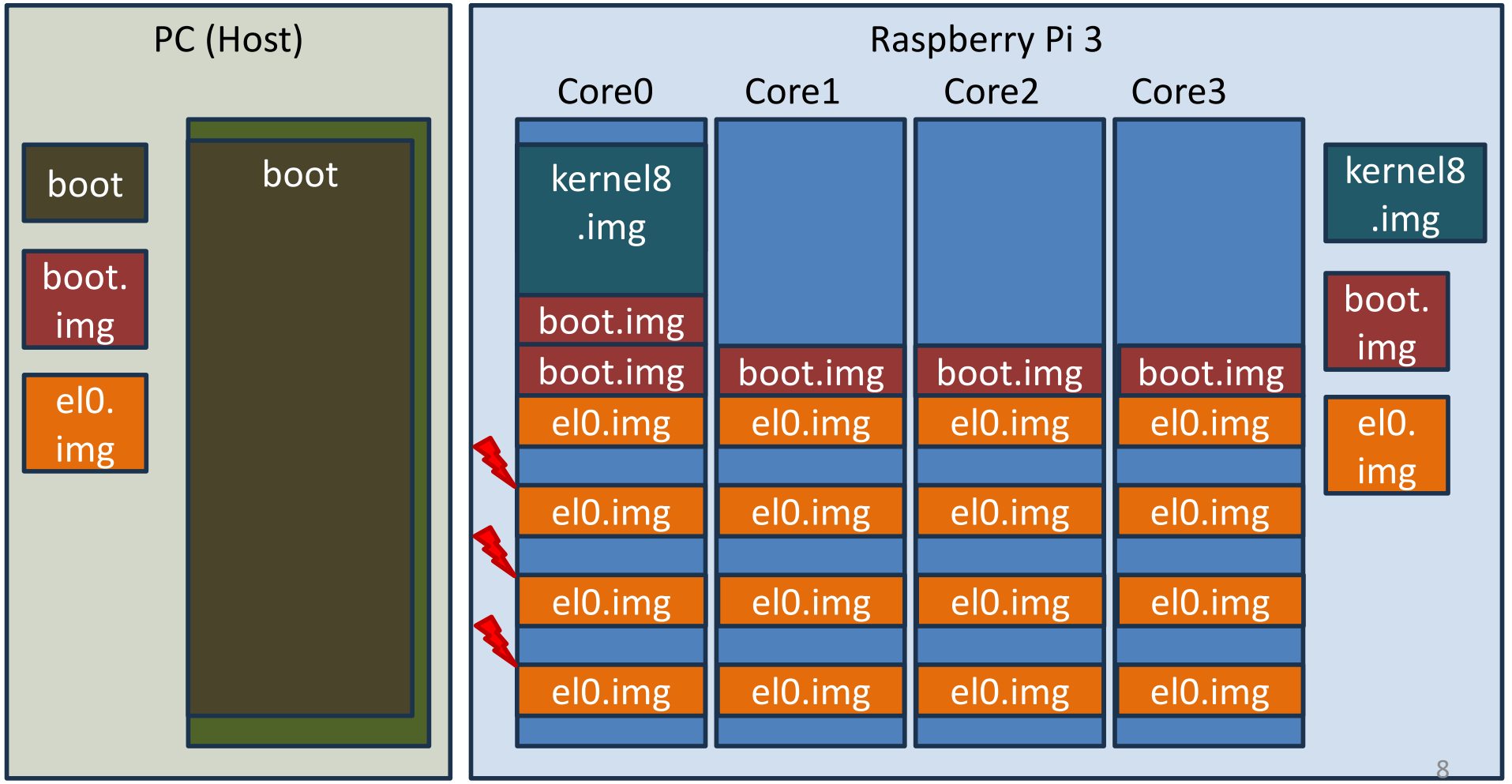
- Implanter correctement et efficacement est un problème difficile
  - Votre travail de TP
    - Mono-cœur – sous Linux/POSIX
    - Multi-cœur – « bare metal » sur ARM Cortex A53 (RPi3)

Contenu du cours

- Concurrence: haut niveau vs. bas niveau
- Implantation multi-cœurs, 3e partie
  - Déjà fait :
    - Principes et programmation d'un ordonnanceur en ligne préemptif
    - Organisation d'un exécutif multi-cœurs simple
    - Synchronisation entre cœurs
  - Aujourd'hui : ordonnancement en ligne, 1e partie
    - RM/EDF sur carte Raspberry Pi
      - Mono-cœur (en n'utilisant qu'un seul cœur des 4 disponibles)
      - Multi-cœurs
    - **Non-préemptif**
  - Préparation du TP

