Trampoline
Un système d'exploitation temps réel pour l'informatique embarquée

Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou

LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

LS2N – UMR CNRS 6004
CNRS, École Centrale de Nantes, École des Mines de Nantes, Université de Nantes

École d'été Temps Réel
Paris, 30 août 2017

Section 1

Contexte, définition

Système embarqué

Système électronique et informatique intégré à un système mécatronique en vue de piloter ses organes

Principales caractéristiques

Ressources matérielles limitées

Matériel adapté aux contraintes opérationnelles (thermiques, vibratoires, etc) et économiques $\longrightarrow$ micro-contrôleur

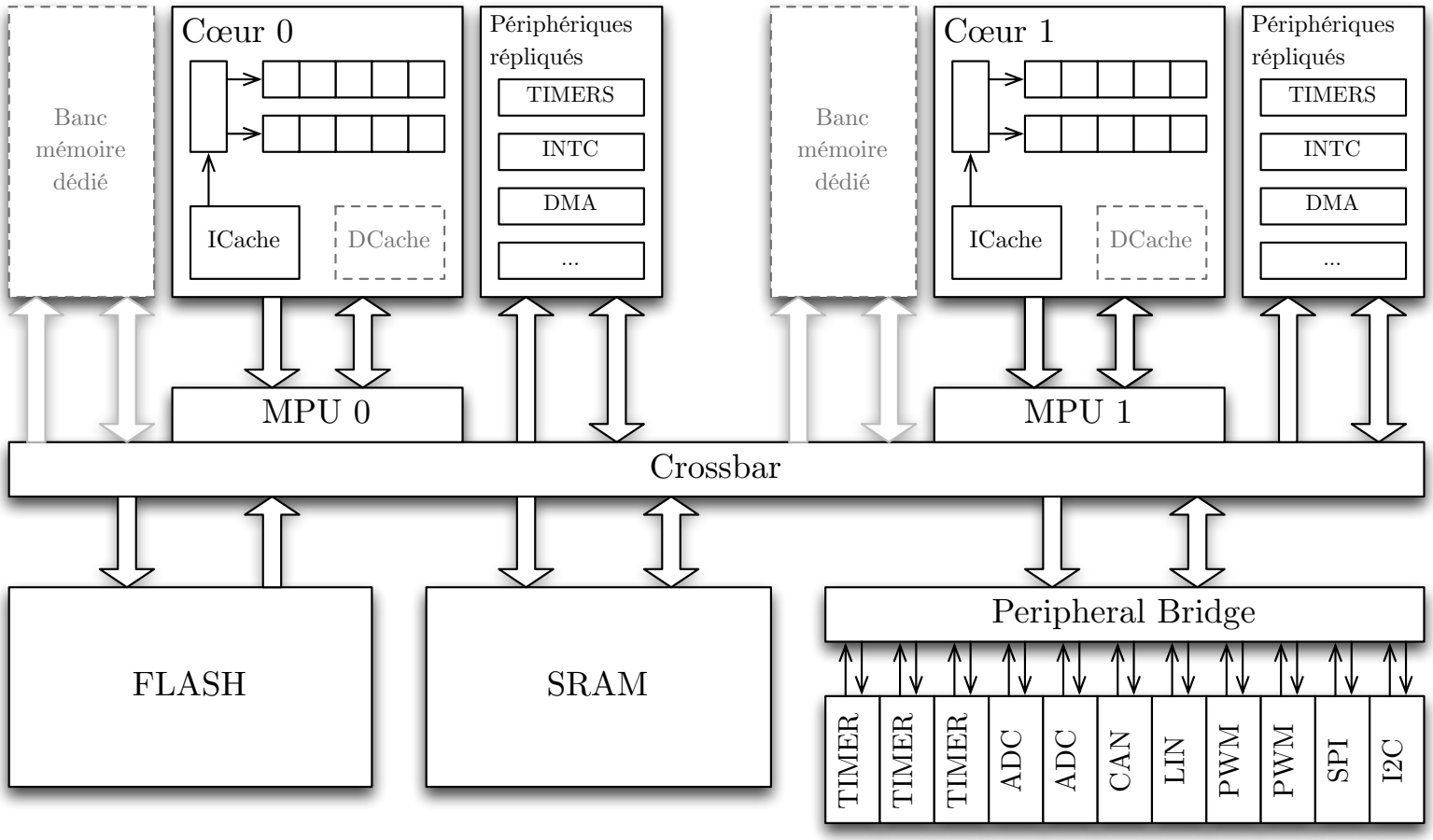
Temps réel

- Contraintes *hard*, *firm*, *soft*
- Dynamique parfois très rapide : les temps de réponse exigés peuvent s'exprimer en centaines de micro-secondes

Contexte industriel

- Normes, standards, certification

Architecture matérielle type : organisation



Architecture matérielle type : caractéristiques

- Mono ou Multicœur
- ISA de type RISC 32 bits
- Micro-architecture superscalaire
- Cache : L1 uniquement, souvent I-Cache uniquement
- Protection mémoire : MPU, plus rarement MMU
- Pas d'assistance matérielle à la virtualisation

Architecture fonctionnelle type

Application = ensemble de fonctions temps réel communicantes

Architecture fonctionnelle type

Application = ensemble de fonctions temps réel communicantes

Fonctions

- Partie commande : régulateurs, filtres, machine à états
- Partie système : pilotes, pile de communication, monitoring, ...

Architecture fonctionnelle type

Application = ensemble de fonctions temps réel communicantes

Temps réel

- Activations périodiques ou sporadiques
- Échéances contraintes ou sur requêtes
- Criticités multiples

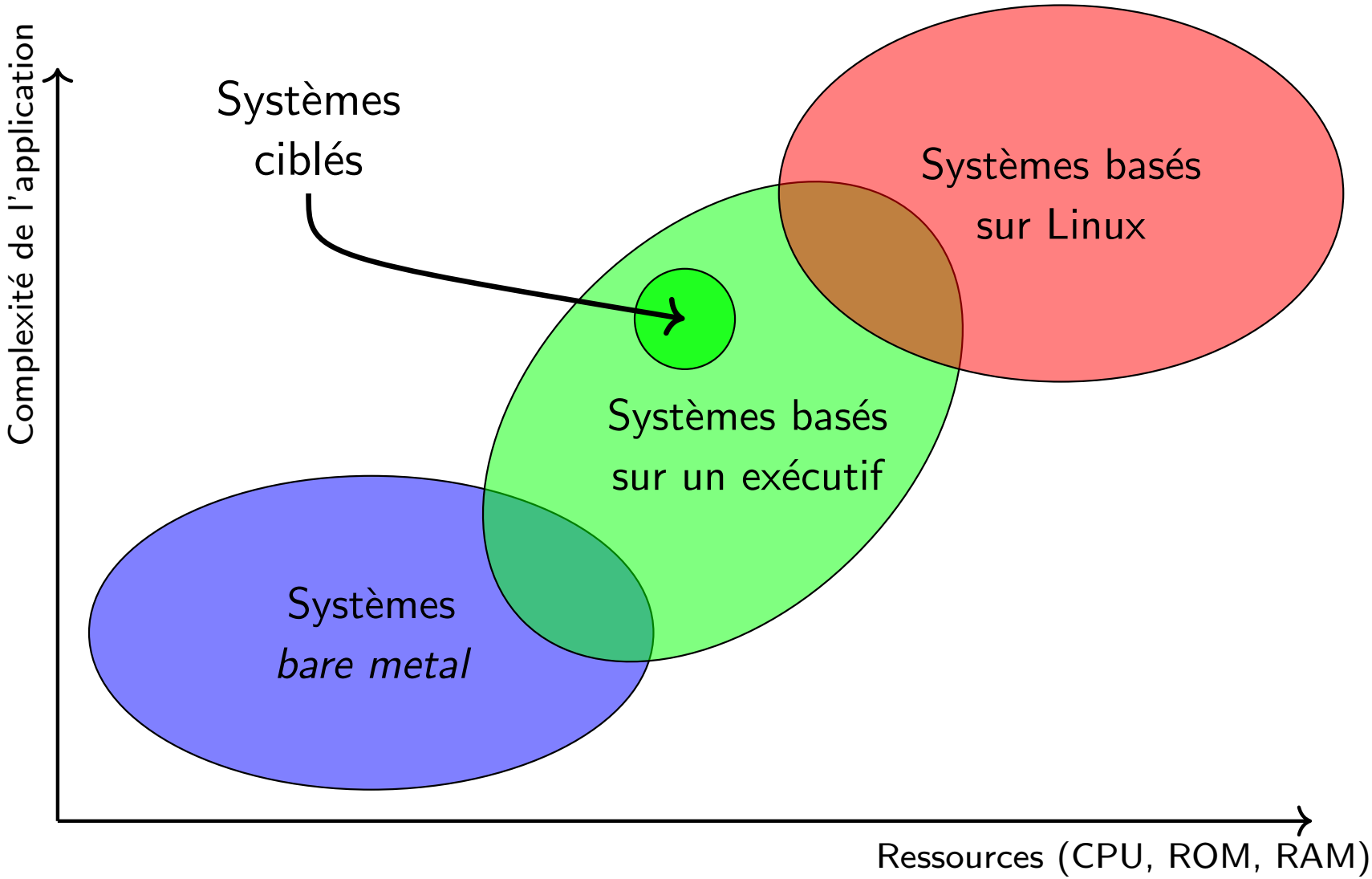
Architecture fonctionnelle type

Application = ensemble de fonctions temps réel communicantes

Communication (et synchronisation)

- Signaux booléens ou numériques, scalaires ou vectoriels
- Utilisation de queues de messages et/ou de blackboard
- Ressources partagées (périphériques)

Positionnement



Système d'exploitation temps réel

Système d'exploitation temps réel (SETR)

Un système d'exploitation (SE) est qualifié de temps réel s'il permet de construire un système aux temps de réponses **prédictibles** et **compatibles avec les dynamiques de l'environnement.**

Chemin critique

- Prise en compte des interruptions
- Ordonnancement
- Changement de contexte

Impact sur la conception

Déterminisme

- Restriction du nombre de services en mode noyau

Prédictibilité

- Éviction des algorithmes / services dont le pire cas est pathologique
- Minimiser le pire-cas, éventuellement au détriment d'autres objectifs : équité, flexibilité, sécurité, etc.

Les services de base

Temps réel

- Ordonnancement
- IPC : synchronisation, signalisation, communication
- Prise en compte des interruptions
- (Mesure du temps : horloges, réveils, chien de garde)

Tolérance aux fautes

- Isolation mémoire
- Isolation temporelle

Et le reste ?

