THESE

pour l'obtention du Grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE
(Diplôme National – Arrêté du 7 août 2006)

Ecole Doctorale : Sciences et Ingénierie pour l'Information, Mathématiques
Secteur de Recherche : Informatique et Applications

Présentée par :

Moustapha BIKIENGA

****

Mise en œuvre applicative de séquences d'ordonnancement hors-ligne

****

Directeurs de Thèse : Annie CHOQUET-GENIET et Dominique GENIET

****

Soutenue le 16 octobre 2014
devant la Commission d'Examen

****

JURY

Yamine AÏT AMEUR	Professeur, IRIT, INP de Toulouse	Président
Maryline CHETTO	Professeur, IRCCYN, Université de Nantes	Rapporteur
Frank SINGHOFF	Professeur, LABSTICC, Université de Bretagne Occidentale	Rapporteur
Annie CHOQUET-GENIET	Professeur, LIAS, Université de Poitiers	Examinateur
Dominique GENIET	Maître de conférences (HDR), LIAS, Université de Poitiers	Examinateur
Gaëlle LARGETEAU-SKAPIN	Maître de conférences, XLIM-SIC, Université de Poitiers	Examinateur

À ma mère.
À la mémoire de mon père.

Remerciements

Je tiens à remercier tout d'abord Annie CHOQUET-GENIET et Dominique GENIET qui sont à l'origine de ma thèse. Merci d'avoir guidé mes pas.

Je remercie Yamine AÏT AMEUR et Ladjel BELLATRECHE qui ont trouvé les arguments pour que ma thèse soit financée.

Je remercie Maryline CHETTO et Frank SINGHOFF qui ont bien voulu consacrer leur temps à la relecture et à la correction, en tant que rapporteurs, de ma thèse. Je remercie également Gaëlle LARGETEAU-SKAPIN qui a bien voulu participer à mon jury.

Je remercie la famille LIAS et plus particulièrement ceux de l'équipe *Systèmes Embarqués Temps Réel* pour les bons moments et les discussions enrichissantes : Ahcène, Allel, Amira, Aymen, Bery, Christian, Claudine, Emmanuel, Frédéric (R et C), Georges, Géraud, Guillaume, Henri, Kevin, Laurent, Mickael, Okba, Olga, Pascal, Patrick, Samia, Selma (B et K), Soumia, Stéphane, Valery, Yassine, Yves, Zahira, Zakaria, Zouhir.

J'adresse mes remerciements à Oumarou SIE pour son soutien et ses conseils.

Je remercie mes parents, camarades, amis et connaissances qui m'ont permis de supporter la dureté de la thèse : Anthony, Angélina, Dimitri, Eva, Frédo, Félix, Isma, Jo, Kadi (B et K), Lydia, Loé, Malick, Raïssa, Safi, Samuel, le TOCP, Youssouf, Yves, Zeynab, et tous ceux qui n'ont cessé de m'encourager.

Je remercie l'ENSMA, l'Université de Poitiers et la Coopération Universitaire Franco-Burkinabé qui m'ont permis d'aller au bout de ma thèse en mettant à ma disposition les moyens financiers nécessaires.

Je remercie l'Université de Koudougou et tous mes collègues pour leurs encouragements et leur aide.

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Depuis l'apparition des premiers ordinateurs, l'importance de l'informatique n'a cessé de croître. De nos jours, de nombreux systèmes informatiques font partie intégrante de nos activités quotidiennes.
Lorsqu'un système informatique interagit avec son environnement extérieur et qu'il doit fonctionner au rythme imposé par son environnement, on parle de *système temps réel*. Il ne s'agit pas d'être rapide, mais de réagir dans des délais qui sont prédéfinis et dépendent de l'environnement.
L'informatique temps réel est utilisée en particulier pour le contrôle de procédés ([FMB+09], [FBH+06], [SR08]) dans de nombreux secteurs tels que l'avionique , l'automobile, la robotique, la médecine, le multimédia. Dans ces cas les délais de réaction exigés dépendent de la dynamique du procédé contrôlé.
Certains procédés contrôlés ont des exigences très fortes en terme de sécurité : une erreur de contrôle peut avoir des conséquences lourdes, qu'il s'agisse de vies humaines mises en péril, de conséquences environnementales ou économiques. Ces systèmes sont qualifiés de critiques. Il est donc vital de garantir que les décisions prises par l'application sont pertinentes. Il faut d'une part s'assurer que les décisions sont algorithmiquement correctes, et d'autre part s'assurer qu'elles sont prises à temps. En effet une décision prise sur la base de valeurs devenues obsolètes n'est pas acceptable. Une application temps réel est donc soumise à des contraintes temporelles, et il faut prouver qu'elles sont satisfaites. Par ailleurs, les démarches usuelles de génie logiciel préconisent la modélisation d'applications sous forme de fonctions ou tâches en interaction. L'approche monolithique c'est à dire une seule tâche est déconseillée pour des raisons de structuration et de maintenance. Nous supposerons donc que les applications temps réel sont modélisées par un ensemble de tâches.
Chaque tâche est soumise à des contraintes temporelles, qui expriment sa date d'activation, sa périodicité si elle est périodique, et le temps dont elle dispose pour s'exécuter (échéance ou suite d'échéances lorsque la tâche est périodique). Les tâches sont concurrentes et se partagent le ou

les processeurs. Le respect des échéances dépend alors étroitement de la stratégie d'exécution choisie, c'est-à-dire de l'ordonnancement.

Deux approches sont généralement utilisées pour déterminer l'ordre d'exécution des tâches. La première approche est dite d'ordonnancement en ligne. Cela correspond au déroulement d'un algorithme qui tient compte des tâches présentes dans une liste de tâches, appelée file d'ordonnancement, lors de chaque décision d'ordonnancement. Cet algorithme d'ordonnancement s'exécute en même temps que l'application. Les algorithmes d'ordonnancement en ligne sont généralement dirigés par des priorités dont le calcul se fait le plus souvent en utilisant les paramètres temporels des tâches.

La deuxième approche est dite d'ordonnancement hors-ligne. Dans cette approche la séquence d'ordonnancement est prédéterminée à l'avance. Dans ce cas l'exécution de l'application suit la séquence préétablie.

De nombreux travaux existent sur la détermination des séquences d'ordonnancement hors-ligne. Ils s'appuient le plus souvent sur des approches dirigées par des modèles. Nous pouvons citer l'approche à base des réseaux de Petri [GCG02].

D'un point de vue théorique, les stratégies d'ordonnancement hors-ligne sont plus puissantes que les stratégies en ligne en ce sens qu'elles ordonnancent correctement, c'est à dire en respectant les contraintes temporelles, un plus grand nombre d'applications. Le gain en terme de puissance s'explique par le fait que les stratégies hors-ligne sont clairvoyantes : elles connaissent l'application dans sa globalité, et peuvent tenir compte d'évènements à venir lors des choix d'ordonnancement. Par contre, les stratégies d'ordonnancement en ligne fondent leur décision d'ordonnancement sur la seule connaisance de l'état instantané de l'application. Pour autant, les applications déployées sont le plus souvent ordonnancées par un algorithme en ligne. En effet, les applications sont implantées sur des plateformes dotées d'un système d'exploitation offrant des services *temps réel* (on parle de système d'exploitation temps-réel). En terme d'ordonnancement les systèmes d'exploitation temps réel proposent classiquement des stratégies en ligne basées sur des priorités fixes. Par contre, ils ne proposent généralement pas de mécanisme permettant d'implanter une séquence d'ordonnancement calculée hors-ligne, ce qui ne permet pas d'exploiter la puissance de ces stratégies. Quelques travaux, tels que ceux sur les sytèmes Mars ([SRG89]), ont abordé ce problème, mais dans des contextes spécifiques.

Compte tenu des apports des stratégies hors-ligne, il nous a semblé intéressant d'étudier la façon dont elles peuvent être mises en œuvre de manière effective sur un système d'exploitation temps réel classique. De façon plus précise, les questions auxquelles nous nous sommes intéressés sont :

- comment embarquer une séquence, c'est à dire comment la fournir au système ;
- comment faire en sorte que l'exécution de l'application suive le schéma décrit par la séquence embarquée.

Nous avons identifié deux approches (voir figure 1).

La première approche (figure 1.A) consiste à agir directement au niveau du système d'exploitation, en modifiant l'ordonnanceur afin d'y ajouter un module qui permet de suivre une séquence

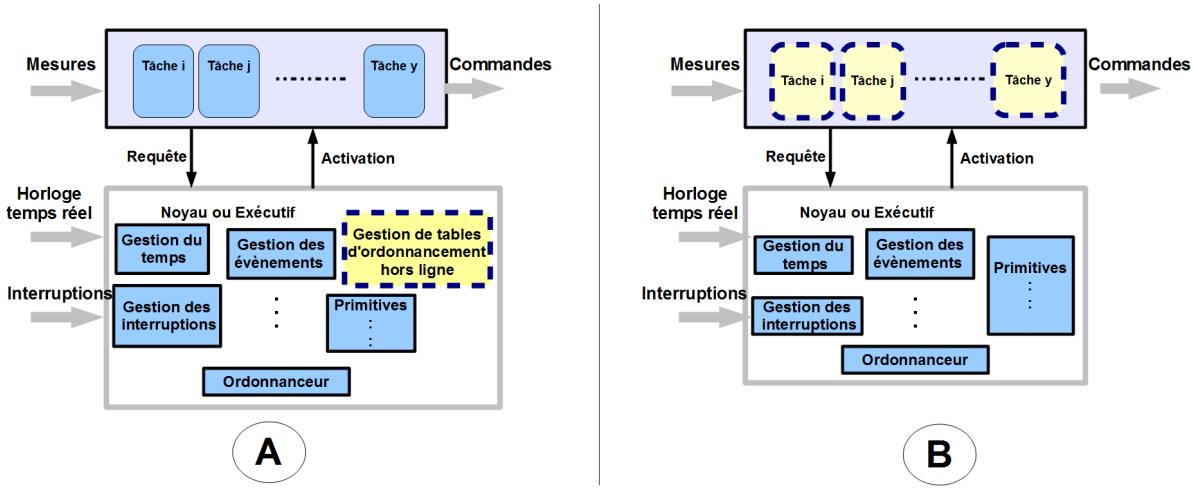


FIGURE 1 – Approches de mise en œuvre d'ordonnancement hors-ligne. A : mise en œuvre système ; B : mise en œuvre applicative

fournie au sytème d'exploitation. Cette approche est lourde à mettre en œuvre car elle revient à mettre au point une nouvelle version du système d'exploitation en garantissant que les services existants ne sont pas altérés. Par ailleurs, cette approche limite fortement la portabilité des applications implantées, qui ne pourront être déployées que sur des systèmes d'exploitation qui auront été modifiés. Aussi du fait de la particularité de chaque sytème d'exploitation, l'effort fourni pour modifier un système d'exploitation sera le même pour en modifier un autre. Nous avons donc renoncé à une telle approche et avons opté pour une seconde approche (figure 1.B), qui consiste à agir au niveau du code des systèmes de tâches. De manière générale, notre approche consiste à traduire la séquence d'ordonnancement hors-ligne par l'ajout de variables, de tâches supplémentaires ou par une restructuration opérationnelle des tâches existantes.
Ce faisant, le système d'exploitation temps réel est préservé, et nous exploitons directement les services proposés.
Afin de mener à bien ce travail, nous nous sommes appuyés sur le noyau temps réel Xenomai dont nous justifions le choix dans les chapitres 1 et 4. Nous nous sommes également intéressés au problème de la validation de notre approche : afin de pouvoir vérifier qu'une exécution est bien conforme aux attentes, il convient d'être à même d'observer le comportement d'une application. Nous nous sommes donc posés la question de l'observation d'une exécution.
Le manuscrit est organisé de la manière suivante : le chapitre 1 présente l'état de l'art sur les systèmes temps réel et les approches existantes de mise en œuvre de séquences hors-ligne ; le chapitre 2 présente plus en détails la problématique du sujet, ainsi que les hypothèses et la méthodologie de travail ; le chapitre 3 présente des algorithmes de mises en œuvres lorsque les tâches sont indépendantes, ou lorsqu'elles partagent des ressources sans préemption et avec préemption ; le chapitre 4 présente les mises en œuvre utilisant une interface de programmation

temps réel standard à savoir POSIX et présente aussi des outils pour l'automatisation et pour l'observation des mises en œuvre . Pour finir, le chapitre 5 présente une étude de cas qu'est une application de gestion de la sécurité d'une mine.

Chapitre

1 État de l'art

Sommaire

Résumé

Dans ce chapitre nous présentons l'état de l'art sur les systèmes temps réel et les mises en œuvre d'ordonnancements hors-ligne. Nous présentons les travaux sur lesquels nous allons nous appuyer, à savoir les travaux sur l'estimation des WCET et la recherche de séquences d'ordonnancement hors-ligne. Nous introduisons également les systèmes d'exploitation temps réel. Enfin nous présentons les travaux existants sur les mises en œuvre d'ordonnancement hors-ligne.

I. Systèmes temps réel

Contrairement aux systèmes informatiques généralistes, pour lesquels le bon fonctionnement est caractérisé par les données produites en sortie, le bon fonctionnement des systèmes informatiques temps réel est caractérisé à la fois par les données produites et le temps mis pour calculer ces données.
Pour cette catégorie de systèmes informatiques, le temps a une grande importance et est au cœur de nombreuses questions : pendant combien de temps s'exécutent les instructions, dans quel ordre exécuter les instructions pour que les contraintes temporelles soient respectées, ...
De nombreuses définitions formelles peuvent être données au concept de systèmes temps réel. Pour résumer ces définitions nous pouvons, comme l'a fait [But97], nous appuyer sur les deux mots *"temps"* et *"réel"* :
- le mot *temps* signifie que l'exactitude du système dépend non seulement de la valeur renvoyée par un calcul mais aussi de la date à laquelle cette valeur est produite ;
- le mot *réel* signifie que la réaction du système aux évèments extérieurs doit intervenir pendant l'évolution de ces évènements.

Nous pouvons donc donner du temps réel la définition suivante [Bur90] :

Définition 1.1 (Système temps réel)**.** *Un système temps réel est une activité de traitement d'information qui doit répondre aux stimuli venant de l'extérieur dans un intervalle de temps donné fini.*

De nombreux systèmes peuvent être considérés comme temps réel. Certains systèmes interactifs au même titre que les systèmes de contrôle de procédés peuvent être considérés comme étant temps réel. Cependant la réponse tardive du système aux stimuli extérieurs n'a pas les mêmes conséquences selon la nature du procédé concerné. Par exemple, s'il s'agit d'une application de diffusion de films, on aura une image moins nette ou une transmission du son moins fluide, on parle alors de temps réel mou. Tandis que s'il s'agit d'une application de pilotage, une réponse tardive du système au stimuli peut conduire à un drame, on parle alors de temps réel dur. Ainsi [Lap04] distingue trois classes de systèmes temps réel. Un système temps réel est dit :
- dur (hard) si le non respect d'une seule contrainte temporelle entraine une mise hors d'usage du système ;
- ferme (firm) si il existe un certain taux de contraintes temporelles manquées à ne pas dépasser. Lorsque ce taux est dépassé on a une mise hors d'usage du système ;
- mou (soft) si le non respect des contraintes temporelles entraine une dégradation de la qualité de service ;

À cause des conséquences dramatiques, dues au contexte d'utilisation, d'un dysfonctionnement, les systèmes temps réel durs sont ceux qui requièrent le plus d'attention. Nous nous intéressons donc principalement aux systèmes temps réel durs. Un système temps réel, qu'il soit dur, mou ou ferme, peut être décomposé en trois sous-systèmes ([DOR10]) à savoir :
- le système de capteurs ;
- le système de contrôle ;

– le système d'actionneurs.

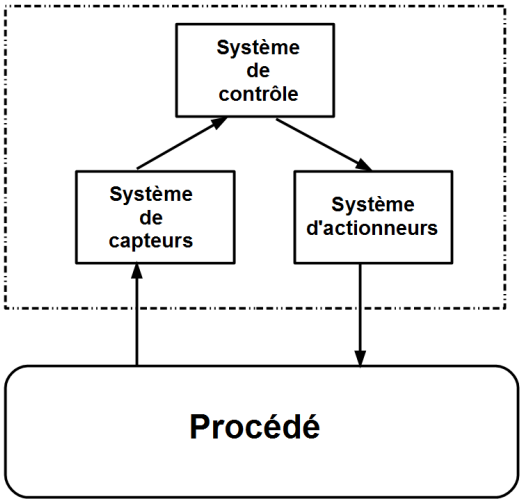


FIGURE 1.1 – Division en sous-systèmes d'un système temps réel

Comme l'illustre la figure 1.1, le système temps réel surveille l'évolution du procédé grâce à un système de capteurs et agit sur le procédé par un système d'actionneurs.

Le système de contrôle est la partie chargée d'effectuer les calculs. Plusieurs architectures matérielles sont possibles. Ainsi on parle d'architecture :

– monoprocesseur si l'architecture matérielle du système de contrôle est constituée d'un seul processeur ;
– multiprocesseur si l'architecture matérielle du système de contrôle est constituée de plusieurs processeurs partageant une même mémoire ;
– distribuée si l'architecture matérielle du système de contrôle est constituée de plusieurs processeurs, chacun possédant sa propre mémoire.

Dans notre thèse, nous ne considérons que l'architecture monoprocesseur, les autres architectures feront l'objet de travaux ultérieurs.

L'architecture logicielle du système de contrôle quant à elle est composée de deux couches (voir figure 1.2) à savoir la couche logiciels systèmes et la couche logiciels applicatifs.

Dans la suite du chapitre nous présentons les systèmes d'exploitation temps réel (section II), puis le modèle temporel d'une tâche (section III.2) avec différentes approches de codage d'applications temps réel (section III.1). Nous expliquons comment mesurer les durées d'exécution (section III.2.b) et nous présentons l'ordonnancement (section IV). Nous terminons le chapitre par une présentation des approches existantes de mise en œuvre d'ordonnancement hors-ligne (section V).

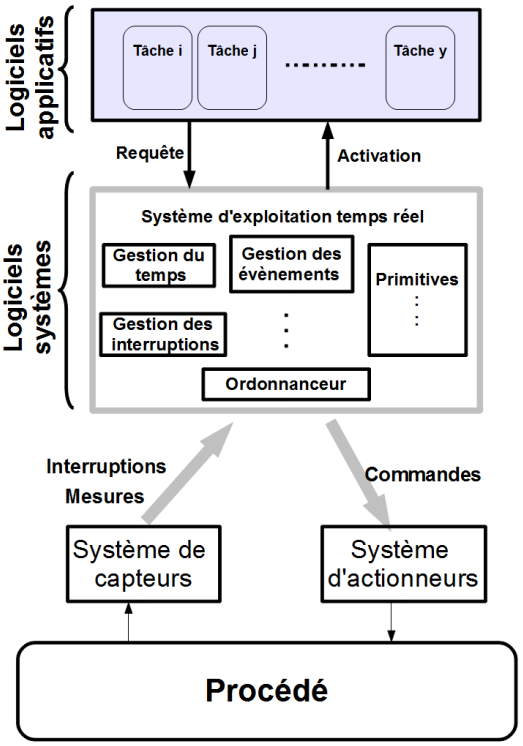


FIGURE 1.2 – Architecture logicielle du système de contrôle

II. Systèmes d'exploitation temps réel

L'offre en systèmes d'exploitation temps réel est très étendue. En effet on peut dénombrer plus d'une centaine de systèmes d'exploitation temps réel [1]. Le tableau 1.1 n'en présente qu'une dizaine parmi les plus utilisés ([Tim99]) en industrie ou dans le monde académique.

Système d'exploitation Temps réel	Matériel d'exécution	Licence
LynxOS	Motorola 68010, x86, ARM, Freescale PowerPC,LEON3	Propriétaire Non gratuit
Nucleus	ARM, PowerPC, MIPS32, MIPS16e, microMIPS, Coldfire, SuperH	Propriétaire Non gratuit
QNX Neutrino	IA32, MIPS, PowerPC, SH-4, ARM, StrongARM, XScale	Propriétaire Non gratuit
RTAI	x86, ARM	Libre Gratuit
μC/OS-II	NIOS, 8051, x86, 68000, PIC, ARM, AVR, Coldfire, Blackfin MSP430, PowerPC, Microblaze	Propriétaire Non gratuit
VRTX	ARM, MIPS, PowerPC, RISC	Propriétaire Non gratuit
VxWorks	PowerPC, SH-4, StrongARM, xScale ARM, IA32, Intel 64, MIPS	Propriétaire Non gratuit
Windows Embedded CE	x86, MIPS, ARM, SuperH	Propriétaire Non gratuit
Xenomai	x86, PowerPC, ARM, Blackfin	Libre Gratuit

TABLE 1.1 – Liste de quelques systèmes d'exploitation temps réel

Un système d'exploitation temps réel ou RTOS (Real Time Operating System) est avant tout un système d'exploitation, c'est-à-dire un ensemble de programmes qui permettent aux logiciels applicatifs de communiquer avec le matériel (ordonnanceur, gestionnaire d'interruptions, gestionnaire de mémoires, ...).
Cependant, un système d'exploitation classique est organisé de la façon suivante ([FK06],[CG05]) :
- un ordonnancement basé sur le partage équitable du processeur (par exemple en utilisant un ordonnancement à temps partagé) ;
- une gestion non déterministe des interruptions (la durée entre la réception et le traitement d'une interruption n'est pas garantie) ;

1. http ://en.wikipedia.org/wiki/List_of_real-time_operating_systems

- un mécanisme de gestion mémoire (cache) et de micro-exécution engendrant des fluctuations temporelles ;
- une gestion de l'horloge avec une grande granularité ;
- un mécanisme de gestion globale basée sur l'optimisation d'utilisation des ressources et du temps de réponse moyen des différents processus en cours.

Un système d'exploitation temps réel ou RTOS (Real Time Operating Systems) est destiné à fonctionner dans un environnement plus contraint qu'un système d'exploitation classique et se doit d'être déterministe, c'est-à-dire avoir des comportements identiques en réponse aux mêmes évènements. Cela rend nécessaire un certain nombre de règles supplémentaires :

- avoir un algorithme d'ordonnancement adapté à la prise en compte des contraintes temps réel avec une complexité limitée ;
- pouvoir respecter des contraintes de temps telles que les échéances, les dates de réveil, les périodes, ... ;
- supporter des surcharges ;

Cela passe à la fois par l'offre de services (API) temps réel aux développeurs d'applications temps réel et par une garantie de performances.

II.1. Interface de programmation des systèmes d'exploitation temps réel

Un effort de normalisation a été fait concernant les interfaces de programmation offertes par les systèmes d'exploitation temps réel et de nombreux standards tels que OSEK/VDX [Gro05], ARINC 653 [ari03], POSIX[Gal95]) ont été proposés. OSEK/VDX et ARINC 653 sont spécifiques à des domaines précis à savoir respectivement l'automobile et l'avionique. POSIX qui est un standard de l'IEEE (IEEE Std 1003.1) n'est pas spécifique à un domaine particulier.

En général, les systèmes d'exploitation temps réel offrent une interface de programmation POSIX en plus d'une API spécifique au système d'exploitation. POSIX permet ainsi la mise en œuvre d'applications temps réel portables, c'est-à-dire qui peuvent s'exécuter sans adaptation spécifique sur différents sytèmes d'exploitation.

Notons également que de nombreux systèmes d'exploitation n'offrant pas POSIX ont cependant des services très proches de POSIX. Dans nos travaux, pour éviter de proposer uniquement des mises en œuvre spécifiques à une interface de programmation particulière, nous avons préféré dans un premier temps utiliser des primitives temps réel plus générales à travers ce que nous avons appelé une *API abstraite* (section IV.2.c). Et dans un second temps, nous avons présenté comment passer de l'API abstraite à POSIX (chapitre 4 section I.1).

II.2. Évaluation des performances des systèmes d'exploitation temps réel

Les sections précédentes ont permis de faire une présentation générale des systèmes d'exploitation et de nous rendre compte qu'il y en a une grande variété. Sachant que les systèmes d'exploitation temps réel se doivent d'être déterministes, une question qui découle de cette variété consiste à identifier les plus déterministes. Cela revient à quantifier le déterminisme temporel de chaque système d'exploitation. Pour cela, il faut disposer de critères et de protocoles de comparaison.
Les protocoles généralement utilisés sont à base d'un dispositif matériel ([RVA09], [Bla00]). Par exemple, le protocole présenté par [RVA09] consiste à quantifier le déterminisme d'un système d'exploitation en évaluant le temps qu'il faut pour traiter une interruption matérielle. Pour cela, le systéme informatique (système d'exploitation et application temps réel) est programmé afin d'émettre un signal de sortie en réponse aux signaux d'entrées. Les signaux d'entrées sont produits par un générateur basse fréquence (GBF) et le signal émis en réponse par le système informatique est monitoré par un oscilloscope (voir figure 1.3-A).
La partie applicative est supposée être la même et la différence principale se situe au niveau système. Cela permet alors de caractériser les systèmes d'exploitation temps réel en utilisant les critères de comparaison suivants :

- *la latence de traitement d'interruption* (voir figures 1.3-B et 1.3-C) qui est la durée maximale entre le moment où l'interruption est reçue et le moment où le signal de sortie associé est produit.
- *la gigue* (voir figure 1.3-D) du système qui est la variation maximale entre les latences de traitement d'interruption ;
- *le temps de réponse du système* (voir figure 1.3-C) qui est le temps maximal nécessaire au traitement de l'interruption.

Considérons deux systèmes d'exploitation temps réel α et β, une latence plus faible pour α garantit que α prend moins de temps que β pour produire la réponse à un évènement externe. Un temps de réponse plus faible pour α garantit que α prendra moins de temps que β pour traiter un évènement externe. Une gigue plus faible pour α signifie que le temps de traitement d'un évènement par α varie moins que le temps de traitement du même évènement par β.
On dit donc qu'un système d'exploitation temps réel α est meilleur qu'un autre système d'exploitation β selon l'un des critères de comparaison (latence, gigue, temps de réponse), lorsque la valeur de ce critère est plus petite pour α.
Notons que le temps de réponse et la latence dénotent beaucoup plus de la rapidité que du déterminisme alors que la gigue dénote beaucoup plus du déterminisme que de la rapidité.
Notons également que d'une plateforme matérielle à l'autre, l'évaluation des critères (latence, gigue, temps de réponse) pour un même système d'exploitation peut produire des valeurs différentes dues aux inter-actions entre le matériel et le logiciel. Il en résulte que le protocole de [RVA09] ne permet pas de comparer deux systèmes d'exploitation temps réel installés sur deux plateformes matérielles différentes.

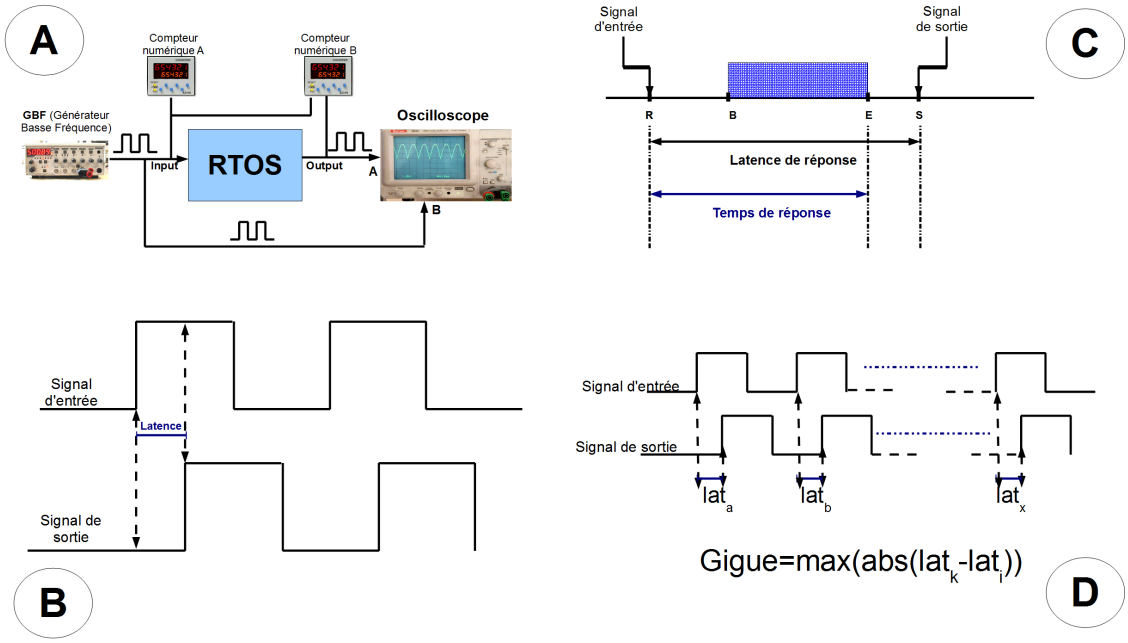


FIGURE 1.3 – Évaluation des performances des systèmes d'exploitation temps réel. A : Dispositif d'évaluation ; B : Latence de traitement d'interruption ; C : Latence et temps de réponse au traitement d'interruption ; D : Gigue

	Win XP	Win CE	Neutrino	μC/OS-II	Linux	RTAI	VxWorks
Temps de réponse	200 μs	20 μs	20 μs	1.92 μs	13.89 μs	5 μs	3.85 μs
Latence	848 μs	99 μs	35.2 μs	3.2 μs	98 μs	11.4 μs	13.4 μs
Gigue	700 μs	88.8 μs	32 μs	2.32 μs	77.6 μs	7.01 μs	10.4 μs

TABLE 1.2 – Exemple 1 d'estimation du déterminisme d'un système d'exploitation ([RVA09])

Le tableau 1.2 extrait de [RVA09] présente des mesures effectuées sur la même architecture matérielle (processeur : Pentium II de 400 MHz ; 256 Mo de RAM). Parmi les systèmes d'exploitation présentés dans le tableau, le moins déterministe est Windows XP parce que sa gigue est la plus élevée.

	Linux	RTAI	Xenomai	VxWorks
Latence	72.80 μs	71.80 μs	73.20 μs	69.20 μs
Gigue	0.40 μs	0.15 μs	0.25 μs	0.50 μs

TABLE 1.3 – Exemple 2 d'estimation du déterminisme d'un système d'exploitation ([BLM+07])

Un deuxième exemple de mesures extrait de [BLM+07] est présenté dans le tableau 1.3. Ces mesures ont été faites sur des processeurs PowerPc MPC7455.
Même si en latence d'activation Xenomai présente les moins bonnes performances, la gigue d'activation montre que Xenomai est assez déterministe.
Dans nos travaux nous utilisons Xenomai qui est un système d'exploitation qui permet d'utiliser POSIX. Ce choix se justifie principalement par le fait que Xenomai est open source et est maintenu grace à une communauté d'utilisateurs assez active. En plus un ordinateur personnel (PC) peut être utilisé comme plateforme de développement et d'exécution. Enfin, de nombreuses API peuvent être utilisées. Il est donc un support idéal d'expérimentation lorsqu'on envisage des mises en œuvres utilisant des API diverses. Xenomai est présenté plus en détail dans le chapitre 4.

III. Applications temps réel

L'application temps réel ou logiciel applicatif est la partie logicielle qui effectue les calculs fonctionnels et pilote le procédé. Sa conception produit en général une subdivision de l'application en un ensemble de traitements appelés tâches. Avant de présenter le modèle de tâche standard, nous présentons différentes approches de mise en œuvre.

III.1. Mise en œuvre synchrone ou asynchrone d'applications temps réel

La mise en œuvre d'une application temps réel peut se faire en considérant un traitement asynchrone ou synchrone des évènements extérieurs. On parle alors d'approche asynchrone ou synchrone.
Dans l'approche synchrone ([Hal98] [BCE+03]) les durées des traitements sont supposées négligeables par rapport à la dynamique du procédé. Autrement dit, la réaction à un évènement externe est considérée comme instantanée et les évènements ne sont observés que lorsqu'il n'y a pas de traitement en cours (voir figure 1.4). Il existe de nombreux langages de programmation

synchrone tels que Esterel [Ber00], Lustre [HLR92], Signal [LGTLL03], ReactiveC [Bou91], SL [BdS96], ReactiveML [MP05].

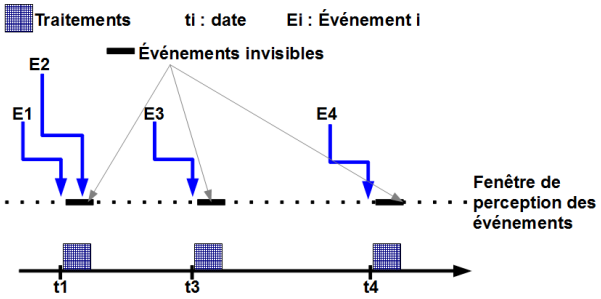


FIGURE 1.4 – Approche d'implémentation synchrone : les évènements E1, E3, E4 sont pris en compte mais E2 n'est pas pris en compte.

Contrairement à l'approche synchrone, dans l'approche asynchrone (voir figure 1.5) aucune hypothèse n'est faite sur la dynamique du procédé et sur le temps de traitement des évènements. Ainsi, des évènements peuvent se produire pendant le traitement d'un autre évènement. En fonction de la priorité à accorder au traitement d'un évènement, il peut être nécessaire d'en interrompre un autre, on parle de *préemption*. Ainsi un traitement peut être préemptée par un autre et reprendre par la suite son exécution sans qu'il y ait une perte d'information.

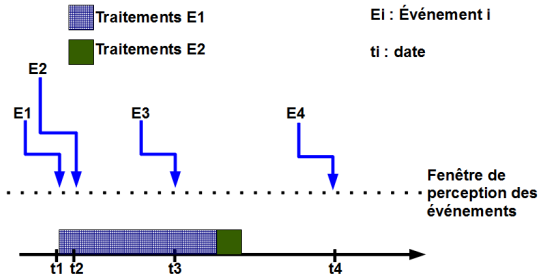


FIGURE 1.5 – Approche d'implémentation asynchrone : tous les évènements sont pris en compte

Dans notre document nous considérons l'approche asynchrone parce que les durées d'exécution des tâches que nous utilisons ne sont pas négligeables par rapport à la dynamique du procédé.

III.2. Modèle standard de tâches

III.2.a. Catégorisation des tâches

Les tâches sont utilisées pour modéliser les traitements d'une application. En fonction du temps séparant les dates d'activation de deux occurrences ou instances consécutives d'une tâche, on parle d'une tâche périodique, d'une tâche sporadique, d'une tâche apériodique. Une tâche est dite périodique lorsque cette durée est constante. Les tâches périodiques correspondent très souvent à des actions de contrôle qui ont vocation à être exécutées régulièrement. Une tâche est dite sporadique si le temps séparant deux instances successives a une borne inférieure. Lorsqu'il n'y a pas de borne, on parle de tâche apériodique. Les tâches sporadiques et apériodiques correspondent généralement à des actions déclenchées en réponse à des flots non réguliers d'évènements.

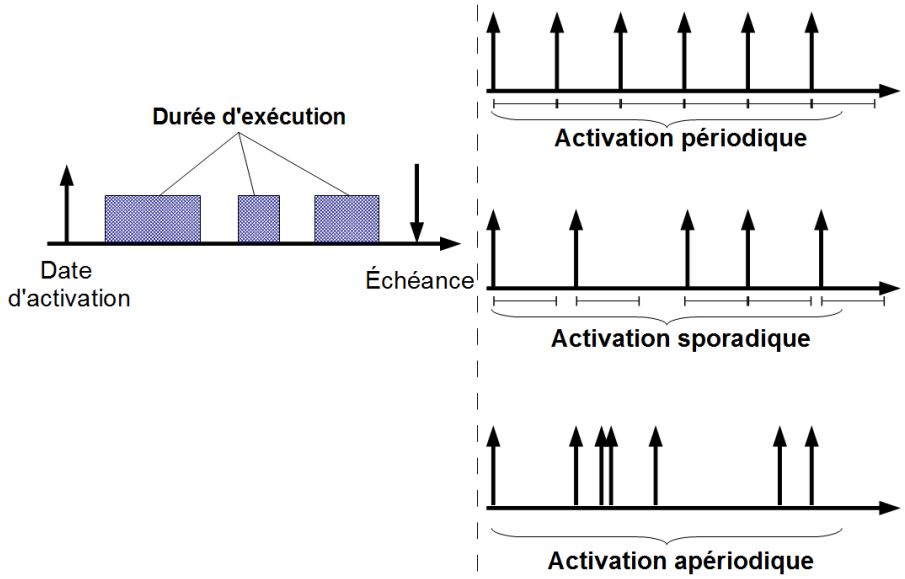


FIGURE 1.6 – Modèles de tâches

Un autre paramètre caractérisant une tâche est sa durée d'exécution (voir figure 1.6). La durée d'exécution étant rarement fixe, on s'intéresse plus à la pire durée d'exécution ou WCET (Worst-Case Execution Time). La pire durée d'exécution d'une tâche est un majorant du temps d'exécution pour l'ensemble des jeux de données en entrée possibles. Son calcul est motivé par le dimensionnement et la validation temporelle du système de tâches. Ainsi, la validation d'un système temps réel consiste généralement en une validation sur la base des pires durées d'exécution, ce qui correspond au comportement le plus pessimiste.

III.2.b. Estimation des durées d'exécution

Pour estimer une pire durée d'exécution, il existe deux grandes catégories de méthodes à savoir les méthodes statiques et les méthodes dynamiques.
Une méthode est dite dynamique lorsqu'elle mesure le temps d'exécution du programme sur un système réel (ou un simulateur). Lorsque le programme utilise en entrée un jeu de données, plusieurs méthodes sont envisagées pour produire ce jeu.
- Laisser l'utilisateur définir le jeu de données qui conduit au pire cas.
- Générer tous les jeux de données possibles en entrée et retenir la pire durée d'exécution. Cette solution n'est envisageable que dans le cas où les domaines de variation des données sont limités (pas d'explosion combinatoire).
- Générer automatiquement un jeu de données qui conduit à la pire durée d'exécution ([WM01] [TCM98]).
- Utiliser un jeu de données symboliques ([LS99]). Ce jeu de données a pour but d'exclure les chemins d'exécution impossibles et de rechercher le nombre de cycles processeur sur le plus long chemin. Ensuite en supposant qu'un cycle processeur a une durée fixe, on détermine la pire durée d'exécution du programme.

Quoi qu'il en soit, de manière pratique, les méthodes dynamiques consistent à utiliser ([Ste06]), soit :
- un appareil de mesure du temps (chronomètre, montre, ...) ;
- un script qui détermine les durées d'exécution en utilisant la date système (commandes Date et Time) ;
- un analyseur logiciel (CodeTest [cod], TimeTrace [Tim], WindView [win]) ou un outil de débogage (Prof et Gprof) ;
- l'instruction clock ;
- une horloge matérielle ;
- un analyseur logique ;
- un oscilloscope.

Une méthode est dite statique lorsqu'elle procède à une analyse statique du code du programme pour pouvoir en déterminer le temps d'exécution sur une architecture matérielle donnée. Cette analyse se base sur une modélisation du matériel. Selon [Nem08] cette détermination se fait en deux étapes :
- une analyse de flot qui évalue les chemins d'exécution possibles d'un programme ;
- une analyse temporelle qui détermine l'impact de l'architecture matérielle (à travers le modèle matériel utilisé) sur les durées d'exécution.

Les méthodes d'estimation des pires durées d'exécution qu'elles soient dynamiques ou statiques, permettent également d'en estimer un minorant ou durée d'exécution au meilleur cas (aussi appelé BCET : Best Case Exécution Time).
La différence entre les méthodes dynamiques et statiques se situe au niveau des estimations qu'elles peuvent faire des durées d'exécution pire cas (WCET) et meilleur cas (BCET). Comme le présente la figure 1.7 extraite de [WEE+08], les durées d'exécution observables sont différentes

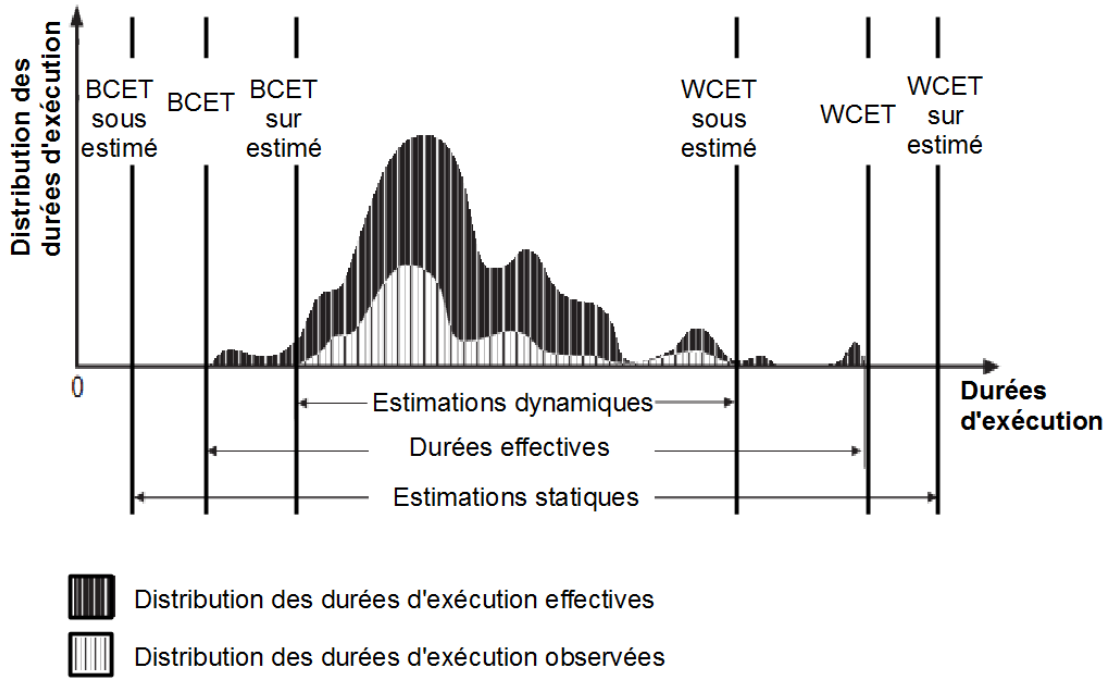


FIGURE 1.7 – Distribution des durées d'exécution : différence entre estimation dynamique et statique.

des BCET et WCET.

Conjecture 1.1 ([WEE+08]). *L'estimation dynamique des durées d'exécution d'une tâche surestime la durée d'exécution meilleur cas et sous-estime la durée d'exécution pire cas. L'estimation statique des durées d'exécution d'une tâche sous-estime la durée d'exécution meilleur cas et surestime la durée d'exécution pire cas.*

Nos travaux ne se focalisent pas sur l'estimation des durées d'exécution et dans la suite nous supposons que les durées d'exécution sont estimées en utilisant l'une des méthodes présentées plus haut.
Dans la suite nous nous intéressons uniquement aux tâches périodiques car ce sont les plus étudiées dans la littérature et surtout parce que les techniques d'ordonnancement hors-ligne ne prennent en compte que ce type de tâches.

III.2.c. Tâche périodique

Dans la littérature l'un des modèles de tâche les plus utilisés est celui de [LL73] que nous retenons.

Définition 1.2 (Modèle temporel simplifié de tâche périodique). *Une tâche périodique τ_i est caractérisée par le quadruplet $(r_i, C_i^{max}, D_i, T_i)$:*
- *r_i est la date d'arrivée de la première instance de la tâche τ_i, encore appelée date de première activation ou offset ;*
- *C_i^{max} est la pire durée d'exécution (WCET), elle spécifie un majorant du temps d'exécution de chaque instance de la tâche τ_i ;*
- *D_i est l'échéance relative ou délai critique, elle dénote la durée séparant l'arrivée d'une instance et son échéance ;*
- *T_i est la période, c'est l'intervalle de temps qui sépare l'arrivée de deux instances successives de τ_i.*

Si la définition que nous avons retenue correspond au modèle de tâche le plus utilisé, il faut cependant noter que les méthodes d'estimation des durées d'exécution permettent également d'estimer un minorant (C_i^{min}) du temps d'exécution de chaque instance de la tâche τ_i. Pour prendre en compte C_i^{min} nous étendons la définition 1.2 par la définition 1.3.

Définition 1.3 (Modèle temporel étendu de tâche périodique). *Le modèle temporel étendu d'une tâche périodique est donné par un quintuplet $\tau_i = (r_i, [C_i^{min}, C_i^{max}], D_i, T_i)$.*

Notons également que le modèle temporel simplifié (définition 1.2) est un cas particulier du modèle temporel étendu (définition 1.3) avec $\tau_i = (r_i, [0, C_i^{max}], D_i, T_i) = (r_i, C_i^{max}, D_i, T_i)$.
Le facteur d'utilisation d'une tâche périodique est $u_i = \frac{C_i^{max}}{T_i}$. Il représente la proportion de temps processeur à prévoir pour l'exécution d'une tâche. L'exécution d'une tâche périodique est concrétisée sous la forme d'une suite d'instances de tâches.

Définition 1.4 (Instance de tâche périodique). *L'instance j (avec $j > 0$), d'une tâche périodique $\tau_i = (r_i, [C_i^{min}, C_i^{max}], D_i, T_i)$, notée τ_{ij} est caractérisée par le triplet (r_{ij}, C_{ij}, d_{ij})) :*
- *r_{ij} est la date d'arrivée de l'instance, avec $r_{ij} = r_i + (j-1) * T_i$;*
- *C_{ij} est la durée d'exécution de l'instance, avec $C_i^{min} \leq C_{ij} \leq C_i^{max}$;*
- *d_{ij} est l'échéance de l'instance, avec $d_{ij} = r_{ij} + D_i$.*

Un système périodique $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ est dit :
- à *échéances sur requête* si toutes les tâches τ_i du système ont un délai critique égal à la période : $D_i = T_i$. Dans ce cas les échéances garantissent la non réentrance : une instance ne peut être activée alors que la précédente n'est pas terminée ;
- à *échéances contraintes* si toutes les tâches τ_i du système ont une échéance inférieure ou égale à la période : $D_i \leq T_i$;
- à *échéances arbitraires*, s'il n'y a pas de relation entre les délais critiques et les périodes ;
- à *départs simultanés*, si toutes les premières instances des tâches démarrent au même instant que l'on prend comme instant initial de l'échelle des temps : $\forall \tau_i$, $r_i = 0$;
- à *départs différés* si les dates de première activation ne sont pas toutes égales.

Un système de tâches périodiques est caractérisé par :
- un facteur d'utilisation $U = \sum_{i=1}^{n} \frac{C_i^{max}}{T_i}$;
- une hyperpériode $H = ppcm(T_1, T_2, \ldots, T_n)$, où *ppcm* est la fonction qui détermine le plus petit commun multiple de plusieurs entiers.

IV. Ordonnancement

IV.1. Étude théorique de l'ordonnancement

Dans une approche asynchrone, il peut arriver que plusieurs tâches soient en concurrence pour l'utilisation du processeur. Il faut donc déterminer l'ordre d'exécution des tâches : on parle d'ordonnancement.
L'étude théorique de l'ordonnancement d'un système temps réel se base généralement sur une hypothèse de discrétisation du temps. Cette discrétisation se base sur l'horloge matérielle qui fonctionne à une fréquence f. L'unité de temps utilisé est un multiple de la période $t = \frac{1}{f}$ de l'horloge matérielle.
D'un point de vue théorique, l'ordonnancement consiste à déterminer à chaque instant, pour un ensemble de tâches concurrentes laquelle exécuter. Plus formellement, dans un contexte mono-processeur, l'ordonnancement peut être défini par :

Définition 1.5 (Ordonnancement). *L'ordonnancement mono-processeur d'un système de tâches $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ est une fonction O définie par*

$$O : N \rightarrow \tau \cup \{\tau_0\}$$

$$O(t) = \begin{cases} \tau_i & \text{si } \tau_i \text{ est exécutée dans l'intervalle de temps } [t, t+1) \\ \tau_0 & \text{si aucune tâche ne s'exécute} \end{cases}$$

Lorsqu'aucune tâche ne s'exécute on parle de *temps creux*.
La suite des valeurs prises par la fonction d'ordonnancement est la séquence d'ordonnancement.

Définition 1.6 (Séquence d'ordonnancement)**.** *Une séquence d'ordonnancement $S(t_1, t_2)$ entre deux dates t_1 et t_2 est la suite des valeurs prises par la fonction d'ordonnancement :*

$$S(t_1, t_2) = O(t_1)O(t_1 + 1) \dots O(t_2 - 1)$$

L'ordonnancement est *préemptif* lorsque les préemptions (suspension de l'exécution d'une tâche) sont autorisées. Dans le cas contraire on parle d'ordonnancement *non-préemptif*. L'ordonnancement est *conservatif* (ou au plus tôt) s'il ne prend jamais la décision de ne rien faire lorsqu'au moins une tâche est active et non bloquée.
Les deux problèmes étudiés en théorie de l'ordonnancement sont :
- l'ordonnançabilité : connaissant les spécifications des tâches et les contraintes sur l'environnement d'ordonnancement (priorités fixes, préemptions admises, ...), il s'agit de déterminer l'existence d'un ordonnancement qui satisfasse toutes les échéances ;
- l'ordonnancement : connaissant les spécifications des tâches et les contraintes sur l'environnement d'ordonnancement d'un système que l'on sait être ordonnançable, il s'agit de déterminer une stratégie d'ordonnancement qui satisfasse toutes les échéances. Cette stratégie peut être décrite par un algorithme ou par une séquence explicite.

Définition 1.7 (Algorithme optimal [DOR10])**.** *Un algorithme est dit optimal pour une classe d'applications et une classe de stratégies si, étant donnée une application de la classe considérée, soit l'algorithme l'ordonnance correctement, soit aucun autre algorithme de la classe d'algorithmes considérée ne le pourra.*

Dans certains contextes, c'est-à-dire pour certaines classes d'applications et de stratégies d'ordonnancement, il existe des algorithmes optimaux. Dans ce cas, vérifier l'ordonnançabilité revient à vérifier si l'algorithme optimal ordonnance correctement l'application. La vérification peut se faire soit via l'utilisation d'une condition nécessaire et suffisante s'il en existe, soit via une simulation.

IV.2. Ordonnancement en ligne

Nous présentons dans un premier temps les principales stratégie d'ordonnancement en ligne (section IV.2.a). Puis nous nous intéressons à la façon dont les tâches sont mises en œuvre de manière effective.

IV.2.a. Algorithmes d'ordonnancement en ligne

La plupart des algorithmes d'ordonnancement en ligne se basent sur la notion de priorité. Ainsi à un instant donné, l'algorithme d'ordonnancement choisit la tâche la plus prioritaire. La séquence suivie est déterminée par l'affectation de priorités aux tâches. De nombreuses méthodes d'affectation de priorités existent, elles sont classées en trois catégories :

– une affectation de priorités fixes aux tâches : à chaque tâche est affectée une priorité constante dans le temps, c'est-à-dire qui ne change pas. Par exemple Rate Monotonic [ABD+95] affecte à chaque tâche une priorité inversement proportionnelle à la période et Deadline Monotonic affecte une priorité inversement proportionnelle au délai critique. La figure 1.8-A présente un exemple d'ordonnancement Rate Monotonic de trois tâches $\tau_1 = (0, 1, 4, 4)$, $\tau_2 = (0, 2, 6, 6)$ et $\tau_3 = (0, 3, 8, 8)$. La tâche τ_1 qui a la plus petite période est la plus prioritaire. Ainsi à chaque activation de τ_1, la tâche τ_3 est préemptée si elle n'a pas fini son exécution. Notons également que l'ordonnancement n'est pas valide parce que la première instance de τ_3 manque son échéance.

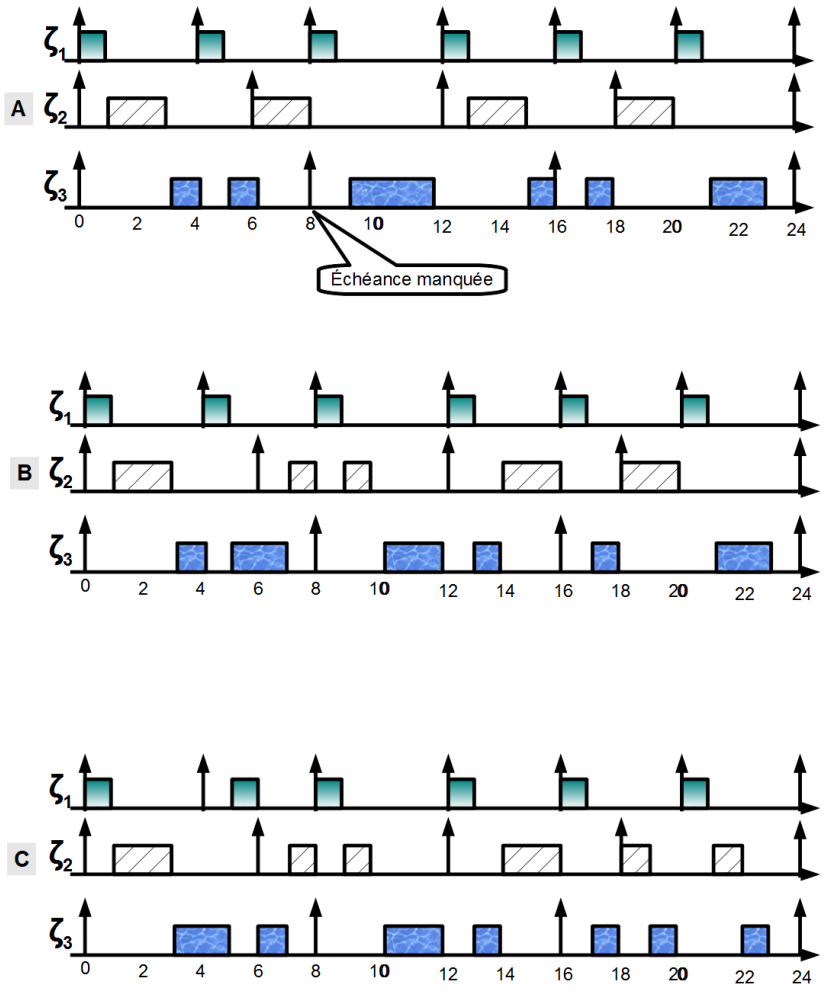


FIGURE 1.8 – Ordonnancement de $\tau_1 = (0, 1, 4, 4)$, $\tau_2 = (0, 2, 6, 6)$ et $\tau_3 = (0, 3, 8, 8)$. A : ordonnancement Rate Monotonic ; B : Earliest Deadline First ; C : ordonnancement Least Laxity

– une affectation de priorités fixes aux instances de tâches : une priorité est affectée à chaque

instance de tâche. Cela signifie que les instances d'une même tâche peuvent être de priorités différentes. Par exemple Earliest Deadline First [SRS98] affecte une priorité plus grande à l'instance de tâche dont l'échéance est la plus proche. La figure 1.8-B présente un exemple d'ordonnancement Earliest Deadline First. Nous avons considéré dans l'exemple que la tâche de plus petit indice est plus prioritaire en cas d'ex aequo. L'ordonnancement produit est valide parce qu'aucune instance ne manque son échéance

– une affectation de priorités dynamiques : la priorité d'une instance peut évoluer. Par exemple l'algorithme Least Laxity [Mok83] augmente la priorité d'une instance lorsque celle-ci se rapproche de son échéance. La figure 1.8-C présente un exemple d'ordonnancement Least Laxity. L'ordonnance produit est également valide, cependant elle produit plus de préemptions de tâches que Earliest Deadline First.

