

Implantation de systèmes synchrones temps réel sur des architectures multiprocesseurs

Mikel Cordovilla Mesonero (mikel.cordovilla@onera.fr)

ONERA–DTIM, Toulouse

ISAE, Toulouse

Thèse encadrée par : Frédéric Boniol, Eric Noulard et Claire Pagetti (ONERA–DTIM, Toulouse)

Résumé *Cet article présente un environnement de développement pour l'implantation d'algorithmes d'ordonnancement temps réel sur des architectures multicœurs. Le choix s'est tourné vers la mise en œuvre d'une librairie en C suivant le standard POSIX. Cette approche assure une portabilité vers de nombreux systèmes d'exploitation temps réel compatibles POSIX, et en particulier le noyau Linux standard. L'utilisateur pourra soit utiliser des ordonnanceurs déjà fournis avec l'environnement, comme *rate monotonic* ou *earliest deadline first*, soit programmer son propre algorithme en appelant des primitives de haut niveau sans avoir à gérer un certain nombre de détails de bas niveau.*

A. INTRODUCTION

L'évolution des systèmes embarqués critiques a conduit à l'augmentation des fonctionnalités et au besoin d'utiliser de nouvelles architectures plus puissantes. Nous nous intéressons dans cette thèse aux architectures de type multicœurs homogènes. On parle de processeur multicœurs s'il est composé d'au moins deux cœurs identiques gravés au sein d'une même puce partageant une mémoire.

La thèse s'inscrit dans le cadre de l'implantation de systèmes embarqués critiques de type contrôle-commande. L'idée est de partir d'une spécification fonctionnelle exprimée dans un langage formel synchrone. Nous avons choisi de partir de systèmes décrits dans le langage PRELUDE défini par J. Forget [6]. Ce langage a la particularité de fournir une couche d'abstraction qui décrit l'architecture logicielle temps réel comme un assemblage de plusieurs systèmes synchrones localement mono-périodiques en un système global multi-périodique. Le langage permet de préciser des schémas de communication déterministes complexes entre plusieurs fonctions externes écrites dans un langage de programmation (comme du C ou du Lustre).

A partir d'une spécification fonctionnelle, on obtient des tâches périodiques avec précédences. Dans sa thèse, J. Forget a proposé une implantation multithreadée mono-cœur. Notre objectif est de passer sur des modèles multithreadés multiprocesseurs. Bien maîtrisé dans le cas monoprocesseur, l'aspect des dépendances est encore peu traité en multiproces-

seurs. Il nous faut donc définir des algorithmes multiprocesseurs corrects en regard de la sémantique du langage et générer le code ad hoc.

Nous devons proposer une méthodologie d'implantation avec les contraintes suivantes :

- conservation de la spécification fonctionnelle exprimée avec un langage de programmation de haut niveau (dans notre cas le langage PRELUDE) ;
- respect des contraintes temps réel ;
- respect des contraintes de sûreté de fonctionnement.

Face à l'inexistence dans la bibliographie d'outils d'exécution adéquats, nous avons programmé une librairie générique pour l'implantation d'algorithmes d'ordonnancement temps réel sur des architectures multicœurs. Cette librairie s'exécute au dessus du système d'exploitation et nous fournit une couche d'abstraction pour la définition des tâches temps réel et leurs interactions avec le système. Cette librairie est programmée selon le standard POSIX qui assure une portabilité vers de nombreux systèmes d'exploitation temps réel et en particulier le noyau Linux standard.

Le papier s'organise comme suit : dans la partie B, nous rappellerons les politiques d'ordonnancement temps réel. Dans la partie C, nous rappellerons la manière dont sont généralement implantés les ordonnanceurs temps réel et nous détaillerons l'architecture logicielle de la librairie. Enfin dans la partie D, nous détaillerons le fonctionnement et l'utilisation de la librairie générique. Dans la conclusion

nous donnerons les pistes de travail en cours de développement.

B. RAPPEL SUR LA THÉORIE DE L'ORDONNANCEMENT

1. Modèle de tâches

Pour un ensemble de tâches temps réel à exécuter, un algorithme d'ordonnancement consiste à organiser la réalisation des tâches tout en satisfaisant les contraintes temporelles. Ainsi un ordonnancement correct ne dépend pas uniquement du résultat fonctionnel, c'est-à-dire des valeurs calculées, mais également du respect des échéances prévues.

