iRbot

LEJOS RT

Cahier Des Charges Fonctionnelles

Lu et Approuvé par Serge MIDONNET le :	Signature :

Clément BESLON
Loïc BODILIS
Benoît BOUSQUET

Xavier DALLA-VECCHIA
Arnaud DELALANDE
Aravindan MAHENDRAN

Sommaire

Suivi des versions

Numéro de version	Date	Modifications apportées
1	2 Novembre 2009	Création
2	16 Novembre 2009	- Modification des objets du domaine - Ajout d'un scénario pour PIP - Ajout de scenarii pour PCE - Ajout de scenarii pour PCP - Ajout de scenarii pour la détection de fautes - Ajout des mots « cost enforcement », « cost overrun detetction », sensibilité et transitivité dans le glossaire
3	23 Novembre 2009	- Modification des objets du domaine (ajout des attributs) - Modification de l'objet du domaine « Thread » : remplacé par « Schedulable ». - Ajout d'un scénario pour EDF - Ajout des use cases de la JVM Lejos

Demande initiale

1. Cadre

Aujourd'hui, l'informatique fait parti entière de notre quotidien. On peut en trouver dans beaucoup d'objets qui nous entourent. Notre téléphone, notre télévision, notre voiture,… en sont tous équipés. Nous avons pu remarquer ces derniers temps que ces outils évoluent pour devenir de plus en plus intelligent. En effet, nous sommes en train de passer le cap de l'utilisation. Maintenant nos systèmes nous rendent des services, nous assistent, aident à la décision. La voiture en est un des meilleurs exemples. Celle-ci est capable d'analyser un certain nombre de paramètres et d'agir en conséquence. Elle est un véritable copilote. Elle assure la gestion des phares en fonction de la luminosité, la mise en route et arrêt des essuies glace, la détection d'obstacles ou le ralentissement des voitures qui nous devancent… Pour ce faire, le système doit être capable de surveiller et analyser tous les paramètres qu'il doit contrôler. Dans le cas ou cela ne fonctionne pas correctement, on peut facilement imaginer les conséquences que cela peut entraîner.

Des systèmes ont été conçus pour assurer le bon fonctionnement de ce type de programme. Ceux-ci s'appellent système temps réel. Les comportements et fonctionnements de ces systèmes sont définis dans une spécification nommée RTSJ (Real Time Spécification for Java).

2. Sujet

Notre projet consiste à implémenter la gestion du temps réel tel qu'il est décrit dans la norme RTSJ dans l'environnement Lego Mindstorms. Lego est une marque de jouet de construction pour enfant qui propose aussi des produits permettant de construire et programmer des robots autonomes. Ceux-ci possèdent un contrôleur qui permet de les commander. Ce contrôleur est à programmer par ses propres soins. Tous les outils nécessaires à la programmation sont fournis avec le jeu et toutes les folies sont permises. Seulement pour aller plus loin, Monsieur MIDONNET aimerait que le robot ait un comportement réaliste, c'est à dire que le système respecte le comportement Temps Réel, comme c'est le cas dans tous les robots. Dans le but de nous faciliter la tache, il nous a proposé de modifier une machine virtuelle qui a été développée pour fonctionner dans l'environnement Lego Mindstorms. Cette machine virtuelle s'appelle LEJOS, et permet de faire fonctionner des programmes Java. Elle devra implémenter différents algorithmes décrits dans la spécification des systèmes temps réel RTSJ. Nous devrons gérer la spécification sous ces différents aspects, c'est à dire mémoire, ordonnancement et synchronisation.

3. Enjeux

Bien que notre projet soit destiné au monde du jeu du grand public, nos avancées sur le projet LEJOS sont grandement attendues. En effet, toute une communauté travaille sur LEJOS. De plus, un certain nombre d'acteurs de l'enseignement utilisent les robots Lego chargés de la machine virtuelle pour effectuer des travaux pratiques et en cours afin de se former au domaine de la robotique et du temps réel.

Présentation de l'équipe

L'équipe iRboT, qui a pour but de développer une partie temps réel pour la machine virtuelle LEJOS, est composée de six personnes, à savoir : Clément BESLON, Loïc BODILIS, Benoit BOUSQUET, Xavier DALLA-VECCHIA, Arnaud DELALANDE et Aravindan MAHENDRAN.

1. Clément BESLON

Clément BESLON est responsable du contact avec le client.

Sa mission est de faire le lien entre le client et l'équipe tout au long du projet.

Il dialogue exclusivement avec le client entre les rendez-vous, gère les listes de questions et suggestions à soumettre au client.

Ses compétences techniques et humaines lui permettront de pouvoir aider le client à comprendre les avancées de l'équipe et de le rassurer ; il souhaite que le client se sente impliqué et confiant dans le projet. Pour lui, un projet réussi implique un client heureux.

Curieux, passionné par les défis, Clément aime quand chaque projet lui apporte de nouvelles connaissances.

2. Loïc BODILIS

Loïc BODILIS est un expert métier mais également un expert technique.

Sa mission est de faire en sorte que le projet respecte les spécificités fonctionnelles du temps réel mais aussi de vérifier la faisabilité et l'homogénéité du projet.

Ses compétences techniques lui permettront de pouvoir valider les spécificités lors des réunions avec le client et également le code produit et ses compétences métier du temps réel permettront de valider les parties fonctionnelles avec le client et de vérifier le respect des spécificités RTSJ.

Curieux, ayant soif de connaissance et de challenge, il ne reculera devant rien pour que le projet réussisse.

3. Benoit BOUSQUET

Benoit Bousquet occupe la fonction d'expert technique.

Son rôle est de garantir les solutions mises en place et leurs qualités. Il doit être capable de comprendre, d'analyser et de choisir des solutions proposées par l'équipe. Ces décisions,

Benoit est à même de les prendre grâce à ses connaissances, acquises aussi bien à l'école qu'à son entreprise.

Grâce au travail pointu qu'il effectue au sein de sa société ainsi qu'à sa nature curieuse, il a chaque jour un peu plus le goût de découvrir les nouvelles technologies. Rigoureux et conscient qu'une des clefs de la réussite d'un projet est la qualité des livrables proposés, Benoit met tout en œuvre pour ne laisser filtrer aucune erreur. De plus, cela permet une maintenance plus aisée pour le client.

Enfin, Benoit a toujours été attiré par le domaine de la robotique, et plus particulièrement leur développement. C'est donc tout naturellement qu'il s'est montré très enthousiaste et motivé à la découverte de ce projet.