IV.2.b. Validation d'ordonnancement en ligne

Valider l'ordonnancement d'un système de tâches consiste à prouver que toutes les échéances sont respectées.
Nous nous intéressons principalement à la validation sur la base de critères analytiques tels que les facteurs d'utilisation ou les pires durées nécessaires à l'exécution des tâches encore appelées temps de réponse.
[LL73] et [DG00] ont démontré que l'algorithme Rate Monotonic est optimal, dans la classe des algorithmes à priorités fixes, pour des systèmes de tâches indépendantes, périodiques, à départs simultanés et à échéances sur requête ($D_i = T_i$).
[LL73] a également présenté une condition suffisante d'ordonnançabilité d'un système de n tâches périodiques à départs simultanés et à échéances sur requêtes par Rate Monotonic :

$$U = \sum_{i=1}^{n} \frac{C_i^{max}}{T_i} \leq n(2^{1/n} - 1)$$

[LW82] a montré que l'algorithme Deadline Monotonic est optimal, dans la classe des algorithmes à priorités fixes, pour des systèmes de tâches indépendantes, à départs simultanés et à échéances contraintes ($D_i \leq T_i$).
[Der74] et [Lab74] ont montré que Earliest Deadline First est optimal, dans la classe des algorithmes d'ordonnancement en ligne, pour des systèmes de tâches indépendantes, à échéances sur requêtes ($D_i = T_i$). La condition nécessaire et suffisante d'ordonnançabilité est ([LW82]) :

$$U = \sum_{i=1}^{n} \frac{C_i^{max}}{T_i} \leq 1$$

[Mok83] et [DM89] ont montré que Least Laxity est optimal, dans la classe des algorithmes d'ordonnancement en ligne, pour des systèmes de tâches indépendantes et à échéances contraintes ($D_i \leq T_i$).

Une autre approche de validation consiste à utiliser une analyse des pires temps de réponse pour déterminer si une application est valide. Notons R_i le pire temps de réponse d'une tâche τ_i. Une application est valide si : $\forall i \in \{1, \ldots, n\}$, $R_i \leq D_i$.
[JP86] ont ainsi établi, dans un contexte de tâches indépendantes avec un algorithme d'ordonnancement à priorités fixes, que le temps de réponse d'une tâche τ_i est égale à la somme de sa charge (C_i) et des interférences des tâches τ_j ($j \in hp(i)$ où hp détermine les indices des tâches plus prioritaires que la tâche τ_i). Plus formellement, R_i est le point fixe de la suite :

$$\begin{cases} R_i^{(0)} = & C_i \\ R_i^{(n+1)} = & C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^{(n)}}{T_j} \rceil C_j \end{cases}$$

La formule de [JP86] permet de calculer une date relative à l'activation de chaque instance et ne prend pas en compte le fait que dans certains cas l'exécution d'une instance peut se poursuivre après l'activation de l'instance suivante. Pour prendre en compte cette possibilité, [Leh90] calcul pour une instance τ_{ik} une date $RA_i(k)$ de fin d'exécution au pire cas relative à l'activation de la première instance :

$$\begin{cases} RA_i^{(0)}(k) = & kC_i \\ RA_i^{(n+1)}(k) = & kC_i + \sum_{j \in hp(i)} \lceil \frac{RA_i^{(n)}(k)}{T_j} \rceil C_j \end{cases}$$

En présence de ressources partagées la validation devient plus complexe et doit prendre en compte des temps de blocage de tâches en attente d'une ressource. De plus s'il n'y a pas de protocole de gestion des ressources prenant en compte les priorités des tâches, il peut y avoir des annomalies d'ordonnancement telles que l'inversion de priorité. L'inversion de priorité correspond au fait que l'exécution d'une tâche τ_2 est retardée par une tâche de priorité inférieure τ_1 sans que τ_2 et τ_1 aient une ressource commune.
La figure 1.9-A présente un exemple d'inversion de priorité.
Pour éviter les inversions de priorité [SRL90] ont proposé deux protocoles de gestion de ressources à savoir le protocole à priorité hérité ou PIP (Priority Inheritance Protocol) et le protocole à priorité plafond ou PCP (Priority Ceiling Protocol).
Ainsi pour le protocole à priorité hérité, lorsqu'une ressource R est utilisée par une tâche τ_1 et qu'une tâche τ_2 de priorité supérieure requiert R, τ_1 hérite de la priorité de τ_2. La figure 1.9-B présente un exemple d'utilisation du protocole à priorité héritée. Le protocole à priorité héritée permet d'éviter l'inversion de priorité, cependant elle ne permet pas d'éviter les inter-blocage (par exemple une tâche τ_1 utilisant une ressource R_1 est bloquée en attente d'une ressource R_2 utilisée par une tâche τ_2 qui attend la ressource R_1). Le protocole à priorité plafond apporte une solution à l'inter-blocage en ajoutant un aspect prévision à l'utilisation des ressources. Pour cela, à chaque ressource R est associée une priorité plafond $pp(R)$ qui est la plus grande priorité des tâches qui l'utilisent. Pendant l'exécution, une autre variable S_p appelée *plafond système* permet d'enregistrer la plus grande priorité plafond des ressources en cours d'utilisation ainsi que la tâche qui détient la ressource correspondante. Le protocole à priorité plafond n'autorise

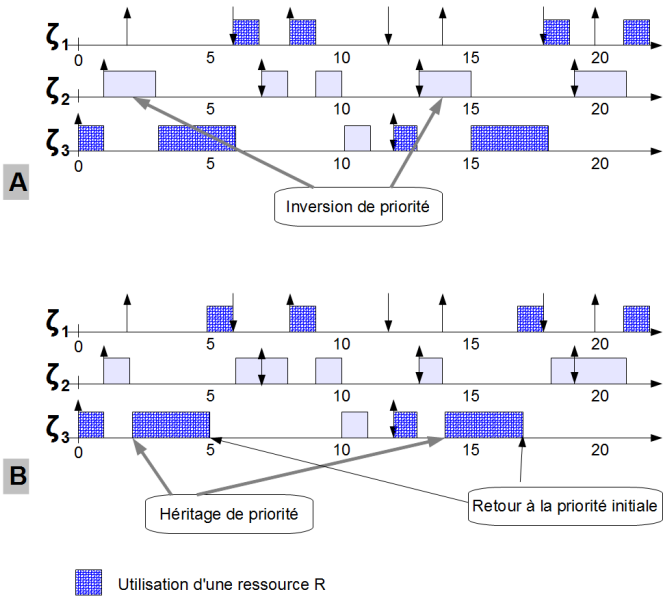


FIGURE 1.9 – Ordonnancement Rate monotonic de $\tau_1 = (0, 1, 4, 4)$, $\tau_2 = (0, 2, 6, 6)$ et $\tau_3 = (0, 3, 8, 8)$. A : Inversion de priorité ; B : Protocole à priorité héritée

alors une tâche τ_1 à accéder à une ressource R que si sa priorité est strictement supérieure à $pp(R)$ ou si τ_1 a comme priorité S_p.
Pour illustrer la difficulté de valider des systèmes de tâches partageant des ressources, nous présentons le principe de calcul des temps de réponse dans un contexte de priorités statiques et d'utilisation du protocole à priorité plafond.
Une tâche τ_1 qui utilise une ressource R_1 (notons $\tau_1 \triangleright R_1$ cette relation d'utilisation) peut être bloquée par les tâches de priorités inférieure utilisant R_1. Nous notons $lp(1)$ l'ensemble des tâches de priorité inférieure à la tâche τ_1. Soit $SC_j(R_\alpha)$ la durée d'utilisation d'une ressource R_α par une tâche τ_j . Nous avons [Bur95] :

$$\begin{cases} B_i = & max_{j \in lp(i), R_\alpha, \tau_j \triangleright R_\alpha}(SC_j(R_\alpha)) \\ RA_i^{(0)}(k) = & B_i + kC_i \\ RA_i^{(n+1)}(k) = & B_i + kC_i + \sum_{j \in hp(i)} \lceil \frac{RA_i^{(n)}(k)}{T_j} \rceil C_j \end{cases}$$

Notons que le calcul des pire temps de réponse a une complexité pseudo-polynomiale. Mais le principal problème vient du fait que la validation par calcul de temps réponse est pessimiste. En fait, la non clairvoyance (non prise en compte du futur) des algorithmes en ligne les rend moins efficaces dès lors qu'il y a des contraintes de précédence ou de partage de ressources critiques. Cela constitue l'essence du théorème 1.2.

Théorème 1.2 ([Mok83]). *Le problème de l'ordonnancement en présence de ressources critiques est NP-difficile au sens fort, et aucun algorithme non clairvoyant n'est optimal dans ce contexte.*

Les algorithmes d'ordonnancement hors-ligne permettent de palier cette non clairvoyance. Nous les présentons dans la section IV.3.
En ce qui concerne l'ordonnancement en ligne, nous nous intéressons maintenant à la mise en œuvre.

IV.2.c. Mise en œuvre de tâches dans un contexte d'ordonnancement en ligne

Mettre en œuvre un système de tâches temps réel consiste à en déterminer l'équivalent opérationnel c'est-à-dire à déterminer ce qui s'exécute effectivement sur le système d'exploitation temps réel hôte. La terminologie utilisée pour désigner ce qui s'exécute varie d'un système d'exploitation à l'autre. Les expressions généralement utilisées pour désigner ce qui s'exécute sont : *tâche temps réel* (real time task), *processus légers* (thread), *processus lourds* (process). Même si les termes utilisés sont différents, ils représentent essentiellement l'idée d'exécution pseudo-parallèle d'instructions. Dans la suite nous utiliserons le terme *processus*.

Définition 1.8 (Processus[BB90]). *Un processus est une entité dynamique associée à une suite d'instructions.*

Pour gérer les processus, le système d'exploitation leur associe un état. [BB90] distingue trois états à savoir *en cours d'exécution*, *prêt* (le processus est en attente du processeur), *en attente* (le processus est en attente d'une ressource autre que le processeur). Lorsque plusieurs processus sont prêts le choix du processus à exécuter est défini par la politique d'ordonnancement.
La plupart des systèmes d'exploitation temps réel met en œuvre trois politiques d'ordonnancement à savoir la politique d'ordonnancement à priorités fixes, celle du premier arrivé premier servi ou FIFO (First in First Out) et celle du tourniquet (ou Round Robin). Ainsi pour développer une application temps réel utilisant un ordonnancement Rate Monotonic ou Deadline Monotonic, il faut au préalable déterminer une affectation de priorités fixes conforme à la politique que l'on veut utiliser. Quant aux politiques d'ordonnancement à priorités dynamiques telles que Earliest Deadline First First (EDF) ou Least Laxity (LL) elles ne sont pas nativement prises en compte dans les ordonnanceurs des systèmes d'exploitation. Pour mettre en œuvre de telles politiques dans des applications temps réel, il faut réaliser une surcouche applicative qui a en charge la modification dynamique des priorités.
Par ailleurs, l'activation d'un processus peut être déclenchée soit par un évènement (interne ou externe), soit à des dates précises.
Dans le premier cas, on parle de déclenchement évènementiel et dans le deuxième cas on parle de déclenchement dirigé par le temps.
Lorsque tous les processus ont un déclenchement dirigé par le temps on dit que l'application est dirigée par le temps (time triggered). Si une partie ou la totalité des traitements est déclenchée évènementiellement on dit que l'application est dirigée par les évènements (event triggered).

[SSP06] estime que les applications dirigées par le temps sont plus tolérantes aux fautes, alors que les applications dirigées par les évènements utilisent mieux les ressources matérielles.
Dans notre document, nous nous intéressons aux applications dirigées par le temps, et nous supposons que l'ordonnancement, c'est-à-dire les dates d'exécution de tous les traitements, est préetabli (hors-ligne). De plus, nous nous intéressons principalement aux tâches périodiques qui correspondent à des processus dont l'activation est liée à l'horloge.
Les mises en œuvre de tâches temps réel se font par l'intermédiaire de primitives temps réel ou primitives temporelles. Dans notre document nous utilisons un ensemble de primitives temporelles que nous regroupons dans une API dite abstraite (inspirée de [ET10a] et de [ET10b]). Cela nous permet de ne pas restreindre notre étude à un système d'exploitation particulier. Ces primitives peuvent être subdivisées en trois groupes à savoir les primitives de gestion du temps (voir tableau 1.4), les primitives de synchronisation et de communication (voir tableau 1.6), les primitives de gestion des processus (voir tableau 1.5).

Primitives de gestion du temps .

Primitives	Paramètres d'entrées	Résultat de la primitive
$t \leftarrow$Lire_Date()		La date t est lue à partir de l'horloge
Attendre_Date(t)	t : date attendue	Le processus est suspendu et reprend son exécution à la date t attendue
Attendre_Delai(d)	d : délai attendu	Le processus est suspendu pendant d unités de temps

TABLE 1.4 – Primitives de gestion de temps

Les primitives de gestion du temps (tableau 1.4) permettent à l'application d'interagir avec l'horloge du système. Dans la suite, nous ne nous intéressons pas à la gestion interne du temps par les systèmes d'exploitation temps réel mais uniquement aux primitives de gestion du temps. Le lecteur intéressé par la gestion interne pourra consulter les travaux de [CV98] et de [Kod07].
Lorsqu'un processus exécute l'instruction Attendre_Date, si la date actuelle est plus petite que la date attendue, le processus passe à l'état *en attente*. Dans la suite afin de différencier l'attente d'une date de l'attente d'une ressource nous utiliserons deux états différents (voir figure 1.10). L'état *bloqué* signifie que le processus est en attente d'une ressource et l'état *endormi* signifie que le processus n'est ni *prêt* ni *bloqué*, mais en attente d'un évènement tel qu'une date. Tout comme l'instruction Attendre_Date, l'exécution de l'instruction Attendre_Delai(d) entraine le passage du processus de l'état en cours d'éxécution à l'état endormi (figure 1.10-transition a). Le processus reste alors à l'état endormi pendant d unités de temps puis revient à l'état prêt (figure 1.10-transition g).

Primitives de gestion des processus Les primitives de gestion des processus (tableau 1.5) permettent d'agir directement sur les processus et de changer leur état. Ainsi, les primitives

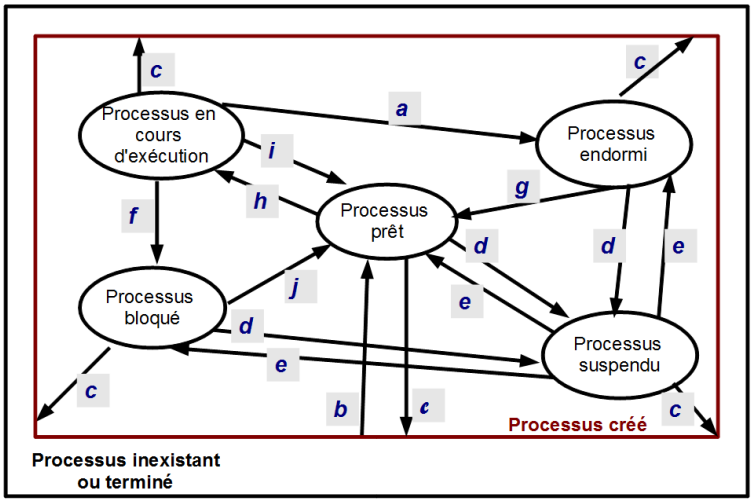


FIGURE 1.10 – États et transitions d'un processus. a : mise en attente ; b : création ; c : suppression ; d : suspension ; e : reprise ; f : blocage ; g : fin d'attente ; h : élection ; i : préemption ; j : fin de blocage

Créer_Processus et Supprimer_Processus permettent respectivement de créer un nouveau processus (figure 1.10-transition b) ou de supprimer un processus existant (figure 1.10-transition c). L'instruction Arrêter_Processus(σ_i) change l'état d'un processus et le met en attente d'un signal de reprise explicite envoyé par la primitive Continuer_Processus(σ_i). Pour distinguer l'attente d'un signal de reprise explicite de l'attente d'une date, nous utilisons l'état suspendu. Ainsi, les instructions Arrêter_Processus(σ_i) et Continuer_Processus(σ_i) font, respectivement, entrer le processus dans l'état suspendu (figure 1.10-transition d), sortir de l'état suspendu (figure 1.10-transition e). Notons qu'un processus peut également s'auto-suspendre ce qui correspond à une transition entre les états en cours d'exécution et suspendu. Cependant pour ne pas trop surcharger la figure 1.10 nous n'y avons pas représenté les transitions entre les états en cours d'exécution et suspendu.
Généralement, l'implémentation d'une tâche périodique consiste à utiliser une variable permettant de temporiser son exécution. La figure 1.11 présente un exemple d'implémentation d'une tâche périodique. Le point d'entrée de l'application est init(). La périodicité de la tâche s'obtient par attente de date.
Dans la suite nous considérons que l'ordre des priorités est croissante : un processus σ_1 de priorité 1 est moins prioritaire qu'un processus σ_2 de priorité 2.

Primitives de synchronisation et de communication Les primitives de synchronisation et de communication (tableau 1.6) permettent la coordination des processus. Les processus peuvent ainsi se synchroniser pour l'utilisation de ressources par exemple ou communiquer par

Primitives	Paramètres d'entrées	Résultat de la primitive
Créer_Processus(σ_i,p_i ,F, A)	σ_i : nom/descripteur de processus p_i : priorité F : code exécuté par le processus A : paramètres de F	Un processus σ_i est créé dans l'état prêt
Supprimer_Processus(σ_i)	σ_i : nom/descripteur du processus	Le processus est supprimé
Arrêter_Processus(σ_i)	σ_i : nom/descripteur du processus	Le processus est suspendu
Continuer_Processus(σ_i)	σ_i : nom/descripteur du processus	Le processus sort de l'état suspendu
Priorité_Processus(σ_i, p_i)	σ_i : nom/descripteur de processus p_i : priorité	La priorité p_i est affectée au processus σ_i

TABLE 1.5 – Primitives de gestion des processus

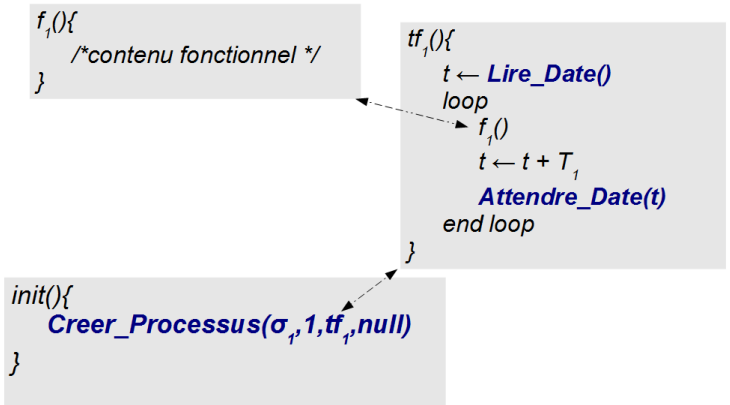


FIGURE 1.11 – Mise en œuvre classique d'une tâche périodique.

l'intermédiaire de messages. Ainsi, les primitives Créer_Boîte et Supprimer_Boîte permettent respectivement de créer et supprimer les objets permettant de stocker des données. Déposer_Lettre et Recevoir_Lettre sont des primitives bloquantes qui permettent d'envoyer et recevoir des données. Les données lues sont retirées de la boîte. Pour avoir des primitives non bloquantes il suffit d'ajouter des tests qui vérifient s'il y a des données avant d'en recevoir ou si la boîte est pleine avant d'en déposer.

Primitives	Paramètres d'entrées	Résultat de la primitive
Créer_Boîte(B,t)	B : nom/descripteur de boîte aux lettres t : taille de la boîte	Une boîte aux lettres B de taille t est créée
Supprimer_Boîte(B)	B : nom/descripteur de boîte aux lettres	La boîte aux lettres est supprimée
Déposer_Lettre(B,δ)	B : nom/descripteur de boîte aux lettres δ : donnée	δ est envoyée à la boîte B
$\delta \leftarrow$Recevoir_Lettre(B)	B : nom/descripteur de boîte aux lettres	δ est lue de la boîte
$\beta \leftarrow$Boîte_Vide(B)	B : nom/descripteur de boîte aux lettres	β est vrai si la boîte est vide
$\beta \leftarrow$Boîte_Pleine(B)	B : nom/descripteur de boîte aux lettres	β est vrai si la boîte est vide
Créer_Semaphore(R,μ)	R : nom/descripteur de sémaphore μ : nombre d'unités utilisables	Un sémaphore est créée avec μ unités
Prendre_Semaphore(R)	R : nom/descripteur de sémaphore	Prise d'une unité du sémaphore R
Vendre_Semaphore(R)	R : nom/descripteur de sémaphore	Relâchement d'une unité du sémaphore R

TABLE 1.6 – Primitives de synchronisation et de communication

Les sémaphores permettent aux processus qui utilisent des ressources communes de synchroniser l'utilisation de ces ressources. La partie d'un processus utilisant une ressource partagée est appelée section critique. La figure 1.12 présente un exemple de deux processus périodiques qui partagent une même variable G et utilisent pour maintenir le système dans un état cohérent les primitives temporelles de synchronisation. Un processus avant d'entrer en section critique appelle la primitive *Prendre_Semaphore*. Si le nombre d'unités de sémaphore demandées par le processus n'est pas disponible, le processus est bloqué (figure 1.10-transition f) en attente de libération des instances nécessaires au processus pour entrer en section critique.

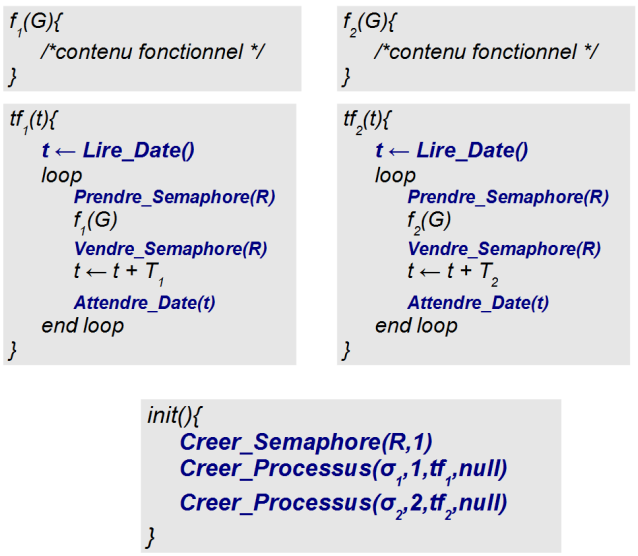


FIGURE 1.12 – Mise en œuvre de deux tâches périodiques avec une synchronisation par sémaphore

Les boîtes aux lettres ou files de messages permettent aux processus de communiquer. La taille d'une boîte aux lettres est le nombre maximum de messages qu'elle peut contenir. Plusieurs processus peuvent accéder (lecture/écriture) à une même boîte aux lettres.

IV.3. Ordonnancement hors-ligne

IV.3.a. Cyclicité des séquences d'ordonnancement

Contrairement aux algorithmes d'ordonnancement en ligne, les algorithmes d'ordonnancement hors-ligne déterminent à l'avance l'ordonnancement permettant aux tâches de respecter leurs contraintes temporelles. Afin de respecter les contraintes temporelles, les algorithmes d'ordonnancement hors-ligne peuvent décider de n'exécuter aucune tâche même lorsque certaines sont actives : on parle d'injection de temps creux. Cette détermination à l'avance se base sur le fait que pour les tâches périodiques l'ordonnancement est également périodique de période $H = ppcm(T_i)$.
Pour les tâches à départs simultanés, la séquence est cyclique à partir de $t = 0$. Il suffit dans ce cas de déterminer une séquence valide $S(0, H)$.
Pour les tâches à départs différés, si la séquence n'est pas déterminée avec injection de temps creux supplémentaires (c'est-à-dire sans injection de plus $H(1 - U)$ temps creux toutes les H unités de temps), toute séquence d'ordonnancement valide présente un régime transitoire et un régime stationnaire cyclique ([CGG04]).

Pour repérer le début du régime stationnaire, [CGG04] utilise la notion de temps creux acyclique. Ainsi, les temps creux qui ne se répètent pas tout les H unités de temps sont dits acycliques et les $H(1-U)$ temps creux qui se répètent toutes les H unités de temps sont dits cycliques.
[CGG04] a alors montré que le nombre de temps creux acyclique est borné et que le regime stationnaire cyclique commence juste après le dernier temps creux acyclique, à $t_c + 1$. Il a également montré que la date du dernier temps creux acyclique t_c est indépendante de la stratégie choisie et est telle que $t_c \leq max(r_i) + H$.

Théorème 1.3 ([CGG04]). *Pour un système de tâches périodiques à échéances contraintes avec des contraintes de précédences, toute séquence d'ordonnancement valide générée par un algorithme déterministe et conservatif, est périodique de période H après le dernier temps creux acyclique, arrivant à la date $t_c \in [-1..r + H[$. Le régime périodique est la première fenêtre temporelle de taille H sans temps creux acyclique, et démarre exactement après la date t_c. La date t_c est indépendante de l'algorithme d'ordonnancement choisi.*

Pour déterminer le régime stationnaire, on peut :
- construire la séquence $S(0, H)$;
- si la séquence ne contient que $H(1-U)$ temps creux alors la séquence calculée est périodique et $t_c = -1$;
- si la séquence contient plus de $H(1-U)$ temps creux, continuer la construction de la séquence jusqu'à trouver une séquence $S(\alpha, H + \alpha)$ qui ne contient que $H(1-U)$ temps creux. On a alors $t_c = \alpha - 1$.

La figure 1.13 présente un exemple de détermination de régime stationnaire pour un système de deux tâches $\tau_1 = (0, 1, 4, 4)$ et $\tau_2 = (4, 3, 6, 6)$.

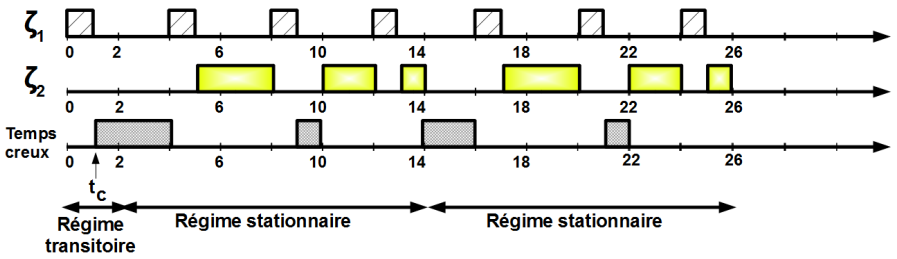


FIGURE 1.13 – Détermination de régime stationnaire d'ordonnancement de deux tâches $\tau_1 = (0, 1, 4, 4)$ et $\tau_2 = (4, 3, 6, 6)$: $t_c = 1$

Par souci de simplification, nous ne considérons dans nos travaux que des séquences périodiques à partir de $t = 0$, c'est-à-dire les tâches périodiques à départs simultanés ou à départs différés et dont $t_c = -1$. Les cas où le régime stationnaire ne commence pas à 0 seront l'objet de travaux à venir.

IV.3.b. Techniques de recherche de séquences

De nombreuses techniques de recherche de séquences d'ordonnancement hors-ligne existent. Elles peuvent être classées en deux grandes catégories [Gro99], à savoir les techniques exactes et les techniques approchées. Les techniques exactes sont basées principalement sur des techniques d'énumération complète. Entre autres techniques nous pouvont citer celles utilisant la programmation linéaire ([GLLR79] [Mar82]), les techniques de séparation et évaluation ([BS74], [XP90]), et celles à base de modèles : réseaux de Petri ([GCG02]), automates finis ([GL07]), chaînes de Markov ([CG07]) ou géométrie discrète ([LGA05]).

Les techniques approchées sont des algorithmes de recherche non exhaustives tels que les algorithmes de mise en sac ou de liste, ou bien des algorithmes stochastiques tel que les algorithmes génétiques ou le recuit simulé [Bea96]. Les techniques approchées ne traitent pas du temps réel dur et gèrent surtout le placement. Nous nous intéressons donc principalement aux techniques exactes.

La technique de recherche de séquences d'ordonnancement hors-ligne présentée par [BS74] considère des tâches non préemptives. Cette technique procède par l'énumération de toutes les combinaisons possibles d'ordonnancement. Pour cela, elle construit un arbre dans lequel chaque nœud fils s'obtient à partir du noeud père par ajout d'une instance de tâche. En définitive, la recherche produit un arbre de profondeur égale aux nombres d'instances à ordonnancer.

La technique de recherche de séquences d'ordonnancement présentée par [XP90] considère quant à elle des tâches préemptibles pouvant partager des ressources ou présentant des contraintes des précédences. Pour cela, une tâche est constituée de plusieurs segments. Les contraintes de précédences et d'exclusion sont des opérateurs entre segments. L'algorithme recherche dans un arbre une solution au problème d'ordonnancement. La racine de l'arbre est la séquence d'ordonnancement produite en utilisant l'algorithme ED (Earliest Deadline). Les noeuds fils s'obtiennent à partir des noeuds pères en rajoutant des contraintes (précédence, exclusion). Les séquences d'ordonnancement produites peuvent prévoir des préemptions.

La technique de recherche d'ordonnancement présentée par [GCG02] modélise l'application par un réseau places/transitions (réseau de Petri), avec ensemble terminal et fonctionnant sous la règle du tir maximal. La règle du tir maximal permet de modéliser le temps. L'ensemble terminal permet de tenir compte des délais critiques, et le réseau permet de modéliser l'ensemble des tâches ainsi qu'un système d'horlogerie qui permet de gérer les dates de l'activation et les périodes. Les séquences valides s'obtiennent ensuite par construction du graphe de marquages terminaux.

IV.3.c. Mise en œuvre d'ordonnancement hors-ligne

La très grande majorité des sytèmes d'exploitation ne prend pas en compte les ordonnancements hors-ligne. A notre connaissance seules quelques approchent le font, telles que Mars [KB03], OSEK Time Triggered [BSD+01] et l'approche par modification d'ordonnanceur [Sou01].

L'approche Mars ou Time Triggered Architecture Mars ([SRG89], [KB03]) est un système d'exploitation temps réel pour une architecture matérielle distribuée. Chaque calculateur exécute une instance du système d'exploitation.
L'approche Mars propose une architecture matérielle et logicielle mettant en œuvre des séquences d'ordonnancement précalculées (hors-ligne). D'un point de vue matériel, Mars repose sur une architecture distribuée constituée de clusters, d'objets contrôlés et d'un ordinateur de maintenance. Un cluster est un groupe de composants (ordinateurs) interconnectés par un bus. Sur chaque composant s'exécute le système d'exploitation Mars et les logiciels applicatifs. Le système d'exploitation Mars repose sur un ordonnanceur dirigé par le temps. Au niveau applicatif, un modèle transactionnel est utilisé. Une tâche est vue comme une transaction constituée d'actions élémentaires. Elle est déclenchée par un stimulus et produit une réponse. Les processus sont périodiques et communiquent par échange de messages. Il y a deux types de messages à savoir les messages évènementiels (survenue d'un problème) et les messages d'états (gestion du flux et des collisions, ordonnancement). Mars ne gère pas les interruptions matérielles autres que l'interruption d'horloge et l'allocation de la mémoire se fait de manière statique. De plus, Mars intègre un système de tolérance aux fautes. Le changement d'une séquence précalculée à une autre séquence (ou changement de mode) est également pris en compte.
Mars utilise un matériel spécifique dédié et n'a donc pas une utilisation généralisée.

OSEK Time Triggered OSEK time triggered est une spécification décrivant le concept de système d'exploitation à déclenchement temporel (OS Time-triggered). Il s'agit d'un complément de la spécification OSEK/VDX [BSD+01] utilisé dans l'industrie (automobile principalement). [BSD+01] présente un ensemble de services rendant conforme un RTOS à la spécification. Un RTOS OSEKtime offre en plus des services du RTOS distribué OSEK/VDX (gestion d'interruptions, dispatching, synchronisation d'horloge, communication par messages) des services de gestion de table d'ordonnancement et un service de détection d'erreurs. Les tâches sont exécutées séquentiellement et sont déclenchées par un évènement. Il y a deux types de tâches à savoir des tâches déclenchées par interruptions matérielles (ISR), et des tâches déclenchées par un dispatcher qui lit une table d'ordonnancement. Les tâches déclenchées par le dispatcher sont dites "Time Triggered Task" (TT task) et ont trois états possibles (running, preempted, suspended). Une TT task est préemptée lorsqu'elle n'a pas fini son exécution et qu'une autre TT task est activée. La table d'ordonnancement indique les dates auxquelles une tâche doit être activée. Il est également possible d'ajouter à la table d'ordonnancement des dates de vérification d'échéances.
A notre connaissance aucun système d'exploitation n'implémente OSEK/TT.

Modification d'ordonnanceur L'approche présentée par [Sou01] consiste à modifier l'ordonnanceur du système d'exploitation temps réel RTAI afin qu'il puisse exécuter une séquence d'ordonnancement précalculée. La séquence est fournie à l'ordonnanceur sous forme d'une liste chainée de fonctions (voir figure 1.14).

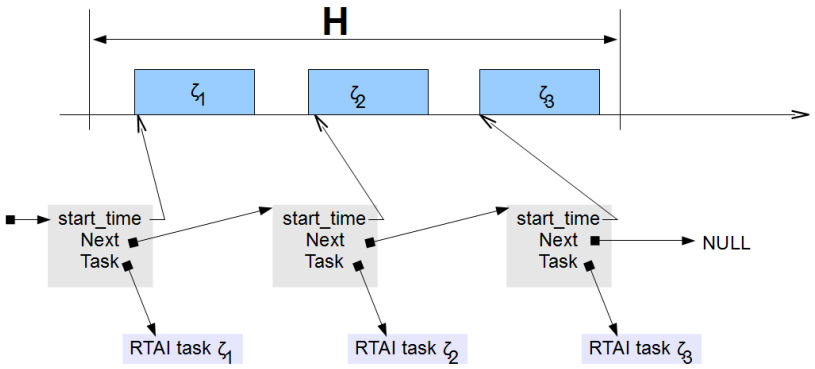


FIGURE 1.14 – Approche de modification de l'ordonnanceur RTAI.

La limite de cette approche est qu'elle ne permet que la mise en œuvre des séquences d'ordonnancement qui ne prévoit pas de préemption. De plus, pour l'appliquer à un autre système d'exploitation il faut au préalable le modifier. Chaque système d'exploitation ayant une structure particulière, l'activité de modification ne peut être automatisée et l'effort déployé pour modifier une version d'un système d'exploitation sera le même pour modifier une mise à jour de ce système.
Compte tenu de la puissance de l'ordonnancement hors-ligne et de sa faible prise en compte dans les systèmes d'exploitation temps réel, d'autres approches de mise en œuvre contournent la difficulté en s'intéressant aux mises en œuvre sur des systèmes d'exploitation classiques. L'application temps réel est développée pour s'exécuter sur un système d'exploitation classique. Nous parlons alors de mise en œuvre applicative d'ordonnancement hors-ligne.

V. Mise en œuvre applicative d'ordonnancement hors-ligne

La littérature, sur les techniques de mise en œuvre applicative d'ordonnancement hors-ligne, bien qu'existante est loin d'être abondante. En effet seuls quelques travaux abordent ce sujet. Nous présentons dans cette section les quelques papiers que nous avons trouvés sur le sujet.

V.1. Mise en œuvre par exécutif cyclique

D'un point de vue historique, la première approche de mise en œuvre d'ordonnancement hors-ligne est l'approche par exécutif cyclique [BS89].
Un exécutif cyclique est une structure de contrôle ou un programme qui implémente une séquence d'ordonnancement d'un système de tâches périodiques. La séquence d'ordonnancement est déterminée hors-ligne de sorte que toutes les contraintes temporelles (période, échéance) soient respectées. De plus, la séquence mise en œuvre est telle qu'elle peut être subdivisée en

plusieurs petites séquences de même longueur m ou frame. La figure 1.15-A présente un exemple. La mise en œuvre consiste à écrire une tâche qui à chaque interruption d'horloge incrémente un numéro de sous-séquence et en fonction du numéro de sous-séquence exécute les tâches de la sous-séquence. La figure 1.15-B présente l'algorithme de mise en œuvre correspondant à la séquence de la figure 1.15-A.
[PNME03] présentent des exemples de mises en œuvre effectives en langage C d'exécutif cyclique.

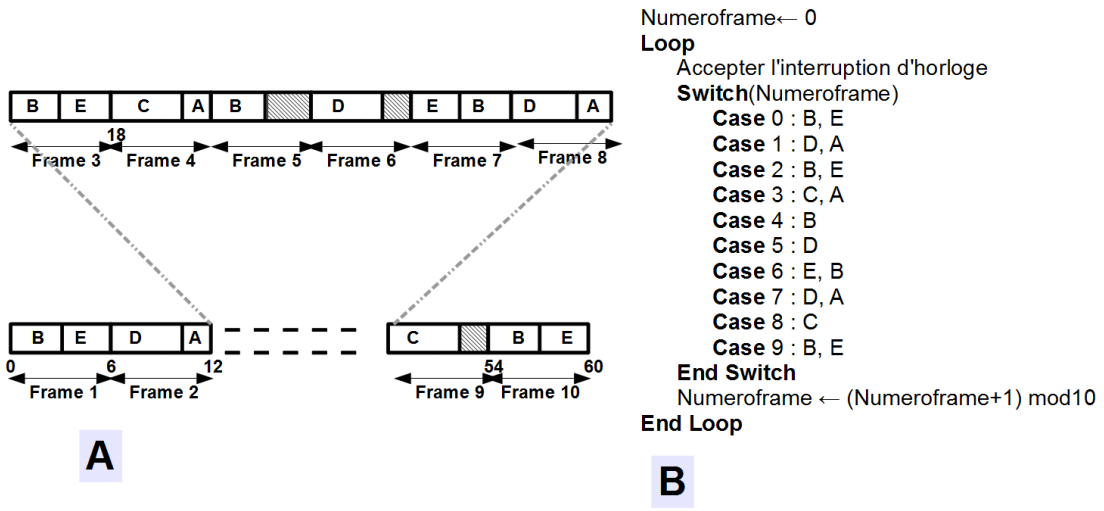


FIGURE 1.15 – Exécutif cyclique. A : Exemple de séquence d'ordonnancement avec division en frames ; B : Mise en œuvre de la séquence par exécutif cyclique

Une version plus élaborée prenant en compte l'exécution de tâches non périodiques (sporadiques ou apériodiques) de la mise en œuvre par exécutif cyclique est présentée par [Liu00]. La différence se situe principalement dans le fait que la séquence est enregistée dans une table où chaque entrée est une tâche périodique. La table enregistre également les temps creux prévus par la séquence. Ces temps sont alors utilisés pour exécuter les tâches non périodiques.
L'inconvénient de la mise en œuvre est qu'elle se limite à un type particulier de séquences à savoir celles que l'on peut subdiviser en sous-séquences de même longueur. Ce qui est assez réduit comme champ d'application. Les techniques de mise en œuvre proposées dans [Xu03] (section V.2) élargissent un peu plus les champs d'application.

V.2. Mise en œuvre d'ordonnancement par un processus

[Xu03] présente une mise une mise en œuvre effective d'ordonnancement hors-ligne par la synthèse d'une application dont le code est une suite séquentielle d'instructions déduite de la

séquence d'ordonnancement. La figure 1.16-A présente un exemple de séquence d'ordonnancement avec sa mise en œuvre (figure 1.16-B). Une version plus modulaire de cette mise en œuvre, proposée par [Sou01], consiste à utiliser une tableau pour enregistrer la suite des fonctions exécutées.

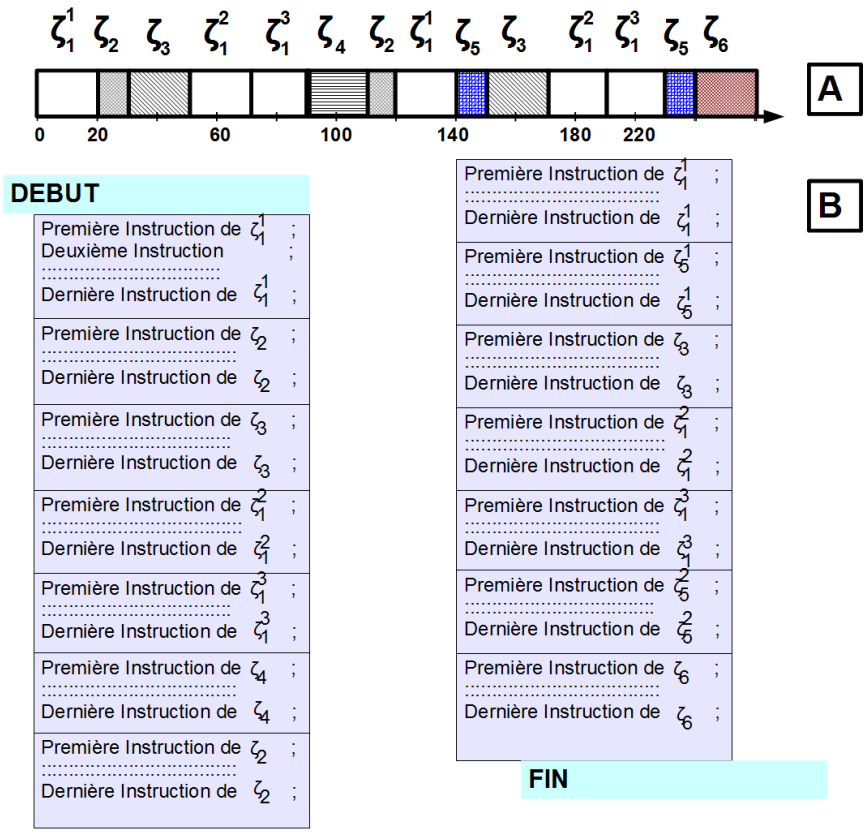


FIGURE 1.16 – Mise en œuvre mono-tâche d'une séquence d'ordonnancement hors-ligne

Les mises en œuvre supposent que l'on peut toujours déduire de la séquence d'ordonnancement un découpage des instructions des tâches. Une telle activité de découpage peut s'avérer très complexe. De plus, la mise en œuvre proposée ne tient pas compte du fait que les tâches peuvent prendre moins de temps que le pire temps prévu et qu'il faut donc prévoir des mécanismes de synchronisation avec le temps. Afin d'assurer cette synchronisation avec le temps, [DGC02] (section V.3) propose d'utiliser une tâche supplémentaire qui décide quelles tâches s'exécutent et à quelles dates.

V.3. Mise en œuvre d'un dispatcher sur VxWorks

La mise en œuvre de [DGC02] propose d'ajouter des synchronisations par sémaphore afin de délimiter dans une tâche les parties non préemptibles.
Une tâche supplémentaire appelée *Scheduler* est chargée de gérer les différents points de préemption. Son fonctionnement peut être décrit comme une boucle qui suit les étapes ci-dessous :
1 le *Scheduler* attend qu'une tâche lui rende la main (en attendant un sémaphore) ;
2 si la tâche n'a pas terminé le temps qui lui était alloué (temps d'exécution inférieur à la durée d'exécution prévue), le *scheduler* effectue une attente active pour finir l'intervalle de temps ;
3 le *scheduler* détermine la prochaine tâche à exécuter et la durée d'exécution prévue ;
4 le *scheduler* active la prochaine tâche à exécuter (en libérant un sémaphore) ;
Si cette mise en œuvre permet une synchronisation avec le temps, elle est cependant peu robuste parce que dès que l'une des tâches est bloquée tout le système est bloqué et la tâche *scheduler* ne peut même pas reprendre la main. Pour éviter un blocage il est possible de garder une indépendance des tâches. C'est la solution de mise en œuvre présentée par [DFP01] que nous présentons dans la section V.4.

V.4. Utilisation de priorités fixes

[DFP01] présente une méthode pour passer d'une séquence construite hors-ligne à un système de tâches périodiques à priorités fixes dont l'ordonnancement réalise la séquence hors-ligne. Pour déterminer l'affection des priorités, [DFP01] examine la séquence d'ordonnancement pour en dériver un certain nombre d'inégalités de niveaux de priorités. Par exemple à une date t si deux instances de tâches τ_{ij} et τ_{kp} sont actives en même temps et si la séquence prévoit l'exécution de τ_{ij} avant τ_{kp}, on peut en déduire que $prio(\tau_{ij}) > prio(\tau_{kp})$. De même, ils vérifient la périodicité de débuts d'exécution de chaque tâche. Dès lors qu'il n'y a pas de périodicité dans les débuts d'exécution ou qu'il y a un conflit de priorité (c'est-à-dire des inégalités contradictoires) ils préconisent de découper les tâches en conflit de sorte que chaque instance de tâche devienne une tâche à part entière et donc avec un niveau de priorité.
Les niveaux de priorités nécessaires peuvent être très grands. En plus, dès que la table d'ordonnancement n'implique pas de conservatisme, il devient difficile d'avoir une configuration de priorités valides. Par exemple si deux instances de deux tâches ont successivement des ordres de priorité inversés il n'y a pas de solution.

VI. Conclusion

Ce premier chapitre nous a permis d'introduire les notions fondamentales en théorie de l'ordonnancement. Il a également permis de faire un tour d'horizon des approches de recherche et de mise en œuvre d'ordonnancement hors-ligne. Nous avons aussi constaté la faible prise en compte de la mise en œuvre effective d'ordonnancement hors-ligne. Pour palier cette insuffi-

sance, dans cette thèse, nous développons une approche la plus générale possible de mise en œuvre d'ordonnancement hors-ligne.

Chapitre

2 Contexte, méthodologie et modélisation

Sommaire

Résumé

Dans ce chapitre nous présentons la problématique de cette thèse. Nous précisons nos hypothèses et notre méthodologie de travail. Nous formalisons également les politiques de mise en œuvre, puis pour finir nous présentons un modèle de coûts dont l'objectif est de servir de support de comparaison des techniques de mise en œuvre d'ordonnancement hors-ligne

Le chapitre 1 nous a permis d'introduire le vocabulaire utilisé et de présenter les travaux en relation avec notre sujet de thèse. Dans ce chapitre nous abordons notre sujet que nous décrivons ainsi que nos hypothèses et notre méthodologie de travail.

I. Problématique

Dans le chapitre 1 nous avons présenté de manière générale des aspects théoriques liés à l'informatique temps réel.
Dans un premier temps nous avons fait un panorama des architectures matérielles et logicielles des systèmes temps réel. En ce qui concerne l'architecture logicielle, nous avons présenté les systèmes d'exploitation temps réel (RTOS) qui sont des systèmes d'exploitation conçus pour la mise en œuvre des systèmes temps réel.
Nous avons également présenté les deux grandes catégories d'algorithmes d'ordonnancement à savoir les algorithmes d'ordonnancement en ligne et hors-ligne.
L'approche hors-ligne se justifie par le fait que dans le cas général du contexte multiprocesseur, même pour les tâches indépendantes, il n'y a pas d'algorithmes polynomiaux optimaux en ligne. Dans le contexte monoprocesseur, il n'y a pas d'algorithmes polynomiaux optimaux dès lors que des ressources critiques sont partagées.
Les algorithmes d'ordonnancement hors-ligne sont développés afin de palier les insuffisances des algorithmes d'ordonnancement en ligne. Par exemple dans [Gro99], un algorithme hors-ligne permet de trouver des séquences hors-ligne valides lorsqu'aucun algorithme en ligne ne permet pas d'en trouver. Ce qui illustre bien leur plus grande puissance.
Si de nombreux travaux existent sur la détermination des séquences d'ordonnancement hors-ligne, il en existe peu sur la réalisation pratique d'applications temps réel utilisant une approche d'ordonnancement hors-ligne. En effet, comme cela est présenté dans le chapitre 1, à notre connaissance, seul le système d'exploitation Mars [SRG89] gère de manière native la mise en œuvre d'ordonnancement hors-ligne. Cependant, son périmètre d'utilisation est restreint du fait que le matériel (stations d'accueil) utilisé est spécifique.
Ce constat nous a amenés à centrer cette thèse sur l'étude de l'exécution d'une application sur un système d'exploitation temps réel classique conformément à une séquence hors-ligne.
Nous abordons les questions suivantes :

- Quelles sont les possibilités techniques (algorithmiques, services des systèmes d'exploitation, etc.) ?
- Les séquences d'ordonnancement pré-déterminées sont-elles directement implémentables ou bien faut-il prévoir des mécanismes supplémentaires lors de leur détermination ? En d'autres termes quelle est l'adéquation entre les modèles théoriques et les possibilités techniques ?
- Quels outils proposer pour l'aide au développement d'applications temps réel basées sur une approche hors-ligne ?

Après avoir présenté nos hypothèses de travail dans la section II, nous présentons la méthodologie (section III) suivie pour répondre aux différentes questions soulevées par notre sujet

de thèse. Nous formalisons l'analyse de l'exécution effective dans la section IV. Pour finir nous présentons notre démarche (section V) et nous introduisons dans la section VI un modèle de coûts qui permet de comparer des mises en œuvre.

II. Hypothèses de base

L'étude que nous menons dans le cadre de cette thèse se limite au contexte monoprocesseur.
Les techniques de recherche de séquences d'ordonnancement hors-ligne ne considèrent que des tâches dont les dates d'activation sont connues à l'avance. Pour cela, nous ne considérons que des tâches périodiques. Si l'application comporte des tâches non périodiques, elles peuvent être intégrées à nos tâches périodiques par l'intermédiaire de serveurs périodiques. En effet les serveurs de tâches apériodiques ou sporadiques ([CG03]) permettent l'exécution en ligne d'éventuelles tâches non périodiques.
Nous considérons que l'application temps réel à mettre en œuvre hors-ligne est disponible sous la forme de tâches dont on connait les paramètres temporels et les instructions fonctionnelles.
Nous supposons que les séquences d'ordonnancement à mettre en œuvre sont périodiques dès le début (c'est-à-dire à partir de 0).
S'agissant de la gestion de la mémoire nous supposons qu'il n'y a pas d'allocation dynamique. En d'autres termes la mémoire est allouée soit de manière statique soit sur la pile. Cela se justifie par le fait que le plan d'exécution de l'application et plus particulièrement les variables globales et locales sont connues à l'avance.
Les mises en œuvre se font dans un contexte où le calculateur n'exécute qu'un seul logiciel applicatif et le système d'exploitation temps réel cible. Nous ne traitons donc pas les cas de mise en œuvre où le système exécute simultannément plusieurs applications. Le comportement de l'application ne dépend que du système d'exploitation cible et de l'application elle même, aucun autre élément n'intervient. Par exemple, un processus unique est élu dès qu'il est activé. Nous supposons que la politique d'ordonnancement est un ordonnancement à priorités (le processus prêt de plus grande priorité est celui qui s'exécute). Ce choix est motivé par le fait que presque tous les systèmes d'exploitation temps réel possèdent un ordonnancement à priorités fixes.

III. Méthodologie

Pour mettre en œuvre une application dans une approche d'ordonnancement hors-ligne, il y a plusieurs étapes. Nous focalisons notre attention sur les étapes en lien direct (celles dont le résultat est nécessaire) avec l'étape de mise en œuvre de séquences d'ordonnancement hors-ligne. En raison de cette focalisation sur les étapes en lien direct avec notre sujet, nous n'évoquerons pas les étapes d'analyse, de conception, de tests.
La figure 2.1 présente un aperçu des quatre étapes que nous considérons.

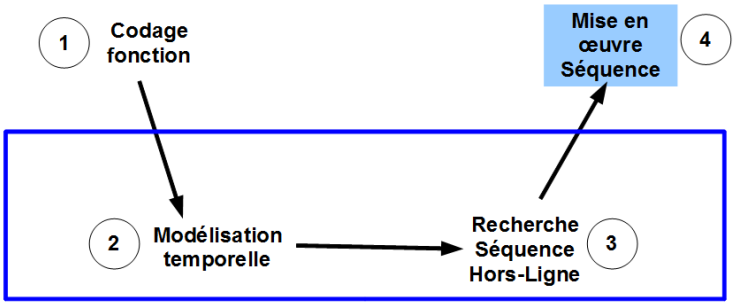


FIGURE 2.1 – Description de notre approche de mise en œuvre d'ordonnancement hors-ligne.