Dans [9], Liu et Layland posent les bases pour l'étude du temps réel. On peut formellement représenter une tâche périodique par quatre paramètres $\tau = (T, C, r, d)$: la période (T), le pire temps d'exécution (C), la date de réveil (r) et l'échéance (d).

2. Ordonnancement monoprocesseur

De nombreuses recherches ont été menées pour proposer des algorithmes simples, efficaces, optimaux (c'est-à-dire que pour un ensemble de tâches temps réel, s'il existe un ordonnancement valide appartenant à la même famille d'algorithmes, un algorithme optimal ordonnancera correctement les tâches), accompagnés de tests d'ordonnancabilité (c'est-à-dire qu'il est possible de décider si un ensemble de tâches est ordonnancable pour la politique choisie). Une présentation détaillée des résultats sur les ordonnanceurs monoprocesseur se trouve dans [11].

A titre d'exemple, on rappelle deux algorithmes d'ordonnancement basés sur des priorités [9]. RM (Rate Monotonic) est un algorithme à priorité fixe où la priorité est inversement proportionnelle à la taille de la période. Cette politique est optimale pour des ensembles de tâches indépendantes telles que $d = T$ et $r = 0$. EDF (Earliest Deadline First) est une politique à priorité dynamique définie en tenant compte du temps d'exécution. La priorité est inversement proportionnelle à l'échéance de la tâche à l'instant de calcul. Cette politique est optimale pour un ensemble de tâches indépendantes avec $d \leq T$.

Il existe plusieurs tests d'ordonnancabilité pour ces deux politiques. Par exemple, pour un ensemble de n tâches $\tau_i = (T_i, C_i, r_i, d_i)$, on peut raisonner sur le facteur d'utilisation $U = U_{sum} = \sum_{i=1}^n \frac{C_i}{T_i}$.

1. pour RM, une condition suffisante est : $U \leq n(2^{1/n} - 1)$
2. pour EDF et $d = T$ une condition nécessaire et suffisante est $U \leq 1$

3. Ordonnancement multicœurs

Dans le cas des systèmes multicœurs les tech-

niques et résultats du cas monoprocesseur ne se généralisent pas toujours. On peut distinguer deux familles de stratégies : partitionnées et globales.

Stratégie partitionnée L'ordonnancement partitionné se divise en deux étapes : (a) répartition de l'ensemble de tâches en plusieurs sous-ensembles, chacun associé à un cœur et (b) ordonnancement standard sur chaque cœur. Cette approche est illustrée dans la figure 1.

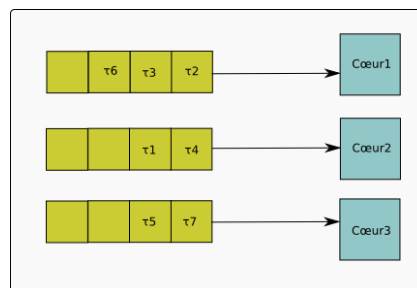


Fig. 1 – Plusieurs files dans le cas partitionné

L'allocation des tâches sur les cœurs (i.e. la définition des partitions) correspond à un problème de *bin packing* [5] : il faut trouver un rangement valide de n articles de taille C_i dans un nombre fini de boîtes de capacité C . Ce problème est NP-complet. Pour un partitionnement, un objet correspond à une tâche, la taille d'une tâche est $u_i = C_i/T_i$, une boîte est un cœur et la capacité dépend de la politique d'ordonnancement sur le cœur.

Plusieurs heuristiques ont été proposées pour trouver rapidement des solutions [10]. Par exemple, l'heuristique First Fit (FF) alloue les tâches séquentiellement sur le premier processeur qui convient. Best Fit (BF) parcourt également les tâches séquentiellement mais les place sur le processeur qui a la plus petite capacité disponible suffisante. Worst Fit (WF) alloue toujours les tâches séquentiellement sur le processeur qui a la plus grande capacité. On peut également ordonner les tâches selon un paramètre. Par exemple, l'heuristique Increasing Utilization (IU) ordonne les tâches par ordre croissant du facteur d'utilisation u_i et Decreasing Utilization (DU) par ordre décroissant.

Une fois les tâches affectées à leur cœur, on se ramène au cas monoprocesseur pour chaque sous-ensemble. La migration, c'est-à-dire, le changement de processeur en cours d'exécution, est interdite. DFF-PRM (on trie les tâches selon DU, on cherche un partitionnement avec l'heuristique First Fit et on ordonnance avec Rate Monotonic) ou IWF-PEDF (tri par IU, heuristique WF et ordonnancement EDF) sont deux exemples d'ordonnanceurs multi-processeurs partitionnés.