Reléguée hors du noyau

- Piles de communication
- Pilotes de périphériques

Parfois/souvent/toujours inutiles

- Système de gestion de fichiers
- Gestion des utilisateurs

Difficilement temps réel

- Mémoire virtuelle (*swapping*)

Section 2

Trampoline : un exécutif temps réel

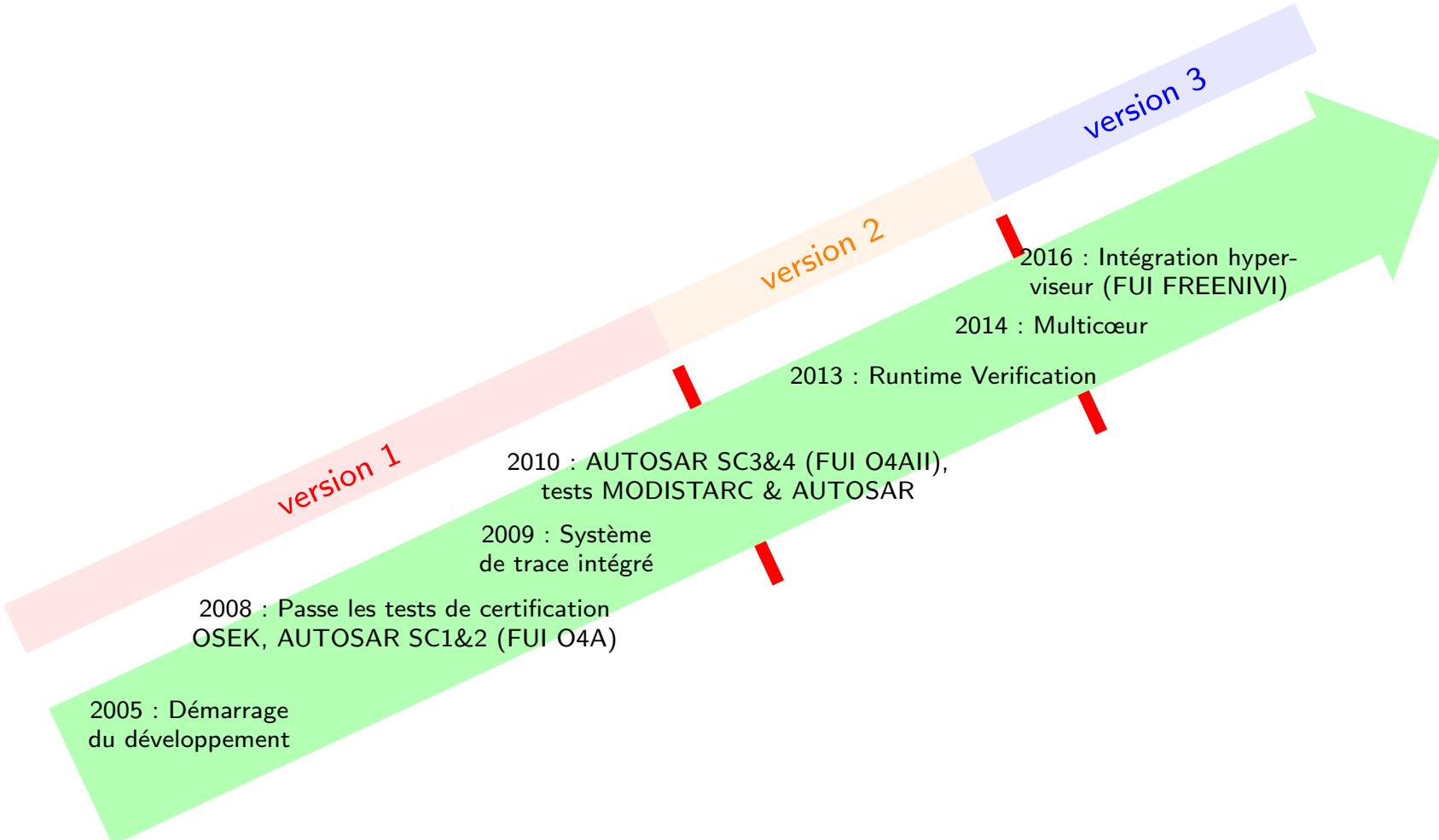
Trampoline : un exécutif temps réel

- Présentation
- Processus de construction d'une application
- Architecture interne

Trampoline

- Exécutif temps réel, développé par l'équipe Systèmes Temps Réel du LS2N (ex-IRCCyN)
- Modulaire et portable
- Licence libre (GPL) + version industrialisée
- Conforme OSEK/VDX OS et AUTOSAR OS
- Nombreuses cibles dont Arduino, Posix, PowerPC, ARM Cortex-M, ...
- Support des architectures multicœurs

Historique


version 1
version 2
version 3
2005 : Démarrage
du développement
2008 : Passe les tests de certification
OSEK, AUTOSAR SC1&2 (FUI O4A)
2009 : Système
de trace intégré
2010 : AUTOSAR SC3&4 (FUI O4AII),
tests MODISTARC & AUTOSAR
2013 : Runtime Verification
2014 : Multicœur
2016 : Intégration hyper-
viseur (FUI FREENIVI)

Utiliser Trampoline : pourquoi ? comment ?

Pourquoi ?

Disposer d'un composant libre pour construire des bancs d'essais

Support d'enseignement

Expérimenter des nouveaux algorithmes / services

Comment ?

Site web : `http://trampoline.rts-software.org`

Dépôt : `https://github.com/TrampolineRTOS/trampoline`

Système statique

Un système d'exploitation est statique s'il n'offre pas la possibilité de créer des objets (tâche, ressource, boîte aux lettres, etc.) en ligne. Il doit donc être entièrement configuré à la compilation.

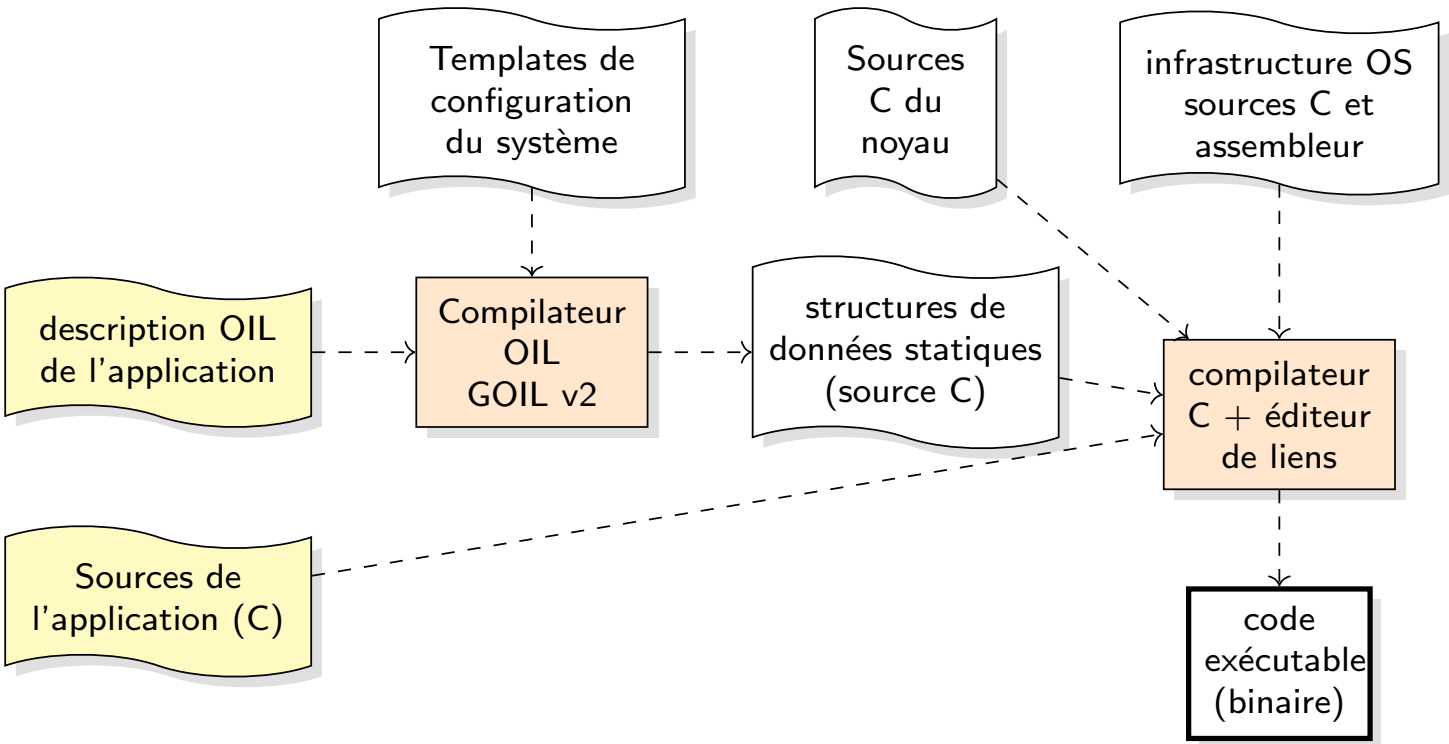
Le noyau d'un système statique peut être « taillé sur mesure » pour l'application

- Services à inclure
- Structure de données
- Et plus encore : chemins d'exécution, variables, etc.

Trampoline est un système statique

Trampoline utilise un langage dédié

- dérivé de la norme OIL
- extensible (basé sur un moteur de template)



Exemple de description

```
ALARM one_second {
  COUNTER = SystemCounter;
  ACTION = ACTIVATETASK { TASK = t0; };
  AUTOSTART = TRUE {
      APPMODE = std;
      ALARMTIME = 100;
      CYCLETIME = 100;
  };
};

TASK t0 {
  AUTOSTART = FALSE;
  PRIORITY = 3;
  ACTIVATION = 1;
  SCHEDULE = FULL;
  MESSAGE = s00;
};

MESSAGE s00 {
  MESSAGEPROPERTY = SEND_STATIC_INTERNAL {
      CDATATYPE = "uint8";
  };
  NOTIFICATION = NONE;
};
```

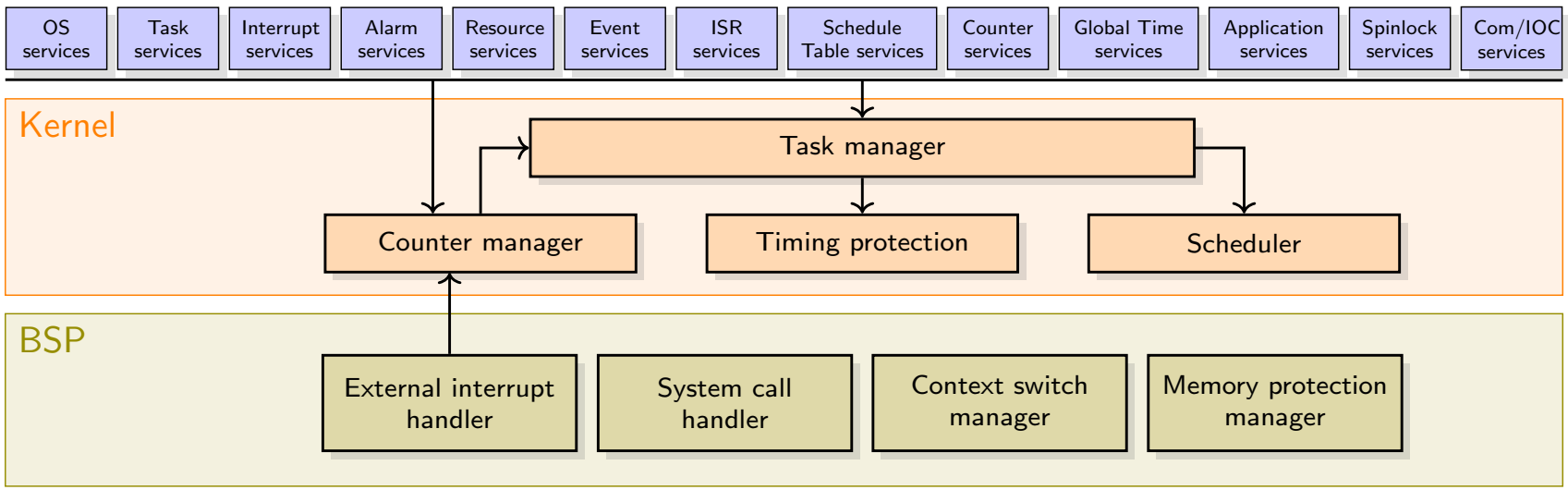
Exemple de template goil

```
/*============================================================================
 * Definition and initialization of process tables (tasks and isrs)
 */
CONSTP2CONST(tpl_proc_static, AUTOMATIC, OS_APPL_DATA)
tpl_stat_proc_table[TASK_COUNT+ISR_COUNT+% ! OS::NUMBER_OF_CORES %] = {
%
foreach proc in PROCESSES do
    %  &% !proc::NAME %_% ![proc::KIND lowercaseString] %_stat_desc,
%
end foreach
if OS::NUMBER_OF_CORES == 1 then
%  &IDLE_TASK_task_stat_desc%
else
  loop core from 0 to OS::NUMBER_OF_CORES - 1 do
    %  IDLE_TASK_% ! core %_task_stat_desc%
  between %,
%
  end loop
end if
%
};
```