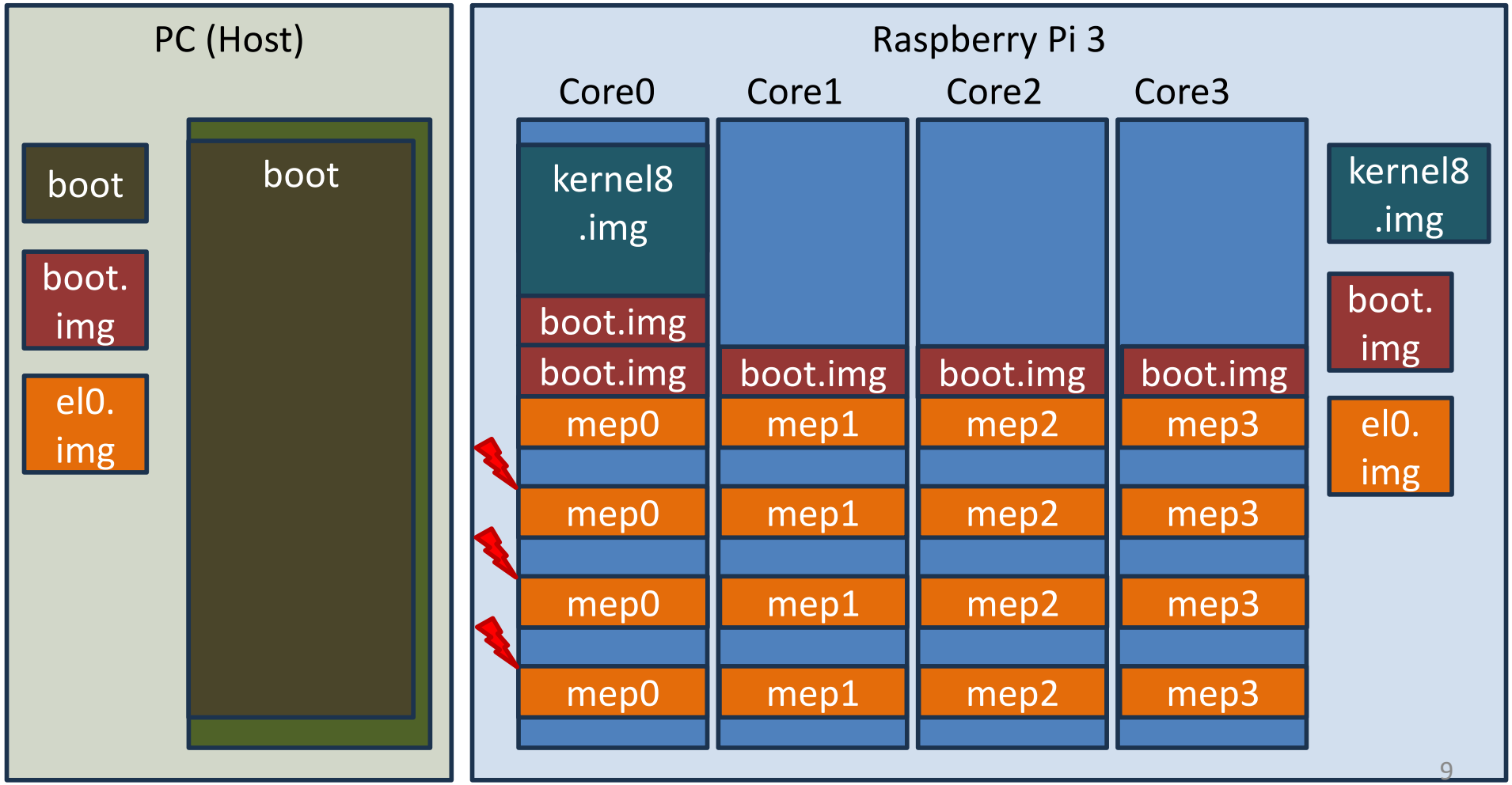
Rappel : exécution multi-cœurs

- Déclenchement timer (période = 1s)