4. Xavier DALLA-VECCHIA

Xavier DALLA-VECCHIA est gestionnaire du site web interne et des outils internes.

Il doit au sein du projet veiller à ce que les outils internes au projet soient correctement utilisés par l'ensemble des élèves qui composent le groupe. Il peut donc user de méthodologie pour veiller à ce que les outils soient disponibles et mis à jour pour l'ensemble des ses coéquipiers. Au sein de son entreprise, il est en charge du développement du Back Office de son entreprise, ce qui lui permet d'avoir une certaine expérience concernant les outils internes de gestion que d'autres personnes doivent utiliser.

Xavier compte sur son sens de l'organisation pour pouvoir regrouper voire recouper les informations qu'il collectera pour alimenter le site web interne si cela s'avère nécessaire. De plus, il possède un bon sens de la camaraderie ce qui permet de faciliter la communication au sein de l'équipe. Mais il sait comment crever les abcès pour résoudre les conflits au plus vite et trouver les solutions aux problèmes qui pourraient apparaître au sein de l'équipe.

Au sein de l'équipe, il possède un bagage technique moins dense que ses camarades. Cependant, il possède un réel intérêt dans les systèmes en temps réel, notamment car il est passionné d'automobile et que de nombreuses applications embarquées doivent être temps réel. Il est novice dans ce domaine, mais adore l'algorithmique et aime trouver les solutions les plus adéquates. Il possède donc une réelle motivation pour étoffer son expérience grâce à ce projet et saura faire les efforts nécessaires pour mener ce projet à bien avec son équipe.

5. Arnaud DELALANDE

Arnaud Delalande occupe la fonction de chef de projet.

Son rôle est d'assumer le pilotage du projet afin de respecter les délais et la qualité des livrables de l'étude de projet jusqu'à l'intégration de l'application.

Le chef de projet est le garant du respect des engagements contractés avec le client. D'autre part, il doit s'assurer de la cohésion de son équipe. Grâce à son expérience acquise au sein de la Société Générale, Arnaud apprend à appréhender les difficultés liées à la gestion de projet. De plus, il bénéficie, grâce à la formation enseignée à Ingénieurs2000 et ses différentes expériences professionnelles telle que développeur application en système de production dans la société MTAE, d'une aptitude à comprendre les aspects techniques du projet. Cette connaissance lui permettra, entre autre, de pouvoir mieux comprendre les problèmes qui peuvent arriver durant les phases de conception, développement et intégration afin de prendre les bonnes décisions. D'autre part, Arnaud possède un sens des relations humaines qui lui permet de prendre en compte les aspects humain du projet. Pour finir son caractère dynamique lui permet de motiver une équipe et ainsi augmenter sa productivité.

6. Aravindan MAHENDRAN

Aravindan MAHENDRAN est le responsable de la qualité de la documentation.

Il doit s'assurer que tous les documents qui doivent être fournis à monsieur Serge MIDONNET et à Ingénieurs2000 respectent le template de l'équipe iRboT et vérifier qu'il n'y ait pas de fautes d'orthographe. Ce rôle lui a été attribué et il l'a accepté car :

- il repère rapidement les fautes dans un texte. En effet, il a d'excellentes compétences en orthographe et il a participé à de nombreux concours-dictée (comme les dicos d'or).
- il rédige beaucoup de documents durant les périodes en entreprise. Ces documents sont de plusieurs types : des manuels utilisateurs, des documents de mise en production ainsi que des documents plus techniques (tests, planning…).
- il a quelques notions de graphisme qui lui ont permis de créer le template de l'équipe.

Outre ces qualités en documentation, Aravindan possède des compétences techniques qui seront utiles pour la phase de développement : il sait programmer en Java et en C qui sont les deux langages principalement utilisés dans le projet LEJOS. Même s'il a plus de difficultés en C qu'en Java, il est persévérant et arrive à surmonter les problèmes qu'il rencontre avec ce langage. De plus, il est très sociable et ouvert, ce qui lui est permet de ne pas se renfermer sur lui-même et de pouvoir aller voir son équipe en cas de problème.

7. Logo de l'équipe

Pour représenter notre équipe, nous avons élaboré ce logo :

Qui a plusieurs significations :

- iR pour représenter notre filière (informatique et réseau).
- RboT pour le travail que l'on va effectuer sur les robots.
- RT pour la partie temps réel (RealTime) que nous allons ajouter.

Description du logiciel

1. Présentation du projet

Notre objectif est de modifier la JVM de Lejos dans le but de lui implémenter un comportement temps réel. Cela signifie que notre système devra vérifier les points suivants :

- Etre préemptif

  Dans un système préemptif, une tâche en cours d'exécution peut être interrompue de manière à exécuter une autre tâche.

- Gérer les événements synchrones

  Certains événements peuvent être exécutés à des périodes fixes.

- Gérer les événements asynchrones

  Certains événements peuvent être exécutés à des périodes non prévisibles (appui sur des capteurs, détection d'un objet …).

- Garantir les délais maximums

  Certaines tâches peuvent avoir besoin de s'exécuter avant une heure précise ou au maximum à un délai défini.

- Garantir la faisabilité

  Il faut garantir qu'un système peut être temps réel. Pour cela, le choix d'un algorithme d'ordonnancement optimal est important. Nous devrons aussi être capables d'indiquer si le système est faisable.

- Gérer les priorités

  Un système temps réel doit gérer vingt huit priorités minimum. Les priorités au dessus de 10 sont considérées comme « temps-réel ». Une tâche de haute priorité pourra interrompre une tâche de priorité plus faible. Cependant il faut garantir que toutes les tâches s'exécutent bien à temps.

- Gérer la mémoire

  La norme RTSJ implique un accès direct à la mémoire, il faut donc garantir cet accès.

2. Acteurs

Les différents acteurs entrant en interaction avec notre système sont :

- Les capteurs

Le robot dispose d'un ensemble de capteurs définis :

- Capteur tactile : Réagit à une pression.
- Capteur photosensible : Mesure l'intensité lumineuse captée ou réfléchie par un objet.
- Capteurs de sons : Mesure l'intensité sonore en décibels.
- Capteurs d'ultrasons : Permet de détecter les objets et de mesurer les distances (en centimètre).
- Les servomoteurs peuvent servir de capteur de rotations.
- Boussole : Permet de connaitre la position du nord par rapport au robot.
- Capteur de couleurs : Distingue les différentes couleurs.
- Capteur accéléromètre : Permet au robot de se repérer dans l'espace et de reconnaître les mouvements du robot ainsi que les accélérations.