III.1. Codage des fonctions

L'étape 1 consiste à déterminer l'implémentation fonctionnelle de l'application. Le résultat de cette étape est un ensemble de fonctions $F = \{f_1, f_2, \ldots, f_n\}$. Chaque fonction est une liste d'instructions qui réalise une fonctionnalité (par un exemple un calcul) de l'application.

III.2. Modélisation temporelle

L'étape 2 consiste à estimer les durées d'exécution des fonctions en utilisant une méthode statique ou dynamique (voir section III.2.b du chapitre 1). À la suite de l'estimation des durées d'exécution, d'autres contraintes temporelles (dates d'activation, échéances, périodes) permettent d'adjoindre un modèle temporel aux fonctions de l'étape 1. Nous avons donc le système de tâches $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ avec pour chaque tâche un modèle fonctionnel.

Définition 2.1 (Modèle fonctionnel de tâche périodique)**.** *Une tâche fonctionnelle périodique* τ_i *est caractérisée par* $< r_i, [C_i^{min} : C_i^{max}], D_i, T_i | f_i >$ *:*

- r_i *est la date d'arrivée de la première instance de la tâche* τ_i*, encore appelée date de première activation ou offset ;*
- C_i^{min} *est la durée d'exécution au meilleur cas, elle spécifie un minorant du temps d'exécution de chaque instance de la tâche* τ_i *;*
- C_i^{max} *est la pire durée d'exécution, elle spécifie un majorant du temps d'exécution de chaque instance de la tâche* τ_i *;*
- D_i *est l'échéance relative ou délai critique, elle dénote la durée séparant l'arrivée d'une instance et son échéance ;*
- T_i *est la période, c'est l'intervalle de temps qui sépare l'arrivée de deux instances successives de* τ_i *;*
- f_i *est la fonction modélisée par la tâche.*

La définition 2.1 étend les définitions 1.2 et 1.3 (chapitre 1, section III.2) et permet ainsi de garder le modèle fonctionnel dans la modélisation du système de tâches. Les autres définitions (instances de tâches, taux de charge, etc.) restent les mêmes.

III.3. Recherche de séquence hors-ligne

L'étape 3 consiste à obtenir un modèle théorique d'exécution. À partir des modèles fonctionnels (voir définition 2.1) des tâches, une stratégie de recherche de séquences d'ordonnancement, telles que celles présentées dans la section IV.3.b du chapitre 1, permet de déterminer une séquence d'ordonnancement valide. Le plus souvent les techniques de recherche d'ordonnancement produisent la séquence d'ordonnancement pire cas (en considérant que toutes les tâches s'exécutent pendant C_i^{max} unités de temps) sous la forme d'une suite de triplets que nous appelons scénario d'exécution.

Définition 2.2 (Scénario d'exécution)**.** *Un scénario d'exécution est une suite de triplet* $Se(t_1, t_2) = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), (start_2, end_2, \tau_{\alpha(2)\beta(2)}), \ldots, (start_q, end_q, \tau_{\alpha(q)\beta(q)})\}$*. La sémantique associée est la suivante : l'instance* $\beta(i)$ *de* $\tau_{\alpha(i)}$ *s'exécute entre* $start_i$ *et* end_i *; avec* $end_i \leq start_{i+1}$ *et* $start_1 \geq t_1$*,* $end_q \leq t_2$*. Chaque triplet* $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$ *est appelé un bloc d'exécution et le nombre de blocs d'exécution* $q = |Se|$ *est appelé la dimension du scénario.*

Notons que l'on peut matérialiser, dans les scénarios d'exécution, les temps creux par τ_0. Nous utilisons alors la notation $\overline{Se}$. Par exemple en matérialisant les temps creux du scénario d'exécution $Se(0, 10) = \{(0, 3, \tau_{11}), (5, 7, \tau_{21})\}$ nous avons $\overline{Se(0, 10)} = \{(0, 3, \tau_{11}), (3, 5, \tau_0), (5, 7, \tau_{21}), (7, 11, \tau_0)\}$.
Dans la suite nous ne nous intéressons qu'aux scénarios d'exécution Se qui ne matérialisent pas les temps creux c'est-à dire :

$$Se(t_1, t_2) = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), (start_2, end_2, \tau_{\alpha(2)\beta(2)}), \ldots\}, \forall i \in [1, |Se|]\alpha(i) \neq 0$$

Les techniques de recherche de séquences d'ordonnancement calculent des scénarios d'exécution pire cas (c'est-à-dire en considérant les C_i^{max}) sur l'hyper-période. Nous notons $Se_{theorique}$ ce scénario d'exécution calculé hors-ligne sur l'hyper-période.
Dans la suite nous ne considérons que des scénarios d'exécution $Se_{theorique}$ valides c'est-à-dire qui prévoient l'exécution de toutes les instances de tâches sur l'hyper-période avec un respect de toutes les contraintes temporelles. Plus formellement nous aurons la propriété 2.3.

Propriétés 2.3 (Scénario d'exécution valide)**.** *Un scénario d'exécution théorique est valide si et seulement si :*

$$\begin{cases} \forall i \in [1, |Se_{theorique}|] & \alpha(i) \in \{1, \ldots, n\}, \beta(i) \in \{1, \ldots, \frac{H}{T_{\alpha(i)}}\} \\ \forall k \in \{1, \ldots, n\}, l \in \{1, \ldots, \frac{H}{T_k}\} & \exists i \in [1, |Se_{theorique}|], (\alpha(i), \beta(i)) = (k, l) \\ \forall i \in [1, |Se_{theorique}|] & r_{\alpha(i)\beta(i)} \leq start_i \\ \forall i \in [1, |Se_{theorique}|] & d_{\alpha(i)\beta(i)} \geq end_i \\ i = 1, \ldots, |Se_{theorique}| & \forall (k, l), \sum_{\alpha(i)=k, \beta(i)=l} (end_i - start_i) = C_k^{max} \end{cases} \quad (2.1)$$

III.4. Mise en œuvre de séquence

L'étape 4 est celle que nous abordons dans cette thèse. Elle consiste à utiliser la séquence d'ordonnancement pour construire un modèle opérationnel. En d'autres termes elle consiste à déterminer un ensemble $\sigma = \{\sigma_1, \sigma_2, \ldots\}$ de processus destiné à suivre le scénario d'exécution théorique lors de l'exécution de l'application.
L'exécution du modèle opérationnel conduit à un scénario d'exécution que nous notons $Se_{effectif}$. Le scénario d'exécution théorique a été déterminé à partir des C_i^{max}. Lors d'une exécution de l'application les durées effectives sont très souvent différentes de la pire durée d'exécution et il y a deux situations possibles.

Situation 1 : Les durées d'exécution peuvent être hors des bornes prévues. Dans ce cas on parle d'*erreur de sous-calibrage* et d'*erreur de sur-calibrage* de l'application. Un scénario d'exécution présente une erreur de sous-calibrage lorsque la durée d'exécution d'une instance de tâche est plus grande que la pire durée prévue. Plus formellement nous avons la propriété 2.4. Une erreur de sur-calibrage correspond au fait que la durée d'exécution d'une instance est inférieure à la durée d'exécution meilleur cas (Propriété 2.5).

Propriétés 2.4 (Erreur de sous-calibrage)**.** *Un scénario d'exécution effectif fait apparaitre une erreur de sous-calibrage si et seulement si :*

$$\left\{ \; i = 1, \ldots, |Se_{theorique}| \quad \exists (k,l), \sum_{\alpha(i)=k, \beta(i)=l} (end_i - start_i) > C_k^{max} \right. \tag{2.2}$$

Propriétés 2.5 (Erreur de sur-calibrage)**.** *Un scénario d'exécution effectif fait apparaitre une erreur de sur-calibrage si et seulement si :*

$$\left\{ \; i = 1, \ldots, |Se_{theorique}| \quad \exists (k,l), \sum_{\alpha(i)=k, \beta(i)=l} (end_i - start_i) < C_k^{min} \right. \tag{2.3}$$

Nous reviendrons sur l'impact des erreurs de calibrage dans la section VI.5.

Situation 2 : Les durées d'exécution peuvent être inférieures aux C_i^{max}. Il faut donc décider de la politique à suivre dans ce cas et intégrer cette politique dans le modèle opérationnel. De plus, se pose la question de déterminer si un scénario d'exécution effectif suit ou ne suit pas un scénario d'exécution théorique. Nous reviendrons plus en détail sur ce point dans la section IV.

Notons pour conclure cette partie que, pour faire une mise en œuvre dans une approche d'ordonnancement en ligne la même méthodologie (figure 2.1) peut être utilisée. L'étape 3 de calcul de séquence est remplacée par une étape de validation pire cas, qui consiste à vérifier qu'au pire cas toutes les contraintes temporelles sont respectées.

IV. Politiques de mise en œuvre de séquences d'ordonnancement hors-ligne

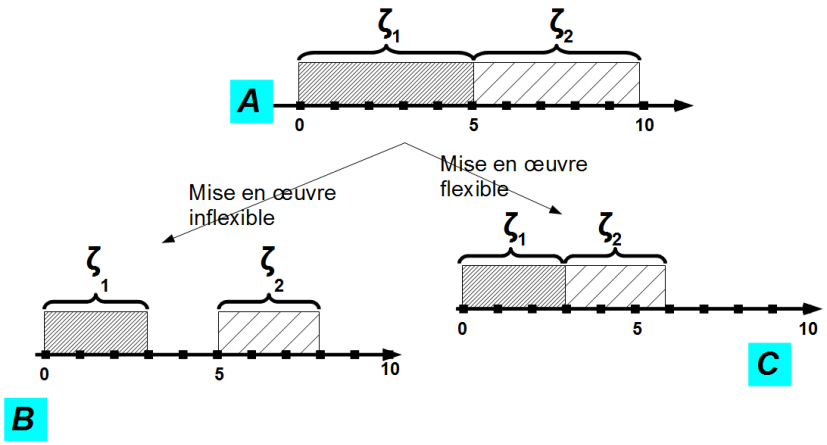


FIGURE 2.2 – Ordonnancement pire cas (A) et mises en œuvre effectives (B, C)

Pour illustrer la nécessité d'intégrer des politiques de mise en œuvre de séquences d'ordonnancement, nous utilisons l'exemple d'ordonnancement de deux tâches dont les paramètres temporels sont : $\tau = \{\tau_1 =< 0, [3:5], 10, 10|f_1, >; \tau_2 =< 0, [3:5], 10, 10|f_2 >\}$. Ces paramètres temporels impliquent que l'étape de modélisation temporelle a estimé que les durées d'exécution de chaque tâche sont comprises entre $C^{min} = 3$ et $C^{max} = 5$.

Supposons que l'étape de calcul de séquence d'ordonnancement produise le scenario d'exécution proposé en figure 2.2-A : $Se_{theorique} = \{(0, 5, \tau_{11}), (5, 10, \tau_{21})\}$ Les deux tâches étant périodiques de période 10, la séquence d'ordonnancement est aussi périodique. Les tâches étant à départs simultanés, la séquence est cyclique à partir de $t = 0$ avec une période $H = ppcm(10, 10) = 10$.

Considérons un scenario d'exécution où les durées d'exécution des deux tâches sont minimales c'est-à-dire égale à 3. Il y a deux grandes possibilités de mise en œuvre.

La première (figure 2.2-B) consiste à faire une mise en œuvre qui impose d'attendre toujours la date 5 avant d'exécuter τ_{21}.

La deuxième (figure 2.2-C) consiste à faire une mise en œuvre qui permette l'exécution de τ_{11} dès que possible.

Nous qualifions la première mise en œuvre de mise en œuvre *inflexible* et la deuxième mise en œuvre de mise en œuvre *flexible*.

La mise en œuvre d'une séquence d'ordonnancement $Se_{theorique} = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), (start_2, end_2, \tau_{\alpha(2)\beta(2)}), \ldots\}$ est inflexible si pour tout i on ne peut exécuter que $\tau_{\alpha(i)\beta(i)}$ entre $[start_i, end_i)$. Dans ce cas, si une instance $\tau_{\alpha(i)\beta(i)}$ termine plus tôt que prévu, même si l'instance

suivante $\tau_{\alpha(i+1)\beta(i+1)}$ est prête, on attend la date $start_{i+1}$ spécifiée par la scénario d'exécution théorique.
Dans le cas de la mise en œuvre flexible d'un scénario d'exécution théorique $Se_{theorique} = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), (start_2, end_2, \tau_{\alpha(2)\beta(2)}) \ldots, \}$, si une instance $\tau_{\alpha(i)\beta(i)}$ termine plus tôt que prévu par le scénario et que l'instance suivante $\tau_{\alpha(i+1)\beta(i+1)}$ est prête, on peut démarrer son éxécution sans attendre la date $start_{i+1}$.
Dans le premier cas (mise en œuvre inflexible), on a la garantie de vérifier les contraintes temporelles et aucune vérification supplémentaire n'est nécessaire. Dans le second cas (mise en œuvre flexible), il faut s'assurer que l'instance est active et donc vérifier les dates de réveil chaque fois que l'on voudra démarrer l'instance de tâche plus tôt que prévu par le scénario d'exécution théorique.
Par contre, dans le premier cas, les temps creux engendrés par les exécutions plus courtes seront disséminés alors que dans le second cas, ils seront d'avantage regroupés. Si l'on souhaite coupler l'exécution de la séquence périodique à un système de gestion en ligne de tâches apériodiques, l'exécution des tâches apériodiques sera davantage morcelée dans le premier cas (mise en œuvre inflexible) que dans le second cas (mise en œuvre flexible). Si les changements de contexte ont un coût non négligeable, la politique de mise en œuvre flexible sera alors préférable.
Dans la suite, nous envisageons les deux politiques de mise en œuvre. Nous les formalisons dans le cas non préemptif (section IV.2) puis dans le cas préemptif (section IV.3).
Nous utilisons pour cette formalisation des propriétés issues de la théorie des mots. En effet, un scénario d'exécution peut être vu comme une suite d'éléments à deux composantes : une première composante qui indique l'instance à exécuter et une deuxième composante qui indique les fenêtres temporelles d'exécution.
La projection du scénario d'exécution sur la première composante est un mot dont les lettres sont les instances de tâches. Vérifier qu'un scénario d'exécution effectif suit un scénario d'exécution théorique nécessite donc de comparer deux mots.
Avant de formaliser les politiques de mise en œuvre, nous présentons dans la section IV.1 des définitions issues de la théorie des mots que nous utilisons dans notre formalisation. Nous y présentons également les définitions qui nous permettent de représenter des scénarios d'exécution sous forme de mots.

IV.1. Mots et scénarios d'exécution

Nous présentons dans la suite les définitions qui nous permettront de déterminer si un scénario d'exécution effectif respecte un scénario d'exécution théorique.

Alphabet Un alphabet A est un ensemble fini non vide de symboles appelés lettres.

Mot Un mot w est une suite de lettres d'un alphabet A. Comme nous nous limitons à des analyses sur l'hyperpériode nous ne considérons dans la suite que des suites finies et nous notons A^* l'ensemble des mots écrits sur un alphabet A.

Mot vide Le mot vide ϵ (avec $\epsilon \in A^*$) est le mot constitué d'une suite de 0 lettre.

Longueur d'un mot La longueur $|w|$ d'un mot w est le nombre de lettres qui composent le mot.

Égalité "=" Deux mots $w_1 = a_1a_2...a_{|w_1|}$ et $w_2 = b_1b_2...b_{|w_2|}$ (avec $w_1, w_2 \in A^*$) sont égaux si et seulement si leurs lettres sont égales deux à deux :

$$\begin{cases} |w_1| = |w_2| \\ \forall i \in [1, |w_1|] \quad a_i = b_i \end{cases} \tag{2.4}$$

Nombre d'occurrences Le nombre d'occurrences $|w|_a$ d'une lettre $a \in A$ dans un mot w est le nombre de fois où a apparait dans w.

Concaténation "." La concaténation de deux mots $w_1 = a_1a_2...a_{|w_1|}$ et $w_2 = b_1b_2...b_{|w_2|}$ est le mot $w_1.w_2 = w_1w_2 = a_1a_2...a_{|w_1|}b_1b_2...b_{|w_2|}$

Sous-mot Un mot $w_1 = a_1a_2...a_{|w_1|}$ est un sous-mot de w_2 si et seulement si il existe pour tout $i \in [1, |w_1| + 1]$ un mot s_i tel que $s_i \in A^*$ et $w_2 = s_1a_1s_2a_2...s_{|w_1|}a_{|w_1|}s_{|w_1|+1}$. Nous notons : $w_1 \subseteq w_2$

Afin de ramener la comparaison de deux scénarios d'exécution à la comparaison de deux mots, nous définissons la notion de *mot topologique*.

Définition 2.6 (Mot topologique d'un scénario d'exécution)**.** *Le mot topologique M d'un scénario d'exécution Se est le mot $M = m_1m_2...m_{|Se|}$ avec $m_i = \tau_{\alpha(i)\beta(i)}$.*

Notre étude se portant sur des ordonnancements cycliques sur l'hyper-période H, nos scénarios d'exécution auront comme alphabet l'ensemble des instances de tâches actives sur l'hyper-période. Ainsi, l'alphabet pour un système de n tâches périodiques est :

$$A = \{\tau_{ij}, i \in \{1, ..., n\}, j \in \{1, ..., \tfrac{H}{T_i}\}\} \tag{2.5}$$

Par exemple, l'alphabet des mots topologiques des scénarios d'exécution du système de tâches $\tau = \{\tau_1 =< 0, [3:5], 10, 10|f_1, >; \tau_2 =< 0, [3:5], 10, 10|f_2 >\}$ est : $A = \{\tau_{11}, \tau_{21}\}$. Le mot topologique du scénario d'exécution $Se = \{(0, 5, \tau_{11}), (5, 10, \tau_{21})\}$ est $w = \tau_{11}\tau_{21}$.
Dans la suite, nous formalisons le respect d'un scénario d'exécution suivant une politique de mise en œuvre flexible et inflexible. Autrement dit, nous précisons sous quelles conditions on peut dire qu'un scénario d'exécution effectif $Se_{effectif}$ (obtenu avec des durées effectives C_i, $C_i^{min} \leq C_i \leq C_i^{max}$) suit, selon une politique flexible ou inflexible, un scénario d'exécution $Se_{theorique}$ calculé hors-ligne (avec des durées théoriques C_i^{max}). Nous considérons pour cette formalisation que les fonctions s'exécutent entièrement.
Afin de simplifier la présentation nous utiliserons deux symboles différents :
– $Se_{effectif} \approx Se_{theorique}$: $Se_{effectif}$ suit de manière inflexible $Se_{theorique}$;
– $Se_{effectif} \sim Se_{theorique}$: $Se_{effectif}$ suit de manière flexible $Se_{theorique}$.
De plus, nous notons M' le mot topologique du scénario d'exécution effectif $Se_{effectif} = \{(start'_1, end'_1, \tau_{\alpha'(1)\beta'(1)}), ...\}$ et M le mot topologique du scénario d'exécution théorique $Se_{theorique} = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), ...\}$.

IV.2. Scénarios d'exécution non préemptif

Un scénario d'exécution théorique $Se_{theorique}$ d'un système de tâches périodiques τ est non préemptif si chaque bloc d'exécution prévoit l'exécution complète d'une instance de tâche :

$$\forall i \in [1, |Se_{theorique}|] \; end_i - start_i = C^{max}_{\alpha(i)}$$

Notons dans ce cas qu'il y a exactement un bloc par instance de tâche.

IV.2.a. Scénario inflexible

Dans un contexte d'ordonnancement non préemptif, les instances de tâches s'exécutent en un trait. Pour déterminer si un scénario d'exécution effectif suit de manière inflexible un scénario d'exécution théorique, il faut vérifier que toutes les instances prévues sont exécutées et qu'aucune instance ne s'exécute avant la date prévue.
Plus formellement nous avons :

$$Se_{effectif} \approx Se_{theorique} \Leftrightarrow \begin{cases} M = M' & (i) \\ \forall i \in [1, |M|] \quad start'_i = start_i & (ii) \end{cases} \tag{2.6}$$

Soit un système de deux tâches $\tau = \{\tau_1 =< 0, [1:2], 4, 4 | f_1 >, \tau_2 =< 0, [1:3], 6, 6 | f_2 >\}$. Considérons le scénario d'exécution théorique (figure 2.3-A) $Se_{theorique} = \{(0, 2, \tau_{11}), (2, 5, \tau_{21}), (5, 7, \tau_{12}), (7, 10, \tau_{22}), (10, 12, \tau_{13})\}$. L'hyper-période est $H = ppcm(4, 6) = 12$. Le scénario d'exécution ne prévoit aucun temps creux et est cyclique sur $[0, 12)$. La figure 2.3-B présente un scénario d'exécution effectif pour les durées d'exécution $C_{11} = 2$, $C_{21} = 2$, $C_{12} = 2$, $C_{22} = 2$, $C_{13} = 1$: $Se_{effectif} = \{(0, 2, \tau_{11}), (2, 4, \tau_{21}), (5, 7, \tau_{12}), (7, 9, \tau_{22}), (10, 11, \tau_{13})\}$.
On constate que les mots topologiques des deux scénarios sont égaux : $M = M' = \tau_{11}\tau_{21}\tau_{12}\tau_{22}\tau_{13}$. De plus toutes les instances commencent aux dates prévues par le scénario théorique. On remarquera par exemple que l'instance τ_{12} bien qu'étant prête à $t = 4$ ne commence son exécution qu'à $t = 5$.

IV.2.b. Scénario flexible

Un scénario d'exécution est flexible si certaines instances de tâches démarrent plus tôt que prévu. Une instance de tâche peut s'exécuter dès lors qu'elle est active et que l'instance précédente a terminé son exécution. Plus formellement :

$$Se_{effectif} \sim Se_{theorique} \Leftrightarrow \begin{cases} M = M' & (i) \\ start_1 = start'_1 & (ii) \\ \forall i \in [2, |M|] \qquad Max(end'_i, r_{\alpha(i)\beta(i)}) \leq start'_i \leq start_i & (iii) \end{cases} \tag{2.7}$$

La figure 2.3-B présente un scénario d'exécution flexible. On constate que les temps creux sont regroupés en fin de séquences alors que dans le cas inflexible (figure 2.3-A) il y avait deux temps creux de durée 1. Une tâche apériodique de durée 3 serait préemptée 2 fois dans la mise en œuvre inflexible et ne le serait pas dans la mise en œuvre flexible.

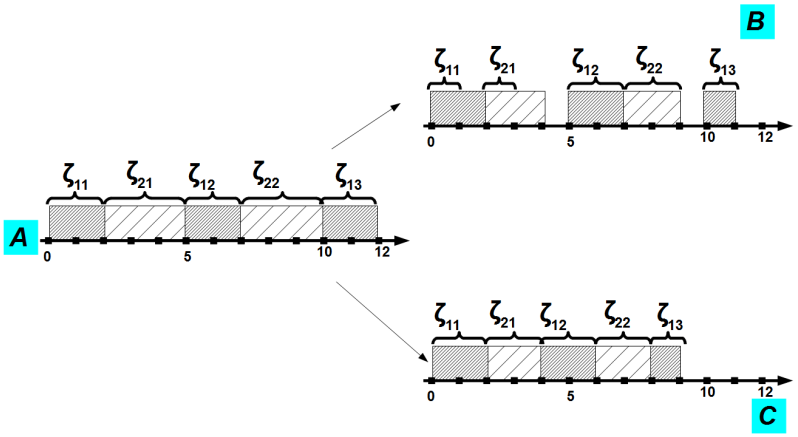


FIGURE 2.3 – Scénario d'exécution théorique (A) et politiques de mise en œuvre flexible (C) et inflexible (B)

IV.3. Scénarios d'exécution préemptif

Nous supposons maintenant que l'exécution de certaines instances peut être décomposée.

IV.3.a. Scénario inflexible

De manière intuitive, le scénario d'exécution effectif suit le scénario d'exécution théorique de manière inflexible si :

- les instances de tâches s'exécutent dans l'ordre prévu : on dit alors que le scénario effectif respecte la topologie du scénario théorique ;
- lorsqu'une instance a une durée d'exécution effective plus courte que la durée maximale, ce sont les derniers blocs qui lui ont été affectés qui sont impactés : un ou plusieurs blocs peuvent disparaitre, et le dernier bloc effectif peut être plus court que prévu.
  Soit la tâche $\tau_1 =< 0, [1:5], 10, 10 | f_1 >$ et le scénario d'exécution théorique $Se_{theorique} = \{(0,1,\tau_{11}), (2,4,\tau_{11}), (5,6,\tau_{11}), (7,8,\tau_{11})\}$. Considérons des scénarios d'exécution effectifs lorsque la durée d'exécution est 3. Un scénario effectif $Se_1 = \{(2,4,\tau_{11}), (5,6,\tau_{11})\}$ ne suit pas le scénario théorique parce que ce ne sont pas les derniers blocs qui disparaissent. Par contre un scénario $Se_2 = \{(0,1,\tau_{11}), (2,4,\tau_{11})\}$ suit le scénario théorique.
- les blocs restants démarrent aux dates spécifiées par le scénario théorique. Considérons à nouveau notre système de tâches $\tau = \{\tau_1 =< 0, [1:2], 4, 4 | f_1 >, \tau_2 =< 0, [1:3], 6, 6 | f_2 >\}$ avec le scénario d'exécution théorique $Se_{theorique} = \{(0,2,\tau_{11}), (2,4,\tau_{21}), (4,5,\tau_{12}), (5,6,\tau_{21}), (6,7,\tau_{12}), (7,10,\tau_{22}), (10,12,\tau_{13})\}$. Le mot topologique du scénario d'exécution est $M = \tau_{11}\tau_{21}\tau_{12}\tau_{21}\tau_{12}\tau_{22}\tau_{13}$. Considérons un scénario effectif tel que $C_{11} = 1$, $C_{12} = 1$, $C_{13} = 2$, $C_{21} = 2$, $C_{22} = 2$. Le scénario d'exécution effectif (voir figure 2.4) devrait dans le cas in-

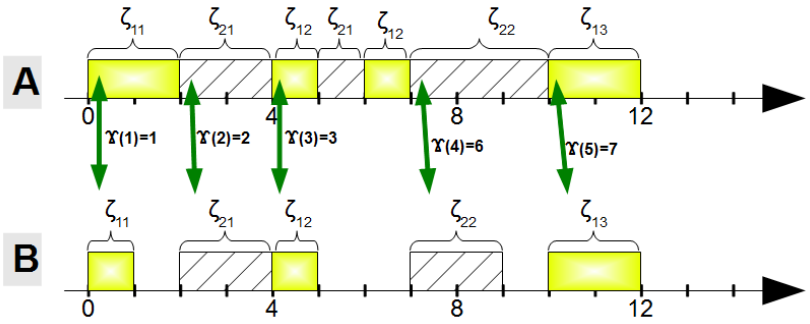


FIGURE 2.4 – Scénario d'exécution théorique avec préemption (A) et politiques de mise en œuvre inflexible (B)

flexible être $Se_{effectif} = \{(0,1,\tau_{11}),(2,4,\tau_{21}),(4,5,\tau_{12}),(7,9,\tau_{22}),(10,12,\tau_{13})\}$ avec comme mot topologique $M' = \tau_{11}\tau_{21}\tau_{12}\tau_{22},\tau_{13}$. Dans la suite nous formalisons le respect de la topologie.

Nous avons, à travers l'exemple précédent, vu que l'exécution peut entrainer la disparition de certains blocs. Nous avons donc :

$$\begin{cases} \forall i[1,|M|] \quad |M'|_{m_i} \geq 1 \\ |M'| \leq |M| \\ M' \subseteq M \end{cases} \tag{2.8}$$

Nous construisons une fonction γ qui relie chaque bloc effectif à un bloc théorique en indiquant la position d'une lettre de M' dans M :

$$\begin{cases} \gamma(1) & = & inf\{j / m'_1 = m_j\} \\ \gamma(p) & = & inf\{j > \gamma(p-1) / m'_p = m_j\} \quad p = 2,\ldots,|M'| \end{cases} \tag{2.9}$$

Dans notre exemple, $|M'| = 5$, $\gamma(1) = 1$, $\gamma(2) = 2$, $\gamma(3) = 3$, $\gamma(4) = 6$, $\gamma(5) = 7$. Les blocs 4 et 5 du scénario théorique ont été effacés.

Vérifier que les blocs effacés sont à la fin revient à vérifier que :

$$\forall (j,p) / j \notin \{\gamma(1)\ldots\gamma(|M'|)\}, \gamma(p) < j < \gamma(p+1) \Rightarrow \quad |m'_{p+1}\ldots m'_{|M'|}|_{m_j} \quad = \quad 0 \tag{2.10}$$

De plus, si un bloc est, dans le scénario d'exécution effectif, plus court que dans le scénario d'exécution théorique (l'instance a terminé plus tôt que prévu), alors il doit être le dernier bloc d'exécution de l'instance de tâche.

$$\forall j / j \in \{1\ldots|M'|\}, end'_j - start'_j < end_{\gamma(j)} - start_{\gamma(j)} \Rightarrow \quad |m'_{j+1}\ldots m'_{|M'|}|_{m_{\gamma(j)}} \quad = \quad 0 \tag{2.11}$$

La dernière condition est le respect inflexible des dates d'occurence des blocs non effacés :

$$\forall j \in \{1 \dots |M'|\} \Rightarrow \quad start'_j = start_{\gamma(j)} \tag{2.12}$$

Pour finir, un scénario d'exécution théorique suit de manière inflexible un scénario d'exécution théorique si et seulement si l'équation 2.13 est respectée.

$$Se_{effectif} \approx Se_{theorique} \Leftrightarrow \begin{cases} 2.8 \\ 2.10 \\ 2.11 \\ 2.12 \end{cases} \tag{2.13}$$

IV.3.b. Scénario flexible

Dans le cas des scénarios flexibles pour des systèmes préemptifs, nous devons toujours vérifier le respect de la topologie, donc les conditions 2.8, 2.10 restent valides.
Par contre, quand un bloc termine plus tôt, le bloc suivant peut éventuellement être avancé. Et de ce fait, si un bloc intermédiaire disparait, on peut avoir fusion de deux blocs.

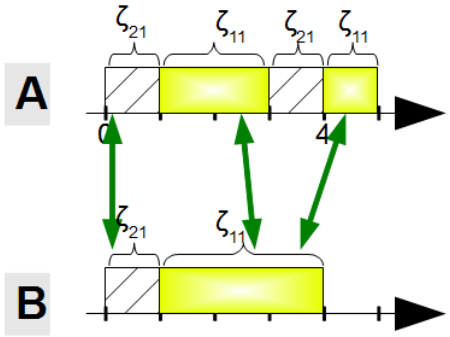


FIGURE 2.5 – Scénario d'exécution théorique avec préemption (A) et politique de mise en œuvre flexible (B)

Considérons l'exemple d'un système de deux tâches périodiques $\tau = \{\tau_1 =< 0, [1:3], 5, 5|f_1 >, \tau_2 =< 0, [1:2], 5, 5|f_2 >\}$ dont le scénario d'exécution théorique est $Se_{theorique} = \{(0, 1, \tau_{21}), (1, 3, \tau_{11}), (3, 4, \tau_{21}), (4, 5, \tau_{11})\}$. Lors de l'exécution si les durées d'exécution sont $C_{21} = 1$, $C_{11} = 3$, un scénario d'exécution effectif pourrait être $Se_{effectif} = \{(0, 1, \tau_{21}), (1, 4, \tau_{11})\}$. Ce scénario effectif (voir figure 2.5) montre la fusion des blocs d'exécution de τ_{11} due au fait que τ_{21} a fini son exécution à $t = 1$. Cette fusion peut éventuellement concerner plusieurs blocs.
Considérons donc les deux scénarios d'exécution $Se_{theorique}$ et $Se_{effectif}$. Pour qu'un bloc d'exécution de $Se_{theorique}$ n'apparaisse pas à l'identique dans $Se_{effectif}$ il y a quatre possibilités. Soit :

– le bloc d'exécution a été effacé ;
– le bloc d'exécution démarre plus tôt ;
– le bloc d'exécution est plus court ;
– le bloc d'exécution est plus long.

Nous adoptons une démarche récursive qui consiste à faire des vérifications pour chaque bloc d'exécution. La vérification de conformité se fait à partir d'une date t qui est initialisée à 0 et est incrémentée récursivement jusqu'à H. Pour simplifier la présentation nous adoptons la notation $Se(5 \ldots 10)$ pour indiquer que la comparaison ne concerne plus que les blocs d'exécution allant du cinquième au dixième. De plus, nous appelons $P(n, n', t)$ la propriété qui signifie que le scénario effectif $Se_{effectif}(n', |M'|)$ suit de manière flexible le scénario théorique $Se_{theorique}(n, |M|)$ à partir de la date t.

Lorsqu'on compare un scénario d'exécution effectif et un scénario d'exécution théorique il y a trois possibilités :

1. $m'_{n'} \neq m_n$: la première lettre du scénario effectif est différente de la première lettre du scénario théorique. Cela signifie que le premier bloc du scénario d'exécution théorique a été effacé. Il faut donc vérifier que la lettre effacée n'apparait plus. Il faut aussi vérifier qu'à partir de t, en restreignant le scénario théorique aux blocs suivants, le scénario effectif suit toujours le scénario théorique. Plus formellement nous avons l'équation 2.14.

$$\left.\begin{array}{l} m'_{n'} \neq m_n \\ |m'_{n'} \ldots m'_{|M'|}|_{m_n} = 0 \\ P(n+1, n', t) \end{array}\right\} \Rightarrow P(n, n', t) \tag{2.14}$$

2. $m'_{n'} = m_n$: la première lettre du scénario effectif est identique à la première lettre du scénario théorique. Il faut donc vérifier dans un premier temps que le premier bloc ne s'exécute pas plus tard que prévu :

$$\left\{ \; Max(t, r_{\alpha(n)\beta(n)}) \;\; \leq start'_{n'} \;\; \leq start_n \right. \tag{2.15}$$

Ensuite en fonction de la longueur du bloc d'exécution il y a d'autres vérifications à faire.

(a) Si les blocs d'exécution théorique et effectif sont de même longueur, il faut continuer la vérification en restreignant les scénarios d'exécution théorique et effectif aux blocs d'exécution suivants. Plus formellement nous avons l'équation 2.16.

$$\left.\begin{array}{l} m'_{n'} = m_n \\ Max(t, r_{\alpha(n)\beta(n)}) \leq start'_{n'} \leq start_n \\ end_n - start_n = end'_{n'} - start'_{n'} \\ P(n+1, n'+1, end'_{n'}) \end{array}\right\} \Rightarrow P(n, n', t) \tag{2.16}$$

(b) Si le bloc d'exécution effectif est plus petit que le bloc théorique, cela signifie que l'instance du bloc termine son exécution plus tôt. Il faut donc que l'instance n'apparaisse plus dans la suite du scénario effectif. Il faut aussi continuer la vérification

en restreignant les scénarios d'exécutions théorique et effectif aux blocs d'exécution suivants. Plus formellement nous avons l'équation 2.17

$$\left.\begin{array}{l} m'_{n'} = m_n \\ Max(t, r_{\alpha(n)\beta(n)}) \leq start'_{n'} \leq start_n \\ end'_{n'} - start'_{n'} < end_n - start_n \\ |m'_{n'+1} \ldots m'_{|M'|}|_{m_n} = 0 \\ P(n+1, n'+1, end'_{n'}) \end{array}\right\} \Rightarrow P(n, n', t) \qquad (2.17)$$

(c) Si le bloc d'exécution théorique est plus petit que le bloc effectif, cela signifie que certains blocs du scénario théorique ont été effacés et qu'il y a eu des fusions. Il faut donc retrouver les blocs qui ont été effacés et vérifier qu'ils n'apparaissent plus.

Pour retrouver les blocs effacés, il faut repérer dans un premier temps tous les blocs d'exécution de l'instance qui ont été fusionnés dans le scénario d'exécution théorique (équation 2.18(d.i) et vérifier que nous les avons tous repérés (équation 2.18(d.iii) : il y en assez et 2.18(d.iv) : on ne peut pas en prendre moins). Il faut aussi vérifier que les blocs intermédiaires du scénario d'exécution théorique sont tous effacés (équation 2.18(d.ii)). Il faut également vérifier que si le dernier bloc n'est pas pris dans son intégralité (l'instance s'est terminée) alors on ne la retrouve plus (équation 2.18(e)). Il faut aussi continuer la vérification en restreignant les scénarios d'exécution théorique et effectif aux blocs d'exécution suivants (équation 2.18(f)).

$$\left.\begin{array}{ll} (a) & m'_{n'} = m_n \\ (b) & Max(t, r_{\alpha(n)\beta(n)}) \leq start'_{n'} \leq start_n \\ (c) & end'_{n'} - start'_{n'} > end_n - start_n \\ (d) & \exists q_0 = n, q_1, \ldots, q_s (s \geq 1) : \\ & \quad (i) \quad \alpha(q_i)\beta(q_i) = \alpha(n)\beta(n) \\ & \quad (ii) \quad \forall j \not= q_i < j < q_{i+1} (i = 0 \ldots s-1) \\ & \qquad \Rightarrow |m'_{n'} \ldots m'_{|M'|}|_{m_j} = 0 \\ & \quad (iii) \quad end'_{n'} - start'_{n'} \leq \sum_{i=0}^{s} end_{q_i} - start_{q_i} \\ & \quad (iv) \quad \sum_{i=0}^{s-1} end_{q_i} - start_{q_i} < end'_{n'} - start'_n \\ (e) & end'_{n'} - start'_{n'} < \sum_{i=0}^{s} end_{q_i} - start_{q_i} \\ & \Rightarrow |m'_{n'+1} \ldots m'_{|M'|}|_{m_n} = 0 \\ (f) & P(q_s + 1, n' + 1, end'_{n'}) \end{array}\right\} \Rightarrow P(n, n', t) \qquad (2.18)$$

3. le mot topologique effectif est vide, ce qui signifie qu'il n'y a plus de bloc effectif à comparer et qu'ils ont été éffacés. Plus formellement :

$$n' = |M'| + 1 \quad \Rightarrow P(n, n', t) \qquad (2.19)$$

En définitif, un scénario d'exécution effectif suit de manière flexible un scénario d'exécution théorique (équation 2.20) si et seulement si $P(1, 1, 0)$ est vrai et que les équations 2.8 et 2.10

sont respectées.

$$Se_{effectif} \sim Se_{theorique} \Leftrightarrow \begin{cases} 2.8 \\ 2.10 \\ P(1,1,0) \end{cases} \tag{2.20}$$

Considérons à nouveau notre système de deux tâches $\tau = \{\tau_1 =< 0, [1:2], 4, 4 | f_1 >, \tau_2 =< 0, [1:3], 6, 6 | f_2 >\}$ dont les scénarios théorique et effectif sont $Se_{theorique} = \{(0, 2, \tau_{11}), (2, 4, \tau_{21}), (4, 5, \tau_{12}), (5, 6, \tau_{21}), (6, 7, \tau_{12}), (7, 10, \tau_{22}), (10, 12, \tau_{13})\}$ et $Se_{effectif} = \{(0, 2, \tau_{11}), (2, 4, \tau_{21}), (4, 5, \tau_{12}), (6, 8, \tau_{22}), (8, 9, \tau_{13})\}$.
Le tableau 2.1 présente les différentes étapes qui permettent de conclure que le scénario d'exécution effectif suit de manière flexible le scénario théorique.

IV.4. Récapitulatif

IV.4.a. Validité de mises en œuvre

Dans cette section nous formalisons à travers les propositions 2.7 et 2.8 le fait que les politiques flexible et inflexible garantissent la validité des mises en œuvre. Nous ne considérons dans cette section que des tâches indépendantes. Le cas des systèmes de tâches en présences de ressources critiques est traité dans le chapitre 3, section III.3.b.
Pour des systèmes de tâches indépendantes, montrer qu'une mise en œuvre est valide revient à montrer que les scénarios d'exécution effectifs respectent les contraintes temporelles (équation 2.21) :

- toutes les instances prévues par le scénario d'exécution théorique sont effectivement exécutées (équation 2.21-i) ;
- le début d'exécution effectif de chaque instance est plus grand que la date d'activation (équation 2.21-ii) ;
- la fin d'exécution effective de chaque instance intervient avant l'échéance (équation 2.21-iii).

$$\begin{cases} \forall i \in [1, |Se_{theorique}|] \quad \exists j \in [1, |Se_{effectif}|] \quad (\alpha'(j), \beta'(j)) = (\alpha(i), \beta(i)) & (i) \\ r_{\alpha'(j)\beta'(j)} \leq start'_j & (ii) \\ d_{\alpha'(j)\beta'(j)} \geq end'_j & (iii) \end{cases} \tag{2.21}$$

Proposition 2.7 (Mise en œuvre non préemptive)**.** *Dans un contexte non préemptif, avec des systèmes de tâches indépendantes et sans erreur de calibrage, une mise en œuvre inflexible (respectivement flexible), d'un scénario d'exécution théorique valide, est valide.*

Preuve Le point 2.21-(i) vient de l'équation 2.6-(i) (respectivement de l'équation 2.7-(i)) qui impose une égalité des mots topologiques : $M' = M$. Le point 2.21-(ii) vient de l'équation 2.6-(ii) (respectivement des équations 2.7-(ii) et 2.7-(iii)) qui garantit la préservation des dates de début d'exécution et du fait que le scénario théorique est valide (équation 2.1). Le point 2.21-(iii) vient

Système de tâches Scénarios d'exécution Alphabet Mots topologiques	$\tau = \{\tau_1 =< 0, [1:2], 4, 4 \mid f_1 >, \tau_2 =< 0, [1:3], 6, 6 \mid f_2 >\}$ $Se_{theorique} = \{(0, 2, \tau_{11}), (2, 4, \tau_{21}), (4, 5, \tau_{12}), (5, 6, \tau_{21}), (6, 7, \tau_{12}), (7, 10, \tau_{22}), (10, 12, \tau_{13})\}$ $Se_{effectif} = \{(0, 2, \tau_{11}), (2, 4, \tau_{21}), (4, 5, \tau_{12}), (6, 8, \tau_{22}), (8, 9, \tau_{13})\}$ $A = \{\tau_{11}, \tau_{21}, \tau_{12}, \tau_{22}, \tau_{13}\}$ $M = \tau_{11}\tau_{21}\tau_{12}\tau_{21}\tau_{12}\tau_{22}\tau_{13}$ $M' = \tau_{11}\tau_{21}\tau_{12}\tau_{22}\tau_{13}$
Condition 2.8	$\lvert M\rvert = 7 \leq \lvert M'\rvert = 5$ $\tau_{11}\tau_{21}\tau_{12}\tau_{22}\tau_{13} \subseteq \tau_{11}\tau_{21}\tau_{12}\tau_{21}\tau_{12}\tau_{22}\tau_{13}$ $\lvert M'\rvert_{\tau_{11}} \geq 1$, $\lvert M'\rvert_{\tau_{21}} \geq 1$, $\lvert M'\rvert_{\tau_{12}} \geq 1$, $\lvert M'\rvert_{\tau_{22}} \geq 1$, $\lvert M'\rvert_{\tau_{13}} \geq 1$
Positions lettres Lettres effacée	$\gamma(1) = 1$, $\gamma(2) = 2$, $\gamma(3) = 3$, $\gamma(4) = 6$, $\gamma(5) = 7$ $j = \{4, 5\}$
Calcul de $P(1,1,0)$	$j = 4, \lvert\tau_{22}\tau_{13}\rvert_{\tau_{21}} = 0$ $j = 5, \lvert\tau_{22}\tau_{13}\rvert_{\tau_{12}} = 0$ $m'_1 = m_1 = \tau_{11}$ $end_1 - start_1 = end'_1 - start'_1 = 2 - 0 = 2$ $P(2,2,2)$ à calculer
Calcul de $P(2,2,2)$	$m'_2 = m_2 = \tau_{21}$ $end_1 - start_1 = end'_1 - start'_1 = 4 - 2 = 2$ $P(3,3,4)$ à calculer
Calcul de $P(3,3,4)$	$m'_3 = m_3 = \tau_{12}$ $end_1 - start_1 = end'_1 - start'_1 = 5 - 4 = 1$ $P(4,4,6)$ à calculer
Calcul de $P(4,4,6)$	$m'_4 = \tau_{22} \neq m_4 = \tau_{21}$ $P(5,4,6)$ à calculer
Calcul de $P(5,4,6)$	$m'_4 = \tau_{22} \neq m_5 = \tau_{12}$ $(6,4,6)$ à calculer
Calcul de $P(6,4,6)$	$m'_4 = m_6 = \tau_{22}$ $end_6 - start_6 = 10 - 7 = 3 > end'_4 - start'_4 = 8 - 6 = 2$ $P(7,5,8)$ à calculer
Calcul de $P(7,5,8)$	$m'_5 = m_7 = \tau_{13}$ $end_7 - start_7 = 12 - 10 = 2 > end'_5 - start'_5 = 10 - 9 = 1$ $P(6,6,9)$ à calculer
Calcul de $P(6,6,9)$	Il n'y a plus de lettre à comparer on a donc : $P(1,1,0)$ est vrai $Se_{effectif} \sim Se_{theorique}$

TABLE 2.1 – Analyse d'un scénario d'exécution effectif du système de tâches $\tau = \{\tau_1 =< 0, [1:2], 4, 4 \mid f_1 >, \tau_2 =< 0, [1:3], 6, 6 \mid f_2 >\}$

de l'absence d'erreur de calibrage : les blocs d'exécution effectifs ont des durées inférieures où égales aux blocs d'exécution théoriques ; les dates de fin des blocs effectifs sont donc inférieures ou égales aux dates théoriques. Comme le scénario théorique respecte les échéances, le scénario effectif les respectent également.

Proposition 2.8 (Mise en œuvre préemptive)**.** *Dans un contexte préemptif, avec des systèmes de tâches indépendantes et sans erreur de calibrage, une mise en œuvre inflexible (respectivement flexible), d'un scénario d'exécution théorique valide, est valide.*

Preuve Le point 2.21-(i) vient directement de l'équation 2.8. Le point 2.21-(ii) vient de l'équation 2.12 (respectivement équation 2.15) : les dates de début des blocs d'exécution non effacés sont conservés ou avancés par la mise en œuvre ; le scénario théorique étant valide, la mise en œuvre respecte la date d'activation. Le point 2.21-(iii), tout comme dans le cas non préemptif, vient de l'absence d'erreur de calibrage et de la validité du scénario théorique : les durées d'exécution effectives sont plus petites que les durées théoriques ; les dates de fin des blocs d'exécution effectifs sont inférieures ou égales aux dates de fin théoriques ; comme les dates de fin sont inférieures ou égales aux échéances on peut en déduire que la mise en œuvre respecte les échéances.

IV.4.b. Liens entre politiques de mise en œuvre

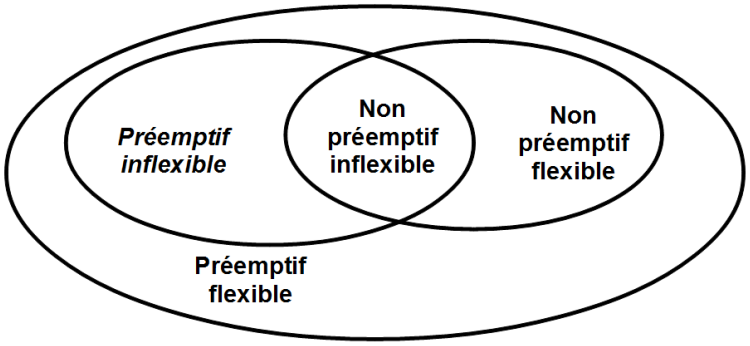


FIGURE 2.6 – Relations entre scénarios flexibles et inflexibles

La figure 2.6 récapitule les relations entre scénarios inflexibles et flexibles dans les contextes non préemptif et préemptif. En effet lorsque l'équation 2.6 (respect inflexible en contexte non préemptif) est satisfaite, cela implique que l'équation 2.7 (respect flexible en contexte non préemptif) ainsi que les équations 2.8, 2.10, 2.11, 2.12 (respect inflexible en contexte préemptif) sont vérifiées. De même, si les équations 2.8, 2.10, 2.11, 2.12 (respect inflexible en contexte

préemptif) sont vérifiées alors l'équation 2.20 (respect flexible en contexte préemptif) est également vérifiée. Pour finir l'équation 2.7 (respect flexible en contexte non préemptif) implique l'équations 2.20 (respect flexible en contexte préemptif).

V. Démarche

La section IV nous a permis de définir les notions de mise en œuvre flexible et inflexible. Dans cette section nous présentons notre démarche.
Notre problème peut être réformulé par la question suivante : *Quelle est l'organisation opérationnelle d'une application temps réel qui s'exécute sur un système d'exploitation temps réel, connaissant une séquence produite par un algorithme d'ordonnancement hors-ligne et son organisation fonctionnelle ?*
Pour répondre à cette question, il faut mettre en place des techniques de mise en œuvre effective. En d'autres termes, il nous faut trouver comment combiner les instructions temporelles (section IV.2.c du chapitre 1) et le modèle fonctionnel pour forcer les processus implémentés à suivre la séquence d'ordonnancement de manière flexible ou inflexible.

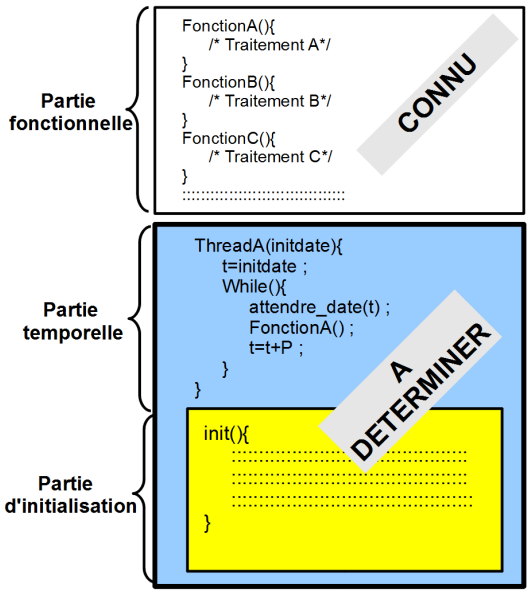


FIGURE 2.7 – Organisation générale de la mise en œuvre d'une séquence d'ordonnancement

Les techniques de mise en œuvre que nous présentons utilisent les possibilités suivantes offertes par les systèmes d'exploitation temps réel :
– gérer l'ordonnancement à travers les affectations de priorités ;
– synchroniser l'exécution des processus en utilisant des sémaphores ;

– gérer l'ordonnancement à travers des attentes de dates.

Certaines de nos mises en œuvre traduisent la séquence d'ordonnancement de plusieurs tâches en un ou deux processus. D'autres mises en œuvre traduisent la séquence d'ordonnancement par la création d'autant de processus qu'il y a de tâches périodiques.

La figure 2.7 présente l'organisation globale du code. Le point d'entrée de l'application temps réel est la fonction init(). Nous appelons cette partie du code *bloc initialisateur*. En plus des blocs initilisateur et fonctionnel, un troisième bloc appelé *bloc temporel* est utilisé pour imposer un comportement temporel donné.

En plus des hypothèses de base, notre démarche suppose que l'ajout d'instructions temporelles aux instructions fonctionnelles ne modifie pas les paramètres temporels et plus particulièrement les pires durées d'exécution. Cela n'implique pas que les instructions temporelles n'ont pas de coût temporel, mais plutôt que l'évalution des pires durées d'exécution les surestiment suffisamment.

Les techniques de mise en œuvre étant nombreuses, il est nécessaire de trouver un moyen d'évaluation et de comparaison. Pour cela, nous proposons un modèle de coûts (voir section VI).

Chaque technique proposée est évaluée par des fonctions de coût qui mesurent leurs complexités temporelle (voir section VI.1) et spatiale (voir section VI.2). Nous étudions également le comportement de l'application lorsque certains processus ont des durées d'exécution plus grandes que celles qui étaient prévues. Ceci est justifié par la conjecture 1.1.

Par souci méthodologique, nous abordons dans un premier temps les mises en œuvre dans un contexte de systèmes de tâches indépendantes (voir chapitre 3 sections I et II). Dans ce contexte, les seules contraintes sont les contraintes temporelles (date d'activation, échéances, période).

Dans un second temps, nous traitons des techniques de mise en œuvre pour des systèmes de tâches partageant des ressources critiques ou soumises à des relations de précédence (voir chapitre 3 section III). Dans ce contexte, en plus des contraintes temporelles, certaines tâches partagent des ressources critiques ou doivent respecter des contraintes de précédence.

Pour finir, nous traitons de l'utilisation de nos techniques sur un système d'exploitation temps réel qui est Xenomai (voir chapitre 4). Nous y avons choisi comme interface de programmation, l'interface POSIX parce qu'elle est l'une des plus utilisées en temps réel. Pour les mêmes raisons d'utilisation fréquente nous utilisons le langage C. La mise en œuvre effective sur un sytème d'exploitation pose une question supplémentaire : l'exécution de l'application suit-elle effectivement la séquence prévue ? Pour vérifier la conformité des mises en œuvre, nous utilisons un outil d'observation (voir section IV du chapitre 4). Nous traitons également comme cas pratique la mise en œuvre d'une application de gestion de mine (voir chapitre 5).

VI. Modèle de coûts des mises en œuvre

Pour évaluer les techniques de mise en œuvre nous proposons un modèle de coûts qui intègre des coût classiques de complexité temporelle (section VI.1) et spatiale (section VI.2) et d'autres

coûts spécifiques au pseudo-parallélisme d'exécution des processus (sections VI.3, VI.4).