Stratégie globale Les algorithmes globaux exécutent à chaque instant les tâches les plus prioritaires sur les m cœurs. On peut naturellement

étendre les ordonnanceurs monoprocesseur en stratégie globale. On obtient ainsi les politiques GRM (Global Rate Monotonic) et GEDF (Global Earliest Deadline First).

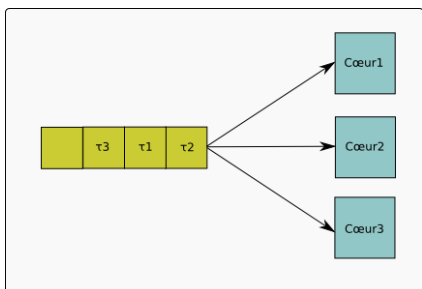


Fig. 2 – Une file dans le cas global

L'article [2] recense l'état des travaux de recherche pour un grand nombre de politiques globales et les tests d'ordonnabilité associés. Il n'existe pas réellement de conditions et d'analyse d'ordonnabilité performantes. On trouve surtout des conditions suffisantes d'ordonnabilité ou des conditions nécessaires et suffisantes de complexité exponentielle.

Seule la politique PFair [3] a la bonne propriété d'être optimale pour $r = 0$ et $T = d$. L'idée est de découper les tâches en fenêtres d'exécution, les ordonner en fonction du facteur d'utilisation et placer des bouts de chaque tâche dans chaque fenêtre. La condition d'ordonnabilité pour $r = 0$ et $T = d$ est alors très simple : $u_{sum} \leq m$ et $u_{max} = \max u_i \leq 1$.

Gestion des précédences Le traitement des précédences, dans un système monoprocesseur, peut se faire en utilisant deux approches différentes : la première consiste à utiliser des sémaphores pour bloquer l'exécution des tâches en précédences. Si on a une précédence $\tau_1 \rightarrow \tau_2$, t_1 relâche le sémaphore à la fin de son exécution et τ_2 peut alors s'exécuter. La deuxième approche consiste à modifier les échéances et les dates de réveil des tâches pour forcer l'ordre d'exécution nécessaire. Chetto et al. [4] ont proposé une solution optimale pour un système ordonné avec EDF. L'utilisation de sémaphores est optimale, mais les tests d'ordonnabilité peuvent être très complexes. La deuxième approche, bien qu'en générale est sous-optimal, réutilise les tests d'ordonnabilité classiques.

Actuellement, dans le cas de multicœur, les précédences sont traitées pour les stratégies partitionnées : on place sur un même cœur les tâches en relation. Malheureusement, si la relation de précédence est très contrainte, on se ramène à un cas monoprocesseur. Pour les stratégies globales, l'approche par encodage ne s'applique plus (ou alors il faut réfléchir à un nouvel algorithme). L'idée est alors d'utiliser

des sémaphores pour bloquer les exécutions. Cette approche compliquera d'autant plus les tests d'ordonnabilité.

C. IMPLANTATION D'UN ORDONNANCEUR TEMPS RÉEL

L'objectif de la thèse est de générer du code multicœurs à partir d'une spécification fonctionnelle temporelle exprimée dans le langage PRELUDE. Un axe de travail est de proposer du code qui puisse être porté sur plusieurs architectures cibles. Nous n'avons pas trouvé d'outils correspondant à notre besoin. Le plus proche est LitmusRT [1] mais il est peu portable et donc difficilement utilisable dans notre cas. Nous nous sommes donc tournés vers la mise en œuvre d'une librairie généraliste qui nous permettra de définir facilement des algorithmes spécifiques.

1. Outils disponibles

De nombreux noyaux temps réel ont été implantés pour le développement de systèmes temps réel. On peut notamment citer RTAI [13], RTEMS [14], Xenomai [18], VxWorks [16], LitmusRT [1] ou MarteOS [12]. Tous ces systèmes d'exploitation disposent du support multiprocesseurs à l'exception de MarteOS. Certains comme RTEMS ont un ordonnanceur du type multi-monoprocesseur et d'autres comme Xenomai ou VxWorks ont un ordonnanceur du type multiprocesseurs symétriques. RTAI quant à lui offre les 2 possibilités.

Le Metascheduler [8] se base sur une autre approche. Il s'agit d'une librairie en C qui permet l'ordonnement temps réel monoprocesseur. Elle a été programmée suivant le standard POSIX. L'architecture du système permet d'ajouter des ordonnanceurs facilement. Si l'idée est prometteuse, les choix d'architecture logiciel nous semblent peu compatibles pour réaliser une extension aisée en multiprocesseurs.