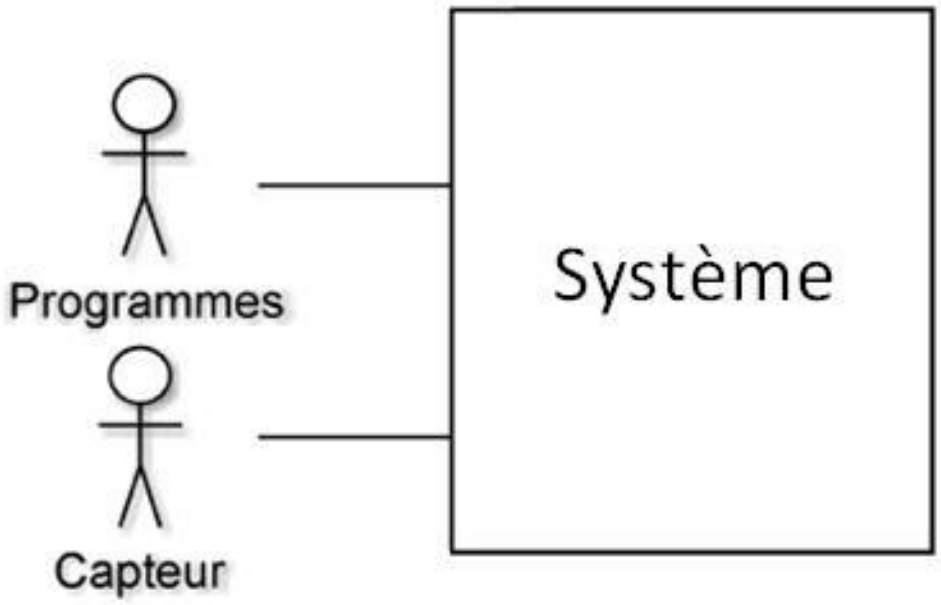
- Autodirecteur infrarouge : Permet de détecter des sources infrarouges, leurs forces et leurs directions.
- Capteur gyroscopique : Permet de détecter les rotations.
- Capteur infrarouge : Permet de communiquer avec certains autres appareils infrarouges.

Les capteurs déclenchent des événements asynchrones qui sont ensuite traités par des AsyncEventHandler de la machine virtuelle de LEJOS.

❖ Les programmes

Des développeurs peuvent insérer des programmes dans LEJOS. Ces programmes sont au format « .nxj ». Lors de leur exécution, ils sont interprétés par la JVM de LEJOS.

3. Interactions entres les acteurs et le système d'information



Chaque acteur peut entrer en contact avec notre système :

- Les capteurs

  Un appui sur un des capteurs peut déclencher une action asynchrone ou synchrone.

- Les threads

  Notre système devant être temps réel, nous devons gérer les différents threads ainsi que leurs priorités. Le but du temps réel étant que chaque thread soit fini d'exécuter avant sa date échéance.

  Des threads créés par le système peuvent également interagir avec la JVM.

- Les programmes

  Des développeurs peuvent insérer des programmes dans LEJOS. Ces programmes sont au format « .nxj ». Lors de leur exécution, ils sont interprétés par la JVM de LEJOS. Ces programmes peuvent déclencher des événements synchrones ou asynchrones qui sont ensuite traités par les RealTimeThread ou les NoHeapRealTimeThread que la JVM instancie selon le mode de gestion de la mémoire choisie.

Architecture

Comme présenté précédemment, LEJOS RT est une version temps réel du projet déjà existant LEJOS, qui devra respecter les normes définit dans RTSJ.

Afin de garantir une portabilité optimale au sein des divers utilisateurs actuels de LEJOS, l'architecture de notre version temps réel restera la même que celle de la solution actuellement existante.

LEJOS repose sur une JVM (machine virtuelle java) légère (ancien projet nommé *tinyVM*). Cette VM est elle-même étroitement liée à un "noyau" permettant l'interaction de celle-ci avec les divers périphériques et ressources système.

La norme RTSJ se caractérise par 7 grands axes :

- l'ordonnancement des threads ;
- la gestion de la mémoire ;
- la synchronisation ;
- la gestion des événements asynchrones ;
- le transfert de contrôle asynchrone ;
- la terminaison asynchrone des threads ;
- l'accès physique à la mémoire.

Dans notre cas, l'accès physique à la mémoire sera complètement ignoré.

Nos modifications porteront donc sur les points suivants :

1. Le Garbage Collector (GC)

Bien que le Garbage Collector facilite grandement la vie des développeurs, il s'avère gênant pour la programmation temps réel. En effet, il est impossible de prédire quand celui-ci se déclenchera ainsi que sa durée d'exécution. Il n'est donc pas possible de le prévoir dans l'ordonnancement des processus au même titre que les autres threads. D'autre part, il empêche la gestion manuelle de la mémoire, qui va à l'encontre de RTSJ. Il sera donc modifier afin de respecter nos contraintes, ou éventuellement débrayer afin de laisser complètement le champ libre aux développeurs.

2. L'ordonnanceur

L'ordonnanceur permet de définir l'ordre de fonctionnement des divers threads dans la JVM. Il sera modifié, dans un premier temps afin de permettre aux différents threads temps réel d'être ordonnancés de manière fixe, en fonction de leur priorité.

Puis, une autre modification entraînera la prise en charge par celui-ci d'un algorithme de synchronisation nommé *PIP*.

3. La mémoire

En temps réel, il existe différents types de mémoire, notamment la *ScopeMemory*, qui est une mémoire attachée à un ou plusieurs threads et qui vie tant qu'elle a des threads vivant qui lui sont rattachés, et l'*ImmortalMemory* qui vit aussi longtemps que la JVM. Les modifications porteront donc sur la prise en charge de tels types de mémoire.

4. Les moniteurs

Les moniteurs permettent de gérer les éléments partagés. Ce sont des jetons qu'il faut impérativement avoir avant de pouvoir prendre une ressource commune et le relâcher ensuite.

Dans une implémentation temps réel, les moniteurs jouent un rôle très important dans la synchronisation. En effet, il existe autant de types de moniteurs que d'algorithmes de synchronisation, il faudra donc mettre en place un moniteur par type de synchronisation qui sera implémenté dans LEJOS RT.