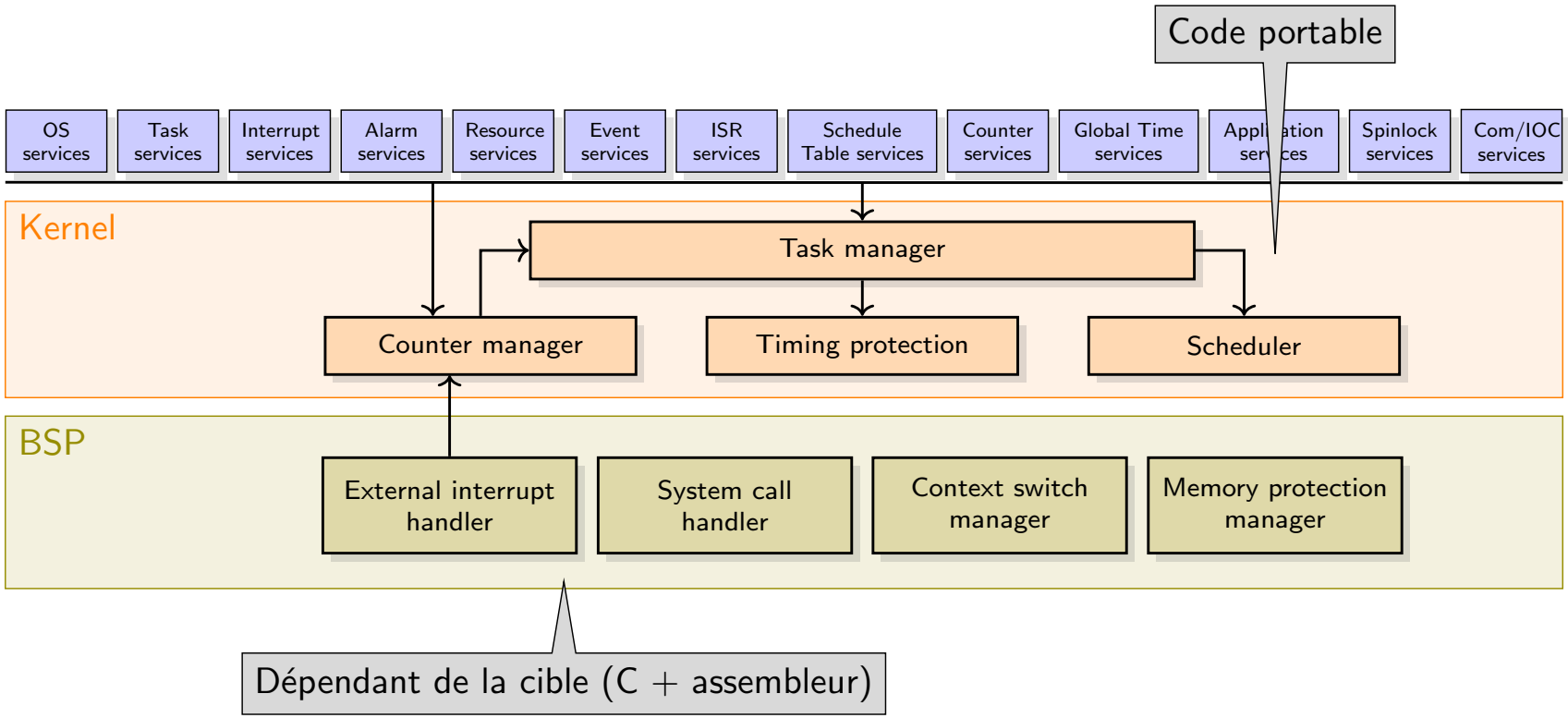
Avantages de `goil` sur une API

- Séparation des préoccupations
- Niveau d'abstraction
- Plus simple à faire évoluer
- Automatisation de la spécialisation du noyau
- Génération automatique : Code, Makefile, *link script*

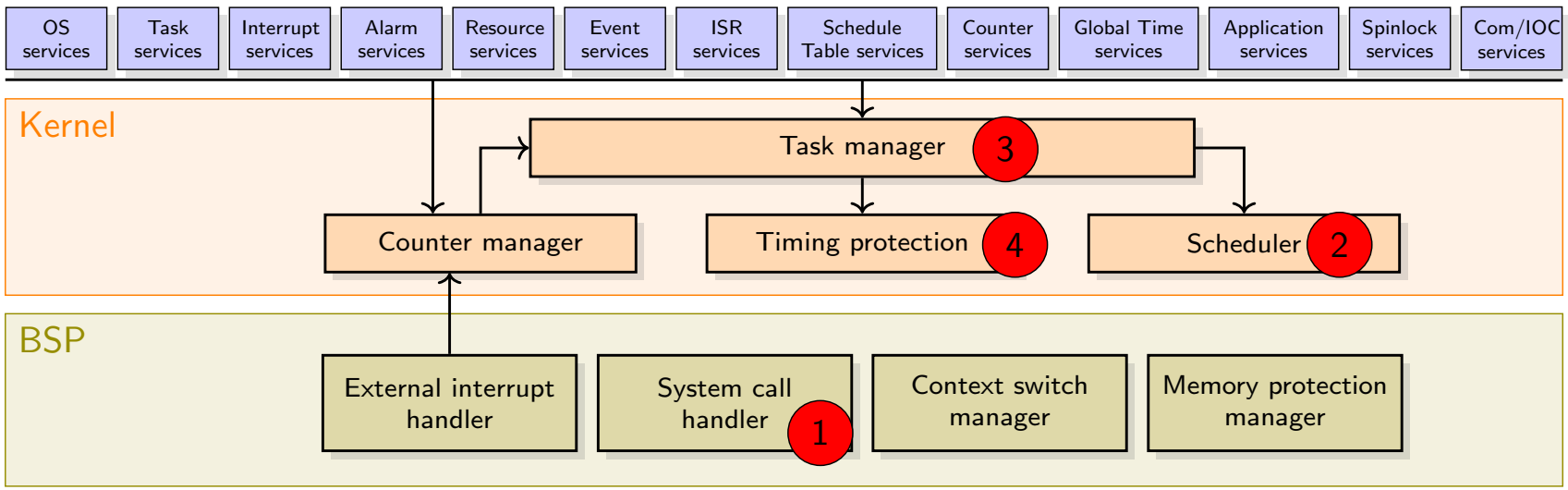
Architecture



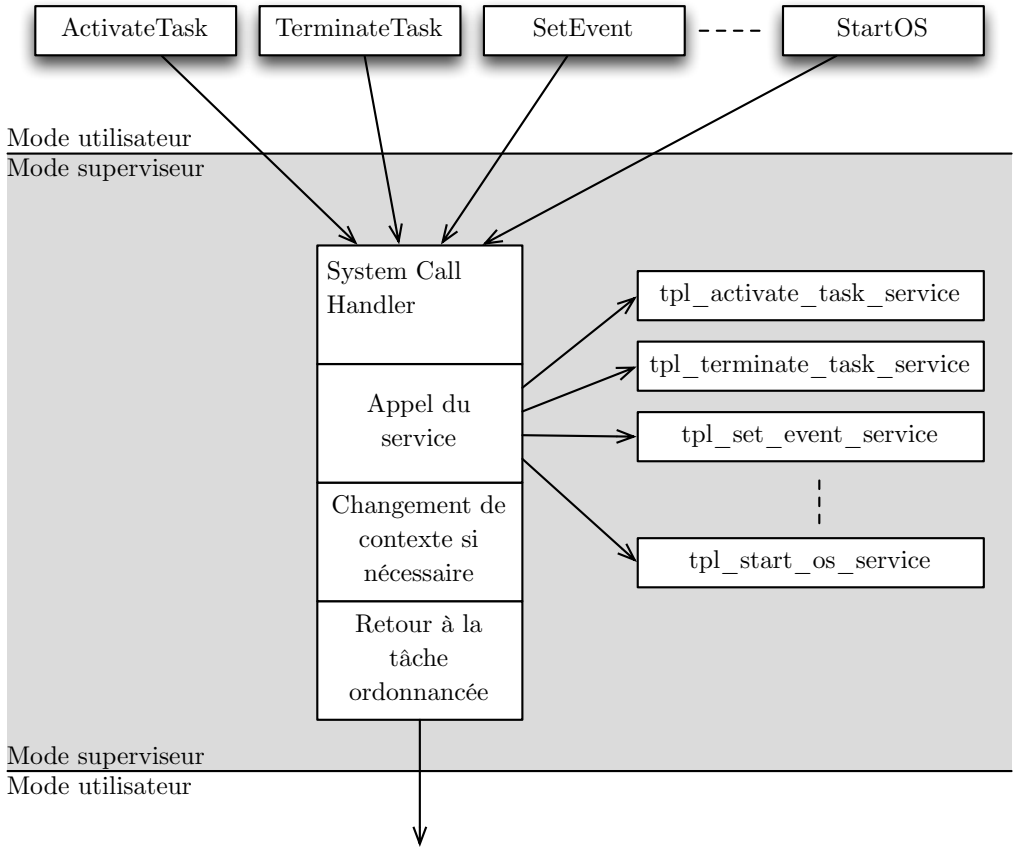
Architecture



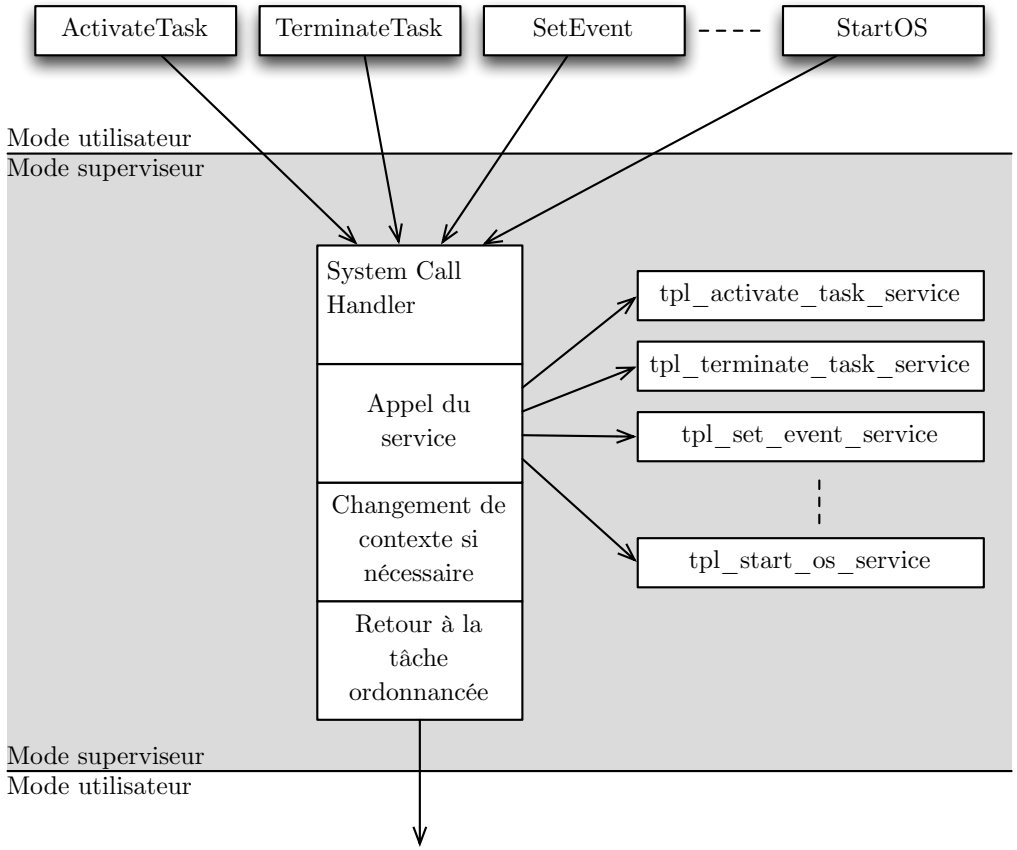
Architecture



Module *System Call Handler*



Module *System Call Handler*



Une fois passé en mode superviseur, le noyau est non préemptible.
Il faut donc minimiser la durée de ce passage pour assurer la réactivité du système