Storm [17] et Cheddar [15] sont deux simulateurs, qui offrent une interface graphique pour la visualisation d'ordonnement multiprocesseurs. Ces outils sont spécifiquement conçus pour que l'utilisateur développe facilement de nouveaux algorithmes mais ne disposent pas de mécanismes pour une exécution sur une machine cible.

2. Standard POSIX

POSIX [7] (Portable Operating System Interface for uniX) est un ensemble de standards consacré à la définition d'API logicielle pour les systèmes d'exploitation. La plupart des systèmes bâtis sur les noyaux temps réel cités précédemment sont conformes soit totalement soit partiellement à POSIX. Les systèmes utilisant les noyaux Linux actuels le sont également.

Dans un souci de portabilité, nous avons codé notre librairie en nous appuyant sur POSIX. Nous avons utilisé les API sur les threads, les primitives de synchronisation (mutex et sémaphores), les objets temporels (chronomètres) et les communications inter-processus (signaux) et bien sûr les fonctionnalités de contrôle de l’ordonnancement (scheduling).

3. Choix réalisés

Nous nous sommes inspirés des philosophies de MarteOS, du Metascheduler et de Storm : un utilisateur doit pouvoir rapidement prototyper son algorithme d’ordonnancement et l’exécuter sur un ensemble de tâches temps réel et un système d’exploitation compatible POSIX. Ce choix permet de porter les programmes sur tous les noyaux temps réel présentés. On s’abstrait ainsi du suivi des évolutions et des patches des systèmes d’exploitation.

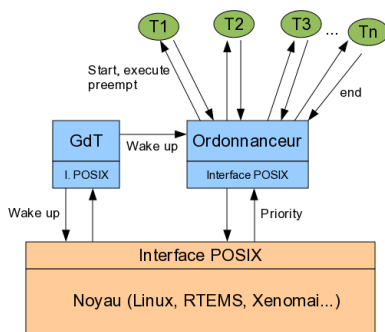


Fig. 3 – Interaction globale

La figure 3 représente la sur-couche de développement. La librairie gère trois entités :

1. tâches à ordonnancer : responsables de la fonctionnalité du programme à exécuter.
2. ordonnanceur : responsable de la gestion de l’exécution des tâches. Il décide quand une tâche accède à une ressource de calcul.
3. gestionnaire du temps (GdT) : responsable d’un compteur temporel. Son objectif est de réveiller l’ordonnanceur aux bons moments.

Comportement des tâches Le cycle de vie d’une tâche se compose de quatre états décrits dans la figure 4. A sa création, la tâche est dans l’état *New*. Quand elle devient exécutable, elle entre dans l’état *Idle*. L’ordonnanceur indique à la tâche *Idle* quand elle peut accéder à une ressource CPU. Elle est alors dans l’état *Executing*. Quand la tâche se termine, elle passe dans l’état *Waiting* en attendant la nouvelle activation à sa prochaine période. Toutes les transitions sont conduites par l’ordonnanceur à part *end* qui est décidée par la tâche.

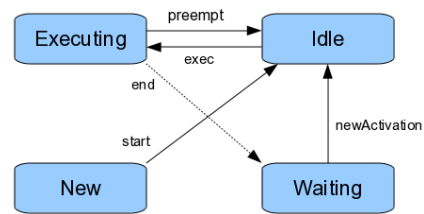


Fig. 4 – Cycle de vie des tâches

Gestionnaire de temps Le gestionnaire de temps est une entité très simple, qui gère des chronomètres et se réveille à leur expiration. Il réveille alors l’ordonnanceur qui doit prendre des décisions d’ordonnancement. Cette politique est adaptée à notre problème car nous visons des ensembles de tâches multipériodiques.

Ordonnanceur L’ordonnanceur gère une file d’attente dans le cas d’une stratégie globale ou plusieurs dans le cas d’une stratégie partitionnée. En tenant à jour ces files et avec les réveils du gestionnaire de temps, il connaît les tâches à exécuter et celle(s) à préempter. Ces choix sont naturels et très similaires à ceux faits dans les noyaux ou les outils classiques de manipulation d’ordonnanceurs. Notre ordonnanceur manipule des structures très simples car nous ne visons pas l’ordonnancement d’un grand nombre de tâches.