5. L'API de développement

Une fois toutes les modifications liées à la JVM et au noyau effectuées, une librairie d'objets Java sera réalisée afin permettre au développeur de créer des programmes temps réel utilisant notre solution. Cette API s'intègrera à l'API existante, et sera dans le paquetage *lejos.realtime*.

6. Lots et livrables

Le projet se découpe en 3 grands lots:

- la JVM LEJOS en version temps réel basé sur RTSJ.
- l'API java permettant d'utiliser les nouvelles fonctionnalités de la JVM.
- le module de faisabilité des systèmes temps réel.

Chacun des lots se découpe en livrable :

1. Lot JVM temps réel

Ce lot comprend la modification de la JVM pour la rendre temps réel en se basant sur les spécifications RTSJ. Cela comprend la modification de la mémoire pour la rendre compatible avec le Garbage Collector et de l'ordonnanceur pour qu'il gère les threads temps réel. Nous aurons :

- Le livrable mémoire qui contiendra les modifications sur la gestion du Garbage Collector et sur la mémoire immortelle.

* Pour rendre la mémoire compatible avec le Garbage Collector, nous allons implémenter la mémoire immortelle (mémoire non atteinte par le Garbage Collector, tout thread qui utilise la mémoire immortelle étant plus prioritaire que ce dernier).

- Le livrable scheduler qui contiendra les modifications sur l'ordonnanceur (l'ordonnanceur décide quel thread exécute son programme et interrompt les threads moins prioritaires pour exécuter le thread prioritaire).

* Pour que l'ordonnanceur gère les threads temps réel, nous allons implémenter l'algorithme EDF (algorithme de gestion des priorités des threads dynamique qui permet une restriction moins forte sur le programme temps réel).

* L'ordonnanceur doit également gérer les threads périodiques et les threads sporadiques. Nous allons également implémenter la fonction waitForNextPeriod() qui renvoie un booléen qui permet de savoir si la date d'échéance a été dépassée.

- Le livrable synchronisation qui contiendra l'ajout de la synchronisation dans l'ordonnanceur.

* Pour que l'ordonnanceur ne rencontre pas d'inter-blocages lors de synchronisation, nous allons implémenter l'algorithme PIP (algorithme de synchronisation qui permet de gérer les inter-blocages entre les threads de différentes priorités).

2. Lot API Java

Ce lot comprend l'ajout de l'API Java temps réel. Il sera composé des classes nécessaires pour faire du temps réel sur la JVM LEJOS. Nous aurons :

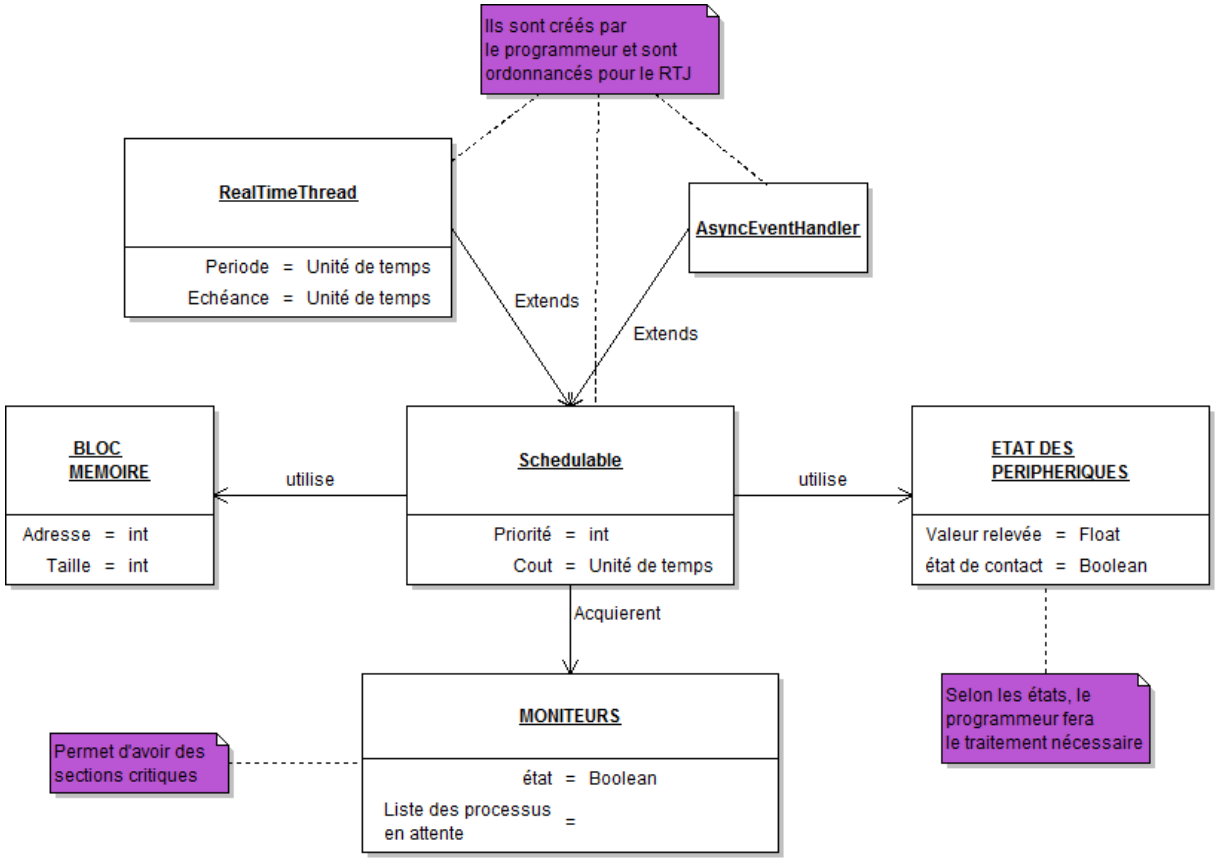
- le livrable mémoire qui utilisera le livrable mémoire de la JVM.
- le livrable scheduler qui utilisera le livrable scheduler de la JVM.

3. Lot Module de faisabilité

Ce lot est à la fois un lot et un livrable. Il permet au développeur de pouvoir vérifier que le programme pourra respecter les contraintes du temps réel avec la configuration d'ordonnancement et de synchronisation choisie.

Objets du domaine

Dans notre système, il existe quatre objets du domaine : la mémoire vive, les threads, l'état des capteurs et les moniteurs.