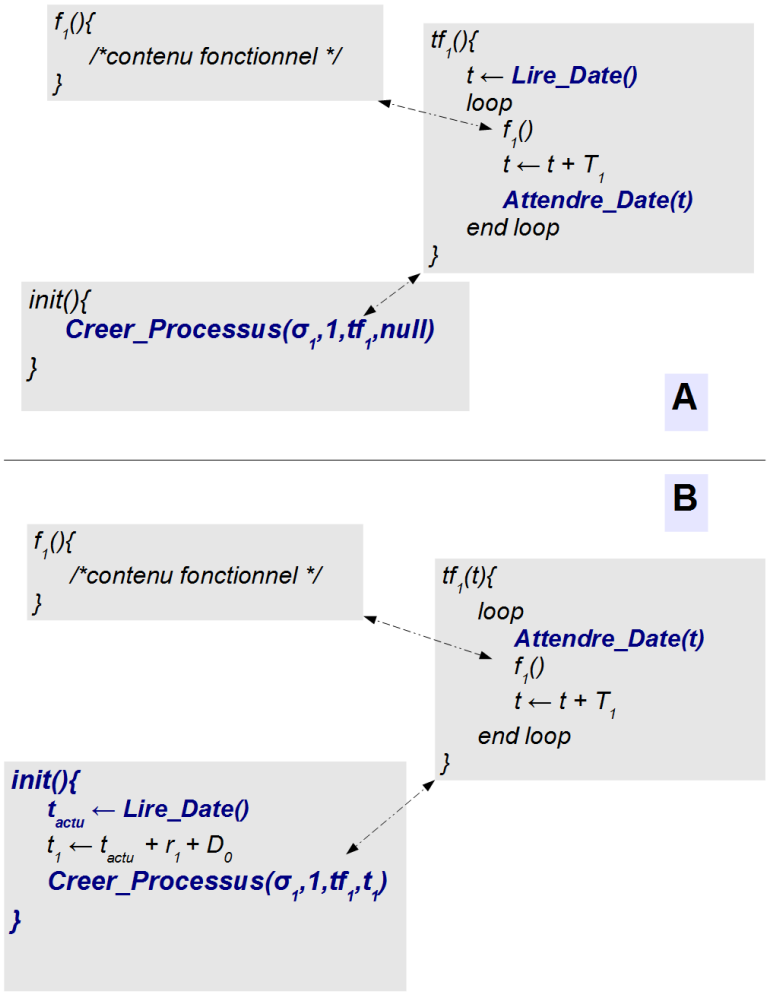


FIGURE 2.8 – Mise en œuvre d'une tâche périodique. A : mise en œuvre classique ; B : mise en œuvre avec attente de date d'activation

Pour illustrer à quoi correspond notre modèle de coût nous utilisons l'exemple de mise en œuvre d'une tâche $\tau_1 =< r_1, [C_1^{min} : C_1^{max}], D_1, T_1 | f_1 >$. L'exemple que nous utilisons est très proche de la mise en œuvre classique (voir figure 2.8-A). L'inconvénient de cette première implémentation est qu'elle ne permet pas de spécifier la première date d'activation.
Pour palier cet inconvénient, nous utilisons une deuxième mise en œuvre (voir figure 2.8-B) qui est une adaptation de la mise en œuvre classique. Cette mise en œuvre utilise la possibilité de passer des paramètres à la création d'un processus. À la création du processus une date t_1 lui

est passée en paramètre. t_1 prend en compte la date courante t_{actu} (obtenue par la primitive Lire_Date), une borne supérieure de la durée d'initialisation (D_0) et l'offset r_1 de la tâche.
Ainsi, nous pouvons spécifier une date d'activation relativement à une date de référence qu'est $t_1 + D_0$.
D_0 représente un temps d'attente nécessaire à l'application pour tout initialiser (création d'objets, affectation d'adresses, ...).
Un autre avantage de la mise en œuvre d'une tâche périodique avec passage d'une date d'activation en paramètre est que la même date peut être passée en paramètre à plusieurs processus et servir de date de référence commune.

VI.1. Coût temporel

Nous appelons coût temporel le temps processeur utilisé par le bloc d'initialisation et le bloc temporel. Au niveau algorithmique, nous utilisons le nombre d'opérations pour évaluer ce coût temporel. Nous nous intéressons plus particulièrement au nombre d'opérations par hyper-période parce que nous ne considérons que des scénarios d'exécution cycliques sur l'hyper-période.

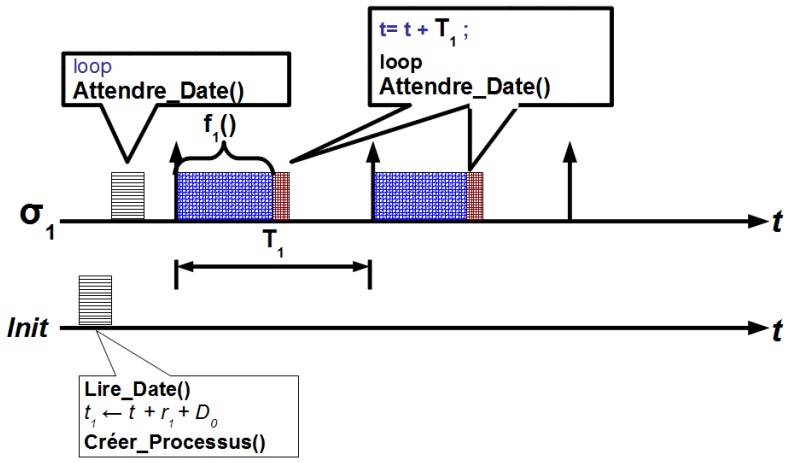


FIGURE 2.9 – Exécution et coût temporel de la mise en œuvre d'une tâche périodique

Pour l'exemple de la mise en œuvre de la tâche périodique, comme il y a une seule tâche, le coût temporel est évalué sur la période de la tâche. À l'exécution de l'application, le processus *init* détermine la date actuelle, calcule la date de référence et crée le processus σ_1 en passant en paramètre cette date. Le processus σ_1 après sa création par *init* entre dans la boucle *loop* et attend la date de référence avant d'exécuter f_1. Si l'estimation de la durée d'exécution est correcte, f_1 utilise au maximum C_1^{max} unités de temps puis la prochaine date à attendre est calculée.

	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date	1	1
Attendre_Date		
Créer_Processus	1	
Autres opérations		
Lecture mémoire	4	3
Écriture mémoire	2	1
Calcul	2	1

TABLE 2.2 – Coût temporel de mise en œuvre de la tâche $\tau_1 =< r_1, [C_1^{min} : C_1^{max}], D_1, T_1 | f_1 >$

Ce scénario d'exécution (voir figure 2.9) nous permet de déterminer le coût temporel de la mise en œuvre que nous résumons dans le tableau 2.2.

VI.2. Coût spatial

Nous appelons coût spatial d'une mise en œuvre, l'espace mémoire utilisé par le bloc d'initialisation et le bloc temporel. L'initialisation a utilisé deux variables de type date à savoir t_{actu} et t_1. De plus la mise en œuvre crée un processus (σ_1). Pour évaluer la mémoire utilisée par le code (ou segment de code), nous comptabilisons les lignes de code de l'algorithme. Ainsi la partie initialisation est écrite sur 3 lignes et la partie temporelle est écrite sur 5 lignes.

	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date	2	1
Processus	1	
Autre coût		
Nombre de lignes	3	5

TABLE 2.3 – Coût spatial de mise en œuvre de la tâche $\tau_1 =< r_1, [C_1^{min} : C_1^{max}], D_1, T_1 | f_1 >$

Le tableau 2.3 présente l'estimation du coût spatial de la mise en œuvre de la tâche τ_1.

VI.3. Niveaux de priorité

Les systèmes d'exploitation temps réel ont des niveaux de priorités limités [BM05]. Il nous faut donc évaluer le nombre de niveaux de priorités requis pour une mise en œuvre. Une mise en œuvre peu être implémentée sur un système d'exploitation si le nombre Pc de niveaux de

priorités de la mise en œuvre est inférieur ou égal au nombre maximal de niveaux de priorités du système d'exploitation temps réel.
Par exemple la mise en œuvre de la tâche périodique (figure 2.8-B) nécessite un seul niveau de priorité : $Pc = 1$.

VI.4. Changements de contexte

Lors d'un changement de contexte entre deux processus σ_i et σ_j, l'état du processeur (pile, registres, ...) correspondant à σ_i est sauvegardé et remplacé par le contexte d'exécution de σ_j. Les opérations de changement de contexte ont un coût non négligeable qu'il faut considérer. Pour évaluer nos mises en œuvre, nous définissons la fonction Rc qui recense le nombre de changements de contexte d'exécution entre processus pendant un cycle.
Pour l'exemple de la mise en œuvre de la tâche τ_1, il n'y a qu'un seul processus σ_1. Il n'y a donc pas de changement de contexte et $Rc = 0$.

VI.5. Robustesse

En plus des critères quantitatifs d'évaluation des mises en œuvre, nous analysons la robustesse. Nous nous intéressons plus particulièrement au comportement du système lorsqu'il y a des erreurs de calibrage. Cette exception est due à une mauvaise estimation des durées d'exécution qui reste possible selon la conjecture 1.1. Nous cherchons l'impact de cette exception sur l'exécution de l'application et sur les scénarios d'exécution effectifs :
- lorsqu'il y a une erreur de sous-calibrage, la tâche en débordement finit-elle son exécution : pour certaines mise en œuvre la tâche en débordement n'est pas interrompue, pour d'autres mises en œuvre la tâche en débordement est interrompue ;
- lorsqu'il y a une erreur de calibrage, le scénario d'exécution est-il bouleversé : pour certaines mises en œuvre le débordement d'une seule tâche peut avoir un effet cascade et même bouleverser totalement le scénario d'exécution, pour d'autres mises en œuvre il n'y a pas de bouleversement. De même le sur-calibrage a-t-il un impact sur le scénario d'exécution effectif.

Nous considérons trois types de robustesse :
- la *robustesse topologique* qui est la propriété de toujours respecter l'ordre d'exécution des instructions ; cette propriété permet de garantir le respect des contraintes d'exclusion mutuelle ou les contraintes de précédence ;
- la *robustesse fonctionnelle* qui est la propriété de ne pas tronquer l'exécution des fonctions ; cette propriété permet de garantir que tous les calculs seront effectués ;
- la *robustesse temporelle* qui est la propriété de ne pas faire empiéter l'exécution d'un bloc sur le bloc suivant ; cette propriété est importante lorsqu'il faut garantir que les calculs se font dans des tranches de temps données.

On dira ainsi qu'une mise en œuvre a une robustesse temporelle si le scénario d'exécution n'est pas perturbé par des erreurs de calibrage.

La mise en œuvre de la tâche τ_1 n'est pas robuste temporellement puisqu'en cas d'erreur de sous calibrage ($C_i > C_i^{max}$) elle n'est pas interrompue et un bloc d'exécution peut empiéter sur le bloc d'exécution suivant. Par contre cette mise en œuvre est robuste topologiquement et fonctionnellement puisque l'ordre d'exécution est respectée et l'exécution de la fonction f_1 n'est jamais tronquée.

VII. Conclusion

Ce deuxième chapitre nous a permis de présenter nos hypothèses et notre méthodologie de travail. Nous avons présenté deux politiques de mise en œuvre à savoir les politiques de mise en œuvre flexible et inflexible. Nous avons présenté les conditions à respecter par une séquence dans le cas flexible et dans le cas inflexible. Nous avons aussi présenté un modèle de coûts qui nous servira de support de comparaison entre techniques de mise en œuvre. Une mise en œuvre est caractérisée par des coûts quantitatifs (coûts temporel, spatial, de changement de contexte et niveaux de priorité) de mise en œuvre et par un coût qualitatif qu'est la robustesse. La suite du document nous permettra de détailler comment mettre en œuvre une séquence d'ordonnancement établie hors-ligne. Dans un premier temps nous présentons nos techniques de mise en œuvre dans un contexte de tâches indépendantes (chapitre 3 sections I et II). Ensuite nous étudions comment prendre en compte l'utilisation des ressources critiques et des relations de précédence (chapitre 3 section III). Enfin, nous présentons la traduction de nos techniques de mise en œuvre dans un contexte réel. Nous traitons en particulier de la mise en œuvre POSIX (chapitre 4).

Chapitre

3 Algorithmes de mise en œuvre

Sommaire

Résumé

Dans ce chapitre, nous présentons nos propositions de mise en œuvre pour des systèmes de tâches indépendantes. Nous proposons un algorithme pour chaque mise en œuvre d'ordonnancement hors-ligne. Nous évaluons chaque mise en œuvre en utilisant le modèle de coûts présenté dans le chapitre 2. Pour finir nous comparons ces différentes propositions et nous les étendons au contexte d'utilisation de ressources critiques ou en présence de contraintes de précédence.

Dans le chapitre 2, nous avons présenté nos hypothèses et notre méthodologie. Le présent chapitre a pour objectif de présenter des algorithmes de mise en œuvre d'ordonnancements hors-ligne. Nous présentons dans un premier temps des mises en œuvre de scénarios d'exécution théoriques pour des tâches indépendantes. Ainsi, pour les tâches indépendantes, nous traitons d'abord le cas des scénarios sans prémption (section I) puis le cas des scénarios avec préemptions (section II). Pour finir, nous traitons de la prise en compte des ressources critiques et des contraintes de précédence (section III).

I. Mise en œuvre d'ordonnancement sans préemption

Dans cette section nous abordons les mises en œuvre de scénarios d'exécution théoriques ne prévoyant aucune préemption de tâches. En d'autres termes, chaque tâche doit s'exécuter sur un bloc d'exécution. Le scénario d'exécution théorique $Se_{theorique} = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), (start_2, end_2, \tau_{\alpha(2)\beta(2)}), \ldots\}$ du système de tâches $\tau = \{\tau_1, \tau_2, \ldots\}$ ne prévoit pas de préemption de tâche si chaque symbole $m_i = \tau_{\alpha(i)\beta(i)}$ apparait une seule fois dans le mot topologique M du scénario d'exécution (équation 3.1).

$$\forall k / 1 \leq k \leq |Se| \Rightarrow \begin{cases} end_k - start_k = C^{max}_{\alpha(k)} \\ |M|_{m_k} = 1 \end{cases} \tag{3.1}$$

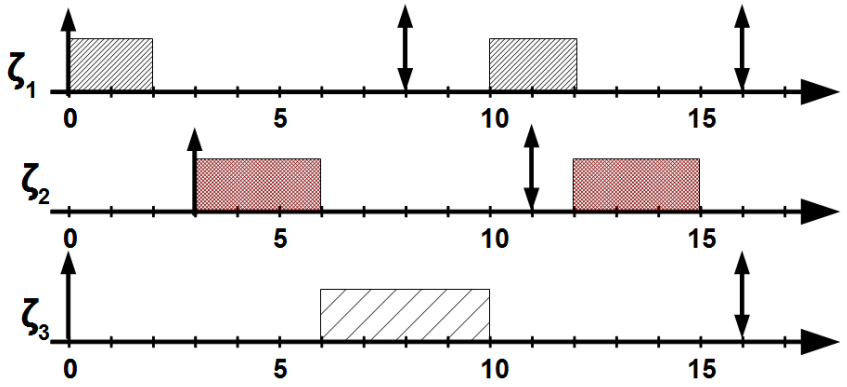


FIGURE 3.1 – Scénario d'exécution théorique du système de tâches $\tau = \{\tau_1 =< (0, [1:2], 8, 8|f_1 >, \tau_2 =< 3, [1:3], 5, 8|f_2 >, \tau_3 =< 0, [2:4], 16, 16|f_3 >\}$

Dans la suite, nous utilisons l'exemple du système de trois tâches du tableau 3.1, dont le scénario d'exécution théorique est représenté par la figure 3.1. L'hyper-période du système de tâches est $H = 16$. Entre $t = 0$ et $t = 16$ il y a exactement 2 temps creux ($H \times (1 - U) = 16 \times (1 - \frac{1}{4} - \frac{3}{8} - \frac{1}{4})) = 2$). Le théorème 1.3 (chapitre 1) nous permet de conclure que l'ordonnancement est cyclique et que le cycle commence à $t = 0$.

Description	exemple
Système de tâches	$\tau = \{\tau_1 =< (0, [1:2], 8, 8 \mid f_1 >,$ $\tau_2 =< 3, [1:3], 5, 8 \mid f_2 >,$ $\tau_3 =< 0, [2:4], 16, 16 \mid f_3 >\}$
Scénario d'exécution théorique	$Se = \{(0, 2, \tau_{11}), (3, 6, \tau_{21})$ $(6, 10, \tau_{31}), (10, 12, \tau_{12}), (12, 15, \tau_{22})\}$ $\lvert Se \rvert = 5\,;\ H = 16$

TABLE 3.1 – Exemple de description d'une application et d'une séquence d'ordonnancement hors-ligne

Pour mettre en œuvre un tel scénario d'exécution théorique il y a plusieurs possibilités.
L'une des techniques consiste à créer un (section I.1, section I.2) ou deux (section I.3) processus pour exécuter les fonctions modélisées. Une autre technique consiste à créer un processus pour chaque tâche. Il est alors possible de diriger l'exécution de ces processus par un processus supplémentaire appelée *répartiteur* (sections I.5, I.4) ou de faire coopérer les processus (sections I.6, I.7). Dans la suite nous décrivons chacune de ces techniques. Nous présentons également pour chaque technique les coûts liés à leur mise en œuvre. Ces coûts sont ensuite résumés dans des tableaux synoptiques. Par souci de clarté ces tableaux synoptiques sont présentés en annexe A.

I.1. Mise en œuvre d'une séquence par un processus

La première technique de mise en œuvre que nous proposons est très proche de la mise en œuvre proposée par [Xu03] à la différence que nous prenons en compte les instructions temporelles d'attente de dates.

I.1.a. Description

Cette technique de mise en œuvre consiste à créer un seul processus σ_1 qui intègre directement le scénario d'exécution théorique $Se_{theorique}$. Pour cela, pour chaque bloc d'exécution théorique $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$, une attente de date et une référence à la fonction $f_{\alpha(i)}$ sont insérées dans le code à exécuter par le processus. Il en résulte que le processus exécute les blocs d'instructions fonctionnelles au fur et à mesure en fonction du scénario d'exécution théorique $Se_{theorique}$. Autrement dit pour chaque bloc d'exécution $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$ le processus :
– attend la date de début d'exécution correspondant au bloc i ; cette date revient à ajouter $start_i$ à une date t_0 qui correspond à $start_1$;
– exécute les instructions fonctionnelles correspondantes ; cela revient à exécuter $f_{\alpha(i)}$.
Après l'exécution de la fonction du dernier bloc $(f_{\alpha(|Se_{theorique}|)})$, le processus doit mettre à jour la date t_0 (date correspondant à l'exécution du bloc 1). Comme le scénario d'exécution est

périodique avec une période égale à l'hyper-période H des tâches, mettre à jour t_0 revient à y ajouter H. Cette mise à jour cyclique de t_0 fait que σ_1 est un processus périodique de période H.
La figure 3.2 présente l'algorithme de mise en œuvre correspondant au système de tâches du tableau 3.1.

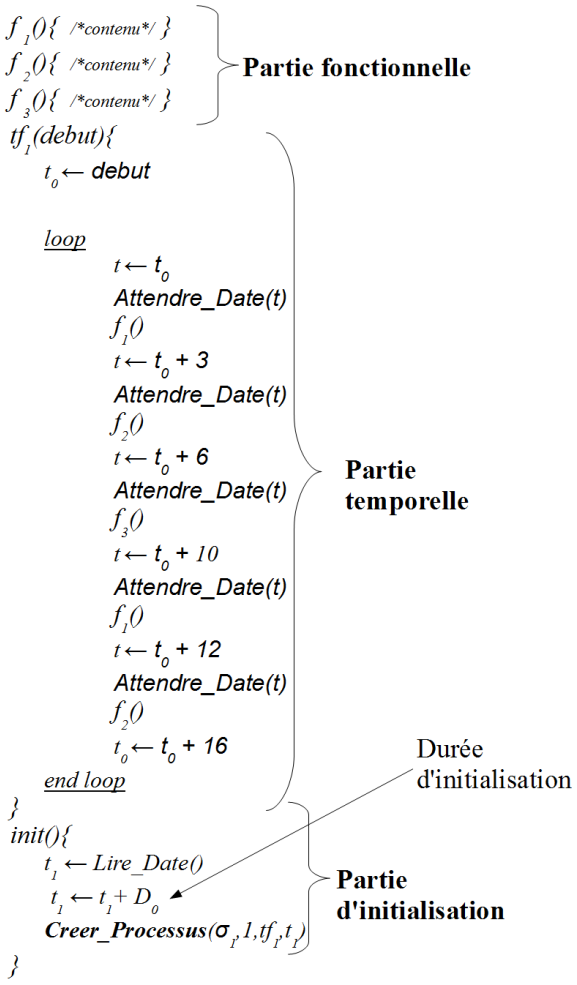


FIGURE 3.2 – Algorithme d'une mise en œuvre par un processus

La mise en œuvre se faisant par un seul processus (σ_1), le scénario d'exécution produit (figure 3.3) ne dépend que des états successifs de ce processus. Après sa création par le processus init, σ_1 passe à l'état endormi jusqu'à ce que la date du système soit égale à la date de référence passée en paramètre par init. Puis la fonction $f_{\alpha(1)}$ correspondant au premier bloc $(start_1, end_1, \tau_{\alpha(1)\beta(1)})$ du scénario d'exécution théorique est exécutée. S'il n'y a pas d'erreur de sous-calibrage, elle

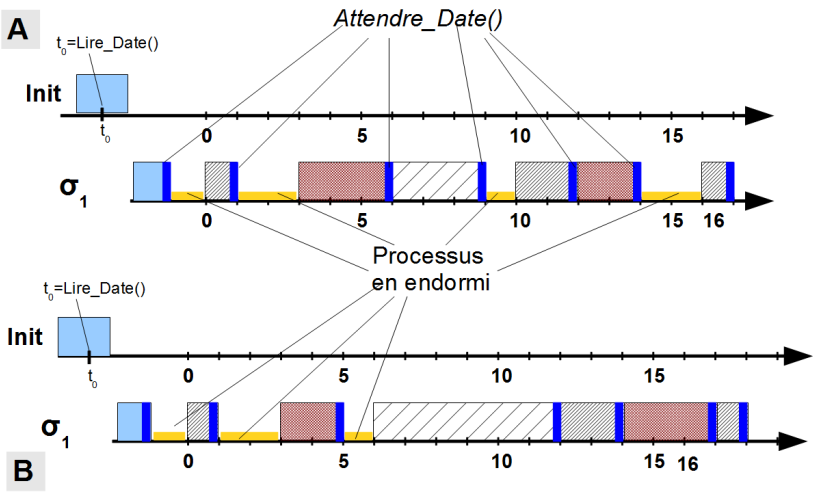


FIGURE 3.3 – Scénario d'exécution d'une mise en œuvre par un processus. A : sans erreur de calibrage ; B : avec une erreur de calibrage ($C_{31} = 6 > C_3^{max} = 4$)

finit son exécution au plus tard à la date de fin prévue (c'est-à-dire à $end'_1 \leq end_1$), puis se met en attente de $start_2$. Les actions d'attente puis d'exécution se répètent alors sans cesse. Le scénario d'exécution produit est inflexible puisqu'aucun bloc i ne commence son exécution avant $start_i$.

I.1.b. Coûts de mise en œuvre

Coût temporel Pour chaque bloc d'exécution $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$ de $Se_{theorique}$ la partie temporelle de l'application intègre une attente de date, un calcul de date et des accès mémoire en lecture (fonctions, dates) et en écriture (dates). On obtient un coût temporel proportionnel à $|Se_{theorique}|$ la dimension du scénario d'exécution.

Coût spatial La mise en œuvre nécessite, pour le bloc d'initialisation, une date et un processus. Le bloc temporel utilise une date mais nécessite un nombre de lignes de code proportionnel à $|Se_{theorique}|$.

Niveaux de priorité Un seul processus étant utilisé, un seul niveau de priorité est nécessaire quel que soit le nombre de tâches et le nombre d'entrées du scénario d'exécution prévu hors-ligne.

Coût des changements de contexte Il n'y a qu'un seul processus qui s'exécute. Le coût dû au changement de contexte est donc nul puisqu'il n'y a pas de changement de contexte entre

processus de l'application.

Robustesse Lorqu'il y a une erreur de sous-calibrage ($C_i > C_i^{max}$), la fonction en débordement n'est pas interrompue et cela conduit à avoir un bloc d'exécution d'une durée plus grande que le bloc d'exécution théorique. La figure 3.3-B présente le scénario d'exécution produit par notre application lorsqu'il y a une erreur de sous-calibrage de la tâche τ_3 avec $C_{31} = 6$. À la suite de l'exécution qui engendre cette erreur, les blocs d'exécution suivants sont décalés. Cette mise en œuvre est robuste topologiquement et fonctionnellement, puisque l'ordre d'exécution des tâches n'est jamais bouleversé. Par contre la mise en œuvre n'est pas robuste temporellement parce qu'un débordement empiète sur les blocs d'exécution proches et il peut avoir des effets cascades.
Le tableau A.1 (annexe A) résume les côuts d'une mise en œuvre à un seul processus.

I.2. Répartiteur non concurrent à un processus

Un inconvénient de la mise en œuvre précédente est que le code produit est de taille proportionnelle à la longueur du scénario d'exécution calculé hors-ligne. Par exemple un scénario d'exécution de longueur 1000 est mis en œuvre par une application de plus de 1000 lignes de code. De plus la mise en œuvre précédente est peu adaptative puisque tout changement de scénario d'exécution nécessite la reprise du code. Pour pallier cela, nous proposons la mise en œuvre par un répartiteur non concurrent.

I.2.a. Description

Dans cette deuxième mise en œuvre utilisant un processus σ_1, le scénario d'exécution théorique $Se_{theorique} = \{(start_1, end_1, \tau_{\alpha(1)\beta(1)}), (start_2, end_2, \tau_{\alpha(2)\beta(2)}), \ldots\}$ est enregistré dans un tableau $Table$. Afin d'enregistrer tous les blocs d'exécution, il nous faut un tableau de $|Se_{theorique}|$ lignes. Chaque ligne i du tableau contient une référence de la fonction $f_{\alpha(i)}$ à exécuter et la date $start_i$ de début d'exécution prévue. Pour suivre le scénario d'exécution théorique, le processus σ_1 utilise une boucle qui parcourt le tableau. À chaque itération de la boucle, la fonction à exécuter ainsi que la date correspondant au prochain bloc d'exécution sont déterminées. Il en résulte que l'ordre d'exécution des instructions fonctionnelles est dicté par le tableau $Table$. Pour cela, $Table$ est appelé *table d'ordonnancement*. Cette technique de mise en œuvre est plus modulaire que la première (section I.1) en ce sens que le bloc temporel et le bloc d'initialisation du code produit sont pratiquement identiques pour toutes les applications. La différence d'une application à une autre se trouvant dans la partie fonctionnelle et dans la table d'ordonnancement. La figure 3.4 présente un exemple de mise en œuvre du système de tâches du tableau 3.1.
Le scénario d'exécution produit est le même que pour la technique de mise en œuvre par un seul processus (voir figure 3.3). Le scénario d'exécution produit est également un scénario inflexible puisqu'aucun bloc i ne s'exécute avant la date $start_i$ prévue.

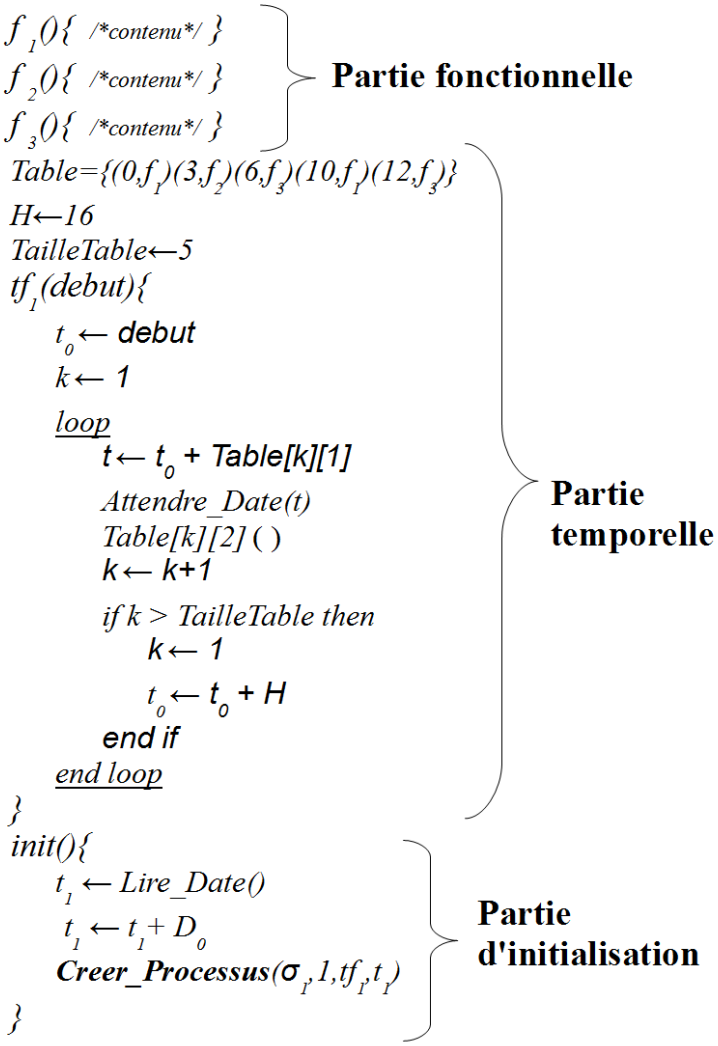


FIGURE 3.4 – Algorithme d'une mise en œuvre du répartiteur non concurrent à un processus

I.2.b. Coûts de mise en œuvre

Les coûts de mise en œuvre sont pratiquement les mêmes que ceux de la technique présentée en section I.1.b. La particularité est qu'il faut intégrer dans le coût temporel le temps de lecture du tableau d'ordonnancement. Il faut également prendre en compte dans le coût spatial le fait que le scénario d'exécution n'est plus intégré dans le code mais enregistré dans une variable tableau. Le tableau A.2 (annexe A) résume les coûts pour la technique de mise en œuvre de répartition non concurrente à un processus. Les robustesses sont les mêmes que pour la technique présentée en section I.1.b à savoir la technique est robuste topologiquement et fonctionnellement mais n'est pas robuste temporellement.

I.3. Répartiteur non concurrent à deux processus

Un inconvénient de la mise en œuvre de répartition à un processus est que lorsque la durée d'exécution d'une fonction est plus élevée que prévu (erreur de sous-calibrage), elle ne peut être arrêtée. Pour pallier cela, nous proposons la mise en œuvre de répartition à deux processus.

I.3.a. Description

La technique du répartiteur non concurrent à deux processus consiste à utiliser deux processus σ_1 et σ_2 et une table d'ordonnancement $Table$ de même structure que celle utilisée dans la technique de répartition non concurrente à un processus (voir section I.2). Le premier processus σ_1 a en charge de suivre le scénario d'exécution théorique. Pour chaque bloc d'exécution i, au lieu d'exécuter directement la fonction $f_{\alpha(i)}$, le processus σ_1 crée un processus qui y est dédié. Pour ne pas avoir à manipuler plusieurs processus, on utilise le même processus σ_2 pour tous les blocs d'exécution. σ_2 est appelé *processus fonctionnel* parce qu'il n'exécute que les instructions fonctionnelles et σ_1 est appelé *processus répartiteur* parce qu'il décide des instructions fonctionnelles à exécuter.
La mise œuvre se faisant par deux processus σ_1 et σ_2, le scénario d'exécution (voir figure 3.6) dépend des états successifs des deux processus. Le processus σ_2 étant le plus prioritaire, dès qu'il est prêt, il s'exécute. Ainsi, après sa création par init, σ_1 s'exécute et s'endort jusqu'à la date de référence qui lui a été passée en paramètre par init. Puis le processus σ_2 est créé pour exécuter la fonction $f_{\alpha(1)}$ du bloc $(start_1, end_1, \tau_{\alpha(1)\beta(1)})$ du scénario d'exécution. Par la suite σ_1 passe encore à l'état endormi jusqu'a $start_2$. S'il n'y a pas d'erreur de sous-calibrage, σ_2 termine son exécution avant end_1. La répétition des actions d'attente et de création de σ_2 par σ_1 produit alors un scénario d'exécution inflexible puisqu'aucun bloc i ne s'exécute avant $start_i$.
À la différence de la mise en œuvre présentée dans la section I.2, le processus répartiteur σ_1 a une priorité plus grande que σ_2 et peut empêcher les éventuels débordement de durées d'exécution. La figure 3.5 présente la mise en œuvre correspondant au sytème de tâches du tableau 3.1.

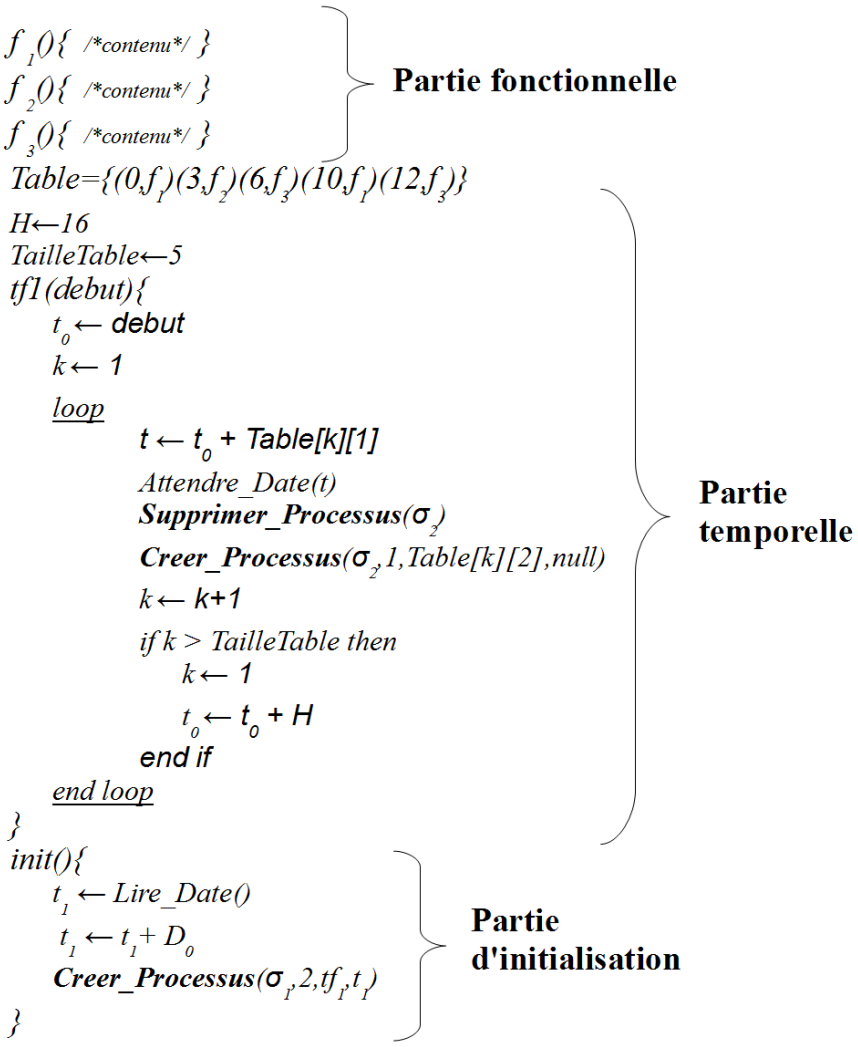


FIGURE 3.5 – Algorithme d'une mise en œuvre de répartition non concurrente à deux processus

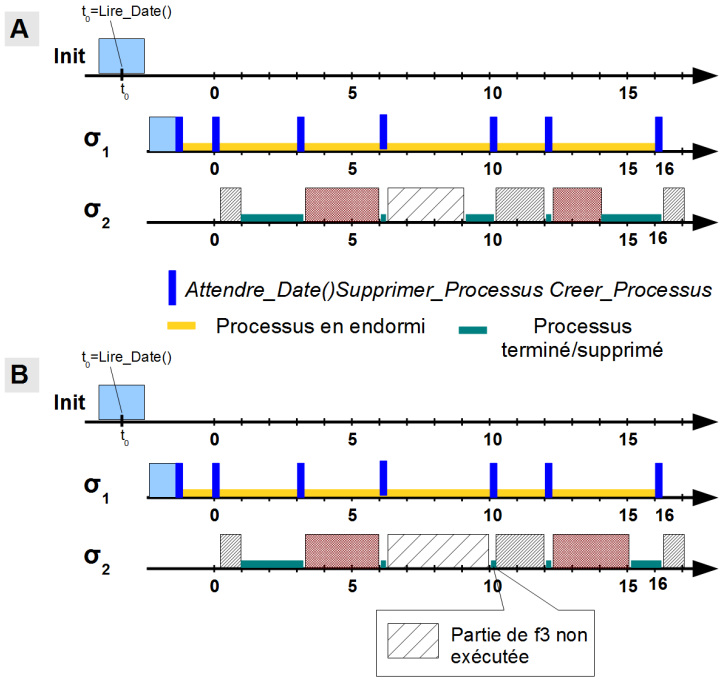


FIGURE 3.6 – Scénario d'exécution d'une mise en œuvre de répartition non concurrente à deux processus. A : sans erreur de calibrage ; B : avec une erreur de calibrage ($C_{31} = 6 > C_3^{max} = 4$)

I.3.b. Coûts de mise en œuvre

Les coûts de mise en œuvre sont pratiquement les mêmes que ceux de la technique présentée en section I.2. La différence est que la technique de répartiteur non concurrent à deux processus comporte deux processus σ_1 et σ_2.

Coût temporel Le coût temporel est pratiquement le même que celui de la technique de répartiteur non concurrent à un processus (section I.2) avec en plus pour chaque bloc d'exécution ($start_i$, end_i, $\tau_{\alpha(i)\beta(i)}$) de $Se_{theorique}$ la durée d'un appel des primitives $Creer\_Processus$ (création de σ_2 pour chaque bloc d'exécution) et $Supprimer\_Processus$ (suppression de σ_2 pour passer au bloc d'exécution suivant).

Coût spatial Le coût spatial est essentiellement dû au stockage de la séquence dans un tableau et l'espace mémoire utilisé pour les contextes des processus.

Niveaux de priorité La technique de mise en œuvre nécessite deux niveaux de priorité à savoir la priorité de la tâche répartiteur et celle de la tâche fonctionnelle.

Coût des changements de contexte À chaque entrée de la table il y a deux changements de contexte à savoir la reprise en main par le processus répartiteur σ_1 et le lancement du processus σ_2 fonctionnelle. Pour $|Se_{theorique}|$ entrées de la table d'ordonnancement le coût dû aux changements de contexte est $2 \times |Se_{theorique}|$.

Robustesse Lorsqu'il y a une erreur de sous-calibrage ($C_i > C_i^{max}$), le processus répartiteur σ_1 interrompt l'exécution de σ_2 pour lancer l'exécution du bloc d'exécution suivant. Ainsi, l'impact d'un débordement se limite toujours à un bloc d'exécution. La figure 3.6-B présente l'exemple du scénario d'exécution produit lorsque τ_3 devrait s'exécuter pendant $C_{31} = 6$. À la date $t = 10$, σ_2 est interrompue et la partie de code correspondant à 2 unités de temps n'est pas exécutée. Il en résulte que le bloc d'exécution théorique est respecté. Cependant, cette technique de mise en œuvre a comme grand inconvénient qu'en cas d'erreur de sous-calibrage, l'exécution est tronquée. Nous avons une robustesse temporelle et topologique mais pas de robustesse fonctionnelle.
Le tableau A.3 (annexe A) résume les coûts de mises en œuvre de la technique à deux processus.

I.4. Répartition par affectation de priorités

L'inconvénient de la mise en œuvre de la répartition à deux processus est que les primitives de création et de suppression de processus sont utilisées systématiquement et qu'en cas d'erreur de sous-calibrage, la tâche exécutée par le processus fonctionnel est arrêtée et son exécution est tronquée. Pour pallier cela nous proposons la mise en œuvre de répartition par affectation de priorités.

I.4.a. Description

Afin que chaque fonction f_i ait un contexte d'exécution propre, dans la technique de répartition par affectation de priorités, nous utilisons autant de processus fonctionnels qu'il y a de tâches périodiques. Pour gérer l'ordre d'exécution des processus fonctionnels, nous utilisons un processus supplémentaire σ_{Rep} ou processus répartiteur. Pour gérer l'ordre d'exécution des processus fonctionnels, le processus répartiteur agit par modification de leurs priorités en augmentant, à une date t_i correspondant à l'exécution d'un bloc $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$, la priorité du processus fonctionnel $\sigma_{\alpha(i)}$. Comme les systèmes d'exploitation temps réel ont un nombre fini de niveaux de priorités, ne faire qu'augmenter les priorités fera atteindre très rapide les limites. Pour y palier, nous diminuons également les priorités du processus fonctionnel $\sigma_{\alpha(i-1)}$. Un autre objectif est d'empêcher un processus qui ne devrait pas s'exécuter de le faire. En effet, si plusieurs processus sont prêts et que le dernier processus qui devrait s'exécuter a terminé son exécution (par exemple $C_i < C_i^{max}$), l'un des processus prêts commencera son exécution même si σ_{rep} n'augmente pas sa priorité. Pour éviter une telle éventualité, nous utilisons un processus supplémentaire oisif (aucune instruction à exécuter) σ_{Idle}, de sorte que σ_{Idle} ait une priorité plus grande que les priorités des autres processus prêts. En définitive, cette mise en œuvre nécessite donc $|\tau| + 2$ processus et quatre niveaux de priorité (la plus faible priorité étant 1) :

- priorité des processus fonctionnels non élus : 1 ;
- priorité du processus oisif : 2 ;
- priorité du processus fonctionnel à exécuter : 3 ;
- priorité du processus répartiteur : 4.

Lorsqu'un processus fonctionnel termine son exécution avant la date de fin prévue, le processus oisif prend la main. Les temps creux correspondent aux instants d'exécution du processus oisif. En diminuant la priorité du processus $\sigma_{\alpha(i-1)}$ du bloc précédent, on empêche le processus en situation de dépassement de temps processeur d'utiliser le processeur au delà du temps qui lui est alloué. Par contre le processus en situation de dépassement peut terminer l'exécution de son code à son prochain réveil.

La figure 3.7 présente l'exemple de mise en œuvre du système de tâches du tableau 3.1.

La mise en œuvre se faisant par $|\tau|+2$ processus, le scénario d'exécution (voir figure 3.8) produit par la mise en œuvre dépendra des états successifs de ces processus. En particulier, le processus répartiteur σ_{Rep} étant le plus prioritaire, dès qu'il passe à l'état prêt, il passe en exécution. Le processus σ_{Idle} est soit à l'état prêt (quand un processus de priorité plus grande est actif), soit en cours d'exécution. Les autres processus qui ont en charge l'exécution des fonctions sont soit endormis (en attente de leur date d'activation périodique), soit prêts (lorsqu'ils sont actifs et en attente de changement de priorité), soit en cours d'exécution (lorsque σ_{Rep} leur affecte une priorité plus grande que celle de σ_{Idle}). Le scénario d'exécution produit dépend donc des dates auxquelles σ_{Rep} effectue les changements de priorités. σ_{Rep}, après sa création par init, passe à l'état endormi jusqu'à la date de référence passée en paramètre. Lors de son exécution, σ_{Rep} augmente la priorité du processus $\sigma_{\alpha(1)}$ correspondant au premier bloc $(start_1, end_1, \tau_{\alpha(1)\beta(1)})$ et diminue la priorité de $\sigma_{\alpha(|Se_{theorique}|)}$. Il passe ensuite à l'état endormi jusqu'à $start_2$. Par la

```
f1(){ /*contenu*/ }
f2(){ /*contenu*/ }
f3(){ /*contenu*/ }
Table={(0,σ1)(3,σ2)(6,σ3)(10,σ1)(12,σ3)}
H ← 16
TailleTable ← 5
r1← 0     r2 ← 3        r3 ← 0
T1 ← 8    T2 ← 8        T3 ← 16
tf1(debut){
    t0 ← debut +r1
    loop
        Attendre_Date(t)
        f1()
        t0 ← t0 + T1
    end loop
}
tf2(debut){
    t0 ← debut+r2
    loop
        Attendre_Date(t)
        f2()
        t0 ← t0 + T2
    end loop
}
tf3(debut){
    t0 ← debut +r3
    loop
        Attendre_Date(t)
        f3()
        t0 ← t0 + T3
    end loop
}
tfIdle(debut){
    loop
    end loop
}
```

```
tfRep(debut){
    t0 ← debut
    k ← 1
    loop
        t ← t0 + Table[k][1]
        Attendre_Date(t)
        if k==1 then
            Priorité_Processus(Table[TailleTable][2],1)
        else
            Priorité_Processus(Table[k-1][2],1)
        end if
        Priorité_Processus(Table[k][2],3)
        k ← k+1
        if k > TailleTable then
            k ← 1
            t0 ← t0 + H
        end if
    end loop
}
init(){
    t1 ← Lire_Date()
    t1 ← t1+ D0
    Creer_Processus(σRep,4,tfRep,t1)
    Creer_Processus(σIdle,2,tfIdle,t1)
    Creer_Processus(σ1,1,tf1,t1)
    Creer_Processus(σ2,1,tf2,t1)
    Creer_Processus(σ3,1,tf3,t1)
}
```

FIGURE 3.7 – Algorithme d'une mise en œuvre de répartition par affection de priorités

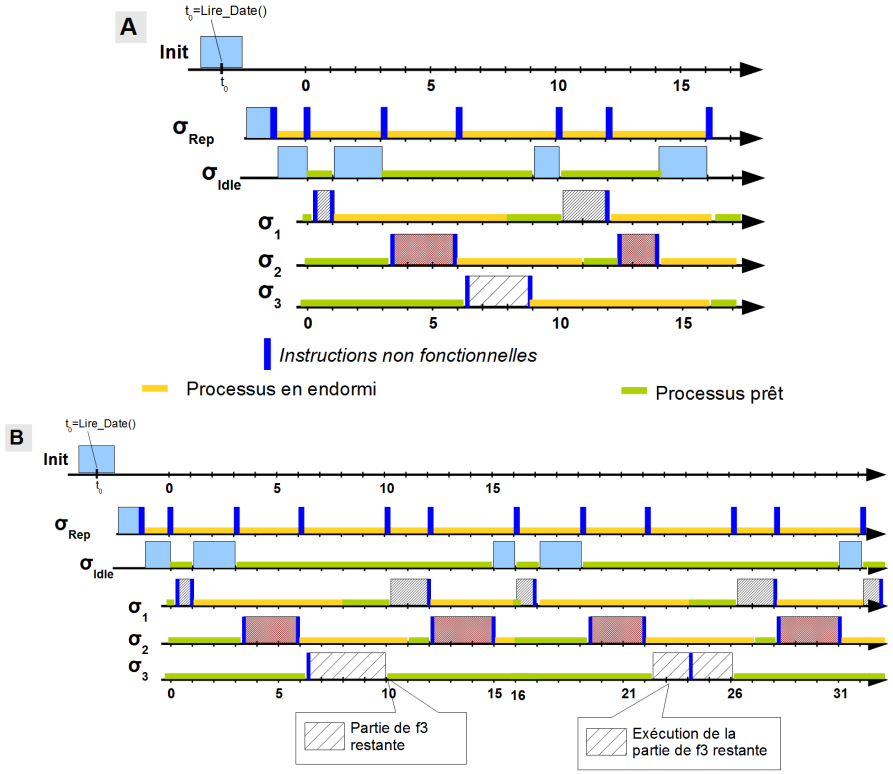


FIGURE 3.8 – Scénario d'exécution d'une mise en œuvre de répartition par affectation de priorités. A : sans erreur de calibrage ; B : avec une erreur de calibrage ($C_{31} = 6 > C_3^{max} = 4$)

suite les actions de modification de priorités se répètent aux dates correspondant au scénario d'exécution théorique. Le scénario d'exécution effectivement produit est inflexible puisqu'un processus d'un bloc i ne peut commencer son exécution avant $start_i$.

I.4.b. Coût de mise en œuvre

Coût temporel Les principaux coûts de mise en œuvre sont dûs au processus répartiteur σ_{Rep} qui effectue $2 \times |Se_{theorique}|$ changements de priorité par cycle. Pour effectuer ces changements σ_{Rep} accède à la mémoire pour déterminer les processus concernés par les changements de priorités (accès en lecture), pour déterminer la date de prochain réveil (calcul et accès en écriture de la mémoire). Ces opérations se répétant à l'identique pour chaque bloc d'exécution, le coût temporel est proportionnel à $|Se_{theorique}|$.

Coût spatial Le coût spatial vient principalement du fait que la mise en œuvre nécessite l'espace mémoire de $|\tau|+2$ processus. En plus de l'espace mémoire des processus, σ_{Rep} requiert deux variables de type date et entier pour le traitement de la table d'ordonnancement $Table$. Chaque processus fonctionnel utilise une date pour gérer sa périodicité. La table d'ordonnancement est de taille $|Se_{theorique}|$.

Niveaux de priorité La mise en œuvre nécessite 4 niveaux de priorités.

Coût des changements de contexte Pour chaque bloc $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$ d'exécution il y a un changement de contexte permettant à σ_{Rep} de s'exécuter et un deuxième changement de contexte pour que $\sigma_{\alpha(i)}$ s'exécute, soit $2 \times |Se_{theorique}|$ changements de contextes.

Robustesse Lorsqu'il y a une erreur de sous-calibrage, le processus en débordement est préempté par le processus répartiteur. En effet, lorsque le processus répartiteur diminue la priorité du processus en cours d'exécution et en débordement de temps d'exécution et augmente la priorité du processus qui doit s'exécuter, cela a pour effet de préempter le processus en débordement. La figure 3.8-B présente le scénario d'exécution effectif lorsque σ_3 devrait s'exécuter avec $C_{31} = 6$. La technique de mise en œuvre de répartition par affectation de priorité permet donc de limiter les effets d'un sous-calibrage aux blocs d'exécution en débordement. Nous avons donc une robustesse temporelle et fonctionnelle. Par contre l'ordre d'exécution des instructions n'est pas gardé puisque les instructions non exécutées le sont dans des bloc d'exécutions différents : nous n'avons pas de robustesse topologique.
Le tableau A.4 (annexe A) résume les coûts de mise en œuvre pour la technique de mise en œuvre de répartition par affectation de priorités.

I.5. Répartition par gestion d'états

L'inconvénient de la mise en œuvre de répartition par affectation de priorité est que le processeur reste actif (exécution du processus oisif σ_{Idle}) même en absence de tâche à exécuter. Cela constitue un gaspillage d'énergie. Afin de diminuer la consommation d'énergie nous proposons la mise en œuvre par gestion d'états.

I.5.a. Description

La technique de répartition par gestion d'état est similaire à la technique de répartition par affection de priorités (section I.4). Elle consiste à utiliser $|\tau|$ processus fonctionnels, un processus répartiteur σ_{rep} et une table d'ordonnancement $Table$.
Le processus répartiteur pour contrôler l'ordre d'exécution des processus fonctionnels agit explicitement sur leurs états en utilisant les primitives Arreter_Processus et Continuer_Processus de l'API abstraite. Comme l'utilisation de ces primitives nécessite les références des processus, $Table$ est un tableau de $|Se_{theorique}|$ lignes. Chaque ligne représente un bloc d'exécution ($start_i$, end_i, $\tau_{\alpha(i)\beta(i)}$) et contient la date de début d'exécution $start_i$ avec une référence du processus $\sigma_{\alpha(i)}$. L'arrêt systématique des processus fonctionnels fait qu'à tout instant, il y a au plus un processus prêt. L'utilisation du processus oisif n'est donc plus nécessaire et le processus répartiteur répète, pour chaque ligne i de la table d'ordonnancement, les actions :
– attente date correspondant à $start_i$;
– suspension du processus $\sigma_{\alpha(i-1)}$ de la ligne précédente ;
– relance du processus $\sigma_{\alpha(i)}$.
L'ordre d'exécution des instructions fonctionnelles est explicitement déterminé par le processus répartiteur qui gère les états des autres processus. Si un processus fonctionnel prend plus de temps que prévu son exécution est suspendue par le processus répartiteur. Les instructions restantes sont exécutées lorsque le répartiteur redonne la main au processus fonctionnel. La figure 3.9 présente la mise en œuvre correspondant au système de tâches du tableau 3.1.
La technique de mise en œuvre nécessitant $|\tau| + 1$ processus, le scénario d'exécution produit (voir figure 3.2) dépend des états successifs de ces processus. Le processus répartiteur σ_{Rep} ayant la plus grande priorité, il peut commencer son exécution dès qu'il passe à l'état prêt. Ensuite l'exécution se poursuit comme pour la technique de mise en œuvre de répartition par affectation de priorités (section I.4), à la seule différence qu'au lieu de diminuer une priorité, σ_{Rep} arrête explicitement un processus et au lieu d'augmenter un priorité, σ_{rep} relance explicitement un processus. Le scénario d'exécution produit est également inflexible puisqu'aucun processus fonctionnel ne s'exécute avant $start_i$.