D. IMPLANTATION

Cette partie décrit la mise en œuvre pratique de la librairie d’aide à la conception d’algorithmes d’ordonnancement temps réel multicœurs.

1. Initialisation

D’un point de vue implantation, toutes les entités, à savoir les tâches temps réel, l’ordonnanceur et le gestionnaire de temps, sont des threads POSIX. Le nombre de threads à créer dépend du type d’ordonnanceur.

Dans le cas d’une stratégie globale (notamment monoprocesseur) on dispose d’un seul ordonnanceur pour toutes les tâches. Le système se compose alors de l’ensemble de tâches et l’ordonnanceur associé, générés au démarrage du système. Dans le cas d’une politique partitionnée, le système se compose de l’ensemble de tâches à ordonnancer et d’autant d’ordonnanceurs que de cœurs d’exécution. La figure 5 résume les possibilités : il y a autant d’ordonnanceurs et de gestionnaire de temps que de partitions. Si la stratégie est globale il n’y en a qu’un.

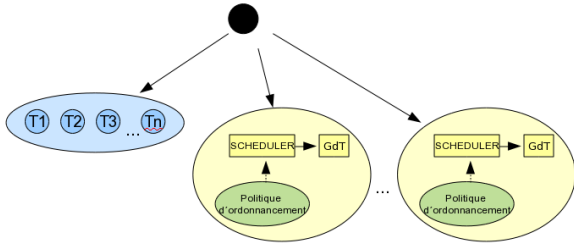


Fig. 5 – Architecture en partitionné

L’algorithme 1 montre la séquence d’initialisation des ordonnanceurs. Chaque ordonnanceur crée son gestionnaire de temps qui le réveille aux bonne instants.

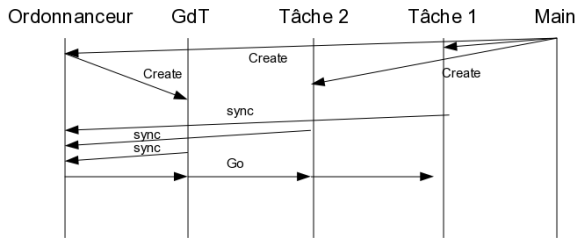


Fig. 6 – Schéma de la barrière d’initialisation

Algorithme 1 : Initialisation

```

m := nombre de cœurs ;
S = {τi}i=1...n;
si Algorithme partitionné alors
  Partitionner S = ∪j=1...m Sj;
  pour j=1 à m faire
    Créer un ordonnanceur;
    Affecter l’ordonnanceur à Sj;
  fin
sinon
  créer un ordonnanceur;
  pour i=1 à n faire
    Affecter l’ordonnanceur à τi;
  fin
fin

```

La création de thread POSIX se fait grâce à l’appel système `pthread_create`. Malheureusement, l’exécution effective d’un thread après sa création se fait de manière assez indéterministe. De plus, pour un fonctionnement correct du système, les tâches à ordonnancer et les gestionnaires de temps doivent être créés avant que l’ordonnanceur commence son travail. On a donc, mis en place une barrière d’initialisation (figure 6), où les tâches et le gestionnaire

de temps notifient leur création et attendent l’autorisation de commencer leur exécution. De cette façon, toutes les tâches commencent au même instant et les références temporelles sont les mêmes, ce qui est crucial pour respecter les dates de réveil et les périodes.

L’algorithme 1 montre la séquence d’initialisation du système.

2. Codage des entités

Gestionnaire du temps La gestion temporelle du système nécessite plusieurs éléments POSIX. La première action est de récupérer la fréquence de réveil imposée par l’ordonnanceur. Il faut ensuite générer un chronomètre qui lance un signal à cette fréquence. Le gestionnaire de temps gère ce chronomètre : il se met alors en attente du signal, quand ce dernier arrive le gestionnaire de temps réveille l’ordonnanceur, réarme le chronomètre et se remet en attente. L’algorithme 2 montre le fonctionnement du gestionnaire du temps.

Algorithme 2 : Gestionnaire de temps

```

P := période de réveil;
Création du chronomètre;
Envoi du signal de création;
Attente du signal GO;
tant que i=1 faire
  attendre le signal du chronomètre;
  réveiller l’ordonnanceur
fin

```

Ordonnanceurs Chaque ordonnanceur calcule, à partir des attributs temps réel des tâches qui lui sont associées, la fréquence à laquelle il sera exécuté. Il crée un gestionnaire de temps en lui envoyant la fréquence de réveil. Une fois que toutes les tâches sont créées, il envoie le signal GO. A partir de sa file de tâches, il donne accès aux plus prioritaires et attend le prochain réveil par le gestionnaire de temps. A chaque réveil, il trie la file de tâches et donne la ou les ressources aux plus prioritaires. L’algorithme 3 résume le fonctionnement d’un ordonnanceur.