1. Mémoire vive

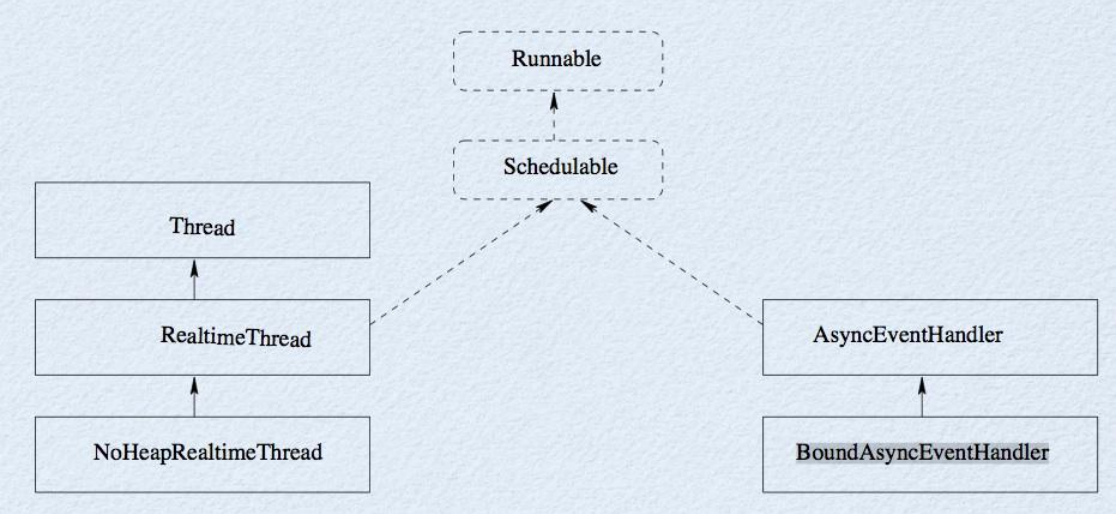
La mémoire vive ou RAM (pour Random Access Memory) est un dispositif électronique qui sert à stocker des données. Elle est rapide et permet de stocker notamment les programmes et les données manipulées par celui-ci lors de son exécution. Elle fonctionne uniquement lorsqu'elle est alimentée en courant électrique. La mémoire est un des acteurs les plus importants du Java Temps Réel. En effet, les fortes contraintes liées entre autre au déterminisme de son exécution doit permettre un accès direct à la mémoire. Différents algorithmes de gestion de la mémoire seront mis en place dans notre projet.

Les attributs associés à cet objet sont :

- L'adresse
- La taille

2. Schedulable

Un thread est un processus léger. Un processus est l'image d'un programme en mémoire. Cette image contient divers informations : le code, une pile d'exécution, un certain nombre de drapeaux et de variables qui caractérisent son exécution. Dans les programmes, certaines tâches n'ont aucune corrélation. Pour des questions de performance, il est parfois intéressant d'exécuter ces tâches en parallèle. Pour se faire il faut créer des Threads. Tous les Threads d'un processus partage la même zone mémoire. Les spécifications du temps réel donnent de fortes contraintes sur l'ordonnancement des Threads. Notre rôle consiste, entre autre, à modifier l'ordonnanceur de la machine virtuelle LEJOS afin qu'elle respecte ces contraintes. Le but de l'ordonnanceur est d'organiser l'exécution des Threads et leur allouer le temps processeur.



Les différents types de threads seront :

- RealTimeThread : Tâche périodique qui a un temps limité pour s'exécuter. Elle aura comme propriétés : une priorité, une fréquence, une échéance.
- NoHeapRealTimeThread : C'est un RealTimeThread qui ne manipulera que des objets en mémoire qui ne seront pas alloués sur le tas.

Les attributs associés aux threads sont :

- La priorité (entier : 10 < priorité < 90)
- La périodicité (temps en millisecondes)
- L'échéance (temps en millisecondes)
- Le coût (temps en millisecondes)

Les différents types de Handler seront :

- AsyncEventHandler : Tâche sporadique, c'est-à-dire qui ne s'exécute pas régulièrement. Elle aura comme propriété : une priorité.

- BoundAsyncEventHandler : C'est un AsyncEventHandler qui est attaché à un Thread. Quand on veut l'exécuter on démarre le Thread. Permet de limiter la latence du démarrage.

Les attributs associés à cet objet sont :

- La priorité (entier : 10 < priorité < 90)
- Le coût (temps en millisecondes)

3. L'état des capteurs

Le but du robot est d'interagir avec son environnement. C'est pourquoi il possède un grand nombre de périphériques. Il en existe plusieurs sortes, de type actionneur comme un moteur par exemple et des capteurs. Le développeur pourra programmer le robot en agissant sur tous ses périphériques. Tous les capteurs retournent des valeurs en fonction de ce qu'il observe : vrai ou faux pour le capteur de contact, un nombre réel pour le capteur température, etc. Pour utiliser cette valeur, il faut que le programme utilisateur, par le biais de notre machine virtuelle, demande l'état du capteur. L'actionnement ou la modification de leur état n'entraîne pas d'action de la part du système. C'est pourquoi l'état du capteur est un objet du domaine et non le capteur en lui même.

Les attributs associés à cet objet sont :

- La température (nombre réel)
- L'état de contact (booléen)

4. Moniteurs

Un moniteur est un élément informatique permettant la manipulation concurrente de données communes à plusieurs threads. C'est un jeton qu'un thread doit acquérir afin d'exécuter un morceau de code nommée *section critique*.

Dans cette section critique, nous avons l'assurance qu'il n'y aura jamais plus d'un thread à la fois, tant que le thread dans cette section ne sera pas sorti, aucun autre ne pourra y entrer. De ce fait, en plaçant ces ressources communes dans une telle partie de code, il est impossible que deux threads les modifient simultanément.

Les moniteurs pourront prendre comme propriété une valeur booléenne qui indique si le moniteur a été pris ou pas et une liste des Threads qui attendent la libération de celui-ci.