I.5.b. Coûts de mise en œuvre

Les coûts de mise en œuvre sont similaires à ceux de la section I.4.b à différence que le répartiteur utilise les instructions temporelles du suspension et de reprise des processus au lieu d'utiliser des primitives de changement de priorités. Une autre différence est qu'en terme de coût de mise

```
f1(){ /*contenu*/ }
f2(){ /*contenu*/ }
f3(){ /*contenu*/ }
Table={(0,σ1)(3,σ2)(6,σ3)(10,σ1)(12,σ3)}
H ← 16
TailleTable ← 5
r1 ← 0      r2 ← 3      r3 ← 0
T1 ← 8      T2 ← 8      T3 ← 16
tf1(debut){
    t0 ← debut +r1
    loop
        Attendre_Date(t)
        f1()
        t0 ← t0 + T1
    end loop
}
tf2(debut){
    t0 ← debut+r2
    loop
        Attendre_Date(t)
        f2()
        t0 ← t0 + T2
    end loop
}
tf3(debut){
    t0 ← debut +r3
    loop
        Attendre_Date(t)
        f3()
        t0 ← t0 + T3
    end loop
}
```

```
tfRep(debut){
    t0 ← debut
    k ← 1

    loop
        t ← t0 + Table[k][1]
        Attendre_Date(t)
        if k==1 then
            Arreter_Processus(Table[Tailletable][2])
        else
            Arreter_Processus(Table[k-1][2])
        end if
        Continuer_Processus(Table[k][2])
        k ← k+1
        if k > TailleTable then
            k ← 1
            t0 ← t0 + H
        end if
    end loop
}
init(){
    t1 ← Lire_Date()
    t1 ← t1+ D0
    Creer_Processus(σRep,2,tfRep,t1)
    Creer_Processus(σ1,1,tf1,t1)
    Creer_Processus(σ2,1,tf2,t1)
    Creer_Processus(σ3,1,tf3,t1)
    Arreter_Processus(σ1)
    Arreter_Processus(σ2)
    Arreter_Processus(σ3)
}
```

FIGURE 3.9 – Algorithme d'une mise en œuvre de répartition par gestion d'états

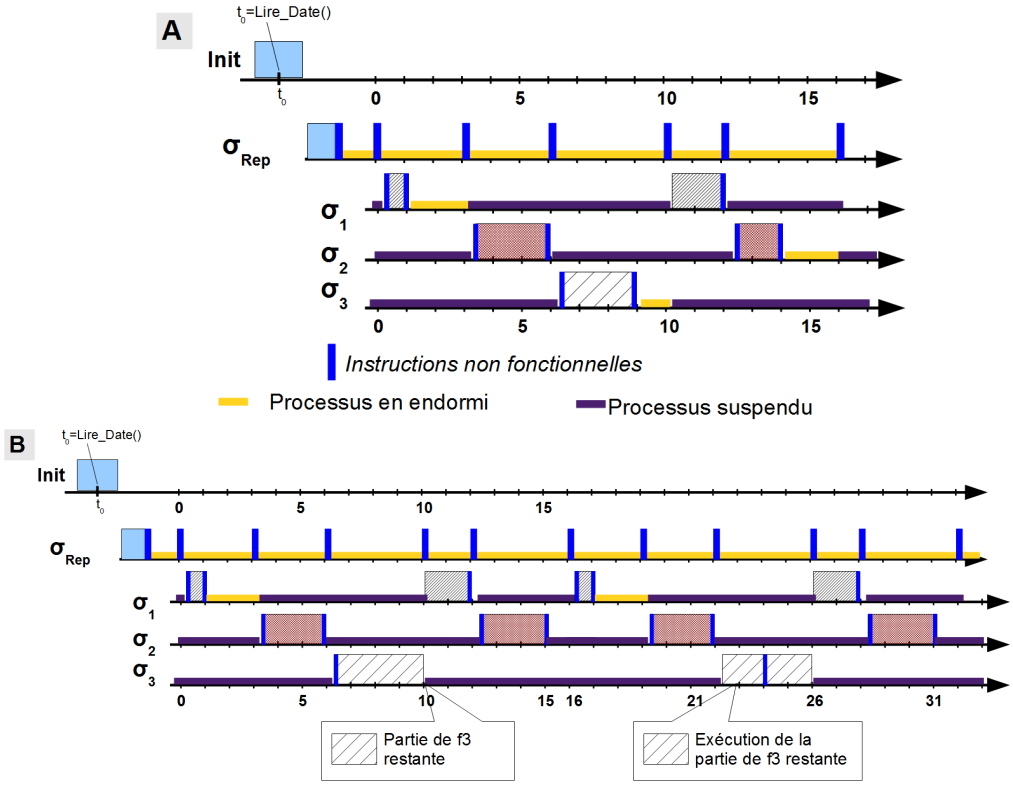


FIGURE 3.10 – Scénario d'exécution d'une mise en œuvre de répartition par gestion d'états. A : sans erreur de calibrage ; B : avec une erreur de calibrage ($C_{31} = 6 > C_3^{max} = 4$)

en œuvre spatiale il y a un processus de moins à créer et que la technique nécessite 2 niveaux de priorités.
Le tableau A.5 (annexe A) récapitule les coûts de mise en œuvre pour la technique de répartition par gestion d'états.

I.6. Synchronisation collective

Un inconvénient de la répartition par gestion d'état est qu'il faut en plus des processus fonctionnels un processus non fonctionnel (le répartiteur). Pour n'avoir que des processus fonctionnels nous proposons la mise en œuvre par synchronisation collective.

I.6.a. Description

La solution de la synchronisation collective consiste à utiliser autant de processus qu'il y a de tâches. Et au lieu d'utiliser un processus répartiteur pour gérer l'ordre d'exécution des processus, la technique met en œuvre une gestion collective (tous les processus participent) de l'ordre d'exécution. Pour cela nous utilisons le principe de synchronisation par sémaphore. En effet les sémaphores permettent de mettre en place des précédences entre deux processus (prédécesseur-successeur) : le processus prédécesseur libère un sémaphore et le processus successeur attend le sémaphore libéré par le prédécesseur. Nous appliquons ce principe à plusieurs processus en établissant pour chaque processus σ_i une table d'ordonnancement $Table_i$. Les précédences n'étant pas nécessairement les mêmes pour toutes les instances de tâche, il est nécessaire d'indiquer pour chaque instance le processus prédécesseur et le processus successeur. La table $Table_i$ est donc un tableau de $\frac{H}{T_i}$ lignes qui contient le sémaphore à attendre et le sémaphore à relâcher. À l'initialisation du sytème, le sémaphore attendu par le premier processus est libéré.
L'ordre d'exécution est garanti par l'utilisation des sémaphores. Il y a un temps creux s'il n'y a pas de processus actif. Lorsqu'un processus s'exécute plus de temps que prévu il retarde tous les autres processus. À l'inverse les dates de début d'exécution effectives peuvent être antérieures à celles prévues par le scénario théorique ($start'_i \leq start_i$) dès lors que certains processus s'exécutent moins de temps que prévu et que les processus successeurs sont actifs.
La figure 3.11 présente l'exemple de la mise en œuvre correspondant au tableau 3.1.
La mise en œuvre nécessitant $|\tau|$ processus, le scénario d'exécution produit (voir figure 3.2) dépend des états successifs des $|\tau|$ processus. Chaque processus après sa création par le processus init passe à l'état endormi jusqu'à sa date d'activation. Puis chacun d'eux passe à l'état bloqué, en attente d'un sémaphore qui doit être libéré par le processus prédécesseur. En particulier, le premier processus à s'exécuter attend un sémaphore qui est libéré par le processus init. Après son exécution, le premier processus libère le sémaphore du processus suivant. Le scénario d'exécution produit est un scénario flexible puisque un processus commence son exécution dès qu'il est actif et que le processus qui le précède a terminé son exécution.

```
f1(){ /*contenu*/ }
f2(){ /*contenu*/ }
f3(){ /*contenu*/ }
Table1={(j1,j2),(j1,j2)}   TailleTable1 ← 2
Table2={(j2,j3),(j2,j1)}   TailleTable2 ← 2
Table3={(j3,j1)}           TailleTable3 ← 1
r1 ← 0   r2 ← 3   r3 ← 0
T1 ← 8   T2 ← 8   T3 ← 16
tf1(debut){
    t ← debut + r1
    NumInstance ← 1
    loop
        Attendre_Date(t)
        Prendre_Semaphore(Table1[NumInstance][1])
        f1()
        Vendre_Semaphore(Table1[NumInstance][2])
        NumInstance ← NumInstance+1
        if NumInstance >TailleTable1 then
            NumInstance ← 1
        end if
        t ← t + T1
    end loop
}
tf2(debut){
    t ← debut + r2
    NumInstance ← 1
    loop
        Attendre_Date(t)
        Prendre_Semaphore(Table2[NumInstance][1])
        f2()
        Vendre_Semaphore(Table2[NumInstance][2])
        NumInstance ← NumInstance+1
        if NumInstance > TailleTable2 then
            NumInstance ← 1
        end if
        t ← t + T2
    end loop
}
```

```
tf3(debut){
        t ← debut + r3
    NumInstance ← 1
    loop
        Attendre_Date(t)
        Prendre_Semaphore(Table3[NumInstance][1])
        f3()
        Vendre_Semaphore(Table3[NumInstance][2])
        NumInstance ← NumInstance+1
        if NumInstance >TailleTable3 then
            NumInstance ← 1
        end if
        t ← t + T3
    end loop
}
init(){
    t1 ← Lire_Date()
    t1 ← t1 + 1
    Creer_Semaphore(j1,1)
    Creer_Semaphore(j2,1)
    Creer_Semaphore(j3,1)
    Prendre_Semaphore(j2)
    Prendre_Semaphore(j3)
    Creer_Processus(σ1,1,tf1, t1)
    Creer_Processus(σ2,1,tf2, t1)
    Creer_Processus(σ3,1,tf3, t1)
}
```

FIGURE 3.11 – Algorithme d'une mise en œuvre par synchronisation collective

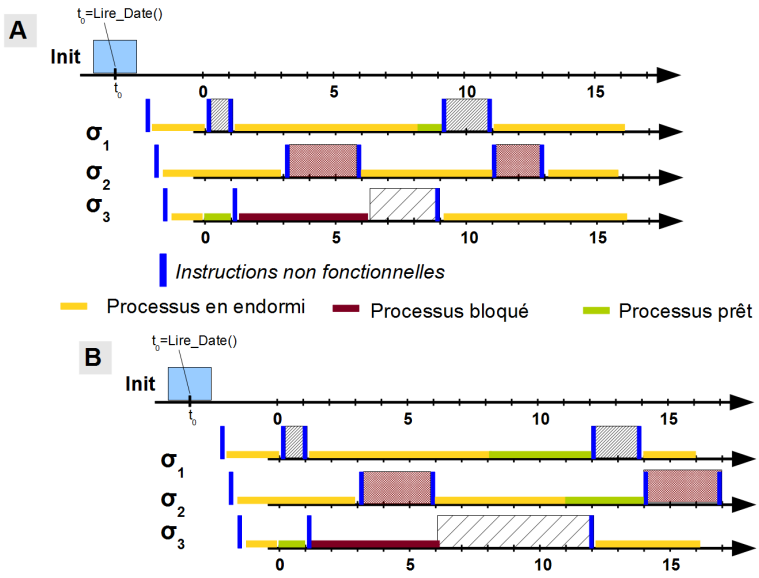


FIGURE 3.12 – Scénario d'exécution d'une mise en œuvre par synchronisation collective. A : sans erreur de calibrage ; B : avec une erreur de calibrage ($C_{31} = 6 > C_3^{max} = 4$)

I.6.b. Coût de mise en œuvre

Coût temporel Les principaux coûts temporels sont dûs à la création des sémaphores, l'attente et la libération de sémaphores, ainsi que la création des processus. Dans notre mise en œuvre nous avons utilisé autant de sémaphores qu'il y a de tâches. Il faut donc utiliser $|\tau|$ sémaphores. Comme la prise et la libération de sémaphores correspondent à un bloc d'exécution, il y a $|Se_{theorique}|$ opérations de prise et de libération de sémaphore.

Coût spatial Le coût spatial vient principalement du fait qu'il faut de la mémoire pour les $|\tau|$ sémaphores et $|\tau|$ processus.

Niveaux de priorité La gestion de l'ordre d'exécution n'utilisant pas les priorités, la technique nécessite un seul niveau de priorité.

Coût des changements de contexte Pour chaque bloc d'exécution $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$ il y a un changement de contexte (σ_{i-1} passe à l'état endormi et σ_i commence son exécution), soit au total $|Se_{theorique}|$ changements de contexte.

Robustesse Lorsqu'il y a une erreur de sous-calibrage, le processus en débordement n'est pas interrompu et continue son exécution jusqu'au bout. La figure 3.12 présente le scénario

d'exécution produit par cette technique lorsque le processus σ_3 s'exécute plus de temps que prévu. Le bloc d'exécution effectif est plus grand que le bloc théorique, mais l'ordre d'exécution est conservé. Nous avons donc une robustesse topologique et fonctionnelle mais pas de robustesse temporelle.
Le tableau A.6 (annexe A) résume les côuts de mise en œuvre pour la technique de mise en œuvre par synchronisation collective.

I.7. Attente coopérative de dates

L'inconvénient de la mise en œuvre par synchronisation collective est qu'il faut utiliser des objets (sémaphores) supplémentaires. Pour éviter d'utiliser des objets supplémentaires, nous proposons la mise en œuvre par attente coopérative.

I.7.a. Description

La mise en œuvre par attente coopérative de dates est similaire à la technique de mise en œuvre par synchronisation collective. Pour chaque tâche τ_i, nous utilisons un processus fonctionnel σ_i. De même l'ordre d'exécution dicté par le scénario d'exécution théorique est géré de manière collective par les processus fonctionnels. Au lieu d'utiliser des sémaphores pour la synchronisation nous utilisons plutôt le principe de répartir le temps processeur. En effet chaque bloc d'exécution $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$ peut être vu comme une fenêtre d'exécution pour le processus $\sigma_{\alpha(i)}$. Les fenêtres d'exécution des processus sont enregistrés dans $|\tau|$ tables d'ordonnancement $Table_i$. Chaque table $Table_i$ est alors constitué de $\frac{H}{T_i}$ lignes qui indiquent les dates de début d'exécution des instances du processus σ_i.
La figure 3.13 présente l'exemple de la mise en œuvre par la technique de mise en œuvre par attente coopérative de dates.
Dans cette mise œuvre l'ordre d'exécution est induit par les dates attendues par chaque processus. Lorsqu'à un instant t tous les processus attendent une date $> t$ on obtient un temps creux.
La technique nécessitant $|\tau|$ processus, le scénario d'exécution produit (voir figure 3.2) dépend des états successifs de ces processus. Après leur création, chaque processus passe à l'état endormi en attente de la date de sa première exécution. Comme les dates sont distinctes, il ne peut y avoir deux processus à la fois à l'état prêt. Dès qu'un processus s'exécute, il passe à l'état endormi et se met en attente de la date de sa prochaine exécution. Le scénario d'exécution produit est un scénario inflexible puisqu'aucun processus ne peut s'exécuter avant la date prévue.

I.7.b. Coût de mise en œuvre

Les coûts de mise en œuvre sont pratiquement identiques aux coûts de la technique par synchronisation collective (section I.6) à la différence que tous les coûts relatifs aux sémaphores sont supprimés et remplacés par les coûts de stockage des fenêtres d'exécution par les tables

```
f1(){ /*contenu*/ }
f2(){ /*contenu*/ }
f3(){ /*contenu*/ }
Table1={0,10}   TailleTable1 ← 2
Table2={3,12}   TailleTable2 ← 2
Table3={6}      TailleTable3 ← 1
H ← 16

tf1(debut){
    t ← debut
    NumInstance ← 1
    loop
        Attendre_Date(t+Table1[NumInstance])
        f1()
        NumInstance ← NumInstance+1
        if NumInstance > TailleTable1 then
            NumInstance ← 1
            t ← t + H
        end if
    end loop
}
tf2(debut){
    t ← debut
    NumInstance ← 1
    loop
        Attendre_Date(t+Table2[NumInstance])
        f2()
        NumInstance ← NumInstance+1
        if NumInstance >TailleTable2 then
            NumInstance ← 1
            t ← t + H
        end if
    end loop
}
```

```
tf3(debut){
    t ← debut
    NumInstance ← 1
    loop
        Attendre_Date(t+Table3[NumInstance])
        f3()
        NumInstance ← NumInstance+1
        if NumInstance > TailleTable3 then
            NumInstance ← 1
            t ← t + H
        end if
    end loop
}
init(){
    t1 ← Lire_Date()
    t1 ← t1 + D0
    Creer_Processus(σ1,1,tf1, t1)
    Creer_Processus(σ2,1,tf2, t1)
    Creer_Processus(σ3,1,tf3, t1)
}
```

FIGURE 3.13 – Algorithme d'une mise en œuvre par attente coopérative de dates

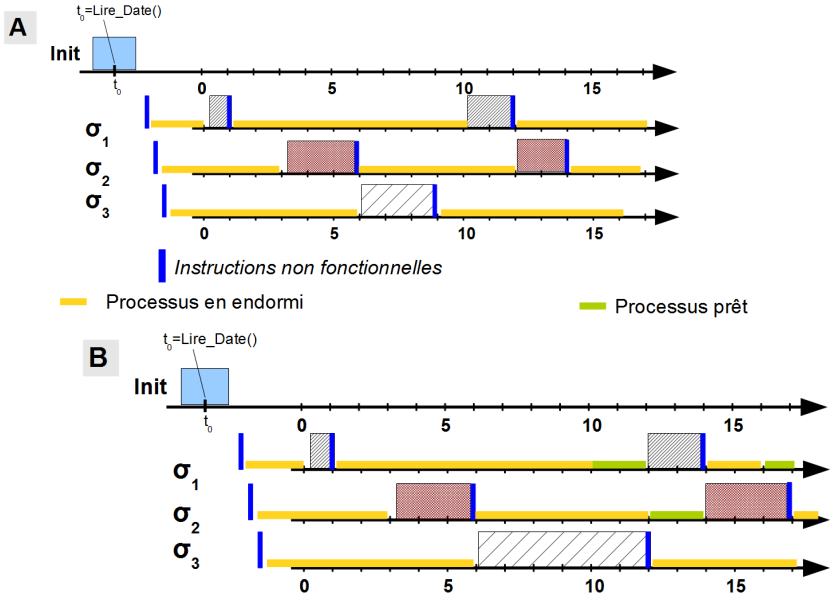


FIGURE 3.14 – Scénario d'exécution d'une mise en œuvre par attente collective. A : sans erreur de calibrage ; B : avec une erreur de calibrage ($C_{31} = 6 > C_3^{max} = 4$)

d'ordonnancement. Une autre différence est que lorsqu'il y a une erreur de sous-calibrage, le débordement de l'un des processus peut avoir pour conséquence que plusieurs processus sont à l'état prêt (voir figure 3.14). L'ordre d'exécution peut être complètement bouleversé par exemple si les processus passent à l'état prêt dans un ordre différents du scénario d'exécution théorique. Nous n'avons donc qu'une robustesse fonctionnelle et pas de robustesse temporelle, ni de robustesse topologique.

Le tableau A.7 (annexe A) récapitule les coûts de mise en œuvre pour la technique d'attente collective de dates.

I.8. Autre mise en œuvre flexible

Parmi nos techniques de mise en œuvre, seule la technique de mise en œuvre par synchronisation collective produit des scénarios d'exécution flexibles.

D'autres mises en œuvres flexibles peuvent se faire en utilisant les techniques des sections I.1 et I.2 en regroupant les blocs d'exécution du scénario d'exécution théorique $Se_{theorique}$ qui ne nécessitent pas d'attente.

I.8.a. Description

En effet, il est possible dans la mise en œuvre d'exécuter les fonctions sans attente intermédiaire si cette exécution ne transgresse pas de contrainte temporelle. On peut enchainer l'exécution de deux blocs $(start_\alpha, end_\alpha, f_\alpha)$ et $(start_\beta, end_\beta, f_\beta)$ si :
– la date d'activation le permet : $r_\beta \leq start_\alpha$;
– ou si la durée d'exécution le permet : $r_\beta \leq start_\alpha + C_\alpha^{min}$.

```
f1(){ /*contenu*/ }
f2(){ /*contenu*/ }
f3(){ /*contenu*/ }
tf1(debut){
    t0 ← debut
    t ← t0

    loop
            Attendre_Date(t)
            f1()
            t ← t0 + 3
            Attendre_Date(t)
            f2()
            f3()
            t ← t0 + 10
            Attendre_Date(t)
            f1()
            f2()
            t0 ← t0 + 16
    end loop
}
init(){
    t1 ← Lire_Date()
     t1 ← t1 + D0
    Creer_Processus(σ1,1,tf1,t1)
}
```

FIGURE 3.15 – Algorithme d'une mise en œuvre flexible par un processus

Pour le scénario d'exécution théorique du tableau 3.1 il est possible d'exécuter f_2 et f_3 sans attente intermédiaire. Cela s'explique par le fait que la date d'activation $r_3 = 0$ est inférieure

à la date de début du bloc d'exécution $start_3 = 2$. De même, il est possible d'exécuter f_2 et f_1 sans attente intermédiaire parce qu'en prenant la durée d'exécution au meilleur cas, on est sûr que f_2 ne s'exécutera pas avant sa date d'activation. En effet la deuxième instance de τ_2 est active à partir de $t = 11$ et f_1 finit son exécution au plus tôt à $t = 11$ (f_1 commence son exécution à $t = 10$ avec un $C_1^{min} = 1$). La figure 3.15 présente la mise en œuvre flexible correspondant aux regroupements.

I.8.b. Coût de mise en œuvre

Les coûts de mise en œuvre sont les mêmes que ceux des mises en œuvre par un processus ou de répartion non concurrente à un et à deux processus. Il y a tout de même une différence dans le fait que s'il y a une erreur de sur-calibrage ($C_i < C_i^{min}$) certaines instances de tâches peuvent s'exécuter avant la date d'activation de l'instance. On n'a donc pas de robustesse temporelle mais on a la robustesse fonctionnelle et topologique.

I.9. Etude comparative des techniques de mise en œuvre

Dans cette section, nous résumons les principales caractéristiques des techniques de mise en œuvre. L'objectif de ce résumé est d'avoir une vue globale permettant de comparer ces techniques. En particulier, notre résumé se décline sous forme de tableaux de bord. Le tableau 3.2 nous permet de faire une étude comparative des scénarios d'exécution effectifs produits par nos techniques. Dans le tableau 3.3 nous comparons les coûts de mise en œuvre et le tableau 3.4 résume les réactions aux erreurs de calibrage.
Le tableau 3.2 récapitule, pour chaque technique de mise œuvre, le type de scénario d'exécution produit.

Techniques de mise en œuvre	Scénarios flexibles par défaut	Scénarios flexibles par regroupement	Scénarios inflexibles
Mise en œuvre par une tâche	Non	Oui	Oui
Répartiteur non concurrent à un processus	Non	Oui	Oui
Répartiteur non concurrent à deux processus	Non	Oui	Oui
Répartition par gestions d'états	Non	Non	Oui
Répartition par affectation de priorité	Non	Non	Oui
Temps partagé	Non	Non	Oui
Synchronisation collective	Oui	Non	Non

TABLE 3.2 – Etude comparative des scénarios d'exécution produits par les techniques de mise en œuvre

Le tableau 3.3 présente un récapitulatif des coûts de mise en œuvre.

Techniques de mise en œuvre	ΔEc	ΔMc	Pc	RC	Embarquement séquence
Mise en œuvre par un processus	$O(\|Se\|)$	$O(\|Se\|)$	1	0	code processus
Répartiteur non concurrent à un processus	$O(\|Se\|)$	$O(\|Se\|)$	1	0	1 tableau
Répartiteur non concurrent à deux processus	$O(\|Se\|)$	$O(\|Se\|)$	2	$O(\|Se\|)$	1 tableau
Répartition par gestions d'états	$O(\|Se\| + \|\tau\|)$	$O(\|Se\| + \|\tau\|)$	2	$O(\|Se\|)$	1 tableau
Répartition par affectation de priorité	$O(\|Se\| + \|\tau\|)$	$O(\|Se\| + \|\tau\|)$	4	$O(\|Se\|)$	1 tableau
Attente coopérative	$O(\|Se\| + \|\tau\|)$	$O(\|Se\| + \|\tau\|)$	1	$O(\|Se\|)$	$\|\tau\|$ tableaux
Synchronisation collective	$O(\|Se\| + \|\tau\|)$	$O(\|Se\| + \|\tau\|)$	1	$O(\|Se\|)$	$\|\tau\|$ tableaux

TABLE 3.3 – Étude comparative des coûts de mise en œuvre ; $|Se|$: nombre de blocs du scénario d'exécution théorique ; $|\tau|$: nombre de tâches périodiques

Le tableau 3.4 présente un récapitulatif des réactions des techniques de mises en œuvre d'ordonnancement hors-ligne lorsqu'une erreur de sous-calibrage se produit.

Techniques de mise en œuvre	Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Mise en œuvre par un processus	Non	Oui	Oui
Répartiteur non concurrent à un processus	Non	Oui	Oui
Répartiteur non concurrent à deux processus	Oui	Non	Oui
Répartition par gestions d'états	Oui	Oui	Non
Répartition par affectation de priorité	Oui	Oui	Non
Attente coopérative	Non	Oui	Non
Synchronisation collective	Non	Oui	Oui

TABLE 3.4 – Étude comparative de la robustesse

II. Mise en œuvre d'ordonnancement avec préemption

II.1. Techniques de mise en œuvre non utilisables

Pour gérer les préemptions nous ne pouvons utiliser que les techniques de mise en œuvre avec processus répartiteur. En effet, si nous voulions utiliser les techniques de la mise œuvre par un processus et les répartiteurs non concurrents à un ou deux processus, il aurait fallu déterminer pour chaque préemption, à quel découpage fonctionnel elle correspond. Cependant, notre modèle de tâches est basé sur le principe que chaque tâche modélise une et une seule fonction (1 tâche $\equiv$ 1 fonction) et déterminer à quel découpage correspond une préemption semble difficile. Il faudrait découper la fonction en une suite de petites fonctions, dont on vérifierait que le WCET est inférieure ou égal à la durée allouée. D'une part, ceci risquerait de générer des pertes, car un découpage exact ne sera pas toujours possible, surtout si la tâche comporte des instructions conditionnelles. De plus, la somme des WCET de chaque petite fonction peut dépasser le WCET global. Enfin, trouver le "bon" découpage c'est-à-dire celui qui regroupe ni trop peu ni pas assez d'instructions nécessiterait de calculer un grand nombre de WCET.
S'agissant des approches de mise en œuvre par synchronisation collective et par attente coopérative de date, elles sont également inutilisables puisqu'il aurait également fallu déterminer un découpage fonctionnel correspondant à la préemption.

II.2. Techniques de mise en œuvre utilisables

Les techniques envisageables, parmi celles que nous avons déjà présentées, sont la répartition par gestion d'états et la répartition par affectation de priorités.
La mise en œuvre utilisant un processus répartiteur gérant les états des autres processus n'est pas très différente de celle que nous avons présentée dans le cas non préemptif (section I.5). La seule différence est que la table d'ordonnancement prévoit des préemptions.
La technique de mise en œuvre utilisant un processus de répartition par affectation de priorités est la même que celle présentée dans la section I.4. Chaque processus gère son activation. Le répartiteur affecte la priorité la plus grande au processus qui doit s'exécuter.

III. Prise en compte des ressources critiques et des contraintes de précédence

Dans cette section nous présentons des primitives permettant de prendre en compte l'utilisation de ressources critiques (section III.1). Nous présentons ensuite comment prendre en compte les ressources critiques et les contraintes de précédence dans notre modèle (section III.2). Pour finir, nous présentons comment intégrer les ressources critiques et les contraintes de précédence dans nos algorithmes de mises en œuvre de séquences d'ordonnancement hors-ligne (section III.3).

III.1. Primitives d'utilisation de ressources

Une *ressource* est un objet utilisé par une application temps réel. Il existe deux types de ressources à savoir les ressources logicielles (variables) et les ressources matérielles (capteur, actionneur). Une ressource est *critique* lorsqu'elle est partagée par plusieurs tâches.
Dans la suite nous nous intéressons essentiellement aux ressources critiques.
Que la ressource soit matérielle ou logicielle, il y a deux modes d'accès :
- en lecture : l'accès en lecture consiste à lire le contenu d'une variable ou lire la valeur renvoyée par un capteur ;
- en écriture : l'accès en écriture consiste à modifier le contenu de la variable ou à envoyer une commande à un actionneur.

Dans nos algorithmes de mise en œuvre, pour représenter l'utilisation des ressources logicielles nous utiliserons l'opération classique d'affection "$\leftarrow$". Et pour distinguer une ressource matérielle d'une ressource logicielle nous introduisons de nouvelles primitives dans notre API abstraite (voir tableau 3.5).

Primitives	Paramètres d'entrées	Résultat de la primitive
$V \leftarrow$Lire_Ressource(R)	R : Nom/descripteur de ressource	V contient les données venant de la ressource R
Ecrire_Ressource(R,V)	R : Nom/descripteur de ressource V : valeurs écrites	La commande V est envoyée à la ressource R

TABLE 3.5 – Primitives de gestion de ressources matérielles

En présence de ressources critiques, le modèle présenté dans le chapitre 2 n'est plus adapté pour la mise en œuvre. Afin de l'illustrer nous pouvons utiliser l'exemple d'un système de deux tâches (voir tableau 3.6).

Description	exemple
Système de tâches	$\tau = \{\tau_1 =< 0, [5:10], 18, 18 \mid f_1 >,$ $\tau_2 =< 0, [3:5], 18, 18 \mid f_2 >\}$
Scénario d'exécution théorique	$Se_{theorique} = \{(0, 5, \tau_{11}),$ $(5, 10, \tau_{21}), (10, 15, \tau_{11})\}$; $\lvert Se_{theorique}\rvert = 3$; $H = 18$

TABLE 3.6 – Exemple de description d'une application et ordonnancée hors-ligne

La première tâche est utilisée pour inter-agir avec l'extérieur (capteur, actionneur) et la deuxième tâche est une tâche de calcul. Un tel système peut être utilisé pour gérer par exemple une poulie : la tâche τ_1 déterminerait la hauteur de l'objet tracté et agirait aussi sur la vitesse de traction ;

la tâche τ_2 utiliserait la hauteur pour calculer la vitesse de traction. Le scénario d'exécution théorique (figure 3.16) prévoit qu'il y ait une préemption de la tâche τ_1 après la première opération qui dure au pire cas 5 unités de temps. Après la préemption la tâche τ_2 s'exécute puis la tâche τ_1 reprend son exécution.

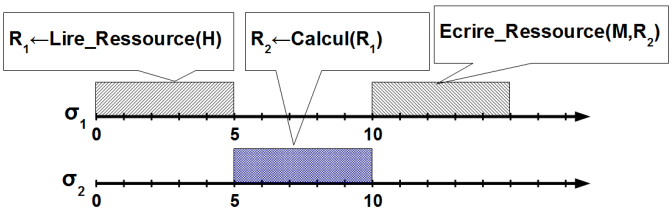


FIGURE 3.16 – Scénario d'exécution prévu

Le scénario d'exécution prévoyant une préemption, nous ne pouvons utiliser que les techniques de répartition par affectation de priorités ou par gestion d'état. La figure 3.17 présente des scénarios d'exécution effectifs qui se produiraient avec l'une de ces techniques de mise en œuvre si les opérations prenaient moins de temps que le pire cas.

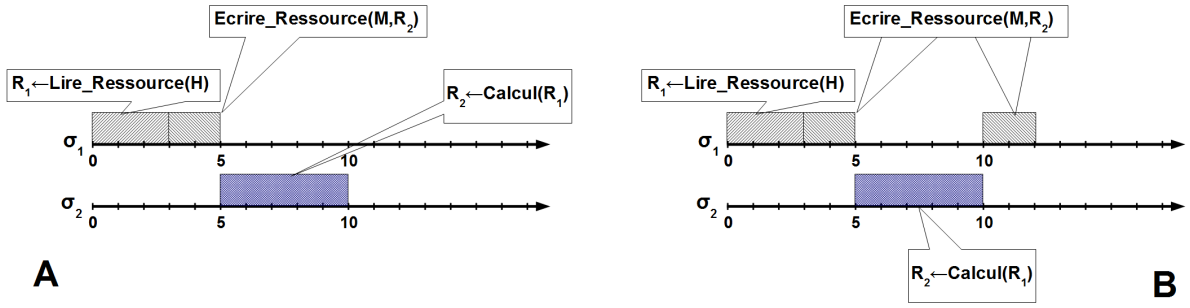


FIGURE 3.17 – Scénarios d'exécution possibles. A : pas de préemption de σ_1 ; B : préemption de σ_1

Dans le scenario effectif 3.17-A, la deuxième partie de la tâche τ_1 qui ne doit pas avoir accès en lecture à la ressource R_2 y a accès avant que la tâche τ_2 n'y ait accès en écriture. Il s'en suit que la valeur utilisée par τ_1 n'est pas bonne. Dans le scénario effectif 3.17-B, la tâche τ_1 est préemptée et la ressource R_2 que τ_1 utilise est modifiée par τ_2. Il en résulte que le comportement du système n'est plus du tout cohérent. Pour éviter que les situations de la figure 3.17 ne se produisent nous devons déterminer un système de tâches "*compatible*" avec nos techniques de mise en œuvre. Un système de tâches est dit compatible avec nos techniques de mise en œuvre si l'utilisation d'une technique de mise en œuvre n'entraine pas le non respect des règles d'utilisation des ressources

critiques ou le non respect des contraintes de précédence. La détermination des systèmes de tâches compatibles avec nos techniques de mise en œuvre doit prendre en compte à la fois le scénario d'exécution théorique et l'utilisation des ressources par les tâches. D'où la nécessité de modéliser l'utilisation des ressources ainsi que les contraintes de précédence. Cette discussion est l'objet de la section III.2.

III.2. Modélisation en présence de ressources critiques et de contraintes de précédence

Les techniques de recherche de séquences d'ordonnancement hors-ligne utilisent généralement un modèle de tâches différent du modèle standard de tâches (présenté dans la section III.2 du chapitre 1). Ce modèle est utilisé pour prendre en compte les contraintes de précédences et de partage de ressources critiques.
Pour cela, les instructions f_i d'une tâche τ_i sont divisées en plusieurs *sous-fonctions* ou sous-blocs d'instructions $f_i^1, f_i^2, \ldots, f_i^{n_i}$. [XP90] parle de *segments* alors que [GCG02] parle de *blocs indépendants*. Nous avons plutôt choisi le terme *sous-fonction* essentiellement pour souligner le fait qu'il s'agit d'un découpage de fonctions. Le découpage en sous-fonctions a pour principal objectif de différencier les instructions d'une fonction ayant une relation (section critique commune, contrainte de précédence) avec une autre fonction. Par exemple, dans [XP90] le découpage en sous-fonction se fait à partir des variables utilisées. [GCG02] utilise une méthode de découpage un peu différente en déduisant le découpage en sous-fonctions à partir des primitives temporelles utilisées par une mise en œuvre. Autrement dit, le découpage en sous-fonctions est généralement obtenu par une analyse de code.
Dans notre cas nous utilisons deux types de relations binaires pour exprimer ces contraintes :
– la relation d'exclusion $\oplus$;
– la relation de précédence $\prec$.
$f_i^j \oplus f_k^l$ signifie que les sous-fonctions doivent s'exécuter en exclusion mutuelle. Dans un ordonnancement mono-processeur, cela revient à dire que f_i^j et f_k^l ne doivent pas s'exécuter en entrelacement.
$f_i^j \prec f_k^l$ signifie que la sous-fonction f_i^j doit s'exécuter entièrement avant que f_k^l ne soit exécutée.
La relation $\oplus$ est une relation symétrique :

$$f_i^j \oplus f_k^l \Leftrightarrow f_k^l \oplus f_i^j$$

La relation $\prec$ est une relation transitive :

$$(f_i^j \prec f_k^l) \wedge (f_k^l \prec f_m^n) \Rightarrow (f_i^j \prec f_m^n)$$

La figure 3.18 présente un exemple de fonctions avec des contraintes de précédences.
Dans la section III.2.a nous présentons, à travers un exemple, la technique de découpage en sous-fonctions utilisée par [XP90] et dans la section III.2.b nous présentons la technique de

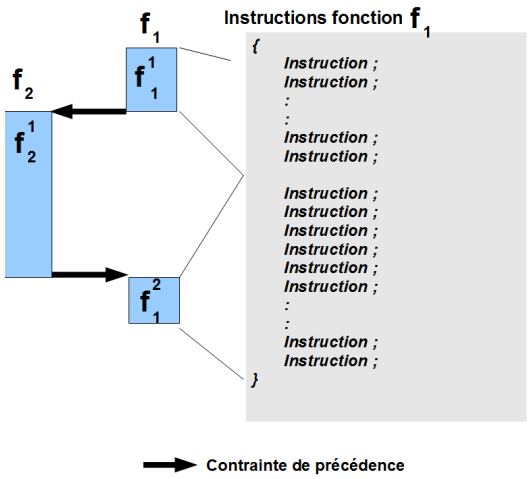


FIGURE 3.18 – Découpage en sous-fonctions avec contraintes de précédences

découpage utilisée par [GCG02]. Le découpage étant une activité manuelle, nous formalisons dans la section III.2.c quelques règles qui permettent de limiter le nombre de sous-fonctions obtenues.

III.2.a. Découpage déduit de l'utilisation des ressources critiques

La technique de découpage en sous-fonctions présentée par [XP90], peut être divisée en trois étapes à savoir :
- identifier les ressources du système ;
- identifier les fonctions utilisant chaque ressource ;
- découper les fonctions en sous-fonctions de sorte à pouvoir exprimer toutes les contraintes dues à l'utilisation des ressources.

Pour illustrer cette technique, nous utilisons le système de tâches dont le modèle fonctionnel est présenté sur la figure 3.19. Les fonctions qui y sont représentées peuvent aussi être utilisées pour la gestion d'une poulie. À la différence du système de tâches du tableau 3.6, la configuration matérielle est différente et nous avons ajouté deux autres tâches d'affichage.

Le système utilise six ressources à savoir H, M, A, R_1, R_2, $Info$. Les seules ressources critiques (utilisées par plusieurs tâches) sont R_1, R_2 et A. f_1 modifie la valeur de R_1 et f_2 a besoin de cette valeur pour un calcul. On peut donc déduire qu'il existe une relation de précédence. De même, f_3 et f_4 utilisent R_1 et R_2. Il faudrait donc qu'il y ait une exclusion mutuelle entre la partie de f_4 utilisant les ressources R_1, R_2 et les parties de f_3 utilisant R_1, R_2. De même, il faudrait une contrainte d'exclusion mutuelle entre f_3 et la partie de f_4 utilisant la ressource A. Le tableau 3.7 résume les trois étapes sus-citées.

Système de tâches	$\tau = \{\tau_1 =< 0, [3:6], 20, 20 \mid f_1 >,$ $\tau_2 =< 0, [1:3], 20, 20 \mid f_2 >\}$ $\tau_3 =< 0, [2:6], 20, 20 \mid f_3 >\}$ $\tau_4 =< 0, [3:5], 20, 20 \mid f_4 >\}$	
Ressources	Lecture	Écriture
H	f_1	
M		f_1
A		f_3 f_4
R_1	f_2 f_3 f_4	f_1
R_2	f_1 f_3 f_4	f_2
$Info$	f_4	f_4
Précédences	$f_1^1 \prec f_2$ $f_2 \prec f_1^2$ $f_1^1 \prec f_3^1$ $f_2 \prec f_3^2$ $f_1^1 \prec f_4^1$ $f_2 \prec f_4^1$	
Exclusions	$f_3^1 \oplus f_4^2$ $f_3^2 \oplus f_4^2$	

TABLE 3.7 – Exemple de découpage des tâches en sous-fonctions utilisant des ressources

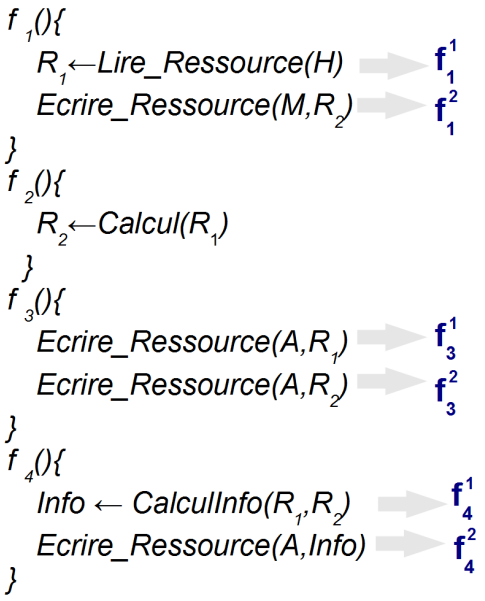


FIGURE 3.19 – Utilisation de ressources et découpage en sous-fonctions

III.2.b. Découpage déduit des instructions temporelles

La technique de découpage en sous-fonctions utilisée par [GCG02], quant à elle part d'une mise en œuvre de tâches dans un contexte d'ordonnancement en ligne. Cela permet d'utiliser les primitives temporelles pour déduire le découpage et les relations entre sous-fonctions. Par exemple, l'utilisation de sémaphores binaires implique une relation d'exclusion mutuelle. De même, l'envoi et la réception de messages impliquent des relations de précédence.
La figure 3.20 présente un exemple de découpage en sous-fonctions et de déduction des relations entre les sous-fonctions. Par souci d'espace nous n'y présentons que l'analyse pour deux fonctions.
tf_1 envoie un message en utilisant Deposer_Lettre. Ce message est reçu par tf_2 (Recevoir_Lettre). Il est donc nécessaire que f_1^1 s'exécute avant f_2^1 d'où la contrainte de précédence.
f_1^2 et f_2^2 sont en exclusion mutuelle parce que tf_1 et tf_2 utilisent le même sémaphore S_2 pour se synchroniser.

III.2.c. Règles supplémentaires de découpage

Afin de formaliser quelques règles de découpage, nous notons $F(f_\alpha)$ l'ensemble des sous-fonctions de f_α.
Pour minimiser le nombre de sous-fonctions nous appliquons une règle de découpage qui consiste à ne découper que lorsque cela se justifie par une relation d'exclusion mutuelle (équation 3.2-

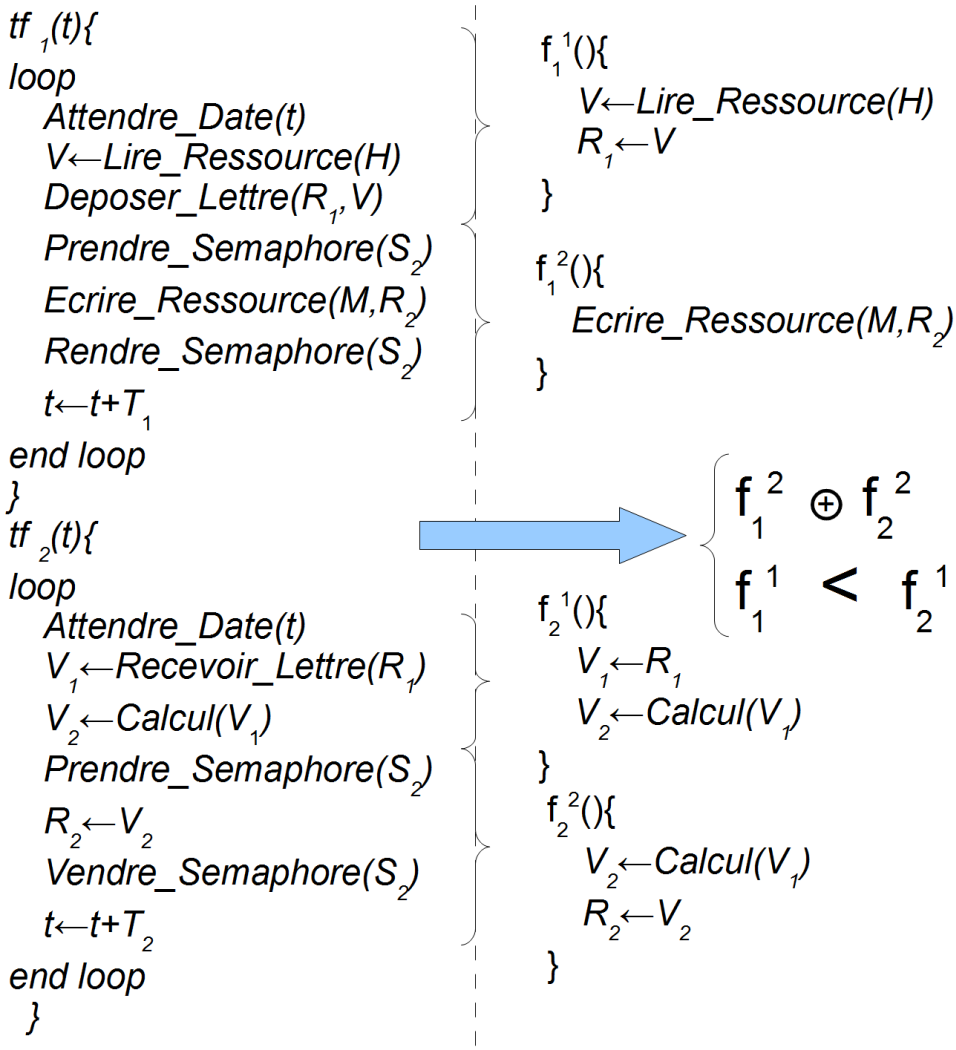


FIGURE 3.20 – Primitives temporelles et déduction d'un découpage sous-fonctions et des relations entre sous-fonctions

i) ou de précédence (équation 3.2-ii). En d'autres termes, si une fonction f_i comporte deux sous-fonctions successives f_i^j et f_i^{j+1} alors :
- soit il existe une sous-fonction f_k^l qui est en exclusion mutuelle avec f_i^j et pas avec f_i^{j+1} ou l'inverse (en exclusion mutuelle avec f_i^{j+1} et pas avec f_i^j) ;
- soit il existe une sous-fonction f_k^l telle que f_i^j la précède et pas f_i^{j+1}.

$$f_i^j, f_i^{j+1} \in F(f_i) \Rightarrow \begin{cases} \exists k \neq i, & f_k^l \in F(f_k) & & & & & \\ ((f_i^j \oplus f_k^l) & \wedge & \neg(f_i^{j+1} \oplus f_k^l)) & \vee & ((f_i^{j+1} \oplus f_k^l) & \wedge & \neg(f_i^j \oplus f_k^l)) \quad (i) \\ ((f_i^j \prec f_k^l) & \wedge & \neg(f_i^{j+1} \prec f_k^l)) & & & & \quad (ii) \end{cases} \tag{3.2}$$

III.3. Mise en œuvre avec ressources critiques et contraintes de précédence

III.3.a. Description d'applications à mettre en œuvre

Comme nous l'avons présenté dans la section III.2, les techniques de recherche de séquences d'ordonnancement hors-ligne pour prendre en compte les ressources critiques divisent la partie fonctionnelle d'une tâche en sous-fonctions. Pour prendre en compte ce découpage fonctionnel nous définissons la notion de *modèle fonctionnel annoté.*

Définition 3.1 (Modèle fonctionnel annoté de tâche périodique)**.** *En présence de ressources critiques ou de contraintes de précédence, une tâche périodique annotée τ_i^{Annote} est caractérisée par $< r_i, [C_i^{min} : C_i^{max}], D_i, T_i | f_i^1 = [C_i^{min_1} : C_i^{max_1}], \ldots, f_i^{n_i} = [C_i^{min_{n_i}} : C_i^{max_{n_i}}] >$:*
- *r_i est la date d'arrivée de la première instance de la tâche τ_i, encore appelée date de première activation ou offset ;*
- *C_i^{min} est la durée d'exécution au meilleur cas, elle spécifie un minorant du temps d'exécution de chaque instance de la tâche τ_i ;*
- *C_i^{max} est la pire durée d'exécution, elle spécifie un majorant du temps d'exécution de chaque instance de la tâche τ_i ;*
- *D_i est l'échéance relative ou délai critique, elle dénote la durée séparant l'arrivée d'une instance et son échéance ;*
- *T_i est la période, c'est l'intervalle de temps qui sépare l'arrivée de deux instances successives de τ_i ;*
- *$f_i^1, \ldots, f_i^{n_i}$ est la liste séquentielle des sous-fonctions modélisées par la tâche, avec $C_i^{min_1}, \ldots, C_i^{min_{n_i}}$ et $C_i^{max_1}, \ldots, C_i^{max_{n_i}}$ leurs durées d'exécution aux meilleurs et aux pires cas.*

De plus, nous considérons que la somme des durées d'exécution ($C_i^{min_k}$ et $C_i^{max_k}$) des sous-fonctions est égale à la durée d'exécution de la tâche ($\sum_k C_i^{min_k} = C_i^{min}$ et $\sum_k C_i^{max_k} = C_i^{max}$). Un système de tâches τ^{Annote} est dit annoté si toutes ses tâches sont annotées.
Nous définissons aussi le scénario d'exécution annoté.

Définition 3.2 (Scénario d'exécution annoté). *Un scénario d'exécution théorique annoté* $Se^{Annote}_{theorique}$ *est une suite de triplet* $Se^{Annote}_{theorique} = \{(start_1, end_1, f^{\mu(1)}_{\alpha(1)\beta(1)}), \ldots, (start_{|Se^{Annote}|}, end_{|Se^{Annote}|}, f^{\mu(|Se^{Annote}|)}_{\alpha(|Se^{Annote}|)\beta(|Se^{Annote}|)})\}$, *où* $f^{\mu(i)}_{\alpha(i)\beta(i)}$ *indique l'instance de sous-fonction qui s'exécute entre* $start_i$ *et* end_i.

Dans la suite nous supposons que nous disposons de systèmes de tâches annotés avec les contraintes associées aux sous-fonctions.
Nous définissons aussi le début et la fin d'exécution d'une instance de sous-fonction :
- le début de l'exécution d'une instance est le minimum des dates de début des blocs d'exécution de l'instance (équation 3.3-(i)) ;
- la fin de l'exécution d'une instance est le maximum des dates de fin des blocs d'exécution de l'instance (équation 3.3-(ii)).

$$\begin{cases} Be(f^k_{ij}) & = & min_{\alpha(p),\beta(p),\mu(p)=i,j,k}(start_p) & (i) \\ Cp(\tau^k_{ij}) & = & max_{\alpha(p),\beta(p),\mu(p)=i,j,k}(end_p) & (ii) \end{cases} \tag{3.3}$$

L'équation 3.4 présente les conditions à respecter par un scénario d'exécution annoté pour être valide. Le scénario d'exécution théorique doit respecter les contraintes temporelles (équations 3.4-(i) et 3.4-(ii)), les contraintes de précédence (équation 3.4-(iii)), les contraintes d'exclusion mutuelle (équation 3.4-(iv)). Le scénario annoté doit également être prévu pour les WCET (équation 3.4-(v)).

$$\begin{cases} \forall j \in [1, |Se^{Annote}_{theorique}|] & r_{\alpha(j)\beta(j)} \leq start_j & (i) \\ \forall j \in [1, |Se^{Annote}_{theorique}|] & d_{\alpha(j)\beta(j)} \geq end_j & (ii) \\ f^l_i \prec f^p_j \Rightarrow \forall k & Cp(f^l_{ik}) \leq Be(f^p_{jk}) & (iii) \\ f^l_i \oplus f^p_j \Rightarrow \forall k, m & [Be(f^l_{ik}), Cp(f^l_{ik})) \cap [Be(f^p_{jm}), Cp(f^p_{jm})) = \emptyset & (iv) \\ i = 1, \ldots, |Se^{Annote}_{theorique}| & \forall (k,l,m), \sum_{\alpha(i)=k,\beta(i)=l,\mu(i)=m}(end_i - start_i) = C^{max_m}_k & (v) \end{cases} \tag{3.4}$$

Dans la section précédente (section III.2) nous avons vu que pour prendre en compte les ressources critiques et les contraintes de précédence, les techniques de calcul de scénarios d'exécution hors-ligne représentent chaque tâche comme un ensemble de sous-fonctions.
Pour prendre en compte cette modélisation, nous avons défini les notions de système de tâches annoté (τ^{Annote}) et de scénario d'exécution annoté ($Se^{Annote}_{theorique}$).
Cependant, nos techniques de mise en œuvre (section I) utilisent des systèmes de tâches à une seule fonction. Pour utiliser nos techniques de mise en œuvre dans un contexte d'utilisation de ressources critiques et de contrainte de précédence, il nous faut déterminer, le système de tâches τ à une fonction qui correspond à τ^{Annote} : *pour chaque tâche annotée* τ^{Annote}_i *quelles sont les tâches* τ_j *(date d'activation, durées d'exécution, échéance, période, fonction modélisée) qui correspondent ?*
En d'autres termes, nous devons déterminer les tâches annotées à scinder et aussi la répartition des sous-fonctions entre les tâches.

La section III.3.b présente une première solution au problème de répartition des sous-fonctions entre tâches. La section III.3.c présente une solution de répartition qui limite le nombre tâches créées. Pour finir dans la section III.3.d nous montrons que nos solutions de répartition des sous-fonctions permettent d'avoir des mises en œuvres valides.

III.3.b. Première solution de répartition des sous-fonctions entre tâches

Dans le chapitre 2-section IV.4, nous avons prouvé que les mises en œuvre flexibles et inflexibles respectent les contraintes temporelles et la topologie dictée par le scénario d'exécution théorique. La particularité dans un contexte de ressources critiques et de contraintes de précédence est que les durées d'exécution plus courtes peuvent entrainer un non respect des contraintes de précédence ou d'exclusion mutuelle (voir exemple 3.17).