Module *System Call Handler*

Passage en mode superviseur (exemple sur PowerPC)

```
  .global SuspendOSInterrupts
SuspendOSInterrupts:
  li   r0,OSServiceId_SuspendOSInterrupts  /* service id in r0 */
  sc                                       /* system call      */
  blr                                      /* returns          */
```

Selon l'ABI utilisée, les paramètres et le code de retour de l'appel système sont passés dans des registres ou sur la pile

Module *Scheduler*

Ordonnanceur

Assure la manipulation de la liste des tâches prêtes grâce à quatre fonctions :

- `tpl_put_new_proc` : Ajout d'une tâche dans la liste des tâches prêtes ;
- `tpl_put_preempted_proc` : Ajout d'une tâche préemptée ;
- `tpl_front_proc` : obtenir la tâche la plus prioritaire ;
- `tpl_remove_front_proc` : enlever la tâche la plus prioritaire.

Module *Scheduler*

Politique d'ordonnancement

FP/FIFO avec seuil de préemption.

En pratique, toute politique *fixed job priority* qui assure un ordre *FIFO* entre les *jobs* successifs d'une même tâche peut être implémentée

Modifier la politique d'ordonnancement

1. Ajouter les informations nécessaires aux descripteurs de tâches à l'aide des *templates* `goil`
2. Adapter le calcul de la priorité dans `tpl_put_new_proc`
3. Adapter le protocole de synchronisation pour l'accès aux ressources partagées (IPCP par défaut)

Module *Task manager*

Task Manager

Fournit les fonctions permettant de faire évoluer l'état des « tâches ».

Utilise les fonctions du module *Scheduler* pour manipuler les *jobs* engendrés.

Tâche ?

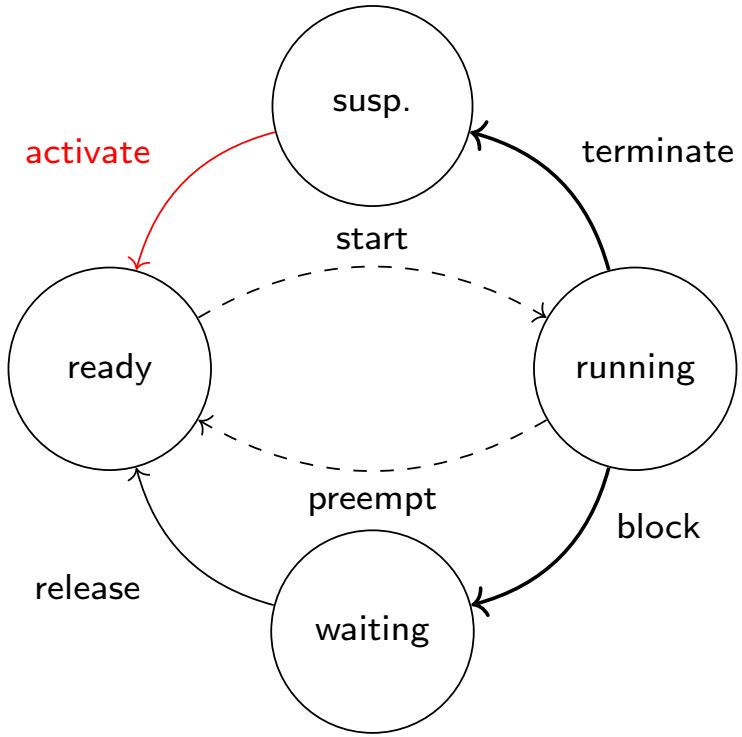
L'API propose 3 types de tâches

- tâche basique : pas d'appel bloquant
- tâche étendue
- ISR : similaire aux tâches basiques mais activée sur une IRQ

En interne, un seul type est utilisé : `tpl_proc`

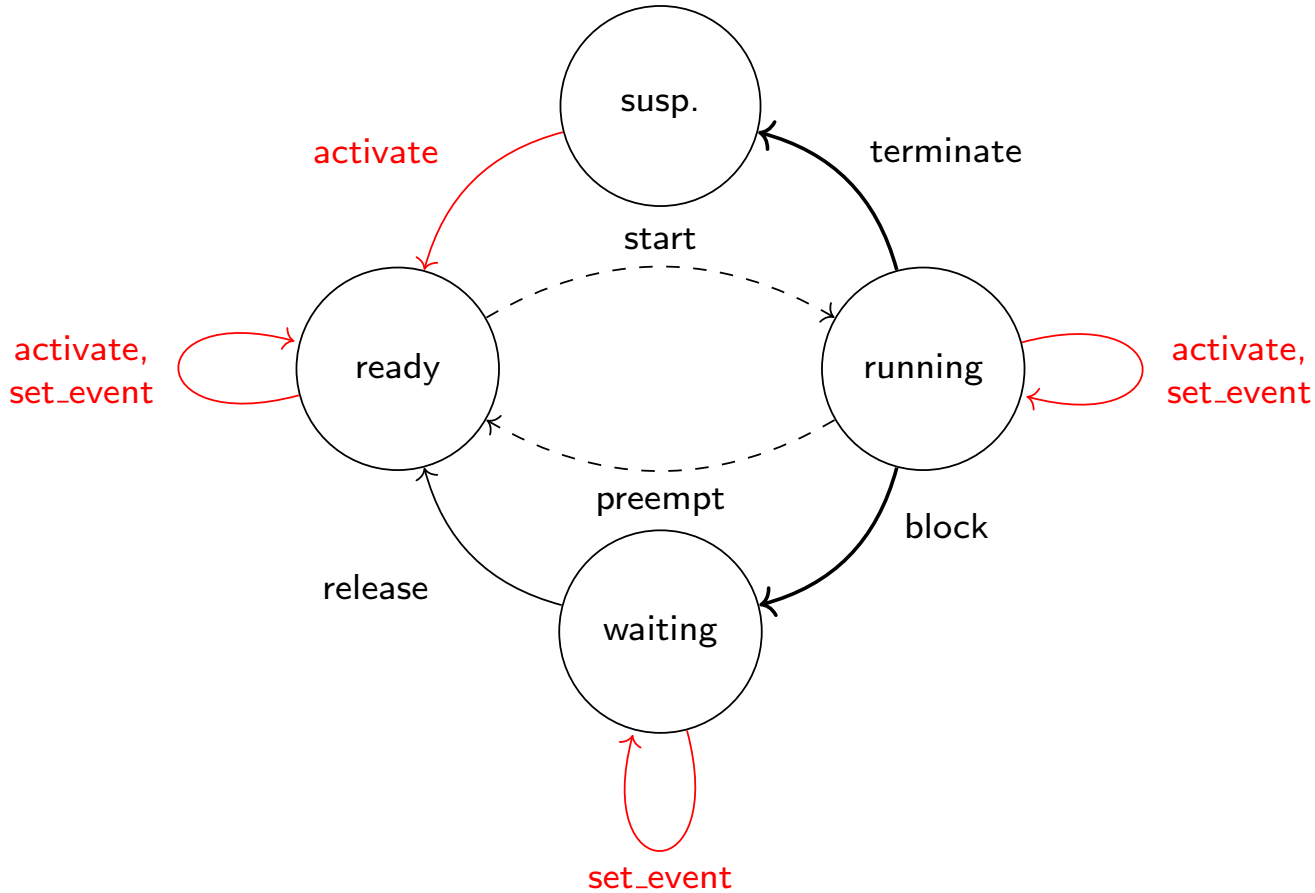
Module *Task Manager*

Le module *Task Manager* fait évoluer l'état des objets `tpl_proc`



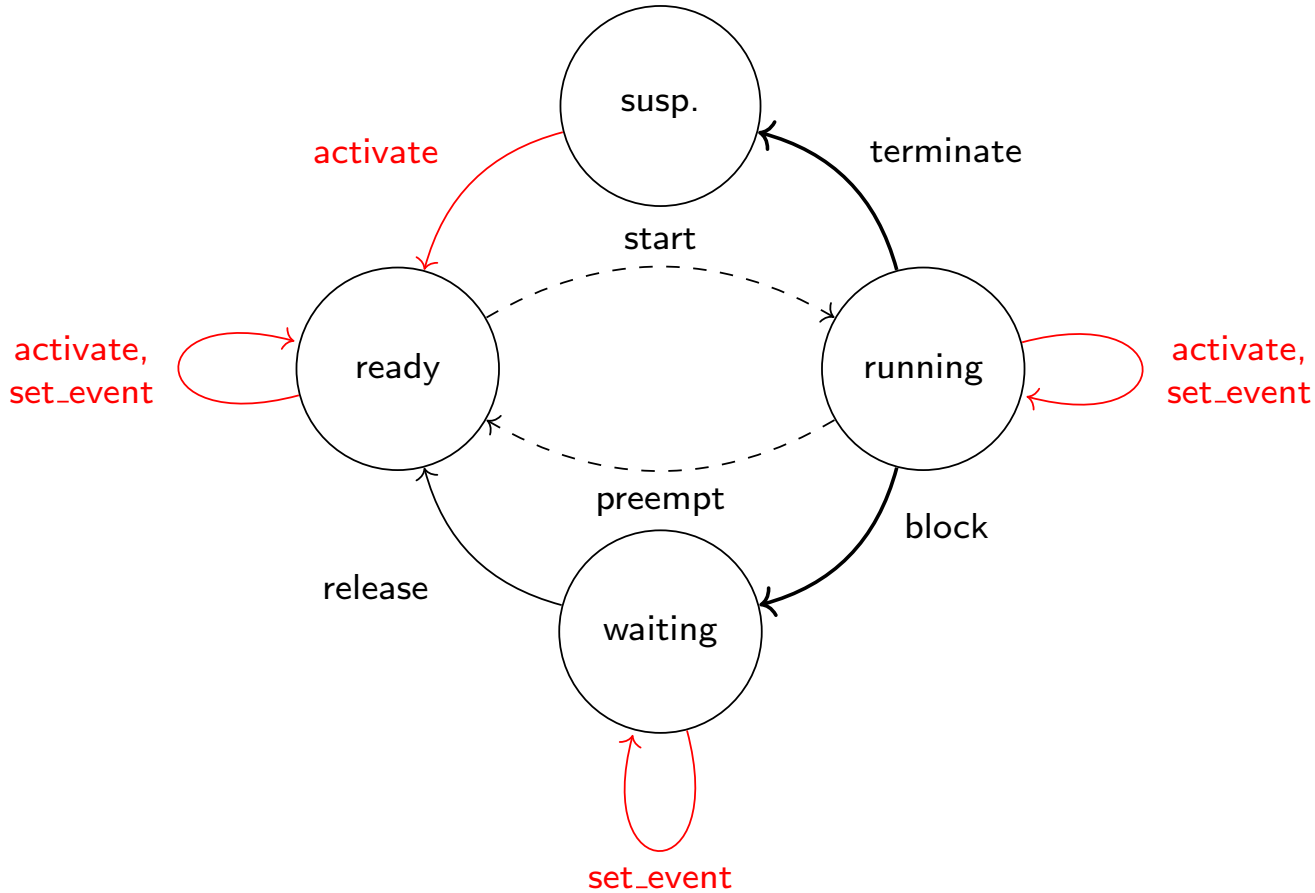
Module *Task Manager*

Le module *Task Manager* fait évoluer l'état des objets `tpl_proc`



Module *Task Manager*

Le module *Task Manager* fait évoluer l'état des objets `tpl_proc`



- Enregistrement des activations jusqu'à une borne fixée
- Enregistrement des événements dans un vecteur de bit