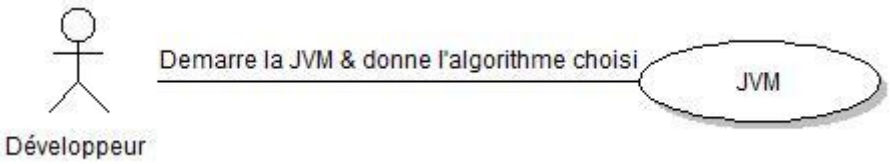
Les attributs associés à cet objet sont :

- L'état (booléen pour indiquer s'il est en cours d'utilisation ou non)
- La liste des processus en attente sur le moniteur

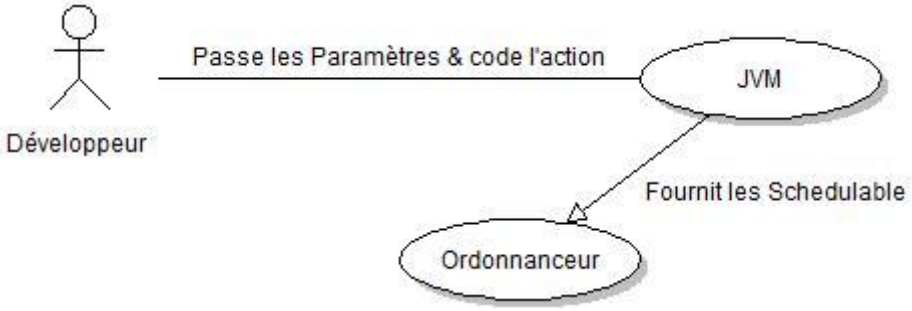
Use cases

La JVM Lejos dégage trois use cases : choix de l'ordonnanceur, création des threads temps réels et faisabilité.

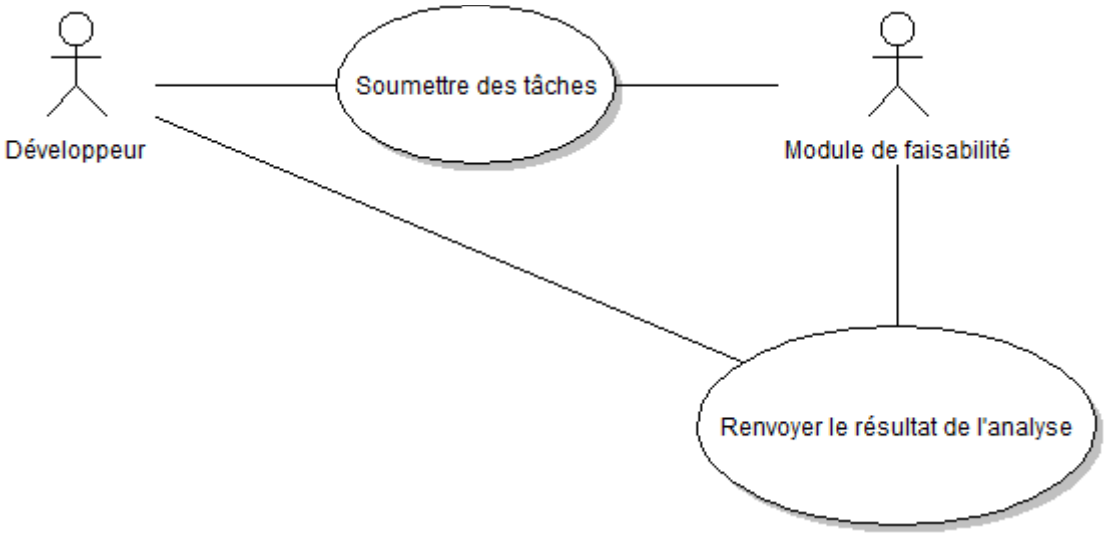
1. Choix de l'ordonnanceur



2. Création des threads temps réels



3. Faisabilité



Scénarii de tests

Cette partie décrit les scénarii de tests concernant les améliorations obligatoires à apporter dans le projet LEJOS. Cela concerne l'implémentation de la mémoire immortelle, du module de faisabilité, des threads périodiques et des algorithmes Rate Monotonic, Deadline Monotonic et du protocole PIP, ainsi que la détection des fautes.

1. Algorithme Rate Monotonic

Rate Monotonic est un algorithme d'ordonnancement temps réel en ligne à priorité fixe. Il est optimal dans la classe des algorithmes à priorité fixe, pour des systèmes de tâches indépendantes, synchrones et périodiques à échéances sur requête. Il attribue la priorité la plus forte à la tâche qui possède la plus petite période. Nous devons donc vérifier que les tâches à plus courtes périodes sont les premières à être exécutées et que toutes les instances des tâches respectent leur échéance.

Une seule instance d'une tâche peut être exécutée à la fois dans ce genre de systèmes. Chaque tâche possède un coût (C), une période (T) et une échéance (D).

On souhaite savoir si le système de tâches indépendantes, synchrones, et périodiques à échéances sur requête est ordonnançable par l'algorithme Rate Monotonic.

Cas nominal – Sans calcul

Scénario à deux threads

On lance deux threads Th1 et Th2. On considère que Th1 a une période inférieure à Th2.

Th1 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th2 est en cours d'exécution, Th1 préempte l'instance de Th2 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th2 n'est en cours d'exécution, Th1 exécute une nouvelle instance de lui-même directement.

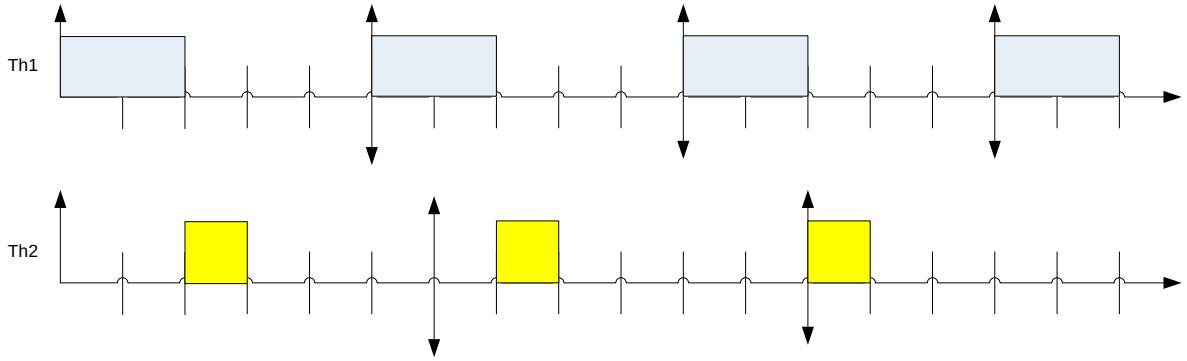
Th2 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th1 est en cours d'exécution, Th2 attend la fin de l'exécution de cette instance de Th1 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th1 n'est en cours d'exécution, Th2 exécute une nouvelle instance de lui-même directement.