Rappel : exécution multi-cœurs

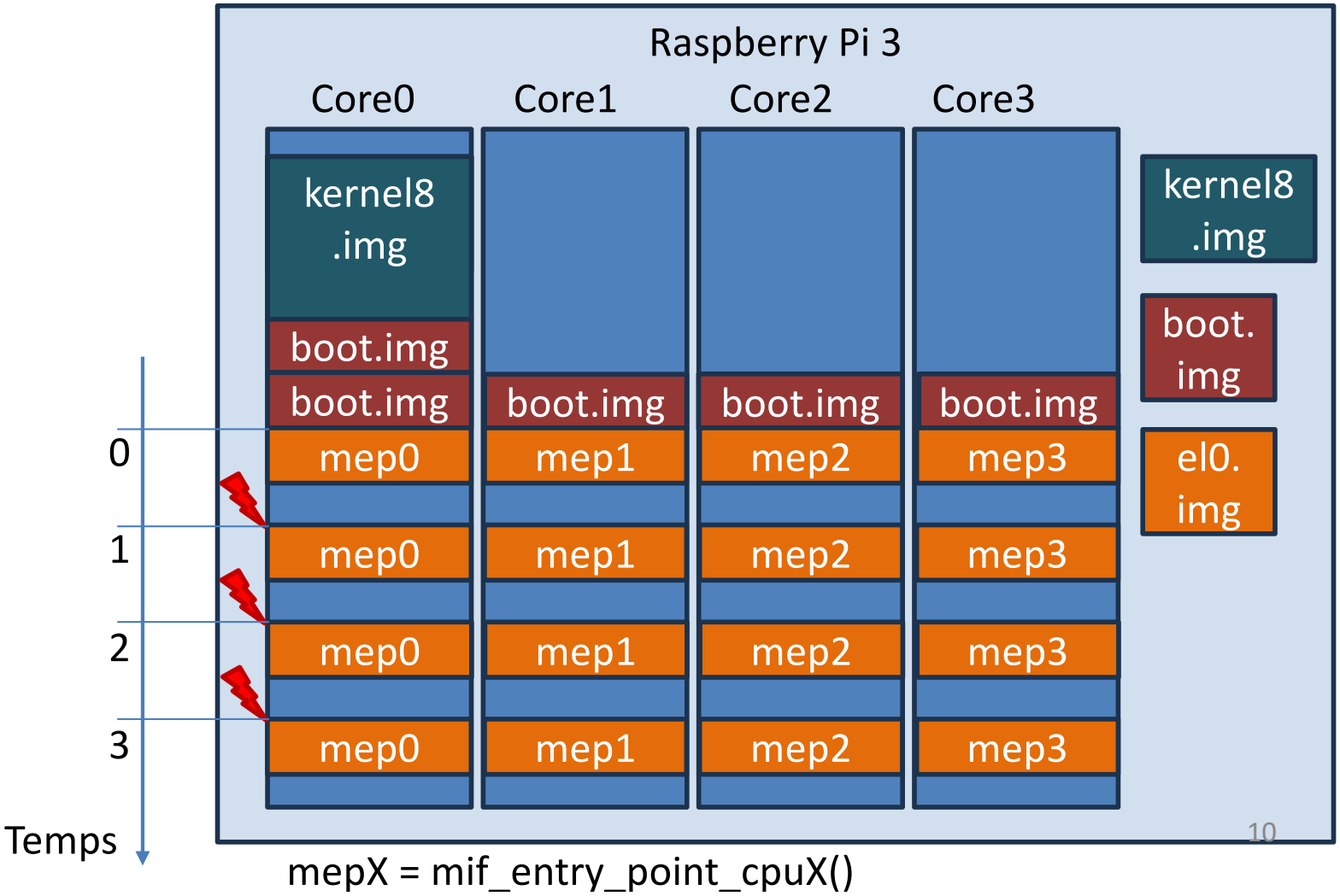
- Déclenchement timer (période = 1s)



mepX = mif_entry_point_cpuX()

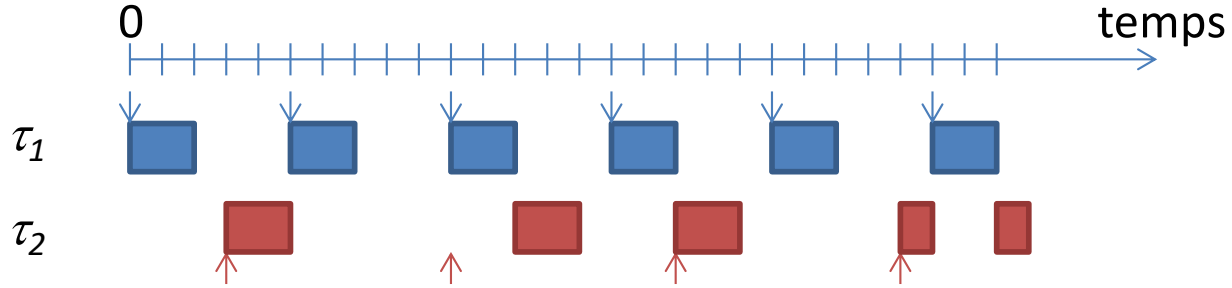
Rappel : exécution multi-cœurs

- Déclenchement timer (période = 1s)