Module *Task Manager*

L'API du module se déduit du modèle

- `tpl_activate_task`
- `tpl_preempt`
- `tpl_start`
- `tpl_terminate`
- `tpl_block`
- `tpl_release`
- `tpl_set_event`

Module *Timing Protection*

Timing protection

Met en œuvre une forme d'isolation temporelle en assurant le respect de trois invariants

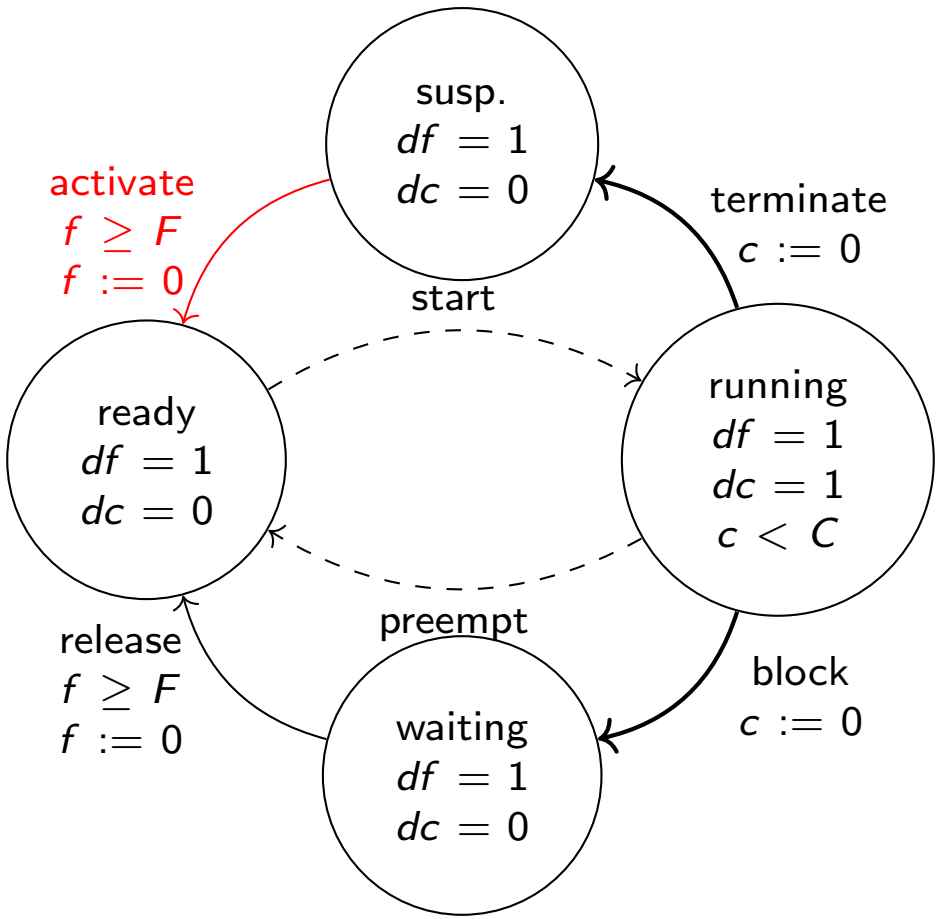
- le *temps d'exécution des jobs* d'une même tâche est inférieur ou égal à un budget C
- le *temps d'exécution de chaque section critique* des *jobs* d'une même tâche est inférieur ou égal à un budget B
- le *délai entre la création de deux jobs* successifs par une même tâche est supérieur ou égal à une borne F

Ces trois invariants renvoient aux attributs du modèle de tâche sporadique

Module *Timing Protection*

L'horloge c mesure le temps d'exécution du *job* $\ll$ actif $\gg$ de la tâche

L'horloge f mesure le temps écoulé depuis la dernière activation d'un *job* de la tâche



Module *Timing Protection*

Recouvrement d'erreur

Appel d'une fonction utilisateur qui choisit :

- Tuer la tâche
- Tuer l'OS-Application
- Re-démarrer l'OS-Application
- Éteindre le système

Section 3

Quelques détails d'implémentations...

Trampoline : un exécutif temps réel

- Présentation
- Processus de construction d'une application
- Architecture interne

Ordonnanceur

L'ordonnanceur dans AUTOSAR a les caractéristiques :

- préemptif à priorité fixe ;
- priorité FIFO pour les jobs de même priorité ;

Ordonnanceur

L'ordonnanceur dans AUTOSAR a les caractéristiques :

- préemptif à priorité fixe ;
- priorité FIFO pour les jobs de même priorité ;

La structure de données doit permettre :

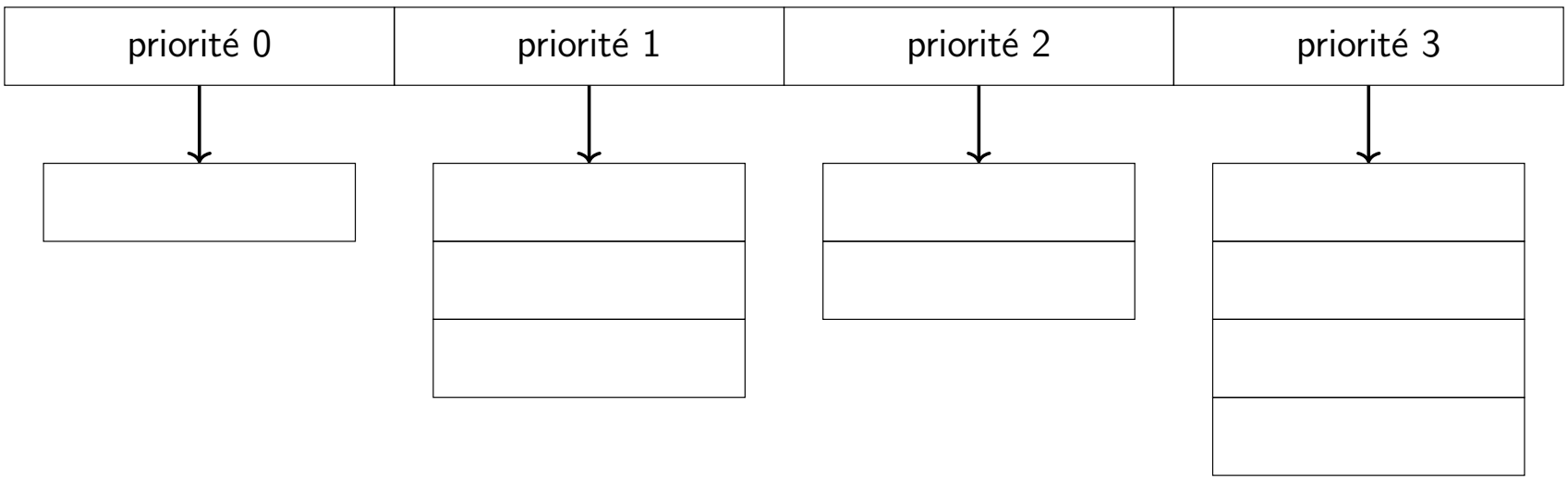
- *d'obtenir* rapidement la tâche la plus prioritaire ;
- *insérer* une nouvelle tâche, pour la priorité donnée, en *fin* de FIFO ;
- *insérer* la tâche préemptée, pour la priorité donnée , en *début* de FIFO ;
- *retirer* de la liste la tâche la plus prioritaire.

Première approche en mono-cœur

La taille de chaque FIFO est calculée par `goil` statiquement en fonction du :

- nombre de tâches de même priorité ;
- nombre d'activations de chaque tâche ;
- priorité des ressources (synchronisation IPCP) ;

Un tableau de FIFO :



Première approche en mono-cœur

- ajout d'une tâche : $O(f)$, où f est le nombre de tâches dans la fifo (avec la même priorité) ;
- consultation de la tâche la plus prioritaire : $O(1)$;
- retrait de la tâche la plus prioritaire : $O(p)$, où p est le nombre de priorités différentes.

Contrainte

Les priorité des tâches doivent être *contigües*.
C'est possible avec `goil` qui recalcule les priorités des tâches en conservant l'ordre relatif.

Approche en multi-cœur

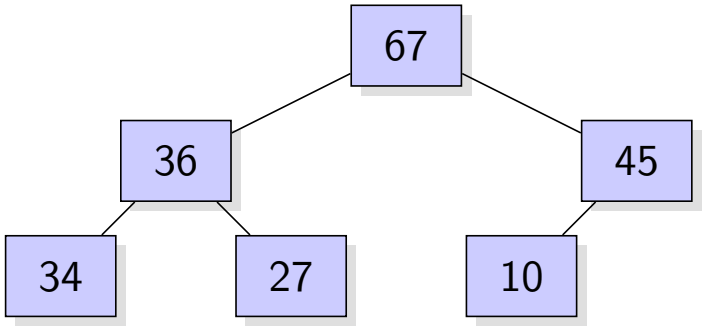
En multi-cœur, l'ordonnancement AUTOSAR est *partitionné* : Il y a une liste des tâche prêtes par tâches, et pas de migration inter-cœur.

Contrainte

La priorité des tâches ne peux pas être recalculée pour être contigüe... car la ressource `RES_SCHEDULER` permettant de rendre une tâche non préemptible est partagée entre tous les cœurs.

Il faut une autre structure de données...

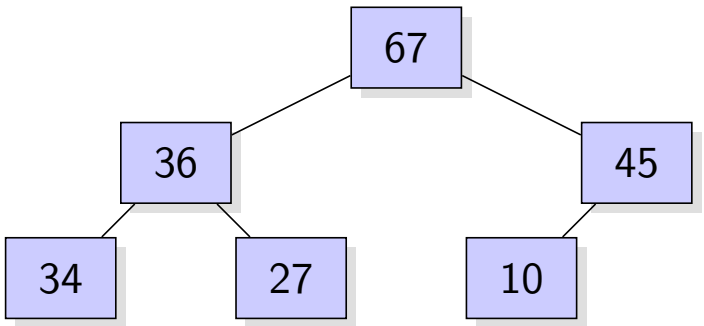
Un tas binaire pour la file de priorités



Structure d'arbre binaire parfait

- structure arborescente ;
- chaque nœud a 2 fils ;
- tous les niveaux sont remplis (sauf le dernier éventuellement), de gauche à droite ;

Un tas binaire pour la file de priorités



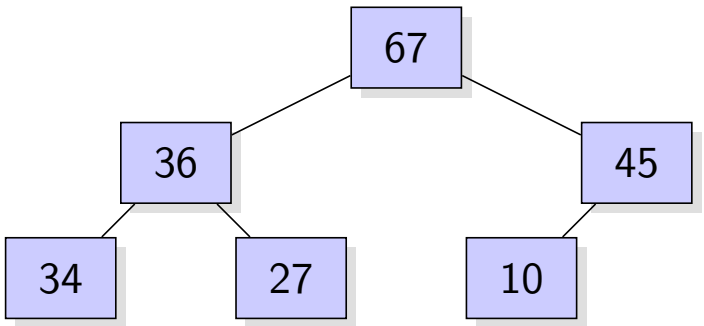
Structure d'arbre binaire parfait

- structure arborescente ;
- chaque nœud a 2 fils ;
- tous les niveaux sont remplis (sauf le dernier éventuellement), de gauche à droite ;

Propriété

La priorité de chaque nœud est *supérieure* à celle de ses fils.