Système de tâches annoté	$\tau^{Annote} = \{\tau_1^{Annote} =< (0,[1:2],8,8\|f_1>,$ $\tau_2^{Annote} =< 3,[2:3],5,8\|f_2^1=[1:1],f_2^2=[1:2]>,$ $\tau_3^{Annote} =< 0,[2:4],16,16\|f_3>\}$
Précédences	Exclusions
$f_2^1 \prec f_1$	$f_3 \oplus f_2^2$
	Système de tâches à mettre en œuvre $\tau = \{\tau_1 =< (0,[1:2],8,8\|f_1>,$ $\tau_2 =< 3,[1:1],5,8\|f_2^1>,$ $\tau_3 =< 0,[2:4],16,16\|f_3>$ $\tau_4 =< 0,[1:2],16,8\|f_2^2>\}$

TABLE 3.8 – Exemple de scénario d'exécution ne prévoyant pas de préemption pour des tâches utilisant des ressources critiques

Une première solution est de créer autant de tâches qu'il y a de sous-fonctions : 1 sous-fonction≡ une tâche.
Le tableau 3.8 présente un exemple de système de tâches annoté. La tâche τ_2^{Annote} qui contient deux sous-fonctions est scindée en deux tâches τ_2 et τ_4.
En créant autant de tâches qu'il y a de sous-fonctions, nous garantissons que quelque soit le scénario d'exécution théorique annoté valide, une mise en œuvre flexible ou inflexible est également valide. Cela vient du fait que les mises en œuvre respectent l'ordre d'exécution des tâches et donc des sous-fonctions. Cependant, créer systématiquement autant de tâches qu'il y a de sous-fonctions n'est pas toujours nécessaire.
Considérons l'exemple du système de tâches et du scénario d'exécution du tableau 3.9. Pour cet exemple, une seule tâche annotée a plusieurs sous-fonctions : il s'agit de τ_2^{Annote}. Comme l'exécution ne prévoit la préemption d'aucune tâche, il n'y a aucune tâche à scinder.
De manière plus générale, dans un contexte ou le scénario d'exécution ne prévoit pas de préemption, le système de tâches annoté peut être gardé tel quel (sans scission). Cela se justifie

Système de tâches annoté	$\tau^{Annote} = \{\tau_1^{Annote} =< (0, [1:2], 8, 8 \mid f_1 >,$ $\tau_2^{Annote} =< 3, [2:3], 5, 8 \mid f_2^1 = [1:1], f_2^2 = [1:2] >,$ $\tau_3^{Annote} =< 0, [2:4], 16, 16 \mid f_3 >\}$
Précédences	Exclusions
$f_2^1 \prec f_1$	$f_3 \oplus f_2^2$
Scénario d'exécution théorique	$Se_{theorique}^{Annote} = \{(0, 2, f_{11}), (3, 4, f_{21}^1), (4, 6, f_{21}^2)$ $(6, 10, f_{31}), (10, 12, f_{12}), (12, 13, f_{22}), (13, 15, f_{22})\}$ $\lvert Se_{theorique}^{Annote} \rvert = 7\,;\ H = 16$
	Système de tâches à mettre en œuvre $\tau = \{\tau_1 =< (0, [1:2], 8, 8 \mid f_1 >,$ $\tau_2 =< 3, [2:3], 5, 8 \mid f_2 >,$ $\tau_3 =< 0, [2:4], 16, 16 \mid f_3 >\}$

TABLE 3.9 – Exemple de scénario d'exécution ne prévoyant pas de préemption pour des tâches utilisant des ressources critiques

par le fait que les sous-fonctions de chaque tâche annotée s'exécuteront sur un seul bloc d'exécution. Dans ce contexte, l'exécution d'une sous-fonction avec une durée effective $C_i^{eff_j}$ (avec $C_i^{min_j} \leq C_i^{eff_j} < C_i^{max_j}$) n'entrainera pas le non respect d'une contrainte d'utilisation de ressources (précédence, exclusion mutuelle).

Afin de minimiser les coûts temporels de mise en œuvre nous devront limiter le nombre de tâches annotées à scinder. La section III.3.c présente une solution prenant en compte le scénario d'exécution annoté pour minimiser le nombre de tâches à scinder.

III.3.c. Répartition des sous-fonctions minimisant le nombre de tâches obtenues

Pour présenter notre solution qui minimise le nombre de tâches à scinder, nous étendons certains concepts aux tâches :

- une tâche précède une autre tâche si il existe deux sous-fonctions liées par une contrainte de précédence (équation 3.5-(i)) ;
- une tâche est en exclusion mutuelle avec une autre tâche si il existe deux sous-fonctions liées par une contrainte d'exclusion mutuelle (équation 3.5-(ii)) ;
- le début de l'exécution d'une instance de tâche est le minimum des dates de début des blocs d'exécution de l'instance (équation 3.5-(iii)) ;
- la fin de l'exécution d'une instance de tâche est le maximum des dates de fin des blocs d'exécution de l'instance (équation 3.3-(iv)).

$$\begin{cases} \tau_i \prec \tau_j & \text{si} & \exists (k,l), f_i^k \prec f_j^l & (i) \\ \tau_i \oplus \tau_j & \text{si} & \exists (k,l), f_i^k \oplus f_j^l & (ii) \\ Be(\tau_{ij}) & = & min_{\alpha(p),\beta(p)=i,j}(start_p) & (iii) \\ Cp(\tau_{ij}) & = & max_{\alpha(p),\beta(p)=i,j}(end_p) & (iv) \end{cases} \tag{3.5}$$

Intuitivement la scission d'une tâche annotée en plusieurs vise à ce que, lors d'une mise en œuvre, une durée d'exécution effective inférieure à la durée maximale n'ait pas pour conséquence le non respect d'une contrainte de précédence ou d'exclusion mutuelle.
Plus formellement, le système de tâche obtenu après les différentes scissions doit respecter l'équation 3.6 :

$$\begin{cases} \tau_i \prec \tau_j & \Rightarrow & \forall k & Cp(\tau_{ik}) \leq Be(\tau_{jk}) & (i) \\ \tau_i \oplus \tau_j & \Rightarrow & \forall k,p & [Be(\tau_{ik}), Cp(\tau_{ik})) \cap [Be(\tau_{jp}), Cp(\tau_{jp})) = \emptyset & (ii) \end{cases} \tag{3.6}$$

L'équation 3.6 pour être respectée nécessite de scinder les tâches dès lors que la préemption d'une tâche annotée est prévue par une autre sous-fonction avec une contrainte (précédence et exclusion).
Lorsqu'un scénario d'exécution théorique annoté prévoit la préemption d'une tâche annotée il y a deux possibilités, soit la préemption est prévue pour se faire entre deux sous-fonctions, soit elle est prévue pour concerner la même sous-fonction.
Considérant les deux types de préemption nous illustrons nos propos à travers des exemples.

Contexte 1 : Préemptions prévues entre sous-fonctions Si la préemption est prévue entre deux sous-fonctions f_k^l et f_k^{l+1}, il faut vérifier si parmi les sous-fonctions qui s'exécutent entre les deux sous-fonctions certaines ont des contraintes (précédence, exclusion mutuelle) avec f_k^{l+1}. Si c'est le cas, il faut scinder τ_k^{Annote} en deux nouvelles tâches :
– la première nouvelle tâche avec comme fonction exécutée la suite séquentielle $f_k^1 \dots f_k^l$;
– la deuxième nouvelle tâche avec comme fonction exécutée la suite séquentielle $f_k^{l+1} \dots f_k^{n_k}$.
Pour illustrer nos propos nous utilisons l'exemple du système de tâches annoté du tableau 3.10. Le scénario d'exécution théorique annoté prévoit la préemption de τ_{11} entre f_{11}^1 et f_{11}^2. La seule fonction qui s'exécute entre f_{11}^1 et f_{11}^2 est f_{21}. Il existe une contrainte de précédence entre f_2 et f_1^2 : $f_2 \prec f_1^2$. Il est donc nécessaire de scinder τ_1^{Annote} :
* $\tau_1^{Annote} =< 0, [1:2], 20, 20 | f_1^1 >$
* $\tau_5^{Annote} = \tau_5 =< 0, [2:4], 20, 20 | f_1^2 >$
Après la scission de τ_1^{Annote} il n'y a plus de préemption et nous avons donc le système de tâches à mettre en œuvre. Il faut également ajouter une contrainte de précédence :$\tau_1 \prec \tau_5$.

Contexte 2 : Préemption de sous-fonction prévue Si la préemption est prévue au sein d'une sous-fonction f_k^l, alors nous savons qu'il n'y a pas de sous-fonction qui s'exécute entre les deux blocs d'exécution de f_k^l (c'est-à-dire pendant la préemption) et qui ait un contrainte de précédence ou d'exclusion avec f_k^l (voir équation 3.4 de validité d'un scénario annoté). Il faut

Système de tâches	$\tau^{Annote} = \{\tau_1^{Annote} =< 0, [3:6], 20, 20 \mid f_1^1 = [1:2], f_1^2 = [2:4] >,$ $\tau_2^{Annote} =< 0, [1:3], 20, 20 \mid f_2 >$ $\tau_3^{Annote} =< 0, [2:6], 20, 20 \mid f_3^1 = [1:3], f_3^2 = [1:3] >$ $\tau_4^{Annote} =< 0, [3:5], 20, 20 \mid f_4^1 = [1:2], f_4^2 = [2:3] >\}$	
	Précédences	Exclusions
	$f_1^1 \prec f_2$ $f_2 \prec f_1^2$ $f_1^1 \prec f_3^1$ $f_2 \prec f_3^2$	$f_3^1 \oplus f_4^1$ $f_3^2 \oplus f_4^1$ $f_3^1 \oplus f_4^2$ $f_3^2 \oplus f_4^2$
	$Se_{theorique}^{Annote} = \{(0, 2, f_{11}^1), (2, 5, f_{21}), (5, 9, f_{11}^2), (9, 12, f_{31}^1),$ $(12, 15, f_{31}^2), (15, 17, f_{41}^1), (17, 20, f_{41}^2)\}$	
	Système de tâches à mettre en œuvre $\tau = \{\tau_1 =< 0, [1:2], 20, 20 \mid f_1^1 >,$ $\tau_2 =< 0, [1:3], 20, 20 \mid f_2 >,$ $\tau_3 =< 0, [2:6], 20, 20 \mid f_3 >,$ $\tau_4 =< 0, [3:5], 20, 20 \mid f_4 >,$ $\tau_5 =< 0, [2:4], 20, 20 \mid f_1^2 >\}$	

TABLE 3.10 – Exemple de système de tâches annoté

cependant vérifier parmi les sous-fonctions qui s'exécutent pendant la préemption si certaines ont des contraintes avec les sous-fonctions $f_k^{l+1} \dots f_k^{n_k}$. Si c'est le cas, il faut scinder τ_k^{Annote} en deux nouvelles tâches. Notons f_k^{ϵ} la sous-fonction qui présente une contrainte avec une sous-fonction qui s'exécute pendant la préemption. Les deux nouvelles tâches crée par scission de τ_k^{Annote} sont telles que :

- la première nouvelle tâche a comme fonction exécuter la suite séquentielle $f_k^1 \dots f_k^{\epsilon-1}$;
- la deuxième nouvelle tâche avec comme fonction exécuter la suite séquentielle $f_k^{\epsilon} \dots f_k^{n_k}$.

Pour illustrer ce point considérons le système de tâches annoté du tableau 3.11. Le scénario d'exécution prévoit une préemption entre f_{11}^1 et f_{11}^2 et une autre préemption est prévue pendant l'exécution de f_3^1. En appliquant la rêgle du paragraphe sur la préemption prévue entre sous-fonctions nous scindons τ_1^{Annote} en deux tâches.

Il ne reste plus qu'à traiter la préemption de f_3^1. Les sous-fonctions qui s'exécutent pendant la préemption sont f_2 et f_1^2. La contrainte $f_2 \prec f_3^2$ nous oblige alors à scinder τ_3^{Annote} en :

* $\tau_3^{Annote} =< 0, [1:3], 20, 20 \mid f_3^1 >$
* $\tau_6^{Annote} =< 0, [1:3], 20, 20 \mid f_3^2 >$

Après la scission de τ_3^{Annote} il n'y a plus de préemption à traiter et nous avons le système de tâches à mettre en œuvre.

Système de tâches	$\tau^{Annote} = \{\tau_1^{Annote} =< 0, [3:6], 20, 20 \mid f_1^1 = [1:2], f_1^2 = [2:4] >,$ $\tau_2^{Annote} =< 0, [1:3], 20, 20 \mid f_2 >$ $\tau_3^{Annote} =< 0, [2:6], 20, 20 \mid f_3^1 = [1:3], f_3^2 = [1:3] >$ $\tau_4^{Annote} =< 0, [3:5], 20, 20 \mid f_4^1 = [1:2], f_4^2 = [2:3] >\}$	
	Précédences	Exclusions
	$f_1^1 \prec f_2$ $f_2 \prec f_1^2$ $f_1^1 \prec f_3^1$ $f_2 \prec f_3^2$	$f_3^1 \oplus f_4^1$ $f_3^2 \oplus f_4^1$ $f_3^1 \oplus f_4^2$ $f_3^2 \oplus f_4^2$
	$Se_{theorique}^{Annote} = \{(0, 2, f_{11}^1), (2, 4, f_{31}^1), (4, 7, f_{21}), (7, 11, f_{11}^2),$ $(11, 12, f_{31}^1), (12, 15, f_{31}^2), (15, 17, f_{41}^1), (17, 20, f_{41}^2)\}$	
	$\tau^{Annote} = \{\tau_1^{Annote} =< 0, [1:2], 20, 20 \mid f_1^1 >,$ $\tau_2^{Annote} =< 0, [1:3], 20, 20 \mid f_2 >,$ $\tau_3^{Annote} =< 0, [2:6], 20, 20 \mid f_3^1 = [1:3], f_3^2 = [1:3] >,$ $\tau_4^{Annote} =< 0, [3:5], 20, 20 \mid f_4^1 = [1:2], f_4^2 = [2:3] >,$ $\tau_5^{Annote} =< 0, [2:4], 20, 20 \mid f_1^2 >\}$	
	Système de tâches à mettre en œuvre $\tau = \{\tau_1 =< 0, [1:2], 20, 20 \mid f_1^1 >,$ $\tau_2 =< 0, [1:3], 20, 20 \mid f_2 >,$ $\tau_3 =< 0, [1:3], 20, 20 \mid f_3^1 >,$ $\tau_4 =< 0, [3:5], 20, 20 \mid f_4 >,$ $\tau_5 =< 0, [2:4], 20, 20 \mid f_1^2 >\}$ $\tau_6 =< 0, [1:3], 20, 20 \mid f_3^2 >\}$	

TABLE 3.11 – Exemple de système de tâches à mettre en œuvre obtenue après analyse du scénario d'exécution annoté

III.3.d. Validité de mise en œuvre

Dans cette section nous montrons que les mises en œuvre flexibles et inflexibles restent valides dans un contexte de contrainte de précédence ou d'exclusion mutuelle. Nous énonçons la proposition 3.3.

Proposition 3.3 (Mise en œuvre avec des ressources critiques)**.** *En présence de ressources critiques et de contraintes de précédence, la mise en œuvre inflexible (respectivement flexible), du scénario d'exécution valide d'un système de tâches bien calibré et qui respecte l'équation 3.6, est valide.*

Preuve La preuve de la validité des mises en œuvre vient du respect des dates d'activation et des échéances par les mises en œuvre inflexibles (respectivement flexibles) : voir preuves des

propositions 2.7 et 2.8. Le respect de la topologie (équations 2.6-(i), 2.7-(i), 2.8) et l'équation de répartition des sous-fonctions (équation 3.6) entrainent que les contraintes de précédences et d'exclusions mutuelles sont conservées par les mise en œuvre. Au total, les contraintes temporelles (activation, échéance) et les contraintes non temporelles (précédence, exclusion mutuelle) sont respectées par les mise en œuvre : les mises en œuvre sont donc valides.

IV. Conclusion

Ce chapitre nous a permis de présenter des techniques de mises en œuvre d'ordonnancement hors-ligne dans un contexte de tâches indépendantes.
Nous avons pu établir une comparaison en utilisant des critères qualitatifs (robustesse) et quantitatifs (coûts temporels, spatial). Nous avons montré également comment utiliser nos techniques dans un contexte d'utilisation de ressources critiques ou avec des contraintes de précédence. Ainsi nous avons montré que l'utilisation de nos techniques nécéssite parfois de redéfinir le système de tâches.

Chapitre

4 Mise en œuvre POSIX

Sommaire

Résumé

Dans ce chapitre nous présentons comment automatiser la mise en œuvre effective de nos techniques. Pour cela nous expliquons dans un premier temps, comment représenter une séquence d'ordonnancement hors-ligne. Et par la suite nous décrivons comment transformer une séquence en code d'application. Nous présentons le RTOS Xenomai que nous avons utilisé comme support de tests. Nous présentons également un outil d'observation d'exécution sur Xenomai et une étude comparative des coûts de mise en œuvre.

Les chapitres précédents nous ont permis de montrer que pour une séquence d'ordonnancement donnée, de nombreuses techniques de mise en œuvre sont possibles. Ce chapitre nous permet de traiter de la mise en œuvre effective (langage machine).
Nous commençons par discuter de la traduction de nos algorithmes et donc des instructions de notre API abstraite en POSIX (section I).
Par la suite dans la section II nous présentons des outils qui permettent de faire de la génération de code.
Ensuite, nous présentons un système d'exploitation temps réel (section III.2) qui nous sert de support d'expérimentations. Enfin, nous utilisons un observateur (section IV) afin de recueillir et analyser les scénarios des exécutions effectives sur le système d'exploitation temps réel. La section III.3 présente l'évaluation pratique des techniques de mise en œuvre en utilisant notre modèle de coût.

I. Mise en œuvre POSIX

Une application est portable lorsqu'elle peut s'exécuter sur plusieurs plate-formes logicielles (système d'exploitation), ou matérielles.
L'idéal est que cette exécution se fasse sans nécessiter la moindre adaptation du code source. La propriété de portabilité est d'autant plus importante que les plate-formes surtout matérielles évoluent très rapidement. Sans portabilité, cette évolution rendrait obsolète très rapidement toute application.
POSIX (Portable Operating System Interface) est un standard qui vise la portabilité des codes sources des applications.
POSIX décrit un ensemble de services (ou d'API : interface de programmation) de références que les systèmes d'exploitation doivent offrir aux développeurs d'applications. Ainsi, une application qui est écrite avec ces services de référence peut s'exécuter sur tout système d'exploitation les proposant.
Dans notre cas nous ne sommes intéressés que par trois parties de POSIX à savoir :

- POSIX.1 (operating system core services) qui décrit l'interface basique d'un système d'exploitation : création et contrôle de processus, signaux, exceptions flottant, violation de segmentation, instructions illégales, erreurs de bus, timers, opérations sur les fichiers et les répertoires, pipes, librairie standard en C, interface d'E/S ;
- POSIX.1b (real-time extensions) qui décrit les fonctionnalités suivantes : ordonnancement à priorité, signaux temps réel, horloges et timers, sémaphores, communication par messages, memoires partagées, E/S asynchrone et synchrone, verrouillage mémoire ;
- POSIX.1c (thread extensions) qui décrit les fonctionnalités suivantes : multi-thread dans un processus ; création de thread, contrôle, suppression, ordonnancement de thread, synchronisation, gestion des signaux.

Dans la suite nous présentons dans la section I.1 les objets et les instructions POSIX que nous utilisons, nous présentons en particulier les primitives POSIX équivalentes aux primitives de

notre API abstraite. Étant plus intéressés par le fait de montrer comment se traduisent de façon générale nos algorithmes, nous présentons juste les primitives et pas l'ensemble des attributs utilisés par chaque primitive. Le lecteur intéressé par plus de détails pourra consulter [Gal95]. Dans la section I.2 nous proposons des structures de données pour représenter les scénarios d'exécution à suivre.

I.1. Traduction des algorithmes

La plupart des objets que nous utilisons trouvent leurs équivalents POSIX. Dans la suite nous discutons successivement de la gestion des dates (section I.1.a), de la synchronisation (section I.1.b) et de la gestion des processus (section I.1.c).

I.1.a. Gestion des dates

Pour manipuler les dates POSIX fournit la structure de données struct timespec et la détermination d'une date se fait par l'intermédiaire de la primitive clock_gettime. La primitive sleep permet de suspendre l'exécution d'un programme pendant un certain temps, et clock_nanosleep suspend l'exécution jusqu'à une daté donnée.

I.1.b. Synchronisation

Pour la technique de mise en œuvre par synchronisation collective, nous utilisons des sémaphores essentiellement parce que c'est le type de données le plus utilisé pour faire de la synchronisation de processus. L'initialisation d'un sémaphore POSIX (sem_t) se fait par l'intermédiaire de la primitive sem_init. Les opérations de prise et de relâchement sont respectivement sem_wait et sem_post.

I.1.c. Gestion des processus

POSIX prévoit deux types de processus que sont les processus légers ou thread (pthread_t) et les processus lourds (pid_t). POSIX prévoit en particulier qu'un processus lourd contient un (par défaut le thread main()) ou plusieurs processus légers. Les processus légers utilisent donc le contexte d'exécution du processus lourd auquel ils appartiennent. Ils peuvent ainsi partager des variables globales tout en ayant un espace mémoire privé (variables locales) qu'est la pile. Du fait que plusieurs processus légers partagent le contexte d'exécution du processus lourd auquel ils appartiennent, la création des processus légers (utilisation de la primitive pthread_create) nécessite moins de mémoire et de temps processeur que la création des processus lourds (utilisation de primitive fork). Pour toutes ces raisons nous utiliserons le plus souvent les processus légers comme l'équivalent des processus de notre API abstraite.
Il y a cependant une exception pour la mise en œuvre de répartition par gestion d'états. En effet, la suspension et la reprise d'un processus léger se font par l'utilisation de la primitive pthread_kill qui envoie un signal à un processus léger. POSIX spécifie que le signal de suspension

API abstraite	Équivalent POSIX
Date	struct timespec
Lire_Date	clock_gettime
Attendre_Date	clock_nanosleep
Attendre_Delai	sleep
Sémaphore	sem_t
Créer_Semaphore	sem_init
Prendre_Semaphore	sem_wait
Vendre_Semaphore	sem_post
Processus	pthread_t pid_t
Créer_Processus	pthread_create ou fork
Supprimer_Processus	pthread_cancel ou kill
Arrêter_Processus	pthread_kill ou kill
Continuer_Processus	pthread_kill ou kill
Priorité_Processus	pthread_setprio ou kill

TABLE 4.1 – Équivalents POSIX d'objets et de primitives de l'API abstraite

doit affecter tout le processus lourd. Dans notre cas le processus répartiteur serait aussi suspendu et ne pourrait plus gérer l'exécution des autres processus. Il s'ensuit que pour la mise en œuvre de répartition par gestion d'état, nous utiliserons des processus lourds pour implémenter chaque processus de notre API abstraite.

Techniques de mise en œuvre	Nombre de processus lourds	Nombre de processus légers
Mise en œuvre par une tâche	1	2
Répartiteur non concurrent à un processus	1	2
Répartiteur non concurrent à deux processus	1	3
Répartition par gestions d'états	$1 + \|\tau\|$	$1 + \|\tau\|$
Répartition par affectation de priorité	1	$2 + \|\tau\|$
Attente coopérative de dates	1	$1 + \|\tau\|$
Synchronisation collective	1	$1 + \|\tau\|$

TABLE 4.2 – Processus légers et lourds utilisés par les techniques de mise en œuvre d'ordonnancement hors-ligne ; $|\tau|$: nombre de tâches périodiques

Les tableaux 4.1 et 4.2 résument la traduction en POSIX de nos algorithmes. Le tableau 4.1 présente les types d'objets et les primitives à utiliser pour chaque type. Le tableau 4.2 présentent pour chaque mise en œuvre le nombre de processus lourds et légers à utiliser.

I.2. Objets structurés

Les techniques de mise en œuvre que nous avons décrites dans le chapitre 3 utilisent le plus souvent des objets structurés de type tableaux. Cependant, la forme (contenu) des tableaux n'est pas la même pour toutes les techniques. Dans cette section nous proposons différents objets structurés pour représenter les scénarios d'exécution.

La figure 4.1 présente les différentes structures que nous proposons.

Pour faciliter la représentation externe des dates, au lieu d'utiliser le type structuré struct timespec nous utilisons le type time_t qui est un entier long. De plus la représentation des dates sous forme d'entier long est celle utilisée par les techniques de calcul de séquences d'ordonnancement hors-ligne.

Pour utiliser la technique de répartition non concurrente à un et deux processus (chapitre 3 sections I.2 et I.3), nous avons besoin d'une structure qui permette de référencer chaque fonction à exécuter et la date à laquelle celle-ci doit être exécutée. La figure 4.1-A présente la structure que nous proposons.

Le processus répartiteur, utilisé dans la technique de répartition par affectation de priorités (chapitre 3 section I.4), a besoin, pour modifier les priorités des autres processus, de connaître les références de ces processus ainsi que les dates de modification. La figure 4.1-B présente la déclaration de la structure que nous utilisons.

Pour la mise en œuvre de répartition par gestion d'état (chapitre 3 section I.5), le processus répartiteur qui envoie les signaux de suspension et de reprise doit disposer des identifiants de chaque processus. Nous proposons la structure de données de la figure 4.1-E pour contenir les informations nécessaire au répartiteur.

Dans la technique d'attente coopérative de dates (chapitre 3 section I.7), chaque instance de processus doit connaitre la date à laquelle elle doit commencer son exécution. Nous utilisons un tableau de dates de début d'exécution pour chaque processus. La figure 4.1-C présente cette structure.

Pour utiliser la technique de mise en œuvre par synchronisation collective (chapitre 3 section I.6), nous avons besoin d'une structure qui renseigne chaque instance de processus sur le sémaphore à attendre avant l'exécution des instructions fonctionnelles et le sémaphore à relâcher à la fin de l'exécution. La figure 4.1-D présente une structure pouvant être utilisée pour les mises en œuvre par synchronisation collective.

Dans cette section nous avons posé les bases de la production de code POSIX mettant en œuvre des ordonnancements calculés hors-ligne.

Afin de faciliter les mises en œuvre effectives, nous nous intéressons, dans la section II, à l'automatisation de la production de code POSIX.

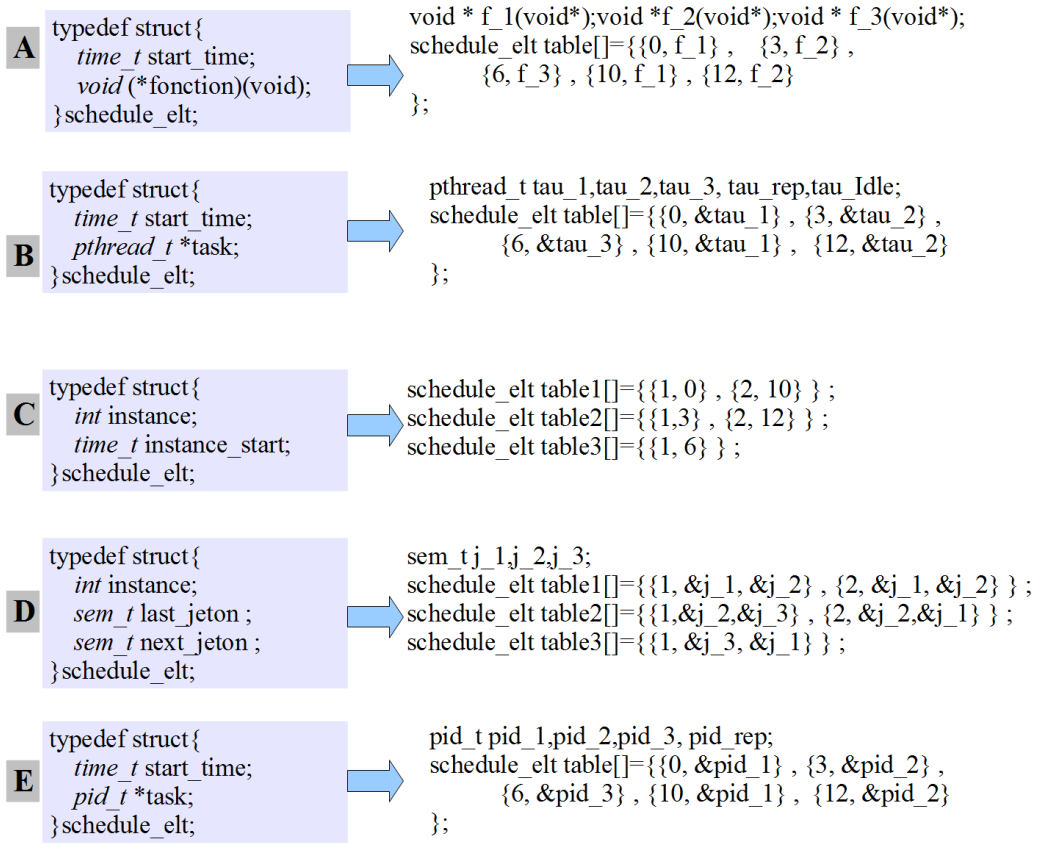


FIGURE 4.1 – Déclarations de structures de données permettant de représenter des tables d'ordonnancement. A : répartition non concurrente. B : répartition à affectation de priorité. C : attente coopérative de dates. D : synchronisation collective. E : répartition par gestion d'états.

II. Automatisation de mise en œuvre POSIX

Pour automatiser la mise en œuvre POSIX d'ordonnancement hors-ligne, nous avons besoin d'outils qui permettent de faire de la génération de code. De tels outils impliquent dans un premier temps une représentation (modèle temporel, scénarios d'exécution) de l'application et dans un second temps des programmes qui transforment cette représentation en code POSIX. Des approches similaires de génération de code, telles que celles utilisant AADL (Architecture Analysis and Design Language) [HZPK07], existent déjà. Cependant, à notre connaissance, la représentation des scénarios d'exécution n'est pas prise en compte par AADL.
Même si AADL autorise son extension, nous n'utiliserons pas cette approche. En effet notre objectif est de montrer que l'automatisation des mises en œuvre effectives d'ordonnancement hors-ligne est réalisable. Pour cela, nous proposons des prototypes et des outils facilitant l'activité de mise en œuvre de scénarios d'exécution calculés hors-ligne.
D'autres travaux pourront par la suite explorer l'utilisation d'AADL pour l'automatisation des mise en œuvre d'ordonnancement hors-ligne.
Nous avons choisi d'utiliser des outils de l'ingénierie dirigée par les modèles [Ouh13].
Cela consiste à :
- définir les objets et les propriétés des objets dont nous avons besoin (section II.1) ;
- définir des patrons de mise en œuvre qui permettent de transformer nos objets en code écrit en langage C (section II.2).

Cette approche est plus facile à mettre en place parce que de nombreux outils existent. Pour définir les objets et les propriétes des objets nous utilisons l'outil Eclipse Modeling Framework ([EMF]). Pour écrire des patrons de mise en œuvre nous utilisons l'outil Acceleo [Acc].
Dans la suite nous ne faisons pas une description détaillée des outils utilisés. Nous présentons de manière globale et par des exemples l'utilisation de ces outils pour automatiser la mise en œuvre de séquences d'ordonnancement hors-ligne.

II.1. Description des applications

Dans cette section, nous présentons les concepts nécessaires à la description d'une application à mettre en œuvre lorsqu'on est dans un contexte d'ordonnancement hors-ligne. Nous utilisons pour cela le formalisme du diagramme de classe UML (voir figure 4.2). Sur cette figure les objets sont représentés par des rectangles et les liens entre objets sont représentés par des lignes.
Chaque application est décrite par une liste :
- de tâches (Task) dont le modèle correspond à la définition 2.1 ;
- de fonctions (Function) exécutées par les tâches. Pour chaque fonction, son nom et les instructions exécutées doivent être connus ;
- de blocs d'exécution (SchedulEntry) caractérisés par des dates de début et de fin d'exécution avec une référence de la tâche à exécuter. L'ensemble des blocs d'exécution constitue le scénario d'exécution théorique (Shedule).

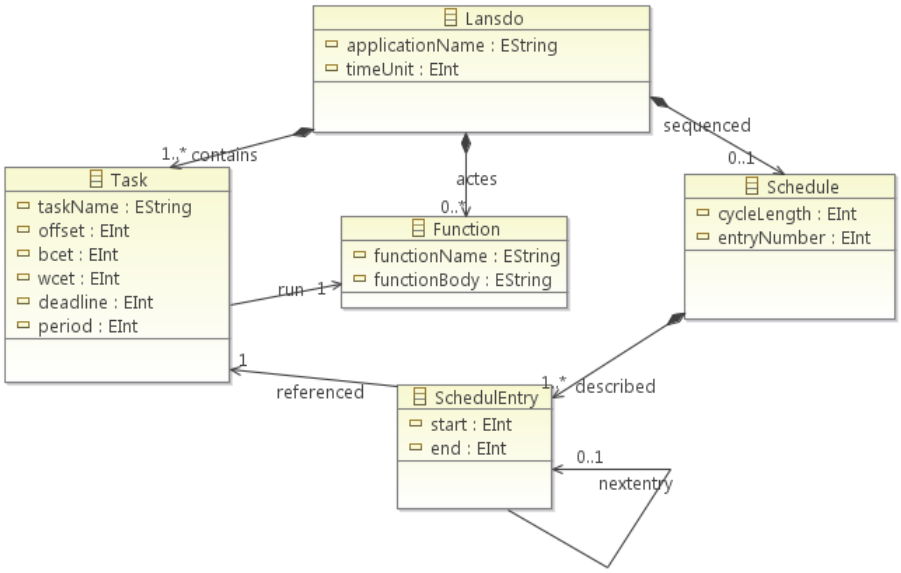


FIGURE 4.2 – Objets utilisés pour la description d'une application et d'un ordonnancement hors-ligne à mettre en œuvre

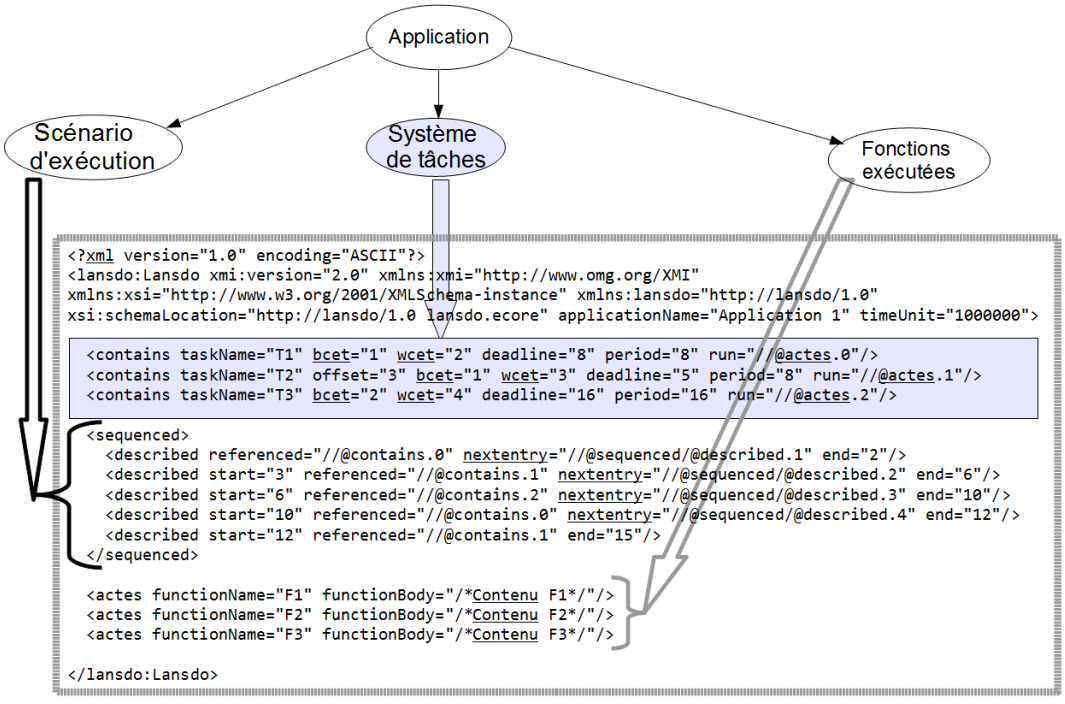


FIGURE 4.3 – Description d'une application à mettre en œuvre

En plus des objets (tâches, fonctions, blocs d'exécution) listés plus haut, chaque application (lansdo) est également caractérisée par un nom et par une unité de temps de discrétisation ut. Comme sous POSIX la plus petite échelle des dates est la nanoseconde, l'unité de discrétisation est exprimée en nanosecondes. Par exemple, si $ut = 1000000$ alors une tâche $\tau_1 =< 0, [3 : 5], 8, 8 | f_1 >$ est périodique de période $T_1 = 8000000ns$.

Nous présentons sur la figure 4.3 un exemple de fichier XML [1] qui décrit une application constituée de trois tâches ($T1$, $T2$, $T3$) qui exécutent trois fonctions ($F1$, $F2$, $F3$). Le scénario d'exécution théorique à mettre en œuvre est constitué de cinq blocs d'exécution.

II.2. Génération de code

La génération de code d'une application suivant l'une des techniques de mise en œuvre proposée dans le chapitre 3 se fait par l'intermédiaire de *patrons de mise en œuvre*. Un patron de mise en œuvre indique comment transformer la description d'une application (exemple de la figure 4.3) en un code POSIX. Pour cela, le patron intègre à la fois :

- du *code statique* : il s'agit de portions de code POSIX indépendantes de l'application à mettre en œuvre. C'est le cas par exemple des déclarations de bibliothèques qui restent les mêmes quelque soit l'application ;
- du *code dynamique* : il s'agit de portions de code POSIX qui dépendent de l'application à mettre en œuvre.

Par souci d'espace, nous ne présentons dans cette section (voir figure 4.4) qu'une partie d'un patron. En annexe C nous présentons des patrons complets.

Sur la figure 4.5-A, nous présentons une portion dynamique de patron de mise en œuvre. Cette portion dynamique parcourt les fonctions à exécuter (figure 4.5-B) et génère la partie fonctionnelle (figure 4.5-C) d'une application à mettre en œuvre.

La distinction entre code statique et code dynamique d'un patron de mise en œuvre est fonction de la technique de mise en œuvre. Comme les fonctions à exécuter ne sont pas identiques pour toutes les applications, la partie fonctionnelle est toujours générée dynamiquement. Les patrons des techniques qui utilisent des tables d'ordonnancement intègrent du code dynamique pour générer ces tables. Les autres parties du patrons (main() et temporelles) sont dynamiques lorsque les instructions dépendent des tâches ou du scénario d'exécution.

Par exemple dans la technique de mise en œuvre par un répartiteur non concurrent à un processus (chapitre 3 section I.2), la partie temporelle et le main() sont statiques alors que la table d'ordonnancement est dynamique.

Nous résumons dans le tableau 4.3, pour chaque technique, les parties statiques et dynamiques des patrons de mise en œuvre.

1. Dans l'approche d'ingénierie dirigée par les modèles on utilise le terme *méta-modèle* pour désigner le diagramme UML et le fichier XML est appelé *modèle*.

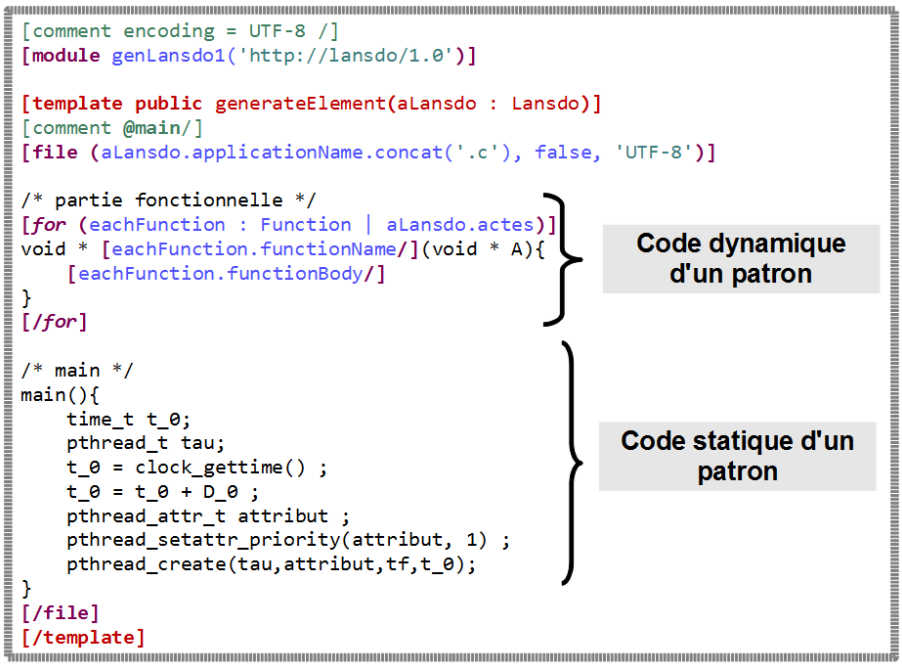


FIGURE 4.4 – Partie statique et dynamique d'un patron de mise en œuvre

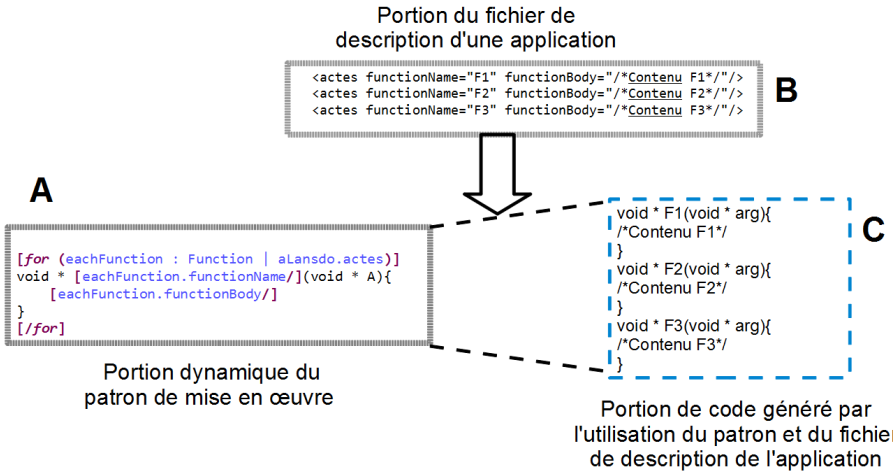


FIGURE 4.5 – Génération de code à partir d'un patron et de la description d'une application. A : Portion dynamique d'un patron ; B : Description d'une application ; C : Code généré

Techniques de mise en œuvre	Partie fonctionnelle	Tables d'ordonnancement	Autre partie temporelle	Partie main()
Mise en œuvre par une tâche	dynamique		dynamique	statique
Répartiteur non concurrent à un processus	dynamique	dynamique	statique	statique
Répartiteur non concurrent à deux processus	dynamique	dynamique	statique	statique
Répartition par gestions d'états	dynamique	dynamique	dynamique	statique
Répartition par affectation de priorité	dynamique	dynamique	dynamique	dynamique
Temps partagé	dynamique	dynamique	dynamique	dynamique
Synchronisation collective	dynamique	dynamique	dynamique	dynamique

TABLE 4.3 – Parties statique et dynamique des patrons de mise en œuvre

III. Mise en œuvre POSIX sur Xenomai

La section précédente (section II) nous a permis de proposer des outils d'automatisation de mise en œuvre POSIX. Dans cette section, nous discutons du choix du système d'exploitation temps réel (section III.1) et nous le décrivons dans la section III.2. Par la suite discutons de l'estimation des coûts de mise en œuvre (section III.3) du code produit.

III.1. Choix du système d'exploitation temps réel

Le système d'exploitation temps réel Linux est un système d'exploitation très utilisé, notamment parce qu'il est robuste et multi-plateformes. Un autre avantage à utiliser Linux est le fait qu'il est libre et que pour l'utiliser dans des projets de développement d'applications, il n'y a pas de coût de licence à prévoir. Ces avantages ont amené la communauté des systèmes embarqués à s'intéresser à Linux. Ils ont été confrontés à un manque de déterminisme du noyau Linux. En effet celui-ci utilise [2] un verrou global (BKL : Big Kernel Lock) dont le but premier était d'empêcher plusieurs processus d'exécuter simultanément des portions de code critique ou appels systèmes. Ce verrou a pour conséquence que dès lors qu'un processus fait un appel système, il n'est plus préemptible même par un processus de priorité supérieure.
La figure 4.6 présente l'exemple de deux processus p_1 et p_2 de priorité respective 1 et 2. Il

2. depuis la version 2.1.23 [LH02]

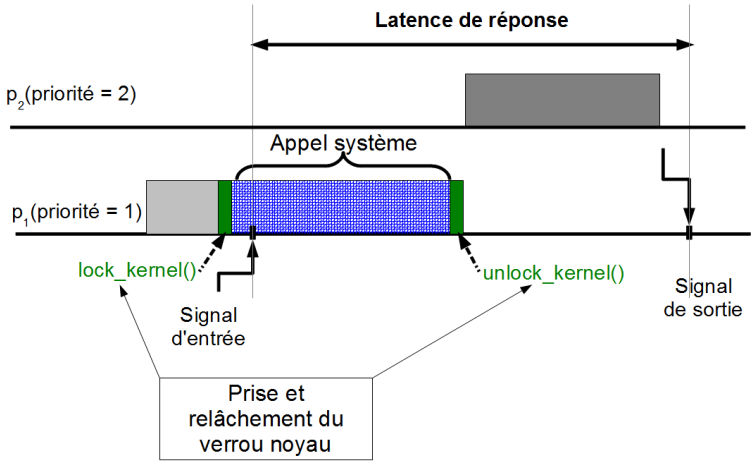


FIGURE 4.6 – Phénomène d'inversion de priorités dû au verrou du noyau

s'agit d'une inversion de priorité puisque le processus p_2 de plus grande priorité est retardé par le processus p_1 de plus faible priorité. De plus la latence de réponse à une interruption n'est plus garantie puisqu'elle dépend du temps pendant lequel un autre processus effectue un appel système. Pour résoudre ce problème de non déterminisme, deux approches ont été utilisées :
– rendre le *noyau préemptible* même lors des appels systèmes ;
– utiliser à côté du noyau Linux un deuxième noyau (*co-noyau*) temps réel.

Noyau préemptible Cette approche [Fra07] a consisté à modifier le comportement du verrou BKL afin de permettre des préemptions. Une autre solution utilisée dans cette approche a consisté à ajouter des points de préemption dans des appels système. Cette approche a permis de réduire les latences mais les performances restent plus faibles ([RLB08]) qu'avec l'approche par co-noyau.

Co-noyau L'approche co-noyau est l'approche utilisée par des systèmes d'exploitation temps réel tel que RTAI [RTA], Xenomai [Xen]. Elle consiste à utiliser un autre système d'exploitation, à "côté" du noyau Linux, qui a en charge l'apport des fonctionnalités temps réel (ordonnancement, gestion du temps, gestion des évèments, héritage de priorité). Ce deuxième système d'exploitation ayant en général une faible empreinte mémoire, on l'appelle généralement *micro-noyau* et le noyau Linux devient un processus du micro-noyau.

Xenomai Nous avons choisi comme support de tests d'éxécution Xenomai parce que c'est un logiciel libre. De plus, il permet d'utiliser l'API POSIX. Un autre avantage est que la communauté d'utilisateurs de Xenomai est très active. Ce qui permet d'avoir un système moderne et évolutif.

La section suivante (section III.2) nous permet de présenter plus en détail Xenomai et les particularités de Xenomai quant à la programmation d'applications avec POSIX.

III.2. Description de Xenomai

La co-existence des deux noyaux (le micro-noyau Xenomai et le noyau Linux) impose qu'il y ait des mécanismes de gestion d'interruptions pour déterminer à quel noyau doit être signalé une interruption. La solution utilisée dans Xenomai consiste à ajouter une couche d'abstraction matérielle appelée *ADEOS* (Adaptive Domain Environnement for Operating System) [ADE05] qui joue un rôle de récepteur/transmetteur d'interruptions. Ainsi, les interruptions reçues sont dans un premier temps transmises au micro-noyau Xenomai et si l'interruption n'est pas traitée, elle est transmise au noyau Linux.
De plus, on se retrouve avec trois types de processus :
– les processus Xenomai ;
– les processus lourds Linux ;
– les processus légers Linux.
D'un point de vue du micro-noyau Xenomai, Linux est un processus, comme tout autre processus applicatif. L'ordonnancement se fait donc de manière hiérarchique. Le premier niveau hiérarchique d'ordonnancement est celui de l'ordonnanceur Xenomai. Lorsque Linux est le processus élu, l'ordonnanceur Linux choisit parmi les processus lourds prêts celui qui est exécuté et il y a également un arbitrage pour un processus lourd de plusieurs processus légers.
La distinction entre ces trois types de processus se fait à leur création en fonction de l'appel système de création de processus. Il y a donc trois primitives de création de processus. Plus généralement, il y a deux classes d'appels système que sont les appels sytème Xenomai et les appels système Linux. Comme l'un des objectifs de l'approche Xenomai était de conserver la possibilité d'utiliser les appels systèmes Linux, un processus Xenomai peut faire un appel système Linux. Par contre l'inverse (appel système Xenomai par un processus Linux) n'est pas autorisé. Un processus Xenomai qui fait un appel système Linux change d'ordonnanceur et passe sous le contrôle de l'ordonnanceur Linux. Un processus Xenomai qui a basculé sous le contrôle de Linux y reste jusqu'à ce qu'il fasse un appel système Xenomai pour rebasculer sous le contrôle de Xenomai. La priorité d'un processus Xenomai qui bascule sous le contrôle de Linux reste constante. Cette constance fait que le nombre de niveaux de priorités disponibles sur Linux est le même sur Xenomai à savoir 100 (de 0 à 99). De plus, pour ne pas tomber dans un cas d'inversion de priorité (c'est-à-dire qu'un processus Xenomai moins prioritaire que le processus qui bascule sous Linux devienne plus prioritaire), le processus Linux hérite de la priorité du processus Xenomai qui y bascule.
De même si plusieurs processus Xenomai basculent sous Linux, la priorité de Linux est égale à la plus grande priorité des processus Xenomai qui ont basculé. La figure 4.7 présente l'exemple des deux processus p_1 et p_2 de priorités respectives 1 et 2. Lorsque p_1 fait un appel système linux, il bascule sous le contrôle de l'ordonnanceur Linux. Comme le noyau Linux est un processus Xenomai, à l'arrivée de l'interruption que p_2 doit traiter, le noyau Linux (et donc le processus p_1)

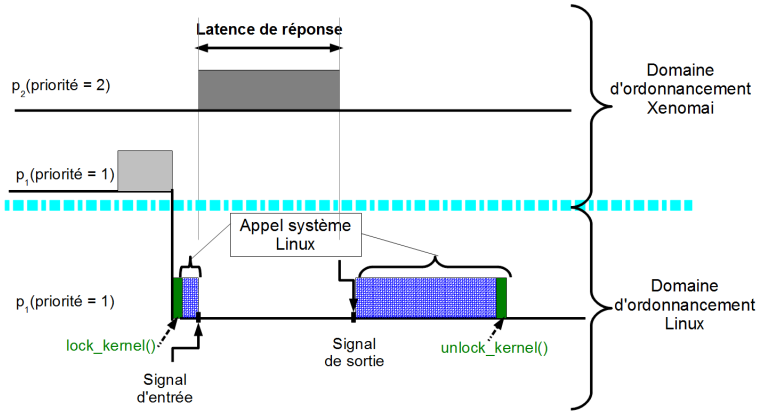


FIGURE 4.7 – Traitement du phénomène d'inversion de priorités dû au verrou du noyau Linux par Xenomai

est préempté. Le basculement de p_1 après l'appel système Linux permet alors à p_2 de s'exécuter immédiatement. Il (le basculement) a cependant l'inconvénient qu'un processus Xenomai puisse être retardé par un processus Linux (si le processus Linux est plus prioritaire). Pour éviter cette situation, lors de la mise en œuvre, il faut créer des processus Xenomai qui ont des priorités plus grandes que la plus grande priorité des processus Linux.

D'un point de vue pratique les appels systèmes Xenomai se font par l'intermédiaire de primitives incluses dans des bibliothèques. Afin d'avoir le plus d'utilisateurs possible, l'approche Xenomai offre des bibliothèques POSIX, RTAI, VxWorks, pSOS, *u*ITRON, VRTX. Ainsi, une application écrite pour le système d'exploitation RTAI devrait, avec pas ou peu de modifications, s'exécuter également sur Xenomai. Nous présentons dans l'annexe B un exemple de script utilisé pour compiler une application utilisant les primitives de la bibliothèque POSIX.

Une particularité de la mise en œuvre d'applications sur Xenomai est qu'elle doivent contenir forcement la ligne mlockall(MCL_CURRENT | MCL_FUTURE) avant toute autre instruction. Cet appel permet d'empêcher l'utilisation d'un système de pagination à la demande. L'inconvénient avec un système de pagination est que lorsqu'un défaut de page survient, des services du noyau Linux (incluant l'allocation dynamique de pages) doivent s'exécuter. L'exception de défaut de page entraine donc un basculement en domaine d'ordonnancement Linux. En utilisant mlockall, toute demande de mémoire entraine un réponse (succès ou échec) immédiate et il n'y a pas de défaut de pages. L'utilisation de mlockall désactive la pagination pour le processus appelant mais pour éviter des erreurs dûes au manque d'espace pendant l'exécution, une taille suffisante de pile doit être attribuée explicitement par la primitive pthread_attr_setstacksize. Le choix de la taille de la pile doit prendre en compte les variables qui seront utilisées

Il est donc nécessaire de bien évaluer les coûts de mise en œuvre qui font l'objet de la section III.3.