Rappel : ordonnancement préemptif

- Exemple: n=2, T_1 =5, C_1 =2, T_2 =7, C_2 =2
- RM: $prio_1$ =1, $prio_2$ =0

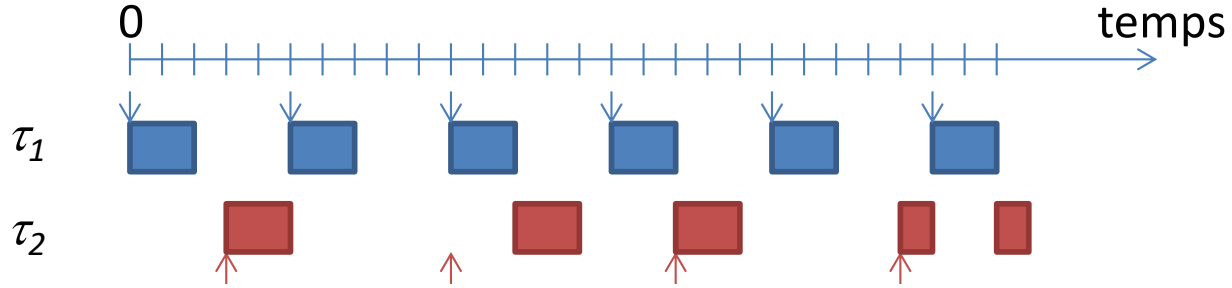


$$\sum_{i=1}^{n} \frac{C_i}{T_i} < n\left(2^{\frac{1}{n}} - 1\right) < 1$$

~0.68% CPU charge

Ordonnancement préemptif

- Exemple: n=2, T_1 =5, C_1 =2, T_2 =7, C_2 =2
- RM: $prio_1$ =1, $prio_2$ =0

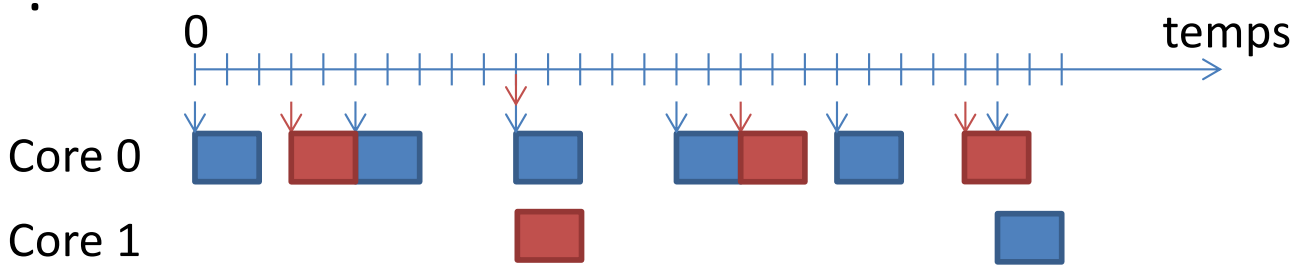


Mono-coeur :

Ordonnancement préemptif

- Exemple: n=2, T_1 =5, C_1 =2, T_2 =7, C_2 =2, RM

Bi-coeur :

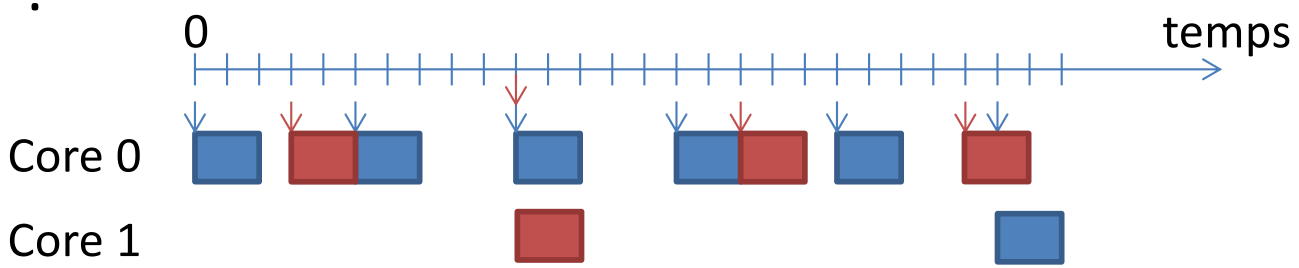


Mono-coeur :