Un tas binaire pour la file de priorités



Structure d'arbre binaire parfait

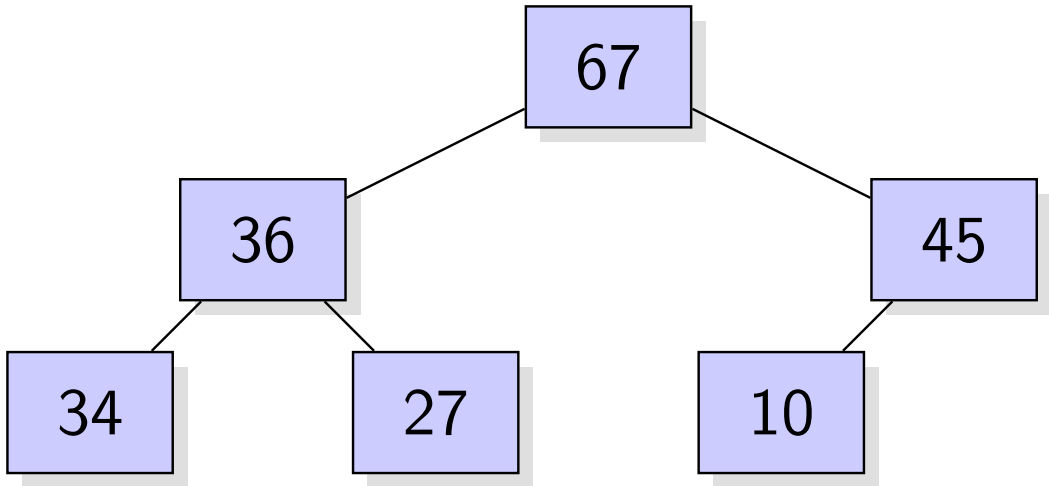
- structure arborescente ;
- chaque nœud a 2 fils ;
- tous les niveaux sont remplis (sauf le dernier éventuellement), de gauche à droite ;

Propriété

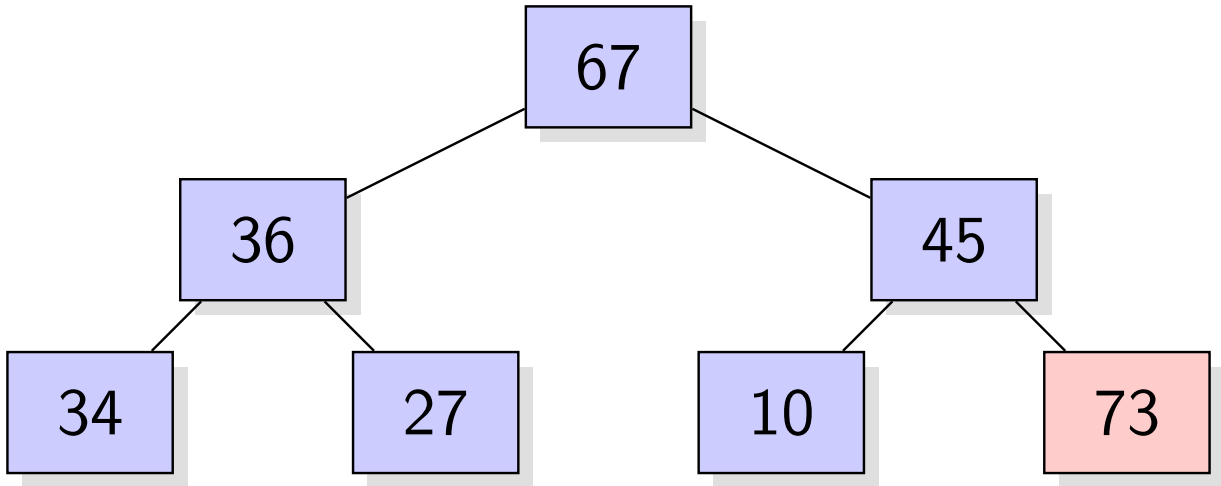
La priorité de chaque nœud est *supérieure* à celle de ses fils.

$\Rightarrow$ L'obtention de la tâche la plus prioritaire est immédiate.

Un tas binaire : Ajout d'un élément

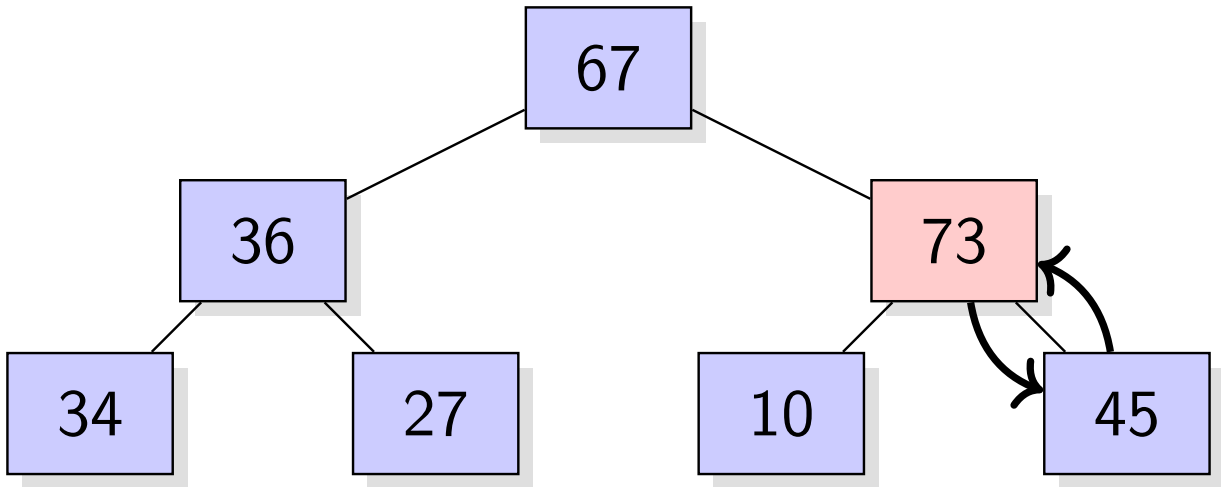


Un tas binaire : Ajout d'un élément



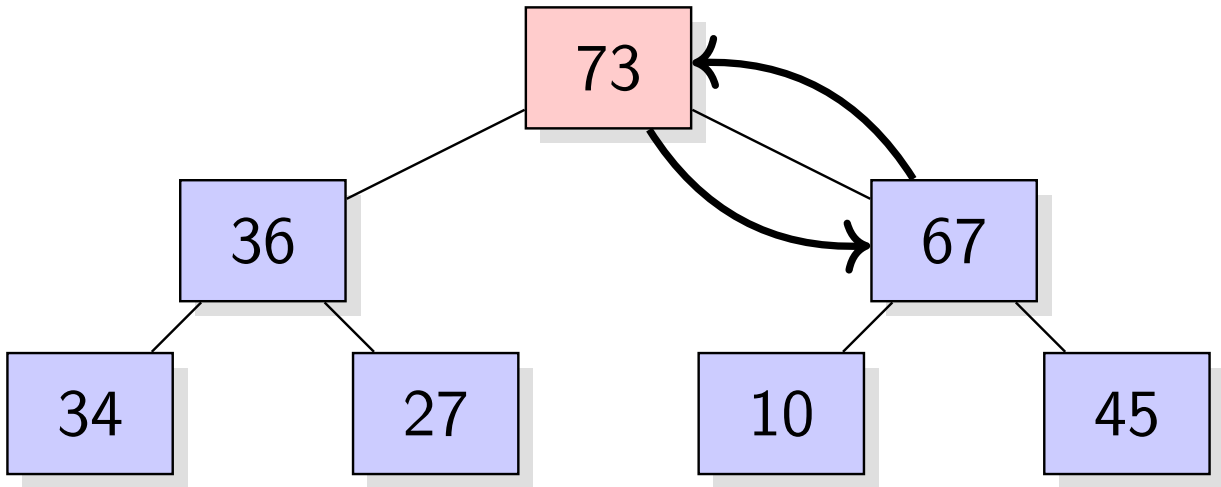
- on ajoute l'élément à la fin du tas ;

Un tas binaire : Ajout d'un élément



- on ajoute l'élément à la fin du tas ;
- on remonte ensuite l'élément à sa place pour respecter l'ordre des priorités

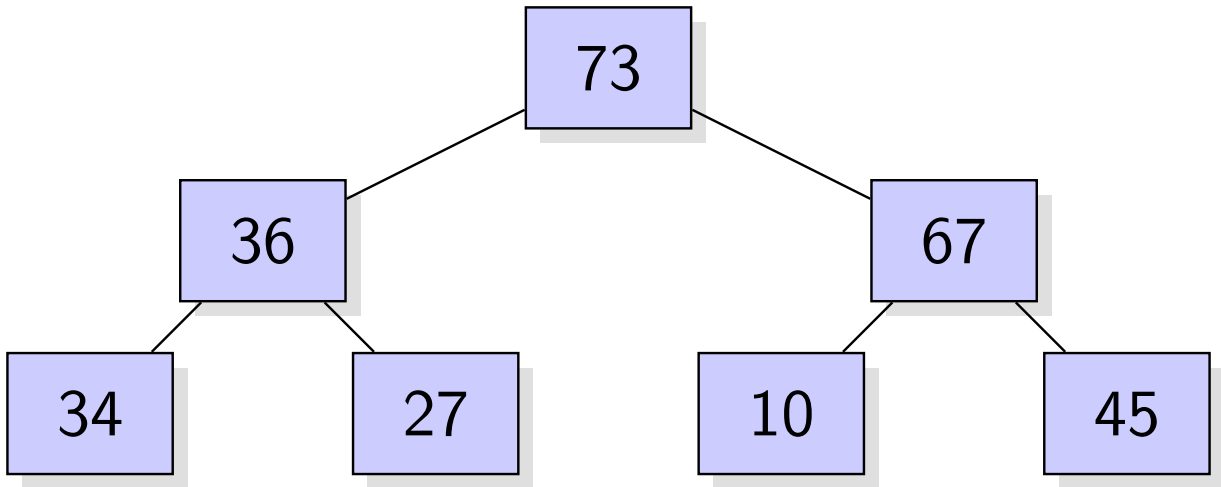
Un tas binaire : Ajout d'un élément



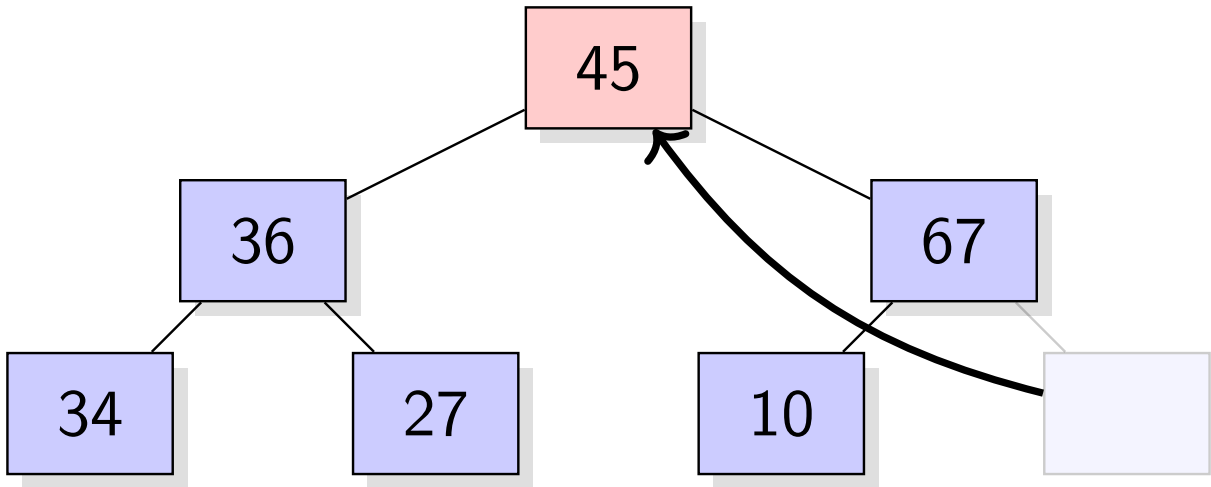
- on ajoute l'élément à la fin du tas ;
- on remonte ensuite l'élément à sa place pour respecter l'ordre des priorités

La complexité est en $O(log(n))$, où n est le nombre de tâches dans la liste des tâches prêtes.