Le système est ordonnançable par l'algorithme Rate Monotonic si toutes les instances de Th1 et Th2 se terminent avant leur échéance respective.

Exemple :

	Coût	Période	Échéance
Th1	2	5	5
Th2	1	6	6



Scénario à trois threads (généralisable pour plus de threads) – Scénario général

On lance trois threads Th1, Th2 et Th3. On considère que Th1 a une période inférieure à Th2 et que Th2 a une période inférieure à Th3.

Th1 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th2 est en cours d'exécution, Th1 préempte l'instance de Th2 puis exécute une nouvelle instance de lui-même.
- Si une instance de Th3 est en cours d'exécution, Th1 préempte l'instance de Th3 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th2 et de Th3 n'est en cours d'exécution, Th1 exécute une nouvelle instance de lui-même directement.

De manière générale, si le thread avec la période la plus courte veut lancer une nouvelle instance de lui-même, il doit préempter toutes les instances des autres threads en cours d'exécution.

Th2 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th1 est en cours d'exécution, Th2 attend la fin de l'exécution de cette instance de Th1 puis exécute une nouvelle instance de lui-même.
- Si une instance de Th3 est en cours d'exécution, Th2 préempte l'instance de Th3 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th1 et de Th3 n'est en cours d'exécution, Th2 exécute une nouvelle instance de lui-même directement.

De manière générale, si un thread avec une période intermédiaire (c'est-à-dire qu'il existe des threads avec des périodes plus courtes et plus longues) veut lancer une nouvelle

instance de lui-même, il doit préempter l'instance du thread en cours d'exécution si celui-ci possède une période plus longue ou bien, laisser l'instance du thread en cours d'exécution se terminer si ce dernier à une période plus courte. Ensuite, si aucun autre thread de période plus courte ne veut exécuter d'instance, le thread à période intermédiaire peut se réactiver.

Th3 veut exécuter une nouvelle instance de lui-même :

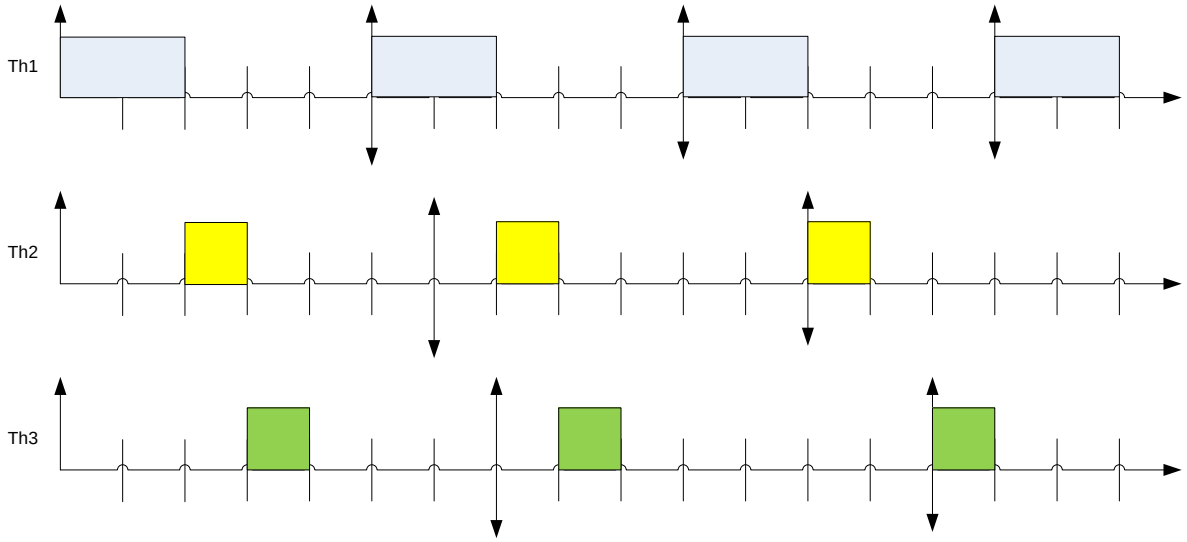
- Si une instance de Th1 est en cours d'exécution, Th3 attend la fin de l'exécution de cette instance de Th1 puis exécute une nouvelle instance de lui-même si une nouvelle instance de Th2 n'est pas lancée. Sinon, Th3 doit aussi attendre la fin de cette dernière.
- Si une instance de Th2 est en cours d'exécution, Th3 attend la fin de l'exécution de cette instance de Th2 puis exécute une nouvelle instance de lui-même si une nouvelle instance de Th1 n'est pas lancée. Sinon, Th3 doit aussi attendre la fin de cette dernière.
- Si aucune instance de Th1 et de Th2 n'est en cours d'exécution, Th3 exécute une nouvelle instance de lui-même directement.

De manière générale, si le thread avec la période la plus longue veut lancer une nouvelle instance de lui-même, il doit laisser toutes les instances des autres threads en cours d'exécution se terminer avant d'exécuter sa nouvelle instance.

Le système est ordonnançable par l'algorithme Rate Monotonic si toutes les instances de tous les threads se terminent avant leur échéance respective.

Exemples :

	Coût	Période	Échéance
Th1	2	5	5
Th2	1	6	6
Th3	1	7	7



Cas nominal – Avec calcul

Rappelons que chaque tâche possède un coût (C), une période (T) et une échéance (D). Il est possible de vérifier que le système est ordonnançable par l'algorithme Rate Monotonic à l'aide de l'analyse de l'utilisation du processeur (U) :

Pour un système à n tâches indépendantes, synchrones, et périodiques à échéances sur requête, si

$$U = \sum_{i=1}^{n} C_i/T_i \leq n(\sqrt[n]{2} - 1)$$

alors le système est ordonnançable par l'algorithme Rate Monotonic.