III.3. Évaluation des coûts de mise en œuvre

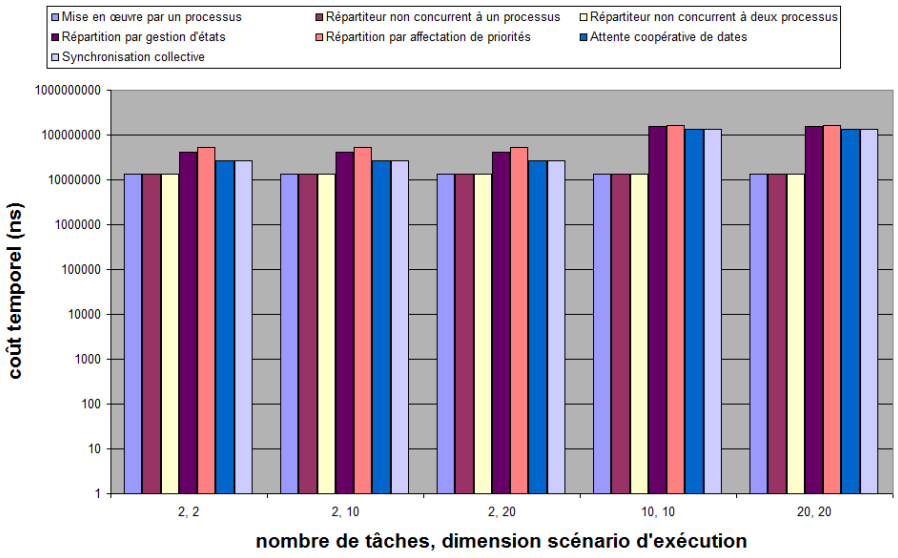
Dans le chapitre 3 où nous avons présenté les techniques de mise en œuvre, nous les avons évaluées analytiquement. L'objectif de cette section est de fournir une première estimation effective de coûts de mise en œuvre.
En rappel, nos techniques de mise en œuvre sont caractérisées par divers coûts à savoir un coût temporel, un coût spatial, un coût de changement de contexte, un nombre de niveaux de priorités et une robustesse. Dans la suite nous ne nous intéressons qu'aux coûts quantitatifs (tout sauf la robustesse). Le nombre de niveaux de priorités est imposé par la technique de mise en œuvre et nous ne nous y intéresserons pas plus. Nous traitons du coût de changement de contexte dans la section III.3.b. Dans la section III.3.a nous présentons l'évaluation du coût temporel et dans la section III.3.c nous présentons l'evaluation du coût spatial.

III.3.a. Étude comparative des coûts temporels

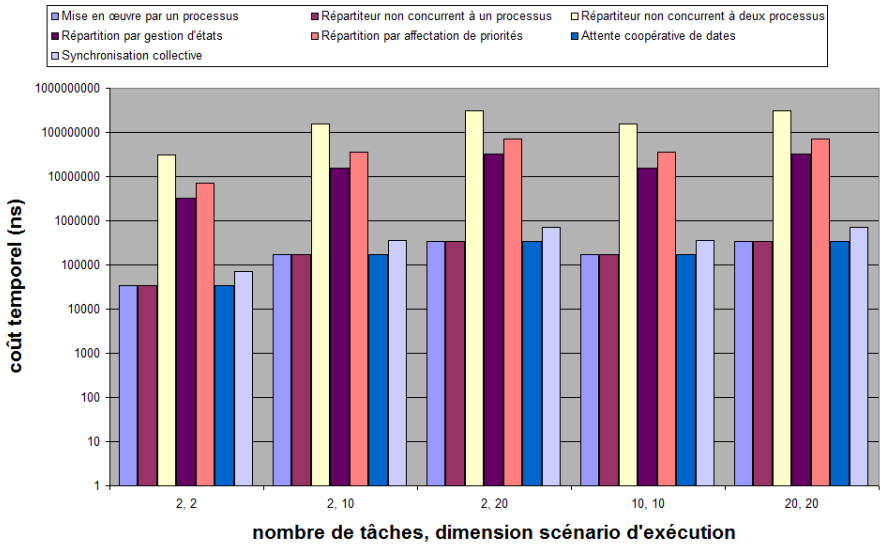
Le coût temporel ou temps processeur utilisé par le bloc d'initialisation et le bloc temporel est par définition une durée d'exécution. Il peut donc être déterminé en utilisant les mêmes méthodes (statiques ou dynamiques) utilisées pour déterminer les pires durées d'exécution (voir chapitre 1 section III.2.b).
Notre objectif étant principalement de pouvoir comparer nos techniques de mise en œuvre, nous utiliserons des méthodes dynamiques qui sont plus faciles à mettre en place.
Pour cela nous utilisons la technique d'estimation dynamique présentée par [AW88]. Cette technique consiste à utiliser un programme de mesure qui effectue des exécutions multiples d'un bloc d'instructions et en fait la moyenne. Par exemple, pour mesurer la durée d'exécution d'un bloc d'instructions B, il faut d'abord déterminer la durée d'exécution $Ec(\phi)$ à vide c'est-à-dire sans aucun bloc d'instructions. Puis, en estimant la durée de M exécutions de B, on déduit la durée d'exécution de B (équation 4.1).

$$Ec(B) = \frac{\sum fin - debut - Ec(\phi)}{M} \tag{4.1}$$

L'annexe D présente le programme que nous utilisons pour estimer les durées d'exécution.
Ce programme nous permet de déterminer les durées d'exécution des primitives POSIX correspondant aux instructions de notre API abstraite. Le tableau 4.4 présente les résultats.
À partir des durées d'exécution des primitives temporelles et du nombre de fois où chaque primitive est exécutée (voir chapitre 3 et annexe A), nous estimons les coûts temporels pour chaque technique de mise en œuvre. Afin de comparer les coûts, l'estimation est faite pour cinq couples $(|\tau|, |Se_{theorique}|)$. La figure 4.8 présente l'histogramme qui en résulte. Notons que les valeurs des coûts sont représentés à une échelle logarithmique afin de visualiser finement les variations.
L'initialisation prend plus de temps pour la technique de mise en œuvre par affectation de priorité. Pour les techniques de mise en œuvre par un processus, de répartition non concurrent à un ou deux processus, l'initialisation est constante et ne dépend ni du nombre de tâches, ni



(a)



(b)

FIGURE 4.8 – Étude comparative des coûts temporels de mise en œuvre. a : bloc d'initialisation ; b : Bloc temporel.

Primitives de POSIX	Ec (en ns)
clock_gettime	607
clock_nanosleep	17 124
pthread_create	13 525 651
pthread_cancel	2 246 230
pthread_kill	786 470
pthread_setprio	1 799 534
mq_open	945
mq_unlink	1 906
mq_send	18 391
mq_receive	18 260
sem_init	1 178
sem_wait	16 747
sem_post	1 923

TABLE 4.4 – Durées d'exécution des primitives POSIX

de la dimension du scénario d'exécution. Ces résultats impliquent que pour ces trois techniques de mise en œuvre (mise en œuvre par un processus, répartition non concurrent à un et à deux processus), la durée de référence D_0 est calculée une fois pour une plateforme donnée et ne nécessite pas de reévaluation. Par contre pour les autres techniques de mise en œuvre D_0 varie en fonction de nombre de tâches et du nombre de blocs d'exécution.

S'agissant des parties temporelles du code, la figure 4.8-b montre que la technique de répartition non concurrente à deux processus est celle qui a le plus grand coût temporel. Les techniques de mise en œuvre par un processus et de répartition non concurrente à un processus sont celles qui ont les coûts temporels les moins élevés.

III.3.b. Étude comparative des coûts de changement de contexte

Le coût de changement de contexte représente la durée nécessaire pour les changements de contexte d'une technique de mise en œuvre. Dans la section IV, qui traite de l'observation, nous présentons la détermination du coût moyen de changement de contexte. Dans la suite, nous supposons que nous connaissons ce coût moyen (nous utilisons comme coût moyen d'un changement de contexte la valeur 1983 ns) et en utilisant les coûts de changement de contexte de chaque technique de mise en œuvre (voir tableaux synoptiques annexe A), nous établissons une étude comparative (voir figure 4.9) pour cinq couples $(|\tau|, |Se_{theorique}|)$.

L'histogramme montre qu'en terme de changement de contexte les techniques qui utilisent un processus répartiteur (répartiteur non concurrent à deux processus, répartition par gestion détat et par affectation de priorités) sont celles qui ont les coûts les plus élevés.

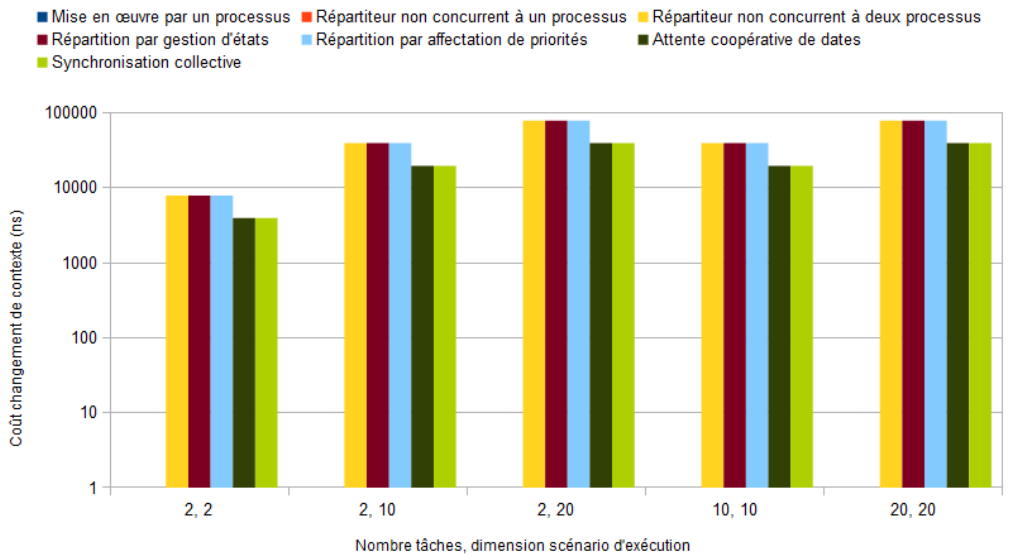


FIGURE 4.9 – Étude comparative du coût de changement de contexte des techniques de mise en œuvre d'ordonnancement hors-ligne

III.3.c. Étude comparative des coût spatiaux

Le coût spatial représente l'espace mémoire lié aux variables locales (pile), au code (segment de code), aux variables globales (segment de données). Par hypothèse, l'allocation de la mémoire est statique, le coût spatial de mise en œuvre n'est donc pas fonction du temps. Notre objectif étant de pouvoir comparer les techniques de mise en œuvre, nous estimerons uniquement la mémoire nécessaire pour les variables globales et locales. Ainsi, en utilisant la primitive sizeof(), nous déterminons dans un premier temps la taille en octets des données que nous utilisons dans nos mises en œuvre (voir tableau 4.5). Par la suite, en utilisant les coûts spatiaux établis dans le chapitre 3 (voir tableaux synoptiques annexe A), nous établissons une étude comparative (voir figure 4.10) pour cinq couples $(|\tau|, |Se_{theorique}|)$. Ainsi, la technique qui demande le moins de mémoire pour stocker les variables est la technique de mise en œuvre par un processus et celle qui nécessite le plus de mémoire pour stocker les variables est la technique de synchronisation collective.

III.4. Utilisation pratique des coûts de mise en œuvre

Dans la section précédente (section III.3), nous avons présenté l'estimation des coûts de mise en œuvre. Dans cette section nous discutons de l'utilisation de ces coûts.

Données	Type	Taille (octets)
Date	time_t	4
processus	pthread_t	4
semaphore	sem_t	16
Entrée d'une table de répartition non concurrente (un/deux processus)	schedule_elt (A)	8
Entrée d'une table de répartition par affectation de priorités	schedule_elt (B)	8
Entrée d'une table d'attente coopérative de dates	schedule_elt (C)	8
Entrée d'une table de synchronisation collective par sémaphore	schedule_elt (D)	12
Entrée d'une table de répartition par gestion d'état	schedule_elt (E)	8

TABLE 4.5 – Taille en octets des types de données utilisés par les techniques de mise en œuvre

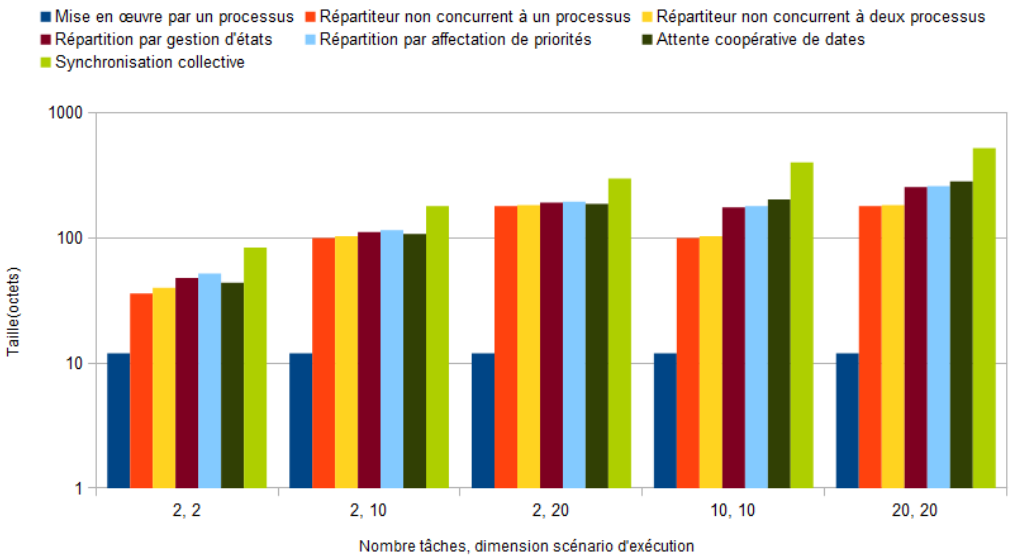


FIGURE 4.10 – Étude comparative du coût spatial des techniques de mise en œuvre d'ordonnancement hors-ligne

Coût temporel et coût de changement de contexte Le coût temporel et le coût de changement de contexte représentent une occupation du processeur par des instructions non fonctionnelles. En principe, ces coûts (temporel et changement de contexte) doivent être pris en compte pendant l'étape de modélisation temporelle de l'application (chapitre 2 section III.2). Dans notre cas nous avons pour hypothèse que l'ajout des instructions temporelles ne modifiait les durées d'exécution (C_i^{max} et C_i^{min}). Cette hypothèse revient à considérer, lors de l'estimation des durée d'exécution, les techniques qui ont les plus grands (respectivement les plus petits) coûts temporels et coûts de changement de contexte.

Coût spatial Le coût spatial permet de déterminer si l'espace mémoire disponible sur une plate-forme matérielle est suffisant pour une mise en œuvre.

Robustesse Comme nous l'avons vu dans le chapitres 3, certaines mises en œuvre permettent de préempter une tâche qui déborde du temps d'exécution qui lui est réservé, d'autres ne le permettent pas. Le choix entre ces deux types de mise en œuvre relève d'un choix politique. En effet, un dépassement de temps d'exécution ne devrait jamais avoir lieu. Cependant, compte tenu de conditions matérielles et logicielles il peut y avoir des erreurs. Il revient alors au concepteur de décider comment doit réagir l'application si tout ne se passe pas comme prévu par la séquence d'ordonnancement. Ainsi lorsqu'une tâche est en situation de dépassement de temps d'exécution, il est possible de :
- d'arrêter la tâche sans qu'elle ne finisse d'exécuter son code ;
- de suspendre la tâche et d'attendre les passages futurs pour finir son exécution ;
- de laisser la tâche s'exécuter jusqu'à son terme.

Recommandations et propriétés Dans ce paragraphe nous utilisons les coûts de mise en œuvre pour formuler des recommandations d'utilisation de nos techniques. Le tableau 4.6 résume ces recommandations. Nous y présentatons aussi les contextes dans lesquels certaines techniques ne sont pas utilisables.
Pour des tâches indépendantes nous n'avons pas de recommandations particulières car toutes nos techniques de mise œuvre sont utilisables. Par contre dans un contexte d'utilisation de ressources critiques ou de contraintes de précédence nous recommandons l'utilisation des techniques ayant une robustesse topologique à savoir la technique de mise œuvre par un processus et la mise en œuvre du répartiteur non concurrent à un processus. Cette recommandation est encore plus forte si l'estimation des durées d'exécution est réalisée dynamiquement (erreur de sous-calibrage possible).
Lorsque le scénario d'exécution théorique est préemptif seules les techniques de répartition par gestion d'états et par affection de priorités sont utilisables. Lorsqu'il est non préemptif toutes les techniques sont utilisables.
À cause du coût spatial élevé de la technique de mise œuvre par synchronisation collective, nous la déconseillons lorsque la mémoire est limitée sur le matériel cible.

Pour finir lorsque les fonctions de l'application nécessitent beaucoup de temps calcul et qu'il faut minimiser les coûts temporels de mise en œuvre (nécessité de grand rendement fonctionnel), nous déconseillons les techniques qui engendrent les coûts temporels les plus élevés à savoir le répartiteur non concurrent à deux processus, la répartition par gestion d'état et la répartition par affectation de priorités.

1-	Mise en œuvre par un processus
2-	Répartiteur non concurrentà un processus
3-	Répartiteur non concurrent à deux processus
4-	Répartition par gestions d'états
5-	Répartition par affectation de priorité
6-	Attente coopérative
7-	Synchronisation collective

	1	2	3	4	5	6	7
Tâches indépendantes	☐	☐	☐	☐	☐	☐	☐
Ressources critiques							
Contraintes de précédence							
Estimation dynamique du WCET	√	√	⊠	⊠	⊠	⊠	⊠
Estimation statique du WCET	√	√	☐	☐	☐	☐	☐
Scénarios théoriques préemptif	X	X	X	√	√	X	X
Scénarios théoriques non préemptif	☐	☐	☐	☐	☐	☐	☐
Mémoire limitée	☐	☐	☐	☐	☐	☐	⊠
Besoin de grand rendement fonctionnel	√	√	⊠	⊠	⊠	√	☐

√ recommandé
☐ possible
⊠ déconseillé
X impossible

TABLE 4.6 – Recommandations d'utilisation des techniques de mise en œuvre

IV. Observation et analyse de scénarios d'exécution

Dans la section I, nous avons présenté la traduction Posix de nos algorithmes de mise en œuvre. Cependant, il reste encore à vérifier que ces mises en œuvre produisent des applications dont l'exécution, sur un système d'exploitation temps réel, produit des scénarios effectifs qui suivent les scénarios théoriques : il est donc nécessaire d'*observer*.
L'observation est définie comme étant "*une activité initiale dans la mise au point d'un système dont l'objectif est de comprendre le fonctionnement de ses éléments constitutifs, afin de déceler des problèmes potentiels et de donner les informations nécessaires pour l'amélioration des performances*" ([PR11]).

L'observation peut être utilisée, lors de la réalisation d'une application, pour détecter et corriger les défauts de l'application. Dans un tel contexte, l'observation consiste à instrumenter (c'est-à-dire modifier afin de tracer les évènements [dACMdA+08] [SLJD08]) l'application ([Jai91], [NS07], [Inc10]) ou le système d'exploitation ([KMPP07]) afin de recueillir ([Kra], [SM06]),[GKM82], [Lev], [VSK+08], [TG06], [93801]) et analyser ([GWWM09], [Fag97]) des traces ([KBB+06], [dKdOSB00]) d'exécution.
Dans notre cas, notre objectif n'est pas d'observer une application en particulier, mais plutôt d'étudier le résultat produit par nos techniques de mise en œuvre. Une solution serait de synthétiser un ensemble de processus qui n'a pour seul objectif que d'observer comment l'ordonnancement se fait ([Reg02]).
Cependant, l'outil d'observation présenté par [Reg02] ne prend pas en compte toutes les instructions temporelles que nous utilisons dans nos techniques de mise en œuvre. De plus les traces d'exécution produites ne suivent pas notre formalisation des scénarios d'exécution. D'où la nécessité de réaliser un outil d'observation qui prend en compte à la fois notre modèle de tâche, les primitives temporelles et le formalisme des scénarios d'exécution.
Dans la suite nous décrivons notre outils d'observation (section IV.1). Nous discutons dans la section IV.3 de l'analyse des données issues de notre outil d'observation.

IV.1. Principes et réalisation de l'outil d'observation interne

L'objectif de notre outil d'observation est qu'il nous fournisse des informations sur les fonctions du modèle de tâches initial exécutées avec les dates de début et de fin d'exécution.
Ce besoin implique que notre outil d'observation doit intégrer des instructions d'observation (c'est-à-dire l'enregistrement des informations sur les dates, les fonctions) et des instructions qui simulent l'utilisation du processeur.
Pour enregistrer les observations (exécution des processus), nous utilisons une structure de données Observation (voir figure 4.11-A). Afin de simuler l'utilisation du processeur par nos processus, nous utilisons une fonction Utiliserprocesseur(dt) qui permet d'occuper le processeur pendant dt unité de temps. La figure 4.11-B présente l'algorithme correspondant. Il s'agit d'une attente active qui s'arrête après dt unités de temps.
Pour se rapprocher le plus possible des modèles théoriques, les durées d'exécution doivent être discrétisées. Se pose alors la question de l'unité de discrétisation ut à choisir. Cette unité de discrétisation dépendra de la précision avec laquelle notre outil pourra observer les évènements. En effet, l'unité de discrétisation doit être suffisamment grande pour que l'observateur puisse enregistrer deux évènnements successifs e_1 et e_2 qui surviennent à des dates d_1 et d_2. Nous reviendrons sur l'unité de discrétisation dans la section IV.2.
Afin d'observer des systèmes de tâches conformes aux modèles théoriques, nous synthétisons pour chaque tâche τ_i une fonction f_i de sorte qu'elle intègre l'exécution de C_i blocs d'exécution appelés blocQ(O,i,j,k) de pire durée d'exécution ut. Afin d'avoir des blocs non préemptibles nous utilisons un sémaphore P. Les blocs d'exécution intègrent les instructions d'observation c'est-à-dire d'enregistrement des évènements à travers les variables passées en paramètres blocQ(O,i,j,k)

(figure 4.11-C) : O une variable de type Observation, i est le numéro de la tâche, j celui de l'instance et k le numéro de bloc Q.

IV.2. Détermination de l'unité de discrétisation

L'unité de discrétisation ut doit être suffisamment grande pour permettre l'enregistrement de deux blocs d'exécution successifs. Notons δ la durée d'exécution des instructions d'observation. Nous devons choisir ut de sorte que $ut > \delta$.
Par exemple en implémentant notre outil d'observation sur un ordinateur équipé d'un processseur x386 avec 512 Mo de RAM nous avons $400ns \leq \delta \leq 700ns$. Sur une telle plateforme, il nous est impossible d'observer des systèmes pour lesquels $ut \leq 700ns$.
Pour avoir plus de marge, nous prenons $ut > 100 \times \delta$.
Pour finir, nous prenons en compte δ dans la fonction blocQ en utilisant $ut - \delta$ comme paramètre de la fonction Utiliserprocesseur.

IV.3. Récupération et traitement des données

Dans cette section nous discutons du traitement des données recueillies lors d'une observation. Comme présenté dans la section IV.1, les données sont enregistrées en mémoire centrale dans une variable O. O est un tableau statique (taille déterminée avant exécution) de type Observation, avec comme taille le nombre de blocs d'exécution blocQ exécuté entre 0 et $H = ppcm(T_i)$.
Les données collectées lors de l'observation ne sont stockées dans des fichiers qu'à la fin du cycle. Elles peuvent être représentées sont sous-forme de tableau d'observations, c'est-à-dire de tableaux dont le nombre de lignes est le nombres de bloc d'exécution et les colonnes sont les attributs de la structure d'observation. Nous obtenons des tableau de cinq colonnes (voir équation 4.2) à savoir : *Début*, *Fin*, *Tâche*, *Instance*, *Bloc*.

$$O = \left(\begin{array}{ccccc} \textit{Début} & \textit{Fin} & \textit{Tâche} & \textit{Instance} & \textit{Bloc} \\ \hline o_{11} & o_{12} & o_{13} & o_{14} & o_{15} \\ \vdots & \vdots & \vdots & \vdots & \\ o_{m1} & o_{m2} & o_{m3} & o_{m4} & o_{m5} \end{array} \right) \tag{4.2}$$

Dans la suite, nous présentons dans la section IV.3.a, à travers un exemple, l'analyse de scénorios d'exécution observés. Nous présentons également dans la section IV.3.b l'utilisation de notre observateur pour évaluer la durée d'un changement de contexte.

IV.3.a. Analyse de scénarios d'exécution

Nous présentons dans cette section un exemple de mise en œuvre d'ordonnancement hors-ligne avec la technique de répartition par synchronisation collective. Nous considérons le système de tâches du tableau 4.7.

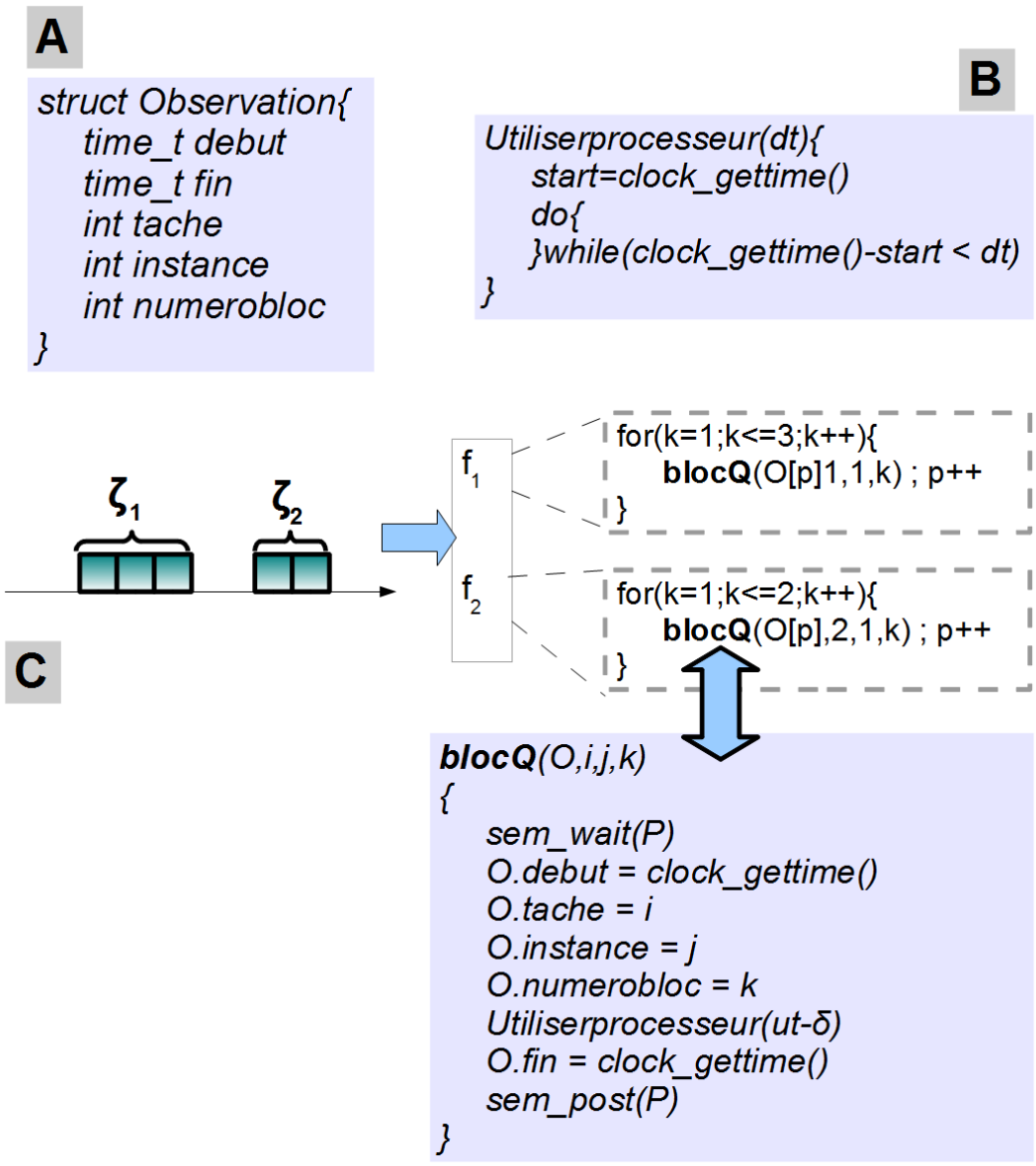
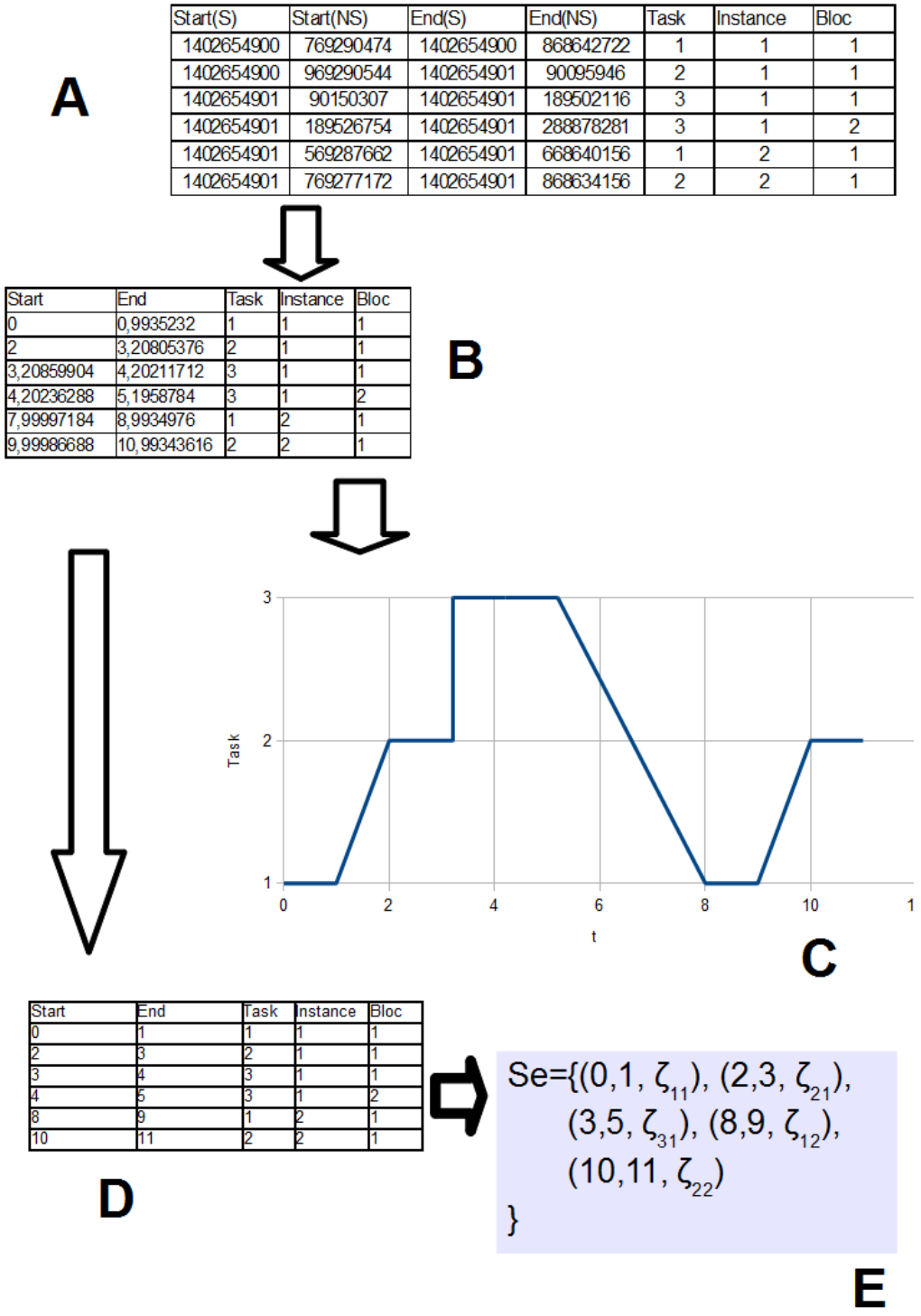


FIGURE 4.11 – Principe de l'outil d'observation. A : structure de données pour l'enregistrement des observations ; B : instructions de simulation de calcul processeur ; C : utilisation des instructions d'observation



Start(S)	Start(NS)	End(S)	End(NS)	Task	Instance	Bloc
1402654900	769290474	1402654900	868642722	1	1	1
1402654900	969290544	1402654901	90095946	2	1	1
1402654901	90150307	1402654901	189502116	3	1	1
1402654901	189526754	1402654901	288878281	3	1	2
1402654901	569287662	1402654901	668640156	1	2	1
1402654901	769277172	1402654901	868634156	2	2	1

Start	End	Task	Instance	Bloc
0	0,9935232	1	1	1
2	3,20805376	2	1	1
3,20859904	4,20211712	3	1	1
4,20236288	5,1958784	3	1	2
7,99997184	8,9934976	1	2	1
9,99986688	10,99343616	2	2	1

Start	End	Task	Instance	Bloc
0	1	1	1	1
2	3	2	1	1
3	4	3	1	1
4	5	3	1	2
8	9	1	2	1
10	11	2	2	1

FIGURE 4.12 – Analyse de scénario d'exécution observé

Description	exemple
Système de tâches	$\tau = \{\tau_1 =< (0, [1:3], 8, 8 \mid f_1 >,$ $\tau_2 =< 2, [1:3], 5, 8 \mid f_2 >,$ $\tau_3 =< 0, [2:4], 16, 16 \mid f_3 >\}$
Scénario d'exécution théorique	$Se = \{(0, 3, \tau_{11}), (3, 6, \tau_{21})$ $(6, 10, \tau_{31}), (10, 13, \tau_{12}), (13, 16, \tau_{22})\}$ $\lvert Se \rvert = 5\,;\ H = 16$

TABLE 4.7 – Exemple d'un système de tâches observé

Le tableau A de la figure 4.12 présente le résultat de l'observation lorsque nous considérons une unité de discrétisation $ut = 100000000$ ns et des durées d'exécution égales aux durées minimales (C_i^{min}) : $C_1 = 1$, $C_2 = 1$, $C_3 = 1$. Les deux colonnes de début (*Start*) et de fin (*End*) s'expliquent par le fait que la structure utilisée pour enregistrer les dates est une structure de type struct timespec. Nous avons donc une composante en seconde S et une composante en nano-seconde NS.
La première étape d'analyse consiste à discrétiser les valeurs des dates. Cette transformation se fait avec les étapes suivantes :
– conversion des dates en ns ;
– utilisation de la première date comme date origine ;
– division des dates par ut.
Le tableau B de la figure 4.12 présente le résultat de la première étape d'analyse.
Nous pouvons aussi visualiser l'exécution effective en utilisant un diagramme de Gantt (figure 4.12-C). Cette visualisation nous permet de voir que l'ordre d'exécution des tâches est respecté.
En partant du tableau B de la figure 4.12, nous pouvons aussi revenir à une représentation des scénarios effectifs sous la forme $(start_i, end_i, \tau_{\alpha(i)\beta(i)})$. Pour cela, nous discrétisons les dates (tableau 4.12-D) et nous regroupons les lignes du tableau qui indique l'exécution d'une même instance de tâche (tableau 4.12-E).

IV.3.b. Évaluation du changement de contexte

Nous présentons dans cette section l'utilisation de l'observateur pour évaluer la durée moyenne d'un changement de contexte. L'idée que nous utilisons vient de la définition du changement de contexte qui est la durée nécessaire pour sauvegarder le contexte d'un processus et restaurer le contexte d'un autre processus. Pour l'évaluer, nous mettons en œuvre deux processus de priorités différentes :
– un premier processus non périodique de faible priorité qui s'exécute par défaut ;
– un deuxième processus périodique de priorité plus grande.
Lorsque le processus de grande priorité est activé, il se produit un changement de contexte.

La figure 4.13 présente sur quelques valeurs notre méthodologie. Le tableau A de la figure 4.13 présente quelques valeurs obtenues par l'observateur interne que nous avons mis en œuvre. Nous n'avons présenté que les colonnes que nous utilisons pour le calcul du changement de contexte à savoir les colonnes *début*, *fin* et *tâche*. En utilisation ce premier tableau nous déterminons les durées entre deux blocs successifs d'un même processus (*delta1*) et de deux processus différents (*delta2*) : voir tableau B de la figure 4.13. Par la suite la différence entre les moyennes de ces valeurs nous donne la durée moyenne de changement de contexte (tableau C de la figure 4.13). Ce calcul nous donne comme valeur moyenne 1983 ns.

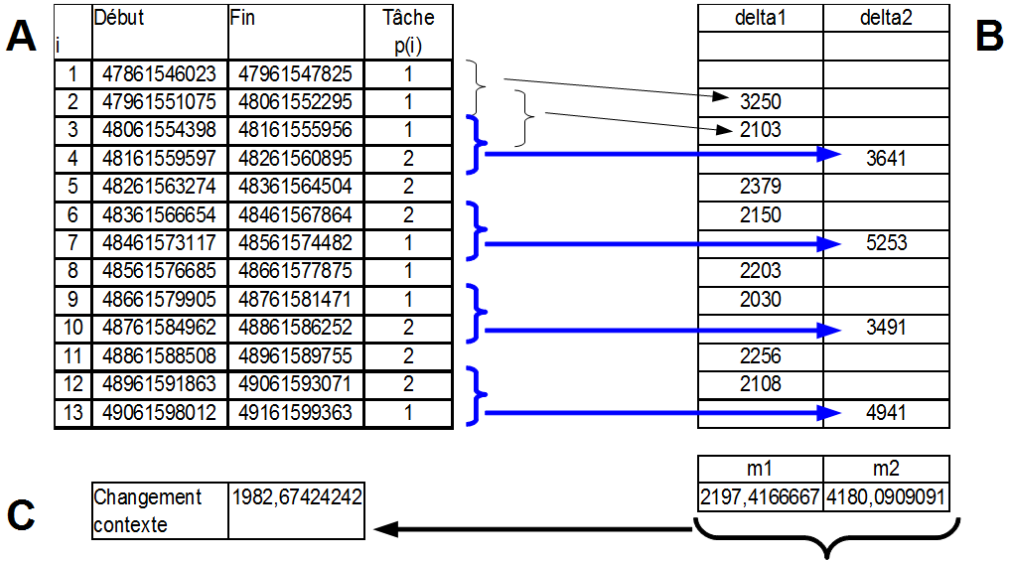


FIGURE 4.13 – Évaluation de la durée de changement de contexte

V. Conclusion

Dans ce chapitre, nous avons présenté des éléments qui permettent de passer de la mise en œuvre algorithmique, utilisant des primitives de notre API abstraite, à une mise en œuvre effective utilisant une API POSIX. Nous avons présenté des éléments d'automatisation des mises en œuvre. Ainsi, à travers des outils qui font de la génération de code nous avons montré que l'automatisation des mises en œuvre de séquences d'ordonnancement hors-ligne est possible. Nous avons également présenté un outil d'observation sur un système d'exploitation temps réel. Pour finir nous avons fait une étude comparative des coûts de mise en œuvre des techniques. Cette étude nous a permis de conclure que la technique de répartition non concurrente à deux processus est celle dont le bloc temporel a le coût temporel le plus élevé et que la technique de synchronisation collective est celle qui a le coût spatial le plus élevé.

Chapitre

5 Étude de cas : application de gestion de mine souterraine

Sommaire

Résumé

Dans ce chapitre nous présentons, à travers un cas pratique d'application de gestion de mine, notre approche de mise en œuvre effective d'ordonnancement hors-ligne.

L'objectif de ce chapitre est de présenter à travers un cas pratique comment appliquer notre approche de mise en œuvre d'ordonnancement hors-ligne. Le cas pratique est utilisé comme support de présentation et montre ainsi la faisabilité de notre approche. Notre étude ne portant pas sur la détermination des paramètres temporels, nous ne nous y attardons pas et nous considérons qu'ils sont connus. De plus notre approche portant sur la mise en œuvre d'ordonnancement hors-ligne, nous utilisons un cas pratique dont l'ordonnancement est fait hors-ligne. Nous utilisons l'exemple de l'application de gestion de la sécurité d'une mine issue de la thèse [Gro99]. Dans la section I nous décrivons le cas pratique. Nous y présentons également les paramètres temporels et le scénario d'exécution théorique à mettre en œuvre. Dans la section II nous discutons des techniques de mise en œuvre et du choix d'une technique.

I. Description du cas pratique : Gestion de la sécurité d'une mine

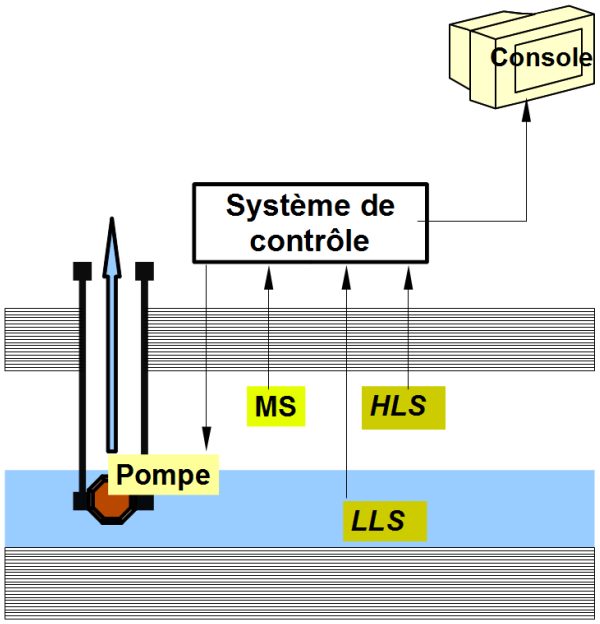


FIGURE 5.1 – Système temps réel de gestion de la sécurité d'une mine

Le contexte dans lequel notre étude de cas se fait est celui d'une mine souterraine dans laquelle il faut contrôler le niveau d'eau et la quantité de méthane. Le système peut être qualifié de *système temps réel dur* puisqu'un taux de méthane trop élevé dans la mine peut être à l'origine de coups de grisou (explosion) et d'intoxication. De plus une mauvaise gestion de l'écoulement de l'eau peut être à l'origine de noyade. On a donc un grand risque de pertes en vies humaines.

Afin d'éviter le risque de noyade, l'eau des galeries est drainée vers l'extérieur par une pompe. Pour éviter que la pompe ne fonctionne à tout instant, un niveau maximal d'eau à ne pas dépasser est défini. Dès que le niveau de l'eau atteint le niveau maximal, la pompe se met en marche. De même, un niveau minimal est défini pour que la pompe s'arrête dès que le niveau d'eau est inférieur au niveau minimal.
Pour éviter l'intoxication par le méthane ou le coup de grisou, la ventilation de la galerie est permanente. En plus de la ventilation, un taux maximal de méthane est défini : une alerte est envoyée à la console de l'opérateur afin qu'il ordonne l'évacuation de la mine. En plus des alertes, la console permet à l'opérateur de suivre le niveau de méthane et le niveau d'eau.
Cette alerte est d'autant plus nécessaire que lorsque le taux de méthane est supérieur à la limite, la pompe est arrêtée (pour éviter une éventuelle explosion dûe à la chaleur dégagée par la pompe). Cet arrêt augmente donc le risque d'inondation.
Afin de founir les fonctionnalités ci-dessus, l'architecture matérielle du système (voir figure 5.1) comprend les éléments suivants :
- deux capteurs digitaux du niveau de l'eau appelés HLS (High Level Sensor) et LLS (Low Level Sensor) ;
- une pompe (appelée Moteur) à eau permettant d'évacuer l'eau ;
- un capteur analogique du taux de méthane appelé MS ;
- une console (ou Terminal) opérateur.

Dans la suite de cette section, nous présentons la mise en œuvre de l'application de gestion de mine en suivant les étapes de réalisation décrites dans le chapitre 2 (chapitre 2-section III) : la section I.1 présente le découpage fonctionnel, la section I.2 donne le modèle temporel de l'application, la section I.3 présente le scénario d'exécution théorique à mettre en œuvre.

I.1. Réalisation de la partie fonctionnelle

Pour réaliser l'application de gestion de la mine, il est nécessaire de lire les valeurs renvoyées par les capteurs, d'activer la pompe lorsque c'est nécessaire et d'afficher toutes les informations sur la console de l'opérateur.
Il est donc nécessaire que l'application :
- accède aux données renvoyées par les capteurs (HLS, LLS, MS) à travers la primitive Lire_Ressource des ressources matérielles ;
- accède aux actionneurs (Terminal, Moteur) pour indiquer une commande à travers la primitive Ecrire_Ressource ;
- stocke les valeurs reçues des capteurs (Evt_Methane, Evt_EauHLS, Evt_EauLLS) et les valeurs à envoyer aux actionneurs (Evt_Moteur, Evt_Alerte, Message_Info).

Nous utilisons le découpage fonctionnel suivant :
- une fonction f_1 pour l'acquisition des données du capteur de méthane ;
- une fonction f_2 pour l'acquisition des données des capteurs d'eau ;
- une fonction f_3 pour commander la pompe ;
- une fonction f_4 pour l'affichage de l'alerte sur le terminal ;

– une fonction f_5 pour l'affichage du taux de méthane et du niveau d'eau sur la console de l'opérateur ;
– une fonction f_6 (ou fonction de contrôle) qui permet de calculer la valeur envoyée sur la pompe et l'évènement alerte qui est affiché sur la console.

La mise en œuvre (utilisant notre API abstraite) de ces fonctions est présentée par la figure 5.2.

```
/* Acquerir capteur methane*/
f1(){
    Evt_Methane←Lire_Ressource(MS)
}

/* Acquerir capteur eau*/
f2(){
    Evt_EauHLS←Lire_Ressource(HLS)
    Evt_EauLLS←Lire_Ressource(LLS)
}

/* Commander Pompe*/
f3(){
        Ecrire_Ressource(Moteur,Evt_Moteur)
}

/* Afficher alerte */
f4(){
    Ecrire_Ressource(Terminal, Evt_Alerte)
}

/* Afficher Informations*/
f5(){
    Message_Info←CalculM(Evt_Methane,Evt_EauHLS,Evt_EauLLS)
    Ecrire_Ressource(Terminal,Message_Info)
}

/* Controle*/
f6(){
    Evt_Moteur←CalculCmdMoteur(Evt_EauHLS,Evt_EauLLS,Evt_Methane)
    Evt_Alerte←CalculCmdAlerte(Evt_Methane)
}
```

FIGURE 5.2 – Liste des fonctions de l'application de gestion de la sécurité d'une mine

Le modèle fonctionnel obtenu après l'étape de codage des fonctions est donc :

$$F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$$

Dans la section I.2, nous présentons le système de tâches correspondant ainsi que les contraintes liées au modèle de tâches.

I.2. Modélisation temporelle

Les paramètres temporels que nous utilisons se basent sur ceux utilisés par [Gro99]. Nous avons juste mis ces paramètres en conformité avec notre modèle de tâches. Dans la suite nous détaillons ces paramètres temporels.

Ainsi, le système de tâches est à départs simultanés et à échéances sur requêtes. Toutes les tâches ont une période $T_i = 100$ ($i \in \{1, 2, 3, 4, 6\}$) à l'exception de la tâche τ_5 qui a une période $T_5 = 500$.

En plus de la date d'activation, de la période et de l'échéance, les tâches sont soumises à d'autres contraintes que sont la précédence et l'exclusion mutuelle.

En effet, le modèle fonctionnel de l'application (voir figure 5.2) fait ressortir qu'il y a des ressources critiques. Pour cela, le modèle temporel est décliné sous forme de sous-fonctions afin d'exprimer les contraintes de précédence et d'exclusion mutuelle.

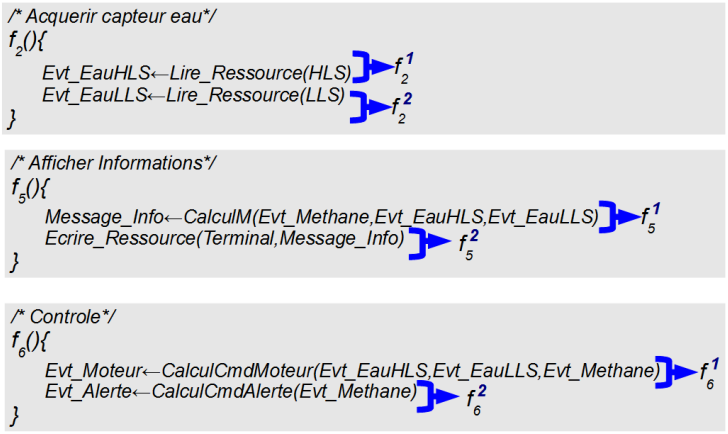


FIGURE 5.3 – Découpage en sous-fonctions de f_2, f_5 et f_6

Ainsi, les trois fonctions f_2, f_5 et f_6 (voir figure 5.3) sont découpées chacune en deux sous-fonctions.

Les ressources critiques sont les suivantes :

- la ressource matérielle Terminal (ou console de l'opérateur) est utilisée en écriture par f_4 et f_5^2. Pour éviter d'afficher des messages incompréhensif par l'opérateur, il y a une contrainte d'accès en exclusion mutuelle.
- la ressource logicielle Evt_Methane est utilisée en lecture par f_6^1 et en écriture par f_1. Pour éviter que la commande envoyée au moteur ne soit erronée, nous ajoutons une contrainte de précédence.
- les ressources logicielles Evt_EauHLS et Evt_EauLLS sont utilisées en lecture par f_6^1 et en écriture par f_2^1 et f_2^2. De même que pour la ressource méthane, une contrainte de précédence est rajoutée.
- la ressource logicielle Evt_Moteur est utilisée en lecture par f_3 et en écriture par f_6^1. Nous ajoutons également une contrainte de précédence.
- la ressource logicielle Evt_Alerte est utilée en lecture par f_4 et en écriture par f_6^2. Nous ajoutons aussi une contrainte de précédence.

Le tableau 5.1 récapitule l'utilisation des ressources et les paramètres temporels.

I.3. Recherche de scénario d'exécution théorique

La dernière étape avant la mise en œuvre effective est le calcul d'un scénario d'exécution théorique. Pour ce cas, l'ordonnancement hors-ligne se justifie par le fait que le système de

Ressources matérielles	Lecture	Écriture	Ressources logicielles	Lecture	Écriture
MS	f_1		Evt_Methane	f_5^1 f_6^1 f_6^2	f_1
HLS	f_2^1		Evt_EauHLS	f_5^1 f_6^1	f_2^1
LLS	f_2^2		Evt_EauLLS	f_5^1 f_6^1	f_2^2
Moteur		f_3	Evt_Moteur	f_3	f_6^1
Terminal		f_4 f_5^2	Evt_Alerte	f_4	f_6^2
			Message_Info	f_5^2	f_5^1
Précédences	$f_1 \prec f_6^1$ $f_2^1 \prec f_6^1$ $f_2^2 \prec f_6^1$ $f_6^1 \prec f_3$ $f_6^2 \prec f_4$		Exclusions	$f_5^2 \oplus f_4$	
Système de tâches	$\tau^{Annote} = \{\tau_1^{Annote} =< 0, [3:10], 100, 100 \mid f_1 >;$ $\tau_2^{Annote} =< 0, [4:12], 100, 100 \mid f_2^1[2:6], f_2^2[2:6] >;$ $\tau_3^{Annote} =< 0, [3:12], 100, 100 \mid f_3 >;$ $\tau_4^{Annote} =< 0, [10:25], 100, 100 \mid f_4 >;$ $\tau_5^{Annote} =< 0, [25:70], 500, 500 \mid f_5^1[8:20], f_5^2[17:50] >;$ $\tau_6^{Annote} =< 0, [10:15], 100, 100 \mid f_6^1[6:8], f_6^2[5:7] >\}$				

TABLE 5.1 – Système de tâches annotées et contraintes entre les sous-fonctions

tâches n'est pas ordonnançable par un algorithme d'ordonnancement en ligne. En effet, dans une approche d'ordonnancement en ligne, pour mettre en œuvre l'exclusion mutuelle, les parties du code de f_4 et de f_5^2 qui accédent au terminal sont protégées par un sémaphore. De même, les contraintes de précédence peuvent être mises en œuvre en utilisant des sémaphores. Cela a pour impact que l'exécution de τ_5 correspondant à f_5^2 retarde τ_4 et lui fait manquer son échéance.

Contrairement à l'approche en ligne qui fait une exécution au plus tôt, en utilisant un outil de calcul de scénarios d'exécution tel que PeNSMARTS [Gro99], nous obtenons tous les scénarios valides.