Un tas binaire : Suppression de la racine

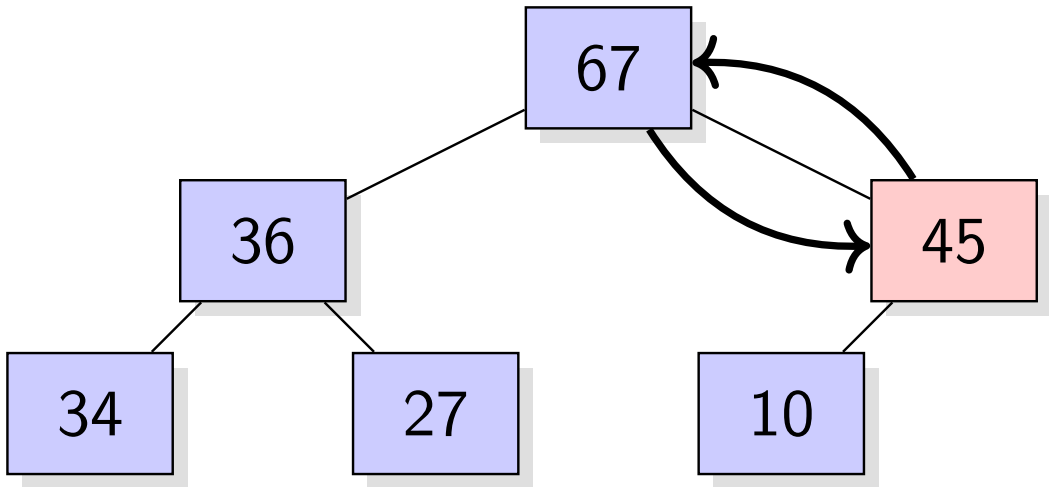


Un tas binaire : Suppression de la racine



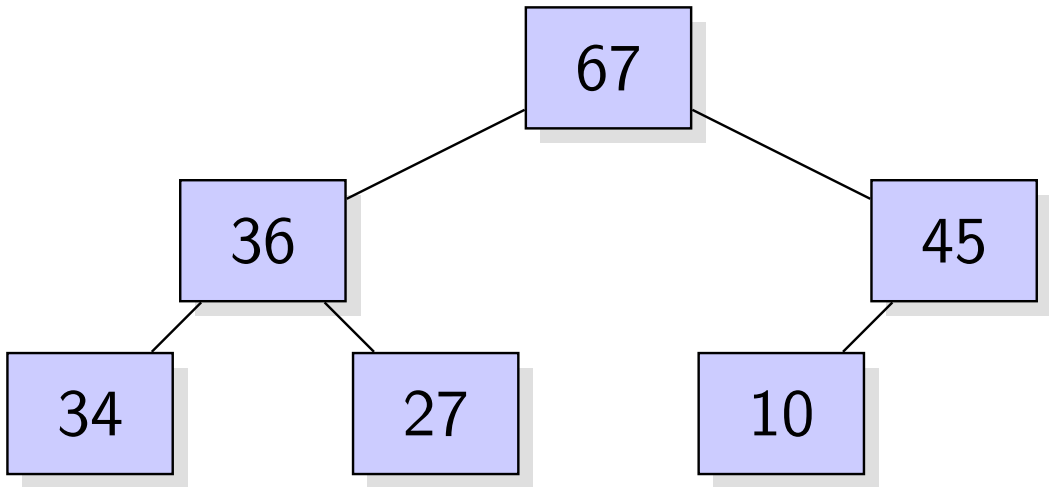
- on déplace le dernier élément à la racine ;

Un tas binaire : Suppression de la racine



- on déplace le dernier élément à la racine ;
- on échange le père avec le fils de plus haute priorité, pour respecter les priorités, récursivement.

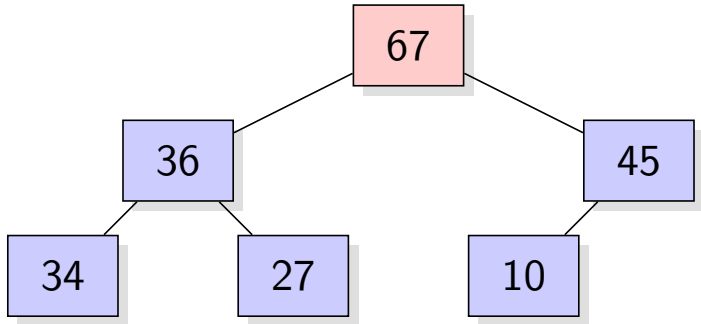
Un tas binaire : Suppression de la racine



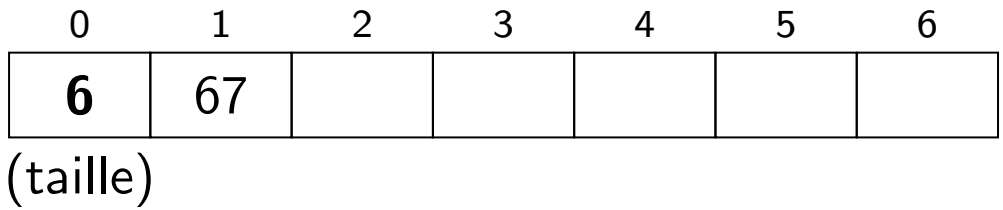
- on déplace le dernier élément à la racine ;
- on échange le père avec le fils de plus haute priorité, pour respecter les priorités, récursivement.

La complexité est en $O(log(n))$, où n est le nombre de tâches dans la liste des tâches prêtes.

Un tas binaire - représentation

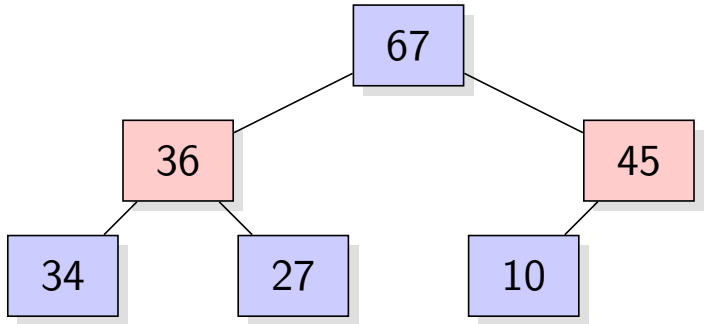


Représentation en mémoire avec un tableau :

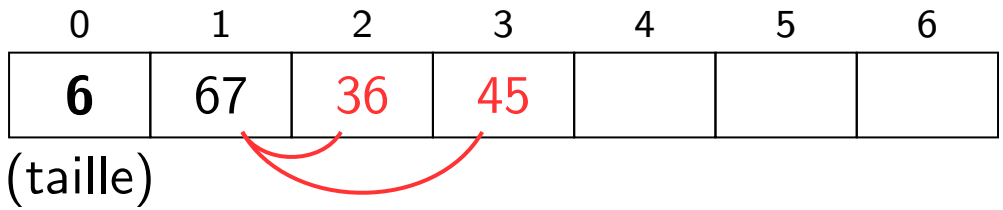


- les fils sont aux index $2n$ et $2n+1$.
- le père du nœud i se situe à l'index $\lfloor \frac{i}{2} \rfloor$

Un tas binaire - représentation

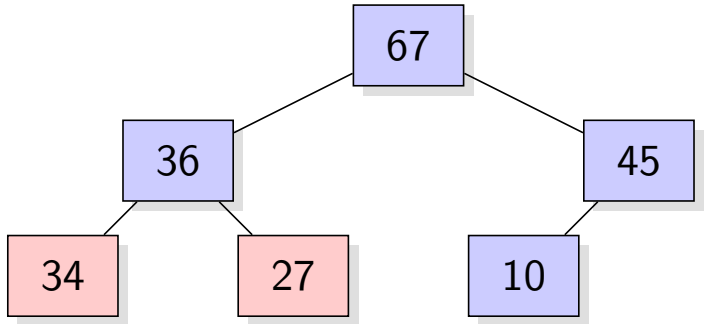


Représentation en mémoire avec un tableau :

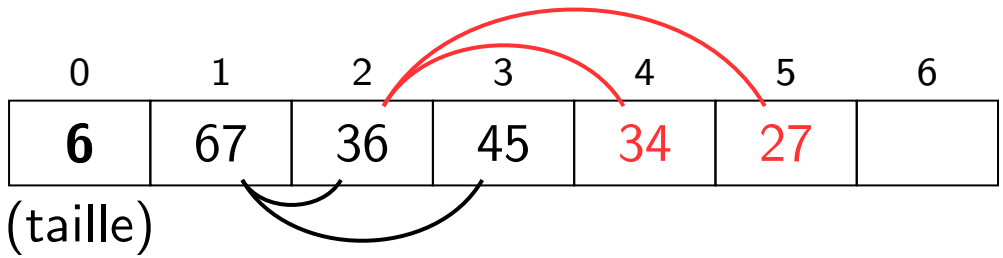


- les fils sont aux index $2n$ et $2n+1$.
- le père du nœud i se situe à l'index $\lfloor \frac{i}{2} \rfloor$

Un tas binaire - représentation

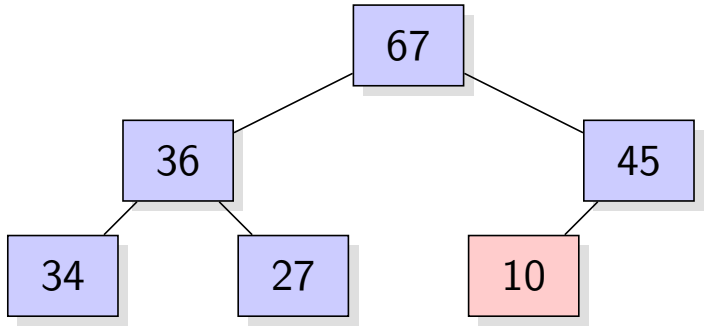


Représentation en mémoire avec un tableau :

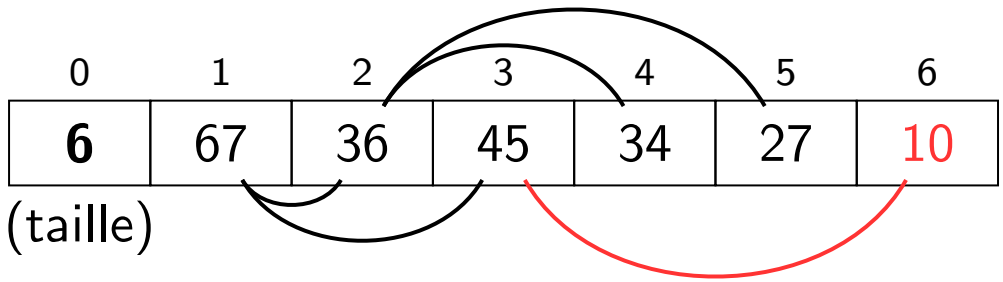


- les fils sont aux index $2n$ et $2n+1$.
- le père du nœud i se situe à l'index $\lfloor \frac{i}{2} \rfloor$

Un tas binaire - représentation



Représentation en mémoire avec un tableau :



- les fils sont aux index $2n$ et $2n+1$.
- le père du nœud i se situe à l'index $\lfloor \frac{i}{2} \rfloor$

Un tas binaire - gestion de la fifo

On ne peut plus intégrer une FIFO simplement avec cette structure de données. On utilise alors une *priorité dynamique* qui dépend de :

- la priorité statique
- l'ordre d'activation (rang)

priorité	rang

- Lors de l'activation d'un processus de priorité P_i, on calcule la priorité dynamique, P_{dyn}, du job. $P_{dyn} = P_i||R^c_{p_i}$ puis on décrémente le rang courant $R^c_{p_i}$;

Comparaison de 2 tâches de même priorités :

$$P_{dyn}c = P_i||(R_{p_i} - R^c_{p_i}) \tag{1}$$

On compare les rang, modulo le nombre de rang maximal

Principe

La communication inter-cœur se fait par interruption.