Ordonnancement préemptif

- Exemple: n=2, T_1 =5, C_1 =2, T_2 =7, C_2 =2, RM

Bi-coeur :



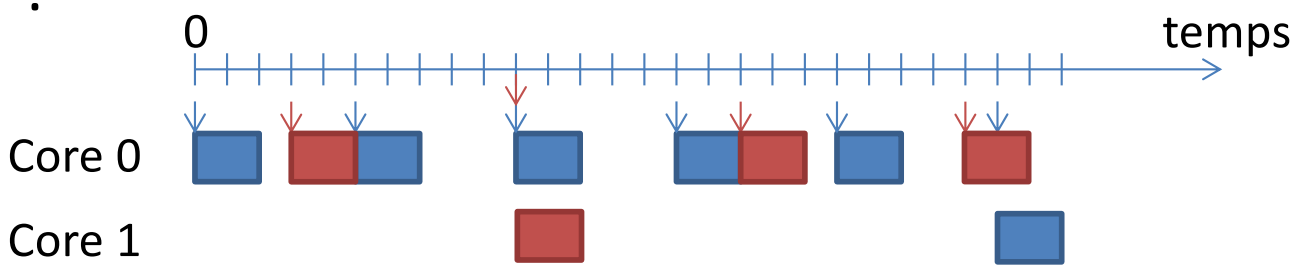
Mono-coeur :

– Remarque : sur deux cœurs, pas de préemption d'une tâche par l'autre

Ordonnancement préemptif

- Exemple: n=2, T_1 =5, C_1 =2, T_2 =7, C_2 =2, RM

Bi-coeur :



Mono-coeur :

  - Impossible à implanter directement par-dessus le simple exécutif fourni
    - Durée des tâches trop longue -> sauvegarde de contexte nécessaire même si les tâches ne s'interrompent pas

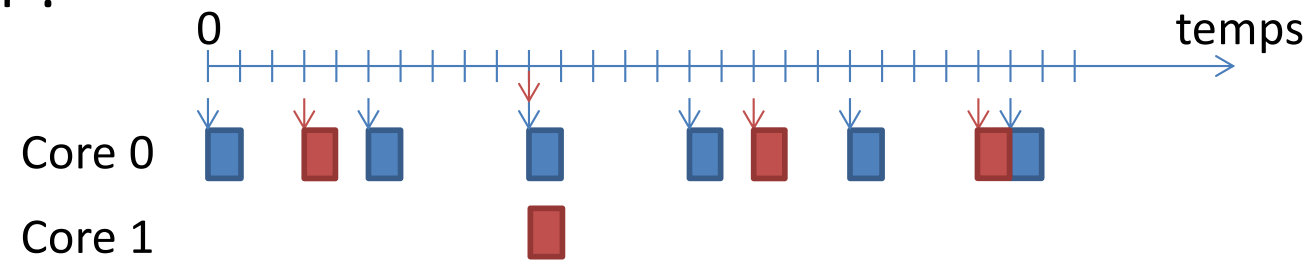
Ordonnancement en ligne non-préemptif

- Contraintes
  - La durée de chaque tâche est plus petite que la période du timer
  - L'ordonnanceur est seulement déclenché sur arrivée du timer
    - Pas à la terminaison d'une tâche, avant l'arrivée du timer
- Conséquences
  - Entre deux arrivées du timer, un cœur ne peut exécuter qu'une instance de tâche
  - Une fois commencée, l'instance de tâche se termine avant la prochaine arrivée du timer.
- Les résultats d'ordonnançabilité s'appliquent encore, malgré la modification de l'algorithme
  - Sous-cas du cas général (même C pour toutes les tâches)

Ordonnancement en ligne non-préemptif

- Exemple:
  - n=2, T_1 =5, C_1 =1, T_2 =7, C_2 =1,
  - Période du timer: 1s, politique RM