Algorithme 3 : Ordonnanceur

```

Calculer période de réveil;
Création gestionnaire de temps;
Attente des signaux de création;
Broadcast GO;
tant que i=1 faire
  Ordonnancement des tâches;
  Attente du signal de gestionnaire de temps;
fin

```

Tâches Les tâches s’exécutent ou attendent bloquées par un mutex. L’algorithme 4 résume leur fonctionnement.

Algorithme 4 : tâche

```
Envoi du signal de création;
Attente du signal GO;
tant que  $i=1$  faire
  Exécution;
  Attente prochaine activation;
```

3. Caractéristiques

La librairie fait 2500 lignes de code. Les ordonnanceurs RM et EDF ont été implantés aussi bien pour le cas monoprocesseur que le cas stratégie globale. Des ordonnanceurs partitionnés sont également disponibles.

Un mode simulateur a été implanté, permettant de ne s'intéresser qu'à l'ordonnancement temps réel. La librairie imite le fonctionnement réel pour obtenir l'ordonnancement temps réel sur l'hyperpériode.

Le résultat est imprimé sur un fichier texte.

E. CONCLUSIONS ET PERSPECTIVES

Cet article présente une librairie générique permettant l'implantation d'algorithmes d'ordonnancement temps réel sur des architectures multicœurs. Son utilisation est aisée car de nombreuses macros ont été mises œuvre. La compatibilité POSIX permet une portabilité vers plusieurs noyaux. Nous avons pour l'instant testé la librairie sur du Linux standard et nous allons très prochainement la tester sur d'autres systèmes. Dans la suite, nous allons étendre les fonctionnalités de la librairie et la rendre disponible sur Internet.

Nous allons travailler sur les aspects plus théoriques en définissant des politiques permettant de prendre en compte les contraintes de priorité.

RÉFÉRENCES

- [1] James H. Anderson and Students. Litmus, Linux testbed for multiprocessor scheduling in real-time systems. <http://www.cs.unc.edu/~anderson/litmus-rt/>, 2007.
- [2] Theodore P. Baker and Sanjoy K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. Technical Report TR-060601, FSU Computer Science, 2007.
- [3] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15(6) :600–625, 1996.
- [4] Houssine Chetto, Maryline Silly, and T. Bouchentouf. Dynamic scheduling of real time tasks under precedence constraints. 1990.
- [5] Johanne Cohen. Le problème du bin packing. <http://webloria.loria.fr/~jcohen/i.php/Main/Enseignement>, 2009.
- [6] Julien Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, ONERA-ISAE (Supaéro), 2009.
- [7] IEEE. IEEE POSIX Certification Authority. <http://standards.ieee.org/regauth/posix/>.
- [8] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Trans. Softw. Eng.*, 30(9) :613–629, 2004.
- [9] Cheng L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.
- [10] José-María López. *Análisis de planificabilidad basado en utilizaciones de sistemas de tiempo real implementados sobre multiprocesadores con técnicas de particionado*. PhD thesis, Universidad de Oviedo, 2000.
- [11] Pascal Richard and Frédéric Ridouard. *Ordonnancement temps réel monoprocesseur*, chapter 1. Hermès, 2006.
- [12] Mario Aldea Rivas and Michael Gonzalez Harbour. A POSIX-Ada Interface for Application-Defined Scheduling. In *International Conference on Reliable Software Technologies, Ada-Europe 2002*, pages 136–150, 2002.
- [13] RTAI. Real-Time Application Interface. <https://www.rtai.org/>.
- [14] RTEMS. Real-Time Operating System for Multiprocessor Systems. <http://www.rtems.com/>.
- [15] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar : a flexible real time scheduling framework. *Ada Lett.*, XXIV(4), 2004.
- [16] Wind River Systems. VxWorks. <http://www.windriver.com/products/vxworks/>.
- [17] Richard Urunuela, Anne-Marie Déplanche, and Yvon Trinquet. Storm, simulation tool for real-time multiprocessor scheduling. Technical report, Institut de Recherche en Communications et Cybernétique de Nantes, september 2009.
- [18] Xenomai. Real-Time Framework for Linux. http://www.xenomai.org/index.php/Main_Page.