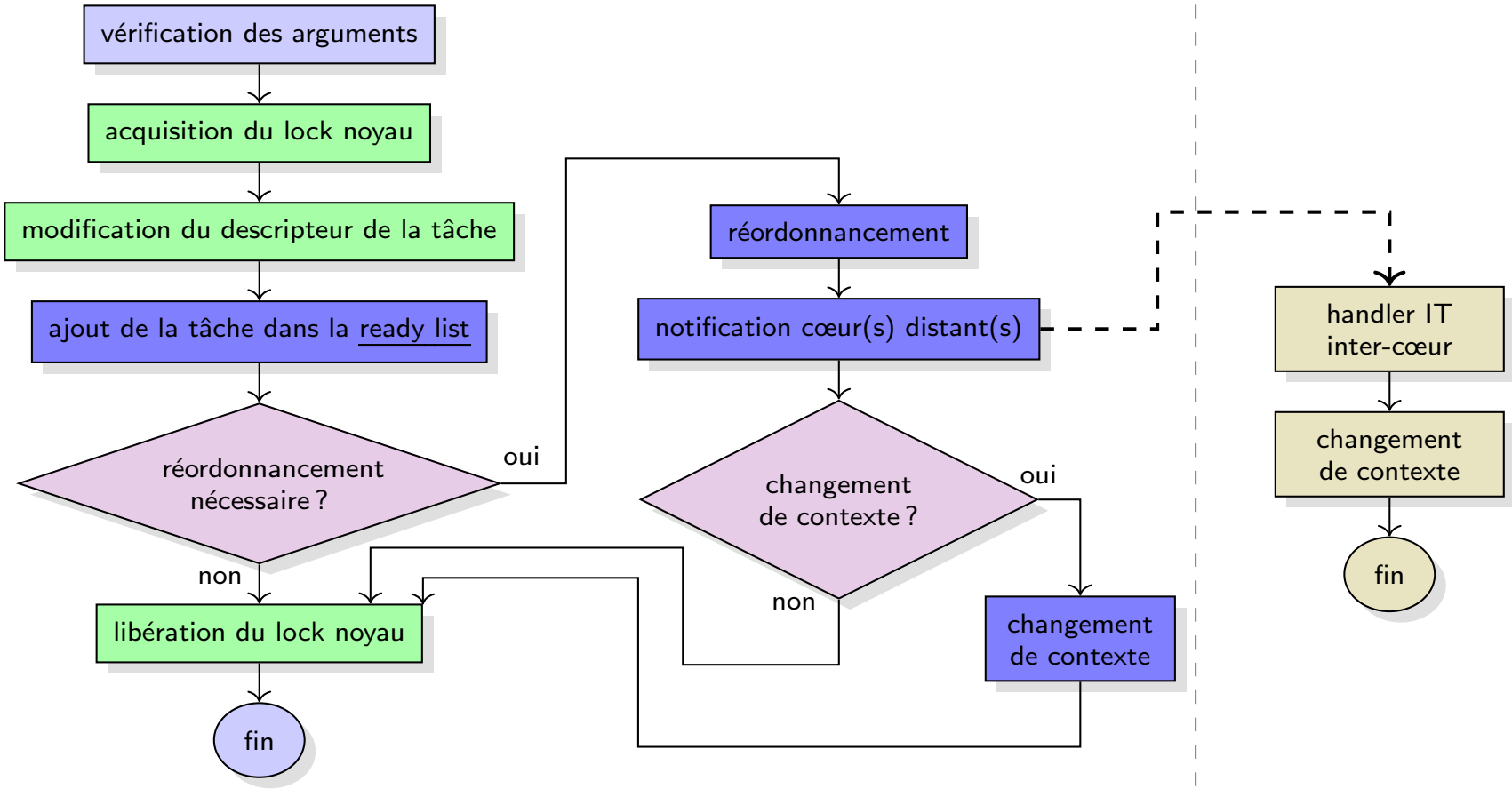
La structure interne `tpl_kern` est dupliquée pour chaque cœur.

L'exécution d'un service conduisant à un changement de contexte est découpé en 2 parties :

- La première partie est exécutée sur le cœur où s'exécute *l'appelant*. L'appelant est soit un processus, soit une alarme ou une schedule table. Elle consiste à exécuter le service en lui même puis à *exécuter le réordonnancement* qui en résulte.
- La seconde partie est exécutée sur le *cœur cible* et consiste à effectuer le *changement de contexte*.

Changement de contexte en multicœur

Cas de l'activation d'une tâche sur un autre cœur :



Évolution de la structure `tpl_kern` (une par cœur)

`running_id` : l'identifiant du processus en cours d'exécution ;

`running` : un pointeur vers le descripteur dynamique du processus en cours d'exécution ;

`s_running` : un pointeur vers le descripteur statique du processus en cours d'exécution ;

`elected_id` : l'identifiant du processus qui réclame le CPU ;

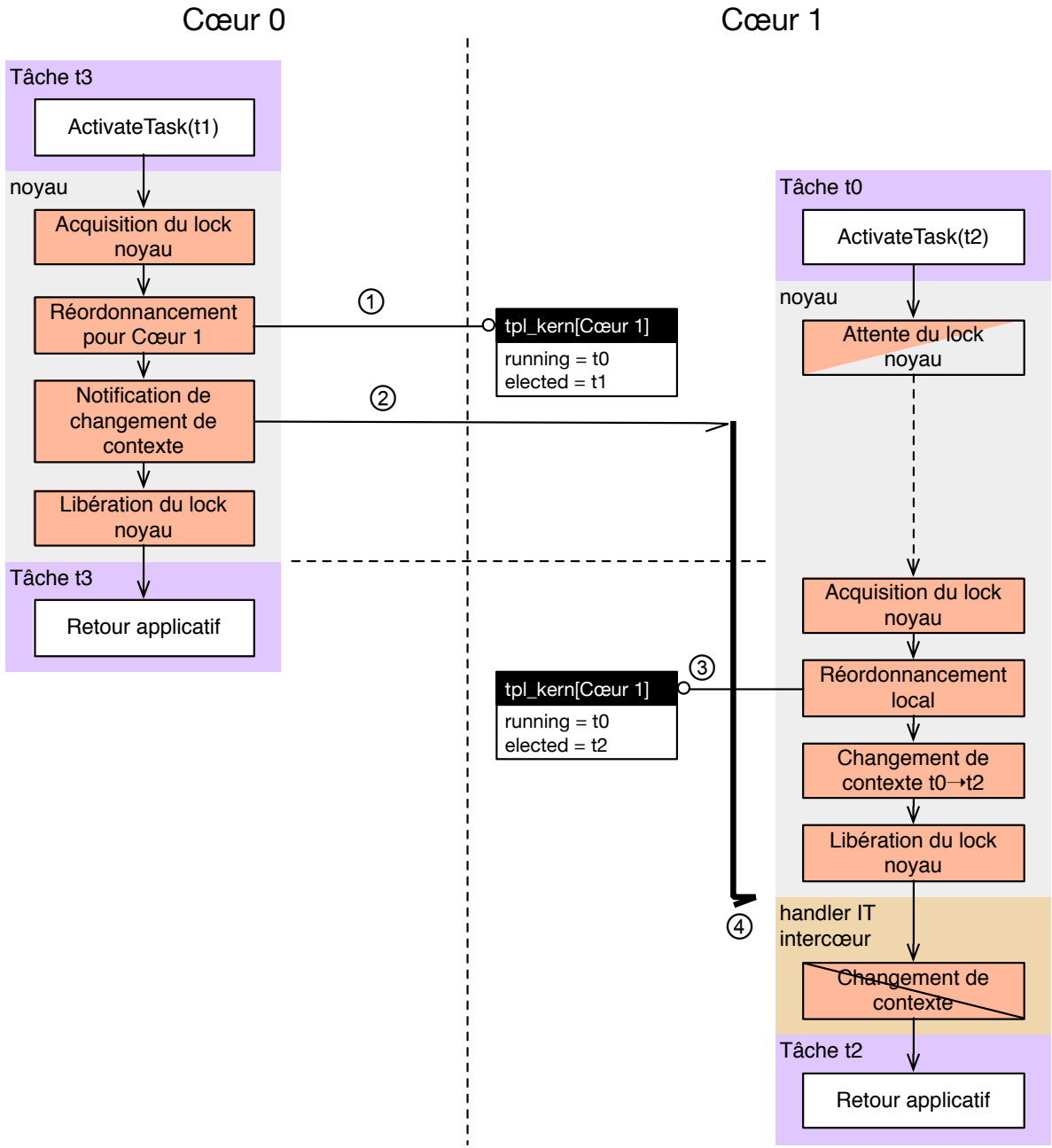
`elected` : un pointeur vers le descripteur dynamique du processus qui réclame le CPU ;

`s_elected` : un pointeur vers le descripteur statique du processus qui réclame le CPU ;

`need_switch` : indique qu'un changement de contexte entre le processus en cours d'exécution, `running`/`s_running`, et le processus qui le réclame, `elected`/`s_elected`, doit être effectué ;

`need_schedule` : indique qu'un réordonnancement doit être effectué.

Scénario d'activation inter-cœur



Section 4

Conclusions

Travaux passés et en cours dans l'équipe

Thèse de Dominique Bertrand

- Dimensionnement des paramètres du module de protection temporelle

Thèse de Sylvain Cotard

- Développement d'un service de vérification en ligne
- Développement d'un service de communication *wait free*

Thèse de Toussaint Tigori

- Génération de noyau spécialisé pour l'application

Thèse de Louis-Marie Givel

- Test de propriétés temps réel par injection de délais

Travaux passés et en cours dans l'équipe

Thèse de Dimitry Solet

- Développement d'un service de vérification en ligne par une approche matérielle (en utilisant un SoPC).

Thèse de Khaoula Boukir

- Mise en œuvre de politiques d'ordonnancement temps réel multiprocesseur prouvées.

Thèse de Joumana Lagha

- Trampoline pour le paradigme normally-off and instant-on

Trampoline

- Exécutif temps réel pour « petit » micro-contrôleurs, conforme OSEK/VDX OS et AUTOSAR OS
- Système statique : ne pas hésiter à abuser de cette propriété
- Architecture modulaire, chaîne de développement extensible
- Logiciel libre sous licence GPL