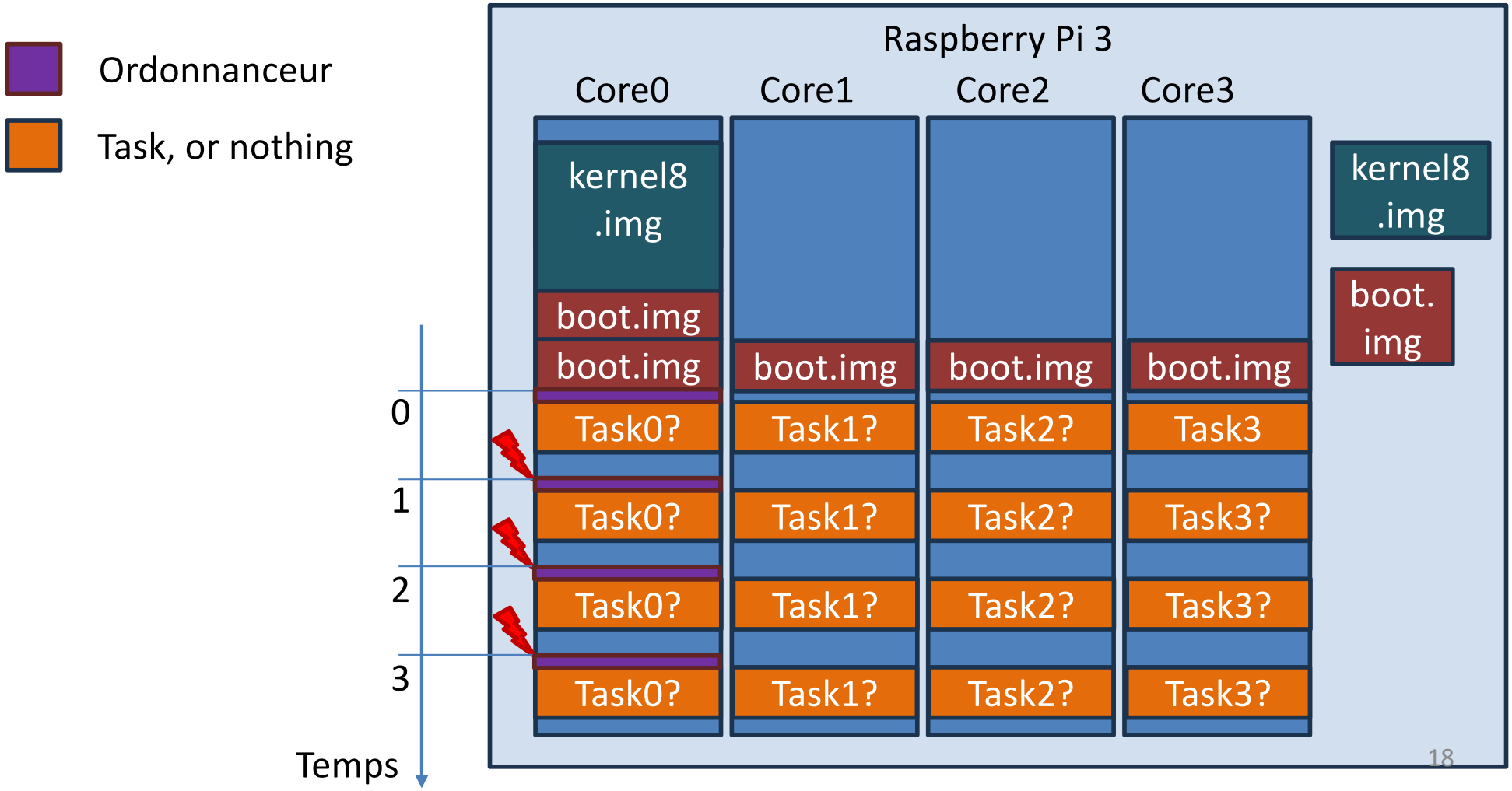
Bi-coeur :



Mono-coeur :

Ordonnancement en ligne non-préemptif

- Exécution de l'ordonnanceur



Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
type task_status =
     { status           : task_state ;
       current_proc     : int ;
       current_deadline : int ;
       left             : int }
```

  - Quand une tâche s'exécute (Running), current_proc donne son processeur
    - Sinon, valeur ignorée

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
node scheduler() returns ()
var ss : scheduler_state ;
    new_date : int ;
    tmp1,tmp2,tmp3,tmp4: task_status^ntasks ;
let
  ss = init_state fby { current_date = new_date ; tasks = tmp4 };
  new_date = ss.current_date + 1 ; (* advance time by 1 *)
  tmp1 = map <<ntasks>> simulate (ss.tasks) ;
  tmp2 = mapi<<ntasks>> check_deadline (new_date^ntasks,tmp1);
  tmp3 = map <<ntasks>> start_inst (new_date^ntasks,tmp2,tasks);
  tmp4 = rate_monotonic   (tmp3) ; (* scheduling policy *)

tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
node scheduler(task_end:bool^ntasks) returns (task_run:int^nproc)
var ss : scheduler_state ;
    new_date : int ;
    tmp1,tmp2,tmp3,tmp4: task_status^ntasks ;
let
  ss = init_state fby { current_date = new_date ; tasks = tmp4 };
  new_date = ss.current_date + 1 ; (* advance time by 1 *)
  tmp1 = map <<ntasks>> complete (ss.tasks,task_end) ;
  tmp2 = mapi<<ntasks>> check_deadline (new_date^ntasks,tmp1);
  tmp3 = map <<ntasks>> start_inst (new_date^ntasks,tmp2,tasks);
  tmp4 = rate_monotonic_mc(tmp3) ; (* scheduling policy *)
  task_run = extract_proc(tmp4) ;
tel
```