Dans notre cas, afin de faciliter la mise en œuvre, nous avons un scénario d'exécution avec le minimum de préemptions de tâches. Le tableau 5.2 présente le scénario d'exécution théorique que nous avons choisi. La représentation graphique du scénario d'exécution calculé hors-ligne (figure 5.4) montre que toutes les échéances sont respectées. De même on remarquera que :

- les blocs d'exécutions pour la tâche τ_6 interviennent toujours après les blocs d'exécution des tâches τ_1 et τ_2 ;
- les blocs d'exécution de la tâche τ_6 précèdent toujours les blocs d'exécution des tâches τ_3 et τ_4 ;
- l'exécution de la tâche τ_5 correspondant à f_5^2 (c'est-à-dire le bloc d'exécution $(174, 224, \tau_{51})$)

$\overline{Se_{theorique}} = \{$	$(0, 10, \tau_{11})$,	$(10, 22, \tau_{21})$,	$(22, 37, \tau_{61})$,	$(37, 49, \tau_{31})$,
	$(49, 74, \tau_{41})$,	$(74, 94, \tau_{51})$,	$(94, 100, \tau_0)$,	$(100, 110, \tau_{12})$,
	$(110, 122, \tau_{22})$,	$(122, 137, \tau_{62})$,	$(137, 149, \tau_{32})$,	$(149, 174, \tau_{42})$,
	$(174, 224, \tau_{51})$,	$(224, 234, \tau_{13})$,	$(234, 246, \tau_{23})$,	$(246, 261, \tau_{63})$,
	$(261, 273, \tau_{33})$,	$(273, 298, \tau_{43})$,	$(298, 300, \tau_0)$,	$(300, 310, \tau_{14})$,
	$(310, 322, \tau_{24})$,	$(322, 337, \tau_{64})$,	$(337, 349, \tau_{34})$,	$(349, 374, \tau_{44})$,
	$(374, 400, \tau_0)$,	$(400, 410, \tau_{15})$,	$(410, 422, \tau_{25})$,	$(422, 437, \tau_{65})$,
	$(437, 449, \tau_{35})$,	$(449, 474, \tau_{45})$,	$(474, 500, \tau_0)$	
$\}$				
$Se_{theorique} = \{$	$(0, 10, \tau_{11})$,	$(10, 22, \tau_{21})$,	$(22, 37, \tau_{61})$,	$(37, 49, \tau_{31})$,
	$(49, 74, \tau_{41})$,	$(74, 94, \tau_{51})$,	$(100, 110, \tau_{12})$,	$(110, 122, \tau_{22})$,
	$(122, 137, \tau_{62})$,	$(137, 149, \tau_{32})$,	$(149, 174, \tau_{42})$,	$(174, 224, \tau_{51})$,
	$(224, 234, \tau_{13})$,	$(234, 246, \tau_{23})$,	$(246, 261, \tau_{63})$,	$(261, 273, \tau_{33})$,
	$(273, 298, \tau_{43})$,	$(300, 310, \tau_{14})$,	$(310, 322, \tau_{24})$,	$(322, 337, \tau_{64})$,
	$(337, 349, \tau_{34})$,	$(349, 374, \tau_{44})$,	$(400, 410, \tau_{15})$,	$(410, 422, \tau_{25})$,
	$(422, 437, \tau_{65})$,	$(437, 449, \tau_{35})$,	$(449, 474, \tau_{45})$	
$\}$				
	$\lvert\overline{Se_{theorique}}\rvert =$	31		
	$\lvert Se_{theorique}\rvert =$	27		

TABLE 5.2 – Scénarios d'exécution théorique complet (avec temps creux) et sans temps creux de l'application de gestion de la sécurité d'une mine

n'est pas préemptée par τ_4.
Ainsi, toutes les contraintes de précédence et d'exclusion mutuelles sont respectées.

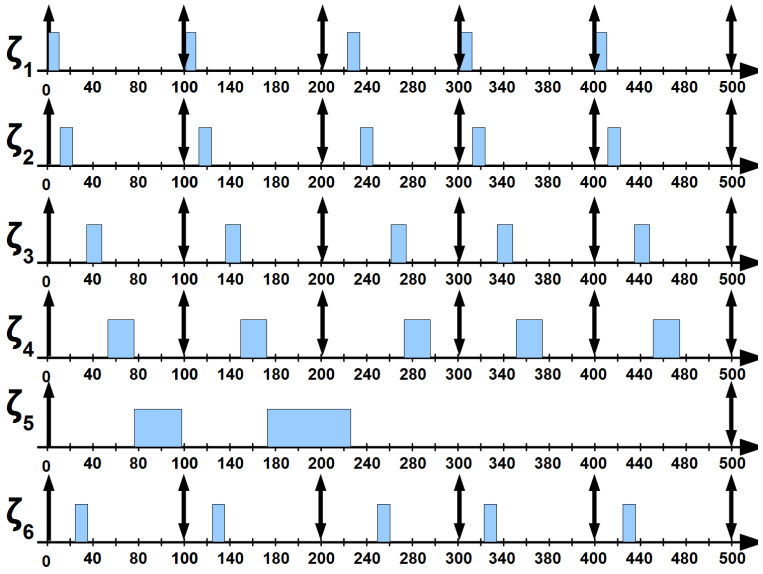


FIGURE 5.4 – Représentation graphique du scénario d'exécution théorique de l'application de gestion de la sécurité d'une mine

II. Mise en œuvre du scénario d'exécution

Dans cette section, nous traitons de la mise en œuvre effective du scénario d'exécution. Comme nous sommes dans un contexte de système annoté de tâches (sous-fonctions partageant des ressources critiques et avec des contraintes de précédence), nous déterminons dans un premier temps (section II.1) le modèle équivalent à une seule fonction. Dans un second temps (section II.2), nous discutons du choix de la technique de mise en œuvre.

II.1. Détermination du modèle de tâches à une seule fonction

Pour déterminer le modèle de tâches à une seule fonction il nous faut d'abord annoter le scénario d'exécution. (voir tableau 5.3).
L'annotation permet de voir que lors d'une mise en œuvre inflexible, si f_5^1 s'exécute pendant C_5^1 unités de temps avec $C_5^{min_1} \leq C_5^1 < C_5^{max_1}$, f_5^2 commence son exécution et est préemptée. Par la suite f_4 s'exécute sans que f_5^2 n'ait fini son exécution : la contrainte d'exclusion mutuelle n'est donc pas respectée. Afin que les mises en œuvre respectent cette contrainte, nous scindons la

$Se^{Annote}_{theorique} = \{$	$(0, 10, f_{11})$,	$(10, 22, f_{21})$,	$(22, 37, f_{61})$,	$(37, 49, f_{31})$,
	$(49, 74, f_{41})$,	$(74, 94, f^1_{51})$,	$(100, 110, f_{12})$,	$(110, 122, f_{22})$,
	$(122, 137, f_{62})$,	$(137, 149, f_{32})$,	$(149, 174, f_{42})$,	$(174, 224, f^2_{51})$,
	$(224, 234, f_{13})$,	$(234, 246, f_{23})$,	$(246, 261, f_{63})$,	$(261, 273, f_{33})$,
	$(273, 298, f_{43})$,	$(300, 310, f_{14})$,	$(310, 322, f_{24})$,	$(322, 337, f_{64})$,
	$(337, 349, f_{34})$,	$(349, 374, f_{44})$,	$(400, 410, f_{15})$,	$(410, 422, f_{25})$,
	$(422, 437, f_{65})$,	$(437, 449, f_{35})$,	$(449, 474, f_{45})$	
$\}$				

TABLE 5.3 – Scénarios d'exécution théorique annoté de l'application de gestion de la sécurité d'une mine

tâche τ_5 en deux tâches τ_5 et τ_7 et nous ajoutons une contrainte de précédence supplémentaire $f_5 \prec f_7$. La figure 5.5 présente les fonctions f_5 et f_7 obtenues (les autres fonctions étant inchangées) et le tableau 5.4 présente le modèle de tâches à une seule fonction obtenu. Le scénario d'exécution à mettre en œuvre est pratiquement le même que celui du tableau 5.2. La seule différence est que le bloc d'exécution $(174, 224, \tau_{71})$ remplace le bloc d'exécution $(174, 224, \tau_{51})$.

```
/* Déterminer Informations à afficher*/
f5(){
    Message_Info←CalculM(Evt_Methane,Evt_EauHLS,Evt_EauLLS)
}
```

```
/* Afficher message */
f7(){
    Ecrire_Ressource(Terminal,Message_Info)
}
```

FIGURE 5.5 – Nouvelles fonctions f_5 et f_7

Nous disposons maintenant d'un modèle fonctionnel et un scénario d'exécution théorique compatibles avec nos techniques de mise en œuvre. Il ne nous reste plus qu'à en choisir une. Ce choix est discuté dans la section II.2.

II.2. Choix de la technique de mise en œuvre

Étant dans l'embarqué, nous supposons dans cette section que la mise en œuvre se fait sur un matériel (calculateur) très limité en mémoire.
Le scénario d'exécution ne présente aucune préemption. Toutes les techniques de mises œuvre que nous avons présentées dans le chapitre 3 peuvent être envisagées.

Système de tâches	$\tau = \{\tau_1 =< 0, [3:10], 100, 100 \mid f_1 >;$ $\tau_2 =< 0, [4:12], 100, 100 \mid f_2 >;$ $\tau_3 =< 0, [3:12], 100, 100 \mid f_3 >;$ $\tau_4 =< 0, [10:25], 100, 100 \mid f_4 >;$ $\tau_5 =< 0, [8:20], 500, 500 \mid f_5 >;$ $\tau_6 =< 0, [10:15], 100, 100 \mid f_6 >$ $\tau_7 =< 0, [17:50], 500, 500 \mid f_7 >\}$

TABLE 5.4 – Modèle de tâches à une seule en sous-tâche

Cependant à cause des multiples contraintes de précédence et d'exclusion mutuelles nous n'utiliserons pas les techniques de répartition par gestion d'états ou par affectation de priorités. En effet, ces deux techniques sont telles qu'en cas d'erreur de sous-calibrage, la tâche en débordement est préemptée. Il s'en suit que les contraintes ne sont plus respectées.
La mise en œvre se faisant sur un matériel limité en mémoire, nous n'utiliserons pas les techniques qui nécessite la création de nombreux objets tel que les sémaphores et les processus.
Comme nous avons un scénario d'exécution théorique grand (27 blocs d'exécution), pour ne avoir une application avec un grand segment de code, nous n'utiliserons également pas la technique de mise en œuvre à un processus.
La technique de répartition non concurrente à un processus est finalement celle que nous utilisons. La figure 5.6 présente le code POSIX obtenu.

III. Conclusion

Dans se chapitre, nous avons présenté, à travers l'exemple de l'application de gestion de mines souteraines, les différentes étapes de la mise en œuvre effective d'un ordonnancement hors-ligne. Cette étude de cas nous a permis en particulier d'utiliser une étape d'annotation de scénario d'exécution théorique afin d'avoir un système de tâches implémentable par les techniques que nous avons présenté dans le chapitre 3. Nous avons également utilisé l'outil de génération de code Posix que nous avons présenté dans le chapitre 4.

A

```
#include <pthread.h>
#include <time.h>

#define TIME_UNIT 1000000

typedef struct{
   time_t start_time;
   void * (*fonction)(void*);
}schedule_elt;

/*Ajouter des ns une date de type timespec */
void time_spec_add_ns(struct timespec *T, time_t ns){
     T->tv_nsec =T->tv_nsec + ns;
     if(T->tv_nsec>=1000000000){
          T->tv_sec=T->tv_sec+T->tv_nsec/1000000000;
          T->tv_nsec=T->tv_nsec%1000000000;
     }
}
void timespec_set(timespec * left,timespec *rigth){
     left->tv_sec = rigth->tv_sec ;
     left->tv_nsec = rigth->tv_nsec;
}
```

B

```
void f_1(void){
     /* Acquerir capteur methane (MS)*/
}
void f_2(void){
     /* Acquerir capteurs (HLS, LLS) eau*/
}
void f_3(void){
          /* Commander Pompe (Moteur)*/
}
void f_4(void){
     /* Afficher alerte (Terminal)*/
}
void f_5(void){
     /* Déterminer Informations à afficher*/
}
void f_6(void){
     /* Contrôle : calcul de commandes */
}
void f_7(void){
     /* Afficher message */
}
```

C

```
schedule_elt table[]={
     (0, f_1 ) , (10, f_2 ) , (22, f_6 ) , (37, f_3 ) ,  (49, f_4 ) , (74, f_5) ,
     (100, f_1 ) , (110, f_2 ) , (122, f_6 ) , (137, f_3 ) , (149,  f_4 ) ,
     (174, f_7 ) , (224, f_1 ) , (234, f_2 ) , (246, f_6 ) , (261, f_3 ) ,
     (273, f_4 ) , (300, f_1 ) , (310, f_2 ) , (322, f_6 ) , (337, f_3 ) ,
     (349, f_4 ) , (400, f_1 ) , (410, f_2 ) , (422, f_6 ) , (437,  f_3 ) ,
      (449, f_4 )
};

int H = 500 ;
int s = 27 ;
```

D

```
/*Répartiteur */
void *tf(void *debut){
     int k=0;
     timespec t, t_0;
     int i;
     timespec_set(&t_0,(timespec*)debut);
     for( ; ; ){
          timespec_add_timespec_ns(&t,t_0,(time_t)table[k].start_time*TIME_UNIT );
          clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,&t,NULL);
          table[k].fonction(NULL) ;
          k=k+1;
          if(k==s){
               k=0;
               timespec_add_timespec_ns(&t_0,t_0,(time_t)H*TIME_UNIT);
          }
     }
     return NULL;
}

/* main */
int main(){
     timespec horloge;
     struct sched_param param;
     clock_gettime(CLOCK_REALTIME,&horloge) ;
     time_spec_add_ns(&horloge,K);
     pthread_attr_t attribut ;
     param.sched_priority = 1;
     pthread_attr_setschedpolicy(&attribut,SCHED_FIFO);
     pthread_attr_setschedparam(&attribut,&param);
     pthread_create(&tau_rep,&attribut,tf,(void*)&horloge) ;
     pthread_join(tau_rep,NULL);
     return 0;
}
```

FIGURE 5.6 – Code POSIX de la mise en œuvre effective d'un scénario d'exécution théorique pour la gestion d'une mine. A- Déclaration de la structure d'embarquement du scénario et définition de fonctions de gestion des dates ; B- Partie fonctionnelle ; C- Embarquement du scénario d'exécution théorique ; D- Initialisation de l'application et fonction de répartition

Conclusion et Perspectives

Conclusion

Ce travail nous a permis de mettre en place une approche de mise en œuvre applicative d'ordonnancement hors-ligne sur les systèmes d'exploitation temps réel. L'ordonnancement hors-ligne se justifie par le fait que dans le cas général des contextes multiprocesseurs, même pour les tâches indépendantes, il n'y a pas d'algorithmes polynomiaux optimaux en ligne et dans les contextes mono-processeurs, il n'y a pas d'algorithmes polynomiaux optimaux dès lors que des ressources critiques sont partagées.
Le premier chapitre sur l'état de l'art nous a permis de nous rendre compte que dans l'approche d'ordonnancement hors-ligne le problème le plus traité est celui de la recherche de séquences d'ordonnancement hors-ligne et que la mise en œuvre effective des ordonnancements calculés hors-ligne est très souvent traitée comme une question secondaire. Ainsi lorsque cette question est abordée, tous les contextes (préemption, partages de ressources, contraintes de précédence) ne sont pas considérés.
De plus, à notre connaissance, il n'y avait pas de modèle de coûts qui permette de comparer les mises en œuvre des applications temps réel.
Forts de ces constats, nous avons, dans le deuxième chapitre, défini un objet (le scénario d'exécution) destiné à représenter les séquences d'ordonnancement. Un scénario d'exécution est une liste de blocs d'exécution qui indique l'instance de tâche exécutée dans un intervalle de temps donné.
Nous avons aussi proposé un modèle de coûts qui permet d'évaluer et comparer des techniques de mise en œuvre. Nous avons ainsi proposé que la comparaison se fasse par rapport :
– aux durées d'exécution des instructions non fonctionnelles (coût temporel) ;
– à la mémoire utilisée par les instructions non fonctionnelles (coût spatial) ;
– aux niveaux de priorité nécessaire ;
– aux coûts des changement de contexte ;

– à l'impact d'une mauvaise estimation des WCET (robustesse) ;
De plus, nous avons fait le constat que les mises en œuvre pouvaient se faire soit en ne permettant aucun écart par rapport à l'ordonnancement calculé hors-ligne (politique de mise en œuvre inflexible), soit en permettant aux tâches de s'exécuter plus tôt que prévu (politique de mise en œuvre flexible). Nous avons formalisé les conditions à respecter par les séquences d'ordonnancement pour qu'une mise en œuvre soit qualifiée de mise en œuvre flexible ou de mise en œuvre inflexible. Nous avons également montré que dans les deux cas (flexible et inflexible) les mises en œuvre sont valides.
Dans le troisième chapitre nous avons présenté sept techniques de mise en œuvre d'ordonnancement utilisables dans un contexte de tâches indépendantes. Nous avons aussi adapté ces techniques aux contextes d'utilisation de ressources critiques et de contraintes de précédence. L'étude de ces techniques nous a permis de déterminer des coûts de mise en œuvre. Nous avons fait le constat que les coûts spatiaux et temporels de nos techniques sont pour la plupart linéaires en nombre de tâches et/ou en nombre de blocs d'exécution.
Dans le quatrième chapitre, nous nous sommes posés la question du passage des algorithmes de mise en œuvre vers un langage et une API de programmation. Nous y avons donc proposé l'utilisation d'outils de génération de code. Afin de vérifier que nos techniques permettaient, sur un système d'exploitation temps réel, de suivre effectivement les ordonnancements hors-ligne, nous y avons également proposé un outil d'observation. Nous avons choisi le système d'exploitation Xenomai comme support des tests. Ce choix nous a aussi permis d'évaluer quantitativement certains coûts de mise œuvre. Ces données chiffrées ont été utilisées en complément de l'estimation des coûts faite dans le troisième chapitre pour émettre des recommandations d'utilisation des techniques. Par exemple en présence de ressources critiques ou de contraintes de précédences nous recommandons l'utilisation de la mise en œuvre par un répartiteur non concurrent à un processus.
Pour finir, nous avons présenté dans le cinquième chapitre l'utilisation de notre approche pour la mise en œuvre d'une application de gestion de la sécurité d'une mine.

Perspectives

Les travaux réalisés durant cette thèse ouvrent de notre point de vue un certain nombre de pistes. Nous présentons ici celles qui nous paraissent les plus pertinentes.

1. Pour valoriser beaucoup plus ce travail et le rendre utilisable en industrie, il faudrait réaliser complètement l'outil de génération de code que nous n'avons fait que prototyper. En effet nous n'avons abordé que la génération de code POSIX. Sachant que dans l'industrie d'autres API/langages telles que OSEK/VDX, VxWorks, ADA sont utilisés, il pourrait être intéressant de réaliser un outil qui permette de produire du code respectant leurs spécifications. Pour y arriver, nous préconisons dans un premier temps de définir un méta-modèle des API temps réel. Ce méta-modèle d'API serait utilisé pour exprimer les algorithmes des techniques de mise en œuvre. La génération de code se ferait alors en deux étapes :

  - génération du code d'une application avec le méta-modèle de l'API ;
  - transformation du code généré en code conforme à une API/langage à partir du modèle de l'API.

2. Une autre perspective est la prise en compte des conditionnelles par les techniques de mise en œuvre. En effet, dans nos travaux nous n'avons considéré que des séquences linéaires. En présence de conditionnelles, les séquences d'ordonnancement sont réprésentées par des arbres dont les branches matérialisent une alternative. Vue la puissance apportée par la prise en compte des conditionnelles dans la détermination des séquences d'ordonnancement, prendre en compte les conditionnelles dans nos techniques de mise en œuvre serait également un gain significatif. *A priori*, prendre en compte les conditionnelles dans un contexte non préemptif pourrait consister à introduire des alternatives dans le choix de la fonction à exécuter.
3. La prise en compte des changements de modes d'exécution c'est à dire les passages entre deux séquences d'ordonnancement serait également très intéressante. En effet, les systèmes embarqués fonctionnent rarement avec une seule configuration d'utilisation. En d'autres termes, sur un système embarqué il y a généralement plusieurs séquences d'ordonnancement à mettre en œuvre. Si le système est critique, les changements de modes sont également critiques et nécessitent d'être aussi pris en compte.
4. Sachant que les plate-formes embarquées évoluent de plus en plus vers le multi-processeur, il faudrait également prendre en compte les ordonnancements multi-processeurs et multi-cœurs ou les plateformes distribuées. On pourrait en particulier étudier la cohabitation de nos techniques de mise en œuvre avec les mises en œuvre en ligne. Il faudrait alors étudier les problèmes de migration de processus, de panne processeurs, d'utilisation de mémoire partagées.

Index

Annexes

Sommaire

A. Coûts des techniques de mise en œuvre

Par souci de simplifier les notations nous utilisons $|Se|$ à la place $|Se_{theorique}|$.

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date Attendre_Date Créer_Processus	1 1	 $\|Se\|$
Autres opérations		
Lecture mémoire Écriture mémoire Calcul	5 4 1	$1 + 3 \times \|Se\|$ $1 + \|Se\|$ $\|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date Processus	1 1	1
Autre coût		
Nombre de lignes	3	$5 + 3 \times \|Se\|$
Niveaux de priorité Pc= 1		
Coûts de changements de contextes Rc= 0		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Non	Oui	Oui

TABLE A.1 – Coûts de la mise en œuvre par un seul processus

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date Attendre_Date Créer_Processus	1 1	 $\|Se\|$
Autres opérations		
Lecture mémoire Écriture mémoire Calcul	5 4 1	$1 + 5 \times \|Se\|$ $2 + 2 \times \|Se\|$ $1 + 2 \times \|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date Processus Entier Table d'ordonnancement	1 1 	2 1 $\|Se\|$
Autre coût		
Nombre de lignes	3	11
Niveaux de priorité Pc= 1		
Coûts de changements de contextes Rc= 0		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Non	Oui	Oui

TABLE A.2 – Coûts de la mise en œuvre du répartiteur non concurrent à un processus

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date	1	
Attendre_Date		$\|Se\|$
Créer_Processus	1	$\|Se\|$
Supprimer_Processus		$\|Se\|$
Autres opérations		
Lecture mémoire	5	$1 + 7 \times \|Se\|$
Écriture mémoire	4	$2 + 2 \times \|Se\|$
Calcul	1	$1 + 2 \times \|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date	1	2
Processus	1	1
Entier		1
Table d'ordonnancement		$\|Se\|$
Autre coût		
Nombre de lignes	3	12
Niveaux de priorité Pc= 2		
Coûts de changements de contextes $Rc = 2 \times \|Se\|$		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Oui	Non	Oui

TABLE A.3 – Coûts de la mise en œuvre du répartiteur non concurrent à deux processus

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date	1	
Attendre_Date		$\|Se\|$
Créer_Processus	$2 + \|\tau\|$	
Prioroté_Processus		$2 \times \|Se\|$
Autres opérations		
Lecture mémoire	$8 + 4 \times \|\tau\|$	$1 + 9 \times \|Se\|$
Écriture mémoire	$4 + \|\tau\|$	$2 + 3 \times \|Se\|$
Calcul	$1 + \|\tau\|$	$1 + 3 \times \|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date	1	$2 + \|\tau\|$
Processus	$2 + \|\tau\|$	
Entier		1
Table d'ordonnancement		$\|Se\|$
Autre coût		
Nombre de lignes	$4 + \|\tau\|$	$16 + 6 \times \|\tau\|$
Niveaux de priorité $Pc = 4$		
Coûts de changements de contextes $Rc = 2 \times \|Se\|$		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Oui	Oui	Non

TABLE A.4 – Coûts de la mise en œuvre de répartition par affectation de priorités

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date	1	
Attendre_Date		$\|Se\|$
Créer_Processus	$1 + \|\tau\|$	
Arrêter_Processus	$\|\tau\|$	$\|Se\|$
Continuer_Processus		$\|Se\|$
Autres opérations		
Lecture mémoire	$5 + 5 \times \|\tau\|$	$1 + 9 \times \|Se\|$
Écriture mémoire	$4 + \|\tau\|$	$2 + 3 \times \|Se\|$
Calcul	$1 + \|\tau\|$	$1 + 3 \times \|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date	1	$2 + \|\tau\|$
Processus	$1 + \|\tau\|$	
Entier		1
Table d'ordonnancement		$\|Se\|$
Autre coût		
Nombre de lignes	$3 + 2 \times \|\tau\|$	$16 + 6 \times \|\tau\|$
Niveaux de priorité $Pc = 2$		
Coûts de changements de contextes $Rc = 2 \times \|Se\|$		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Oui	Oui	Non

TABLE A.5 – Coûts de la mise en œuvre de répartition par gestion d'état

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date	1	
Attendre_Date		$\|Se\|$
Créer_Processus	$\|\tau\|$	
Créer_Semaphore	$\|\tau\|$	
Vendre_Semaphore		$\|Se\|$
Prendre_Semaphore	$\|\tau\| - 1$	$\|Se\|$
Autres opérations		
Lecture mémoire	$1 + 4 \times \|\tau\|$	$6 \times \|Se\|$
Écriture mémoire	$2 + 2 \times \|\tau\|$	$1 + 2 \times \|Se\|$
Calcul	$1 + \|\tau\|$	$2 \times \|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date	1	$\|\tau\|$
Processus	$\|\tau\|$	
Sémaphore		$\|\tau\|$
Entier		$\|\tau\|$
Tables d'ordonnancement		$\|Se\|$
Autre coût		
Nombre de lignes	$2 + 3 \times \|\tau\|$	$13 \times \|\tau\|$
Niveaux de priorité $Pc = 1$		
Coûts de changements de contextes $Rc = \|Se\|$		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Non	Oui	Oui

TABLE A.6 – Coûts de la mise en œuvre par synchronisation collective

Coût temporel		
	Bloc d'initialisation Nombre d'opérations avant cycle	Bloc temporel Nombre d'opérations par cycle
Opérations de l'API abstraite		
Lire_Date Attendre_Date Créer_Processus	1 $\|\tau\|$	 $\|Se\|$
Autres opérations		
Lecture mémoire Écriture mémoire Calcul	$2 + 4 \times \|\tau\|$ $2 + 2 \times \|\tau\|$ 1	$1 + 4 \times \|Se\|$ $2 + \|Se\|$ $1 + 2 \times \|Se\|$
Coût spatial		
	Bloc d'initialisation	Bloc temporel
Type de données de l'API abstraite		
Date Processus Entier Tables d'ordonnancement	1 $\|\tau\|$ 	$\|\tau\|$ $\|\tau\|$ $\|Se\|$
Autre coût		
Nombre de lignes	$2 + \|\tau\|$	$11 \times \|\tau\|$
Niveaux de priorité $Pc = 1$		
Coûts de changements de contextes $Rc = \|Se\|$		
Robustesse temporelle	Robustesse fonctionnelle	Robustesse topologique
Non	Oui	Non

TABLE A.7 – Coûts de la mise en œuvre par attente coopérative de dates

B. Makefile d'une application Xenomai

```
BIN = ApplicationName
CC = gcc

SUBDIR = .
SRC = $(foreach dir, $(SUBDIR), $(wildcard $(dir)/*.c))
OBJ = $(SRC:.c=.o) $(wildcard ../../common/*.o)
INCLUDES =
WARNINGS =
OPTIMISATION =
DEBUG =

XENO_CONFIG = /usr/xenomai/bin/xeno-config
XENO_POSIX_CFLAGS = $(shell $(XENO_CONFIG) --skin=posix --cflags)
XENO_POSIX_LDFLAGS = $(shell $(XENO_CONFIG) --skin=posix --ldflags)

CFLAGS = $(INCLUDES) $(XENO_POSIX_CFLAGS) $(WARNINGS) $(OPTIMISATION)
LDFLAGS = -lm $(XENO_POSIX_LDFLAGS) $(DEBUG)

all:.depend $(BIN)

%.o:%.c
    @echo "CC $<"
    @$(CC) -c $(CFLAGS) $< -o $@
$(BIN):    $(OBJ)
    @echo "Building ${BIN}"
    @$(CC) $(OBJ) -o $@ $(LDFLAGS)
clean:
    rm -f $(OBJ)
distclean: clean
    rm -f $(BIN)
    rm -f ./.depend
.depend: $(SRC)
    @echo "Génération des dépendances"
    @$(CC) $(CFLAGS) -MM $(SRC) > .depend
-include .depend
```

C. Patrons de mise en œuvre d'ordonnancement hors-ligne

```
[comment encoding = UTF-8 /]
[module genLansdo1('http://lansdo/1.0')]
[template public generateElement(aLansdo : Lansdo)]
[comment @main/]
[file (aLansdo.applicationName.concat('.c'), false, 'UTF-8')]
#include <pthread.h>
#include <time.h>
#include <stdio.h>
#include "timespec_util.h"
#define TIME_UNIT [aLansdo.timeUnit/]
#define H [aLansdo.sequenced.cycleLength/]
#define K 1000000000
/*Partie fonctionnelle de l'application */
[for (F : Function | aLansdo.contains.run)]
void * [F.functionName/](void * arg){
[F.functionBody/]
}
[/for]
void * tf(void *arg){
      timespec t;
      timespec t_0;
      timespec_set(&t_0,(timespec*)debut);
      for(;;){
[for(bloc : SchedulEntry | aLansdo.sequenced.described)]
            timespec_add_timespec_ns(&t,t_0,(time_t)[bloc.start/]*TIME_UNIT) ;
            clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,&t,NULL);
            [bloc.referenced.run.functionName/]();
[/for]
            timespec_add_timespec_ns(&t_0, t_0, H*TIME_UNIT);
      }
      return NULL;
}
/* Fonction main */
int main (int argc, char *argv['['/][']'/]){
      mlockall(MCL_CURRENT | MCL_FUTURE);
      timespec horloge;
      struct sched_param param;
      pthread_t tau;
      clock_gettime(CLOCK_REALTIME,&horloge) ;
      time_spec_add_ns(&horloge,K);
      pthread_attr_t attribut ;
      pthread_attr_init(&attribut);
      param.sched_priority = 1;
      pthread_attr_setschedpolicy(&attribut,SCHED_FIFO);
      pthread_attr_setschedparam(&attribut,&param);
      pthread_create(&tau,&attribut,tf,(void *)&horloge);
      pthread_join(tau,NULL);
    return 0;
}
[/file]
[/template]
```

FIGURE C.1 – Patron de mise en oeuvre d'une séquence par un processus

```
[comment encoding = UTF-8 /]
[module genLansdo1('http://lansdo/1.0')]


[template public generateElement(aLansdo : Lansdo)]
[comment @main/]
[file (aLansdo.applicationName.concat('.c'), false, 'UTF-8')]
#include <pthread.h>
#include <time.h>
#include <stdio.h>
#include "timespec_util.h"
#define TIME_UNIT [aLansdo.timeUnit/]
#define H [aLansdo.sequenced.cycleLength/]
#define K 1000000000
/*Partie fonctionnelle de l'application */
[for (F : Function | aLansdo.contains.run)]
void * [F.functionName/](void * arg){
[F.functionBody/]
}
[/for]
typedef struct{
    time_t start_time;
    void * (*fonction)(void*);
}schedule_elt;
/* Table d'ordonnancement */
schedule_elt table['['/]']'/]={
[for(bloc : SchedulEntry | aLansdo.sequenced.described) separator(',')] {[bloc.start/],
[bloc.referenced.run.functionName/]}[/for]
}
/*Partie temporelle de l'application */
void *tf(void *debut){
      int k=0;
      timespec t, t_0;
      timespec_set(&t_0,(timespec*)debut);
      for(;;){
            timespec_add_timespec_ns(&t,t_0,(time_t)
table['['/]k[']'/].start_time*TIME_UNIT );
            timespec_print(t_0);
            clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,&t,NULL);
            table['['/]k[']'/].fonction(NULL) ;
            k=k+1;
            if(k==s){
                  k=0;
                  timespec_add_timespec_ns(&t_0,t_0,(time_t)H*TIME_UNIT);
            }
      }
      return NULL;
}
/* Fonction main */
int main(){
      pthread_t tau_rep;
      timespec horloge;
      struct sched_param param;
      clock_gettime(CLOCK_REALTIME,&horloge) ;
      timespec_print(horloge);
      time_spec_add_s(&horloge,K);
      timespec_print(horloge);
      pthread_attr_t attribut ;
      param.sched_priority = 2;
      pthread_attr_setschedpolicy(&attribut,SCHED_FIFO);
      pthread_attr_setschedparam(&attribut,&param);
      pthread_create(&tau_rep,&attribut,tf,(void*)&horloge) ;
   pthread_join(tau_rep,NULL);
      return 0;
}
[/file]
[/template]
```

FIGURE C.2 – Patron de mise en oeuvre par répartiteur non concurrent à un processus

D. Programme d'estimation dynamique des durées d'exécution

```
#define M 1000  /** facture multiplicatif des mesures */
int err; /**< stockage du code d'erreur */
/*********Definitions pour les mesures******/
timespec debut,fin; pthread_t tau;
/**********Affichage des mesures****************/
void affiche_mesure(){
        int i;
        printf("Erreur\tDebut_s\tDebut_ns\tFin_s\tFin_ns\n");
        printf( "%d\t",err[i]); timespec_print(debut); timespec_print(fin);
}
/******* Code a mesurer *******/
int code(){     return 1; }
/******* Fonction de mesure ******/
void *mesure_code(void *date){
        int i;
        clock_gettime(CLOCK_REALTIME,&debut) ;
        for(j=0;j<M;j++)
                err = err  + code();
        clock_gettime(CLOCK_REALTIME,&fin) ;
}
int main (void) {
        mlockall(MCL_CURRENT | MCL_FUTURE);
        struct sched_param param;
        pthread_t tau;
/*Creer thread de mesure*/
        pthread_attr_t attribut ;
        pthread_attr_init(&attribut);
        param.sched_priority = 1;
        pthread_attr_setschedpolicy(&attribut,SCHED_FIFO);
        pthread_attr_setschedparam(&attribut,&param);
        pthread_create(&tau,&attribut,mesure_code,NULL);
/*Affichage mesure*/
        pthread_join(tau,NULL);
        affiche_mesure();
  return 0;
}
```

Notations

- $\alpha(i)$: numéro de la tâche correspondant au bloc d'exécution numéro i ;
- $\beta(i)$: numéro de l'instance de tâche correspondant au bloc d'exécution numéro i ;
- C_i^{max} : durée d'exécution de la tâche τ_i au pire cas ;
- C_i^{min} : durée d'exécution de la tâche τ_i au meilleur cas ;
- C_{ij} : durée d'exécution de l'instance de tâche τ_{ij} ;
- D_0 : durée de référence permettant l'initialisation d'une application ;
- D_i : échéance de la tâche τ_i ;
- d_{ij} : délai d'exécution de l'instance de tâche τ_{ij} ;
- end_i : date de fin du bloc d'exécution numéro i ;
- f_i : fonction exécutée par la tâche τ_i ;
- $\gamma(i)$: numéro du bloc d'exécution théorique correspondant au bloc d'exécution effectif numéro i ;
- H : hyper-période ou plus petit commun multiple des périodes des tâches ;
- M : mot topologique du scénario d'exécution théorique ;
- M' : mot topologique du scénario d'exécution effectif ;
- r_i : première date d'activation de la tâche τ_i ;
- r_{ij} : date d'activation de l'instance de tâche τ_{ij} ;
- $Se(t_1, t_2)$: scénario d'exécution ou liste de blocs d'exécution entre les dates t_1 et t_2 ;
- $Se_{theorique}$: scénario d'exécution calculé hors-ligne ;
- $Se_{effectif}$: scénario d'exécution effectif ;
- $Se_{theorique}^{Annote}$: scénario d'exécution théorique annoté de toutes les sous-fonctions ;
- $start_i$: date de début du bloc d'exécution numéro i ;
- σ_i : processus numéro i ;
- T_i : période de la tâche τ_i ;
- t_c : date du dernier temps creux acyclique ;
- τ_{ij} : instance numéro j de la tâche numéro i ;

– τ_i : tâche numéro i ;
– τ_i^{Annote} : tâche numéro i annotée de toutes ses sous-fonctions ;
– U : facteur d'utilisation du système de tâche.

Acronymes

- Api : Application Programming Interface
- ARINC : Aeronautical Radio Incorporated ;
- BCET : Best Case Execution Time ;
- DM : Deadline Monotonic ;
- EDF : Earliest Deadline First ;
- FIFO : First In First Out ;
- IEEE : Institute of Electrical and Electronics Engineers ;
- ISR : Interupt Service Routine ;
- LL : Least Laxity ;
- OSEK : Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug ;
- PCP : Priority Ceiling Protocol ;
- PIP : Priority Inheritance Protocol ;
- POSIX : Portable Operating System Interface
- ppcm : plus petit commun multiple ;
- RM : Rate monotonic ;
- RTAI : Real Time Application Interface ;
- RTOS : Real Time Operating System ;
- TT : Time Triggered ;
- VDX : Vehicle Distributed eXecutive ;
- WCET : Worst Case Execution Time.

Bibliographie

[93801] Ieee standard test access port and boundary scan architecture. IEEE Std 1149.1-2001, July 2001.

[ABD+95] Neil C Audsley, Alan Burns, Robert I Davis, Ken W Tindell, and Andy J Wellings. Fixed priority pre-emptive scheduling : An historical perspective. *Real-Time Systems*, 8(2-3) :173–198, 1995.

[Acc] Acceleo - transforming models into code. http ://www.eclipse.org/acceleo/.

[ADE05] Life with adeos. http ://www.xenomai.org/documentation/xenomai-2.3/pdf/Life-with-Adeos-rev-B.pdf, 2005.

[ari03] Airlines electronic engineering commitee (aeec), avionics application software standard interface (arinc specification 653-1), 2003.

[AW88] N. Altman and N. Weiderman. Timing variation in dual loop benchmark. *Ada Lett.*, VIII(3) :98–106, April 1988.

[BB90] J. Beauquier and B. Bérard. *Systèmes d'exploitation - Concepts et algorithmes.* Ediscience international, 1990.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1) :64–83, 2003.

[BdS96] Frédéric Boussinot and Robert de Simone. The sl synchronous language. *IEEE Transactions on Software Engineering*, 22(4) :256–266, 1996.

[Bea96] Jean-Pierre Beauvais. *Etude d'algorithmes de placement de taches temps réel périodiques complexes dans un système réparti.* PhD thesis, Université de Nantes, 1996.

[Ber00] Gérard Berry. The esterel v5 language primer - version v5.91. ftp ://ftp-sop.inria.fr/marelle/Laurent.Thery/esterel/esterel.pdf, June 2000.

[Bla00] William Blacher. Le temps réel. Master's thesis, Ecole Nationale Supérieure d'Informatique et de Mathématique Appliquées de Grenoble, 26 Juin 2000.

[BLM+07] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance comparison of vxworks, linux, rtai and xenomai in a hard real-time application. In *Real-Time Conference, 2007 15th IEEE-NPSS*, pages 1–5. IEEE, 2007.

[BM05] Sanjeev Baskiyar and Natarajan Meghanathan. A survey of contemporary real-time operating systems. *Informatica (Slovenia)*, 29(2) :233–240, 2005.

[Bou91] Frédéric Boussinot. Reactive c : An extension of c to program reactive systems. *Software Practice and Experience*, 21(4) :401–428, 1991.

[BS74] Kenneth R Baker and Zaw-Sing Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Research Logistics Quarterly*, 21(1) :171–176, 1974.

[BS89] T. P. Baker and Alan Shaw. The cyclic executive model and ada. *Real-Time Systems*, 1(1) :7–25, 1989.

[BSD+01] Volker Barthelmann, Anton Schedl, Elmar Dilger, Thomas Führer, Bernd Hedenetz, Jens Ruh, Matthias Kühlewein, Emmerich Fuchs, Yaroslav Domaratsky, Andreas Krüger, Patrick Pelcat, Martin Glück, Stefan Poledna, Thomas Ringler, Brian Nash, and Tim Curtis. Osek/vdx time-triggered operating system 1.0. http ://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf, July 2001.

[Bur90] Alan Burns. *Real-Time Systems and Their Programming Languages*. Addison Wesley Publishing Company, 1990.

[Bur95] Alan Burns. Advances in real-time systems. chapter Preemptive Priority-based Scheduling : An Appropriate Engineering Approach, pages 225–248. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[But97] Giorgio C. Buttazzo. *Hard Real-time Computing Systems : Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[CG03] A. Choquet-Geniet. Panorama de lordonnancement temps réel monoprocesseur. *École d'Été Temps Réel (ETR2003)*, pages 263–276, 2003.

[CG05] Francis Cottet and Emmanuel Grolleau. *Systèmes temps réel de contrôle-commande*. 2005.

[CG07] Bernard Chauvière and Dominique Geniet. Une approche markovienne pour l'étude de systèmes temps-réel à contraintes strictes. *Technique et Science Informatiques*, 26(10) :1269–1303, 2007.

[CGG04] Annie Choquet-Geniet and Emmanuel Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theoretical computer science*, 310(1) :117–134, 2004.

[cod] Codetest embedded software test and analysis tools. http ://www.amc.com/products/embedded_sw_test.html.

[CV98] Adam M. Costello and George Varghese. Redesigning the bsd timer facilities. *Softw., Pract. Exper.*, 28(8) :883–896, 1998.

[dACMdA+08] Gustavo Rau de Almeida Callou, Paulo Romero Martins Maciel, Ermeson Carneiro de Andrade, Bruno Costa e Silva Nogueira, and Eduardo Antonio Guimarães Tavares. A coloured petri net based approach for estimating execution time and energy consumption in embedded systems. In *Proceedings of the 21st annual symposium on Integrated circuits and system design*, pages 134–139. ACM, 2008.

[Der74] Michael L. Dertouzos. Control robotics : The procedural control of physical processes. In *Proceedings of IFIP Congress (IFIP'74)*, pages 807–813, Stockholm, Sweden, 1974.

[DFP01] Radu Dobrin, Gerhard Fohler, and Peter P. Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 225–234. IEEE, 2001.

[DG00] Raymond R. Devillers and Joël Goossens. Liu and layland's schedulability test revisited. *Information Processing Letters*, 73(5) :157–161, 2000.

[DGC02] Laurent David, Emmanuel Grolleau, and Sébastien Constantin. Plateforme d'expérimentation pour l'ordonnancement des applications temps réel à contraintes strictes. In *Real Time Systems (RTS 02)*, pages 33–46, 2002.

[dKdOSB00] Jacques Chassin de Kergommeaux, Benhur de Oliveira Stein, and Pierre-Eric Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. pages 133–140, 2000.

[DM89] Michael L. Dertouzos and Aloysius K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions Software Engineering*, 15(12) :1497–1506, 1989.

[DOR10] François DORIN. *Contributions à l'ordonnancement et l'analyse des systèmes temps réel critiques*. PhD thesis, ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique, 2010.

[EMF] Eclipse modeling framework (emf) project. http ://www.eclipse.org/modeling/emf/.

[ET10a] Jean-Pierre Elloy and Yvon Trinquet. Systèmes d'exploitation temps réel - exemples d'exécutifs industriels. *Techniques de l'ingénieur Supervision des systèmes industriels*, (ref. article : s8052), 2010.

[ET10b] Jean-Pierre Elloy and Yvon Trinquet. Systèmes d'exploitation temps réel - principes. *Techniques de l'ingénieur Supervision des systèmes industriels*, (ref. article : s8050), 2010.

[Fag97] Alain Fagot. *Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers.* PhD thesis, Institut National Polytechnique de Grenoble, 1997.

[FBH+06] Helmut Fennel, Stefan Bunzel, Harald Heinecke, Jürgen Bielefeld, Simon Fürst, Klaus-Peter Schnelle, Walter Grote, Nico Maldener, Thomas Weber, Florian Wohlgemuth, et al. Achievements and exploitation of the autosar development partnership. *SAE Convergence*, 2006 :10, 2006.

[FK06] Pierre Ficheux and Patrice Kadionik. Linux et le temps réel. reloaded. http ://pficheux.free.fr/articles/lmf/hs24/realtime/linux_realtime_reloaded_final_www.pdf, 2006.

[FMB+09] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles*, 2009.

[Fra07] Markus Franke. A quantitative comparison of realtime linux solutions. Technical report, Chemnitz University of Technology, 2007.

[Gal95] Bill O. Gallmeister. *POSIX. 4 : Programming for the Real Word.* O'Reilly & Associates, Inc, 1995.

[GCG02] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules using petri nets. *Discrete Event Dynamic Systems*, 12(3) :311–333, 2002.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof : A call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, volume 17, pages 120–126. ACM, 1982.

[GL07] Dominique Geniet and Gaëlle Largeteau. Wcet free time analysis of hard real-time systems on multiprocessors : A regular language-based model. *Theoretical Computer Science*, 388 :26 – 52, 2007.

[GLLR79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy. Optimization and approximation in deterministic sequencing and scheduling : a survey. In *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[Gro99] Emmanuel Grolleau. *Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur.* PhD thesis, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, 29-11-1999.

[Gro05] OSEK/VDX Group. Operating system specification 2.2.3. February 17, 2005.

[GWWM09] Markus Geimer, Felix Wolf, Brian J.N. Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7) :375–388, 2009.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering*, 18(9) :785–793, 1992.

[HZPK07] Jérôme Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina. In *IEEE International Workshop on Rapid System Prototyping*, pages 106–112, 2007.

[Inc10] Free Software Fondation Inc. Gdb : The gnu project debugger, 2010.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. Wiley, April 1991.

[JP86] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5) :390–395, 1986.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, 2003.

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *International Conference on Computational Science (2)*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer, 2006.

[KMPP07] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. Ptrace, utrace, uprobes : Lightweight, dynamic tracing of user apps. In *Linux Symposium*, pages 215–224, 2007.

[Kod07] A Hariprasad Kodancha. Time management in partitioned systems. Master's thesis, Indian Institute of Science, 2007.

[Kra] Paul Kranenburg. strace : System call tracer.

[Lab74] J. Labetoulle. Un algorithme optimal pour la gestion des processus en temps réel. *Revue Francaise d'Automatique, Informatique et Recherche Opérationnelle*, pages 11–17, Fevr 1974.

[Lap04] Phillip A Laplante. *Real-time Systems Design and Analysis*. Wiley-Interscience, 2004.

[Leh90] John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *11th Real-Time Systems Symposium*, pages 201–209. IEEE, 1990.

[Lev] John Levon. Oprofile manual.

[LGA05] G. Largeteau, D. Geniet, and E. Andres. Discrete geometry applied in hard real-time systems validation. In *DGCI 2005 12th International Conference, Poitiers*, volume 3429 of *LNCS*, pages 23–33. Springer Verlag, 2005.

[LGTLL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(03) :261–303, 2003.

[LH02] Rick Lindsley and Dave Hansen. Bkl : One lock to bind them all. In *Ottawa Linux Symposium*, page 301, 2002.

[Liu00] Jane W. S. Liu. *Real-time systems*. Prentice Hall PTR, 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[LS99] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2) :183–207, 1999.

[LW82] Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4) :237–250, 1982.

[Mar82] Charles U. Martel. Preemptive scheduling with release times, deadlines, and due times. *Journal of the ACM*, 29(3) :812–829, 1982.

[Mok83] Aloysius Ka Lau Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology, 1983.

[MP05] Louis Mandel and Marc Pouzet. Reactiveml : a reactive extension to ml. In *International Conference on Principles and Practice of Declarative Programming*, pages 82–93. ACM, 2005.

[Nem08] Fadia Nemer. *Optimisation de l'estimation du WCET par analyse inter-tâche du cache d'instructions*. PhD thesis, Université de Toulouse III - Paul Sabatier, 2008.

[NS07] N. Nethercote and J. Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[Ouh13] Yassine Ouhammou. *Model-based Framework for Using Advanced Scheduling Theory in Real-Time Systems Design*. PhD thesis, ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique, dec 2013.

[PNME03] Michael J. Pont, Andrew J. Norman, Chisanga Mwelwa, and Tim Edwards. Prototyping time-triggered embedded systems using pc hardware. In *8th European Conference on Pattern Languages of Programms*, pages 691–716. UVK - Universitaetsverlag Konstanz, 2003.

[PR11] Carlos Hernan Prada Rojas. *Une approche à base de composants logiciels pour l'observation de systèmes embarqués*. These, Université de Grenoble, Jun 2011.

[Reg02] John Regehr. Inferring scheduling behavior with hourglass. In *USENIX Annual Technical Conference, FREENIX Track*, pages 143–156, jun 2002.

[RLB08] Paul Regnier, George Lima, and Luciano Barreto. Evaluation of interrupt handling timeliness in real-time linux operating systems. *Operating Systems Review*, 42(6) :52–63, 2008.

[RTA] Rtai - real time application interface. https ://www.rtai.org/.

[RVA09] Glauco Caurin Rafael V. Aroca. A real time operating systems (rtos) comparison. In *XXIX Congresso da Sociedade Brasileira de Computação*, 2009.

[SLJD08] Eric Senn, Johann Laurent, Emmanuel Juin, and Jean-Philippe Diguet. Refining power consumption estimations in the component based aadl design flow. In *Forum on specification and Design Languages*, pages 173–178. IEEE, 2008.

[SM06] Sameer Shende and Allen D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2) :287–311, 2006.

[Sou01] William Soukoreff. A comparaison of two preruntime schedule-dispatchers for rtai. Cosc6390 project report, Dept of Computer Science, York University, 2001.

[SR08] Oliver Scheickl and Michael Rudorfer. Automotive real time development using a timing-augmented autosar specification. In *4th Embedded Real Time Software*, 2008.

[SRG89] Wolfgang Schwabl, Johannes Reisinger, and Guenter Gruensteidl. A survey of mars. Technical Report Nr 16/89, Institut für Technische Informatik Tecnische Universität Wien Austria, October 1989.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions Computers*, 39(9) :1175–1185, 1990.

[SRS98] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline scheduling for real-time systems : EDF and related algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[SSP06] Fabian Scheler and Wolfgang Schroeder-Preikschat. Time-triggered vs. event-triggered : A matter of configuration ? In *GI/ITG Workshop on Non-Functional Properties of Embedded Systems*, pages 1–6. VDE, 2006.

[Ste06] David B. Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference Boston*, 2006.

[TCM98] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation : The optimisation-based approach. In *IFIP International Workshop on Dependable Computing and its Applications (DCIA 98)*, pages 169–180, 1998.

[TG06] MIPI Test and Debug Working Group. Mipi test and debug interface framework v3.2. http ://www.mipi.org/sites/default/files/whitepapers/MIPI_TDWG _whitepaper_v3_2.pdf, April 2006.

[Tim] Timetrace. http ://www.timesys.com/products/timetrace.html.

[Tim99] Martin Timmerman. Rtos market overview-a follow up. *REAL TIME MAGAZINE*, pages 6–8, 1999.

[VSK+08] Bart Vermeulen, Neal Stollon, Rolf Kühnis, Gary Swoboda, and Jeff Rearick. Overview of debug standardization activities. *IEEE Design & Test of Computers*, 25(3) :258–267, 2008.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3) :36, 2008.

[win] Windview. http ://www.wrs.com/pdf/wvGuide.pdf.

[WM01] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3) :241–268, 2001.

[Xen] Xenomai : Real-time framework for linux. http ://www.xenomai.org/.

[XP90] Jia Xu and David Lorge Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3) :360–369, 1990.

[Xu03] Jia Xu. Making software timing properties easier to inspect and verify. *IEEE Software*, 20(4) :34–41, 2003.

Résumé

Nous nous intéressons à la mise en œuvre effective d'applications temps réel dans une approche d'ordonnancement hors-ligne de systèmes de tâches périodiques. L'ordonnancement hors-ligne consiste à rechercher avant l'exécution de l'application une séquence pire cas, c'est-à-dire une suite de blocs indiquant une date de début et de fin d'exécution d'une instance de tâche. Mettre en œuvre une séquence suppose de spécifier ce qui doit se passer quand les durées d'exécution réelles sont inférieures aux durées pire cas prévues par la séquence. Notre première contribution consiste en la proposition de deux politiques de mise en œuvre : une politique inflexible qui respecte strictement les dates de début des blocs ; et une politique flexible qui permet de les avancer. Nous prouvons que ces politiques préservent la validité des séquences. Nous proposons ensuite un modèle de coûts pour l'évaluation et la comparaison de techniques respectant les politiques proposées. La seconde contribution concerne la proposition de techniques de mise en oeuvre. Dans un premier temps, nous proposons sept techniques de mise en oeuvre dans un contexte de tâches indépendantes et séquences sans préemption. Nous étendons ensuite l'utilisation de ces techniques aux séquences avec préemption, et aux tâches partageant des ressources critiques ou soumises à des contraintes de précédence. La troisième contribution concerne la mise en œuvre sous Posix. Nous présentons des outils de génération de code issus de l'ingénierie dirigée par les modèles. Nous proposons également un outil d'observation de séquences effectives. Enfin, une étude de cas présente l'utilisation pratique de notre approche.

Mots-clés : systèmes temps réel, ordonnancement hors-ligne, POSIX, ingénierie dirigée par les modèles

---

Abstract

We address the implementation of periodic task sets for off-line scheduling. Off-line scheduling approach consists in computing a worst-case schedule before runtime. Implementing a schedule requires to specify what must happen when the actual execution times of tasks are lower than the planned execution times. The first contribution consist of the formalization of implementation policies. These policies consider the date by which a task may start execution, which may or not occur before the planned start time. The inflexible policy does not allow a task to run before its planned start time, the flexible policy does. Since many implementations can comply with these two policies, we also propose a cost model which enables to perform some comparisons between these implementations. The second contribution is the proposition and the presentation of a set of algorithms which implement the pre-computed schedules. We first deal with independent task sets in a non preemptive context. These algorithms are then adapted to be used in the context of preemptive scheduling, with shared critical ressources and precedence constraints. Using the model driven engeneering, we next provide a Posix code generation tool. We also present a schedule observation tool. Finally, our work has been tested through a pratical case study.

Keywords : real-time systems, off-line scheduling, POSIX, model driven ingeneering

Secteur de recherche : Informatique et Applications

LABORATOIRE D'INFORMATIQUE ET D'AUTOMATIQUE POUR LES SYSTÈMES
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1, avenue Clément Ader – BP 40109 – 86961 Chasseneuil-Futuroscope Cédex
Tél : 05.49.49.80.63 – Fax : 05.49.49.80.64