  - Remplacer simulation par exécution

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4
  - Nouvelle interface
    - task_end : pour chaque tâche qui se termine, la valeur est « true ». Sinon « false ».
      - C'est la fin des diverses tâches qui détermine ces valeurs
    - task_run : pour chaque processeur i qui démarre une nouvelle tâche d'indice n, task_run[i]=n. Sinon, task_run[i]=ntasks.
      - Sur la base de ces valeurs, les cœurs vont lancer des calculs
    - L'état est internalisé
      - cf. node main, transparent 27, cours 4

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4
  - Modifier la fonction rate_monotonic

```
fun rate_monotonic   (ts:task_status^ntasks)
                          returns (tso:task_status^ntasks)
var selected : int  ;
let
  selected = select_one_task(ts) ;
  tso = mapi<<ntasks>> update_selected (ts, selected^ntasks) ;
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4
  - Modifier la fonction rate_monotonic

```
fun rate_monotonic_mc(ts:task_status^ntasks)
                              returns (tso:task_status^ntasks)
var selected : int^nproc;
let
  selected = select_tasks   (ts) ;
  tso = mapi<<ntasks>> update_selected (ts, selected^ntasks) ;
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
fun update_selected(ts:task_status;selected:int      ;tid:int)
                                    returns (tso:task_status)

let
  tso = if (tid = selected) then


          { ts with .status = Running }
        else ts
tel
```

  - Modifier la fonction update_selected

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
fun update_selected(ts:task_status;selected:int^nproc;tid:int)
                                        returns (tso:task_status)

let
  tso = if (fold<<nproc>> (or)
                (map<<nproc>> (=) (selected,tid^nproc))
                false) then
          { ts with .status = Running }
        else ts
tel
```

  - Le changement du statut traverse « selected »

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
fun update_selected(ts:task_status;selected:int^nproc;tid:int)
                                        returns (tso:task_status)
var proc_id : int ;
let
  proc_id = foldi<<nproc>> update_aux (selected,tid^nproc,nproc);
  tso = if proc_id = nproc then ts
        else {
          ({ ts with .status = Running })
          with .current_proc = proc_id }
tel
```

  - Il faut aussi modifier « current_proc »
  - Le changement du statut traverse « selected »

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
fun update_selected(ts:task_status;selected:int^nproc;tid:int)
                                        returns (tso:task_status)
var proc_id : int ;
let
  proc_id = foldi<<nproc>> update_aux (selected,tid^nproc,nproc);
  tso = if proc_id = nproc then ts
        else {
          ({ ts with .status = Running })
          with .current_proc = proc_id }
tel
```

  - « proc_id » contient nproc si « selected » ne contient pas tid, sinon le processeur pointant vers tid
    - Fonction update_aux à écrire

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Modifier celui du cours 4

```
fun update_selected(ts:task_status;selected:int^nproc;tid:int)
                                        returns (tso:task_status)
var proc_id : int ;
let
  proc_id = foldi<<nproc>> update_aux (selected,tid^nproc,nproc);
  tso = if proc_id = nproc then ts
        else {
          ({ ts with .status = Running })
          with .current_proc = proc_id }
tel
```

  - Prototype de update_aux:

```
fun update_aux(selected:int^nproc,tid:int,proc_id:int,proc_acc_in
:int)
                                        returns (proc_acc_out:int)
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur

```
type select_acc = { tid : int; speriod : int }
fun select_aux(ts:task_status; ta:task_attributes;
               tid:int; acc:select_acc) returns (acc_o:select_acc)
let
  acc_o =
    if (ts.status = Ready) and (ta.period < acc.speriod) then
         { tid = tid; speriod = ta.period }
    else acc
tel
fun select_one_task(ts:task_status^ntasks)
                              returns(selected:int)
var tmp : select_acc ;
let
  tmp = foldi<<ntasks>> select_aux
              (ts,tasks,{ tid = ntasks; speriod = int_max }) ;
  selected = tmp.tid ;
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur

```
type select_acc = { tid : int; speriod : int }
```

```
fun select_one_task(ts:task_status^ntasks)
                                returns(selected:int)
var tmp : select_acc ;
let
  tmp = foldi<<ntasks>> select_aux
              (ts,tasks,{ tid = ntasks; speriod = int_max }) ;
  selected = tmp.tid ;
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur

```
type select_acc = { tid : int; speriod : int }
```

```
fun select_tid(s:select_acc) returns (o : int)
let
  o = s.tid ;
tel
fun select_tasks   (ts:task_status^ntasks)
                              returns(selected:int^nproc)
var tmp : select_acc^nproc ;
let
  tmp = foldi<<ntasks>> select_aux
              (ts,tasks,{ tid = ntasks; speriod =int_max }^nproc);
  selected = map<<nproc>> select_tid tmp ;
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur

```
fun select_aux(ts:task_status; ta:task_attributes; tid:int;
           acc:select_acc      ) returns (acc_o:select_acc)
let

  acc_o =
    if ts.status = Ready then
      if ta.period <  acc        .speriod then
                           { tid = tid ; speriod = ta.period }
      else acc
    else acc
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur

```
fun select_aux(ts:task_status; ta:task_attributes; tid:int;
           acc:select_acc^nproc) returns (acc_o:select_acc^nproc)
let
  lp = select_largest_period(acc);
  acc_o =
    if ts.status = Ready then
      if ta.period < (acc[>lp<]).speriod then
         [ acc with [lp] = { tid = tid ; speriod = ta.period } ]
      else acc
    else acc
tel
```

- Parmi les allocations de tâches aux cœurs, choisir une de période la plus longue. La remplacer avec la tâche courante si sa période est plus courte.

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur

```
type aux_type = { sel_proc : int ; sel_period : int }
fun slp_aux (sa:select_acc; proc:int; acc:aux_type)
                returns (o:aux_type)
let
  o = if acc.sel_period < sa.speriod then
        { sel_proc = proc ; sel_period = sa.speriod }
      else acc
tel

fun select_largest_period(acc:select_acc^nproc) returns (lp:int)
var search_result : aux_type ;
let
  search_result =
    foldi<<nproc>> slp_aux (acc,{ sel_proc = 0 ;
                                  sel_period=(acc[0]).speriod});
  lp = search_result.sel_proc ;
tel
```

Mise en œuvre bi-coeur

- Etape 1 : créer un scheduler RM bi-cœur
  - Il vous reste à écrire les fonctions
    - complete – si une tâche t a terminé son exécution (task_end[t]=true) alors mettre à jour son état
    - extract_proc – extraire de l'état des tâches ce que chaque processeur doit faire
      - Inverser un « map »
  - Tester d'abord le scheduler sur votre PC, en créant une fonction qui lit des entrées du clavier pour vous permettre de vérifier qu'il fonctionne correctement

Mise en œuvre bi-coeur

- Etape 2 : intégrer le scheduler dans le code C multi-coeurs

```
void mif_entry_point_cpu0(){
  Scheduler__scheduler_step(task_end,&out,&mem) ;
  UPDATE_CPU(loc_pc_0,1) ;

  switch(out.task_run[0]) {
  case 0: task0() ; break ;
  case 1: task1() ; break ;
  …
  case Scheduler__ntasks : break ;
  default:printf("Error.\n") ; break ;
  }

  WAIT_CPU  (loc_pc_1,2);
  loc_pc_1 = -1 ;
  UPDATE_CPU(loc_pc_0,-1);
}
```

```
void mif_entry_point_cpu1(){

  WAIT_CPU(loc_pc_0,1);

  switch(out.task_run[1]) {
  case 0: task0() ; break ;
  case 1: task1() ; break ;
  …
  case Scheduler__ntasks : break ;
  default:printf("Error.\n") ; break ;
  }

  UPDATE_CPU(loc_pc_1,2);

  WAIT_END  (loc_pc_0);
}
```

Mise en œuvre bi-coeur

- Etape 2 : intégrer le scheduler dans le code C multi-cœurs
  - A votre charge:
    - Placer le code Heptagon dans le répertoire nodes
    - Créer les variables nécessaires (état, output)
    - Faire l'initialisation de l'état du scheduler (fonction reset) dans global_init.

Préparation du TP

- Objectif 1 – mise en œuvre sur bi-cœur ARM de l'ordonnanceur non-préemptif
  - Configuration du transparent 17

- Objectif 2 – nouvelle configuration:
  - Utilisation de 3 cœurs
  - 5 tâches:

Task id	period	First start date
1	7	1
2	5	1
3	6	3
4	8	2
5	10	2