Exemple :

	Coût	Période	Échéance
Th1	2	5	5
Th2	1	6	6

$n*(2^{1/n} - 1) = 2 * ((2^{(1 / 2)}) - 1) = 0.828427125$

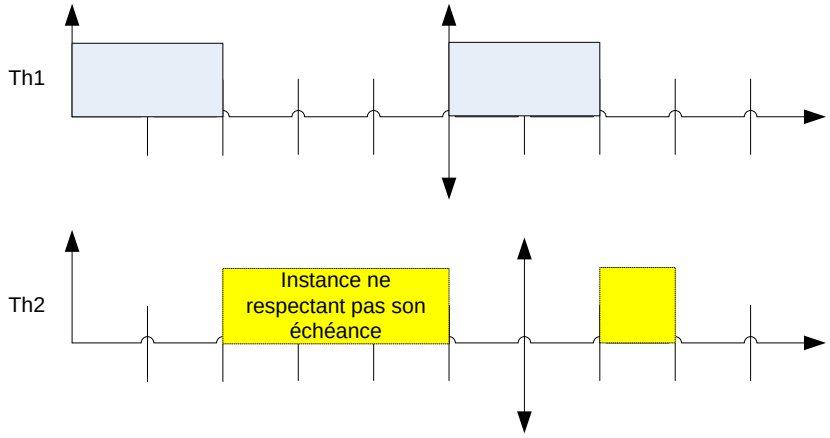
U = 2/5 + 1/6 = 0.566666667 < 0.828427125 donc le système est ordonnançable par l'algorithme Rate Monotonic.

Cas d'erreur – Sans calcul

Si en reprenant les cas nominaux précédents sans calcul on se rend compte qu'une des instances des threads ne respectent pas son échéance, le système n'est pas ordonnançable par l'algorithme Rate Monotonic.

Exemple :

	Coût	Période	Échéance
Th1	2	5	5
Th2	4	6	6



Cas d'erreur – Avec calcul

Dans un système à n threads, si en reprenant la formule précédente :

$$U = \sum_{i=1}^{n} C_i/T_i$$

le résultat obtenu est supérieur à 1, alors le système n'est pas ordonnançable par Rate Monotonic et plus encore, il n'est pas faisable. Mais si le résultat obtenu est compris entre 1 et $n*(2^{1/n} - 1)$, alors on ne peut pas conclure sur le fait de savoir si le système est ordonnançable par Rate Monotonic. Il faut passer au cas nominal sans calcul pour statuer.

Exemple :

	Coût	Période	Échéance
Th1	2	5	5
Th2	4	6	6

$n*(2^{1/n} - 1) = 2 * ((2^(1 / 2)) - 1) = 0.828427125$

U = 2/5 + 4/6 = 1.06666667 > 0.828427125 donc le système n'est pas ordonnançable par l'algorithme Rate Monotonic et il n'est pas faisable.

2. Algorithme Deadline Monotonic

Deadline Monotonic est un algorithme d'ordonnancement temps réel à priorité fixe. Il est optimal dans la classe des algorithmes à priorité fixe, pour des systèmes de tâches indépendantes, synchrones et périodiques à échéances inférieures aux périodes. Il attribue la priorité la plus forte à la tâche qui possède la plus petite échéance. Nous devons donc vérifier que les tâches à plus courtes échéances sont les premières à être exécutées et que toutes les instances des tâches respectent leur échéance.

Une seule instance d'une tâche peut être exécutée à la fois dans ce genre de systèmes. Chaque tâche possède un coût (C), une période (T) et une échéance (D).

On souhaite savoir si le système de tâches indépendantes, synchrones, et périodiques à échéances inférieures aux périodes est ordonnançable par l'algorithme Deadline Monotonic.

Cas nominal – Sans calcul

Scénario à deux threads

On lance deux threads Th1 et Th2. Chacun de ces threads possède une échéance inférieure à leur période et on considère que Th1 a une échéance inférieure à Th2.

Th1 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th2 est en cours d'exécution, Th1 préempte l'instance de Th2 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th2 n'est en cours d'exécution, Th1 exécute une nouvelle instance de lui-même directement.

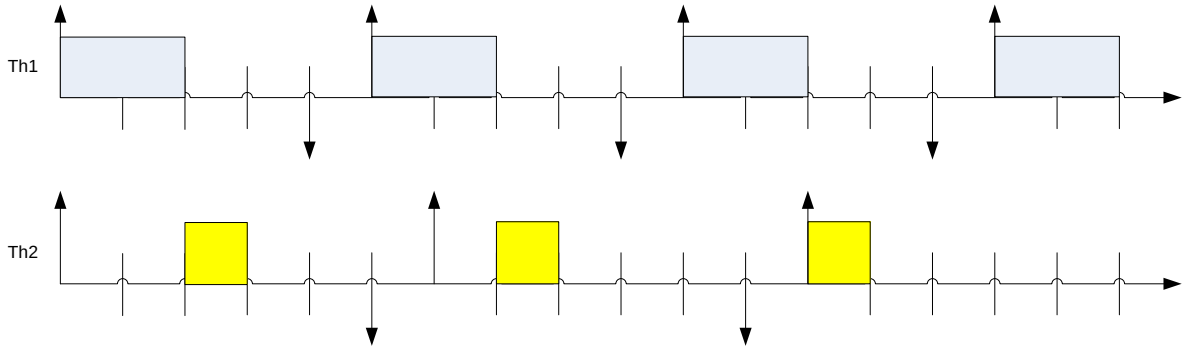
Th2 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th1 est en cours d'exécution, Th2 attend la fin de l'exécution de cette instance de Th1 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th1 n'est en cours d'exécution, Th2 exécute une nouvelle instance de lui-même directement.

Le système est ordonnançable par l'algorithme Deadline Monotonic si toutes les instances de Th1 et Th2 se terminent avant leur échéance respective.

Exemples :

	Coût	Période	Échéance
Th1	2	5	4
Th2	1	6	5



Scénario à trois threads (généralisable pour plus de threads) – Scénario général

On lance trois threads Th1, Th2 et Th3. Chacun de ces threads possède une échéance inférieure à leur période et on considère que Th1 a une échéance inférieure à Th2 et que Th2 a une échéance inférieure à Th3.

Th1 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th2 est en cours d'exécution, Th1 préempte l'instance de Th2 puis exécute une nouvelle instance de lui-même.
- Si une instance de Th3 est en cours d'exécution, Th1 préempte l'instance de Th3 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th2 et de Th3 n'est en cours d'exécution, Th1 exécute une nouvelle instance de lui-même directement.

De manière générale, si le thread avec la plus petite échéance veut lancer une nouvelle instance de lui-même, il doit préempter toutes les instances des autres threads en cours d'exécution.

Th2 veut exécuter une nouvelle instance de lui-même :

- Si une instance de Th1 est en cours d'exécution, Th2 attend la fin de l'exécution de cette instance de Th1 puis exécute une nouvelle instance de lui-même.
- Si une instance de Th3 est en cours d'exécution, Th2 préempte l'instance de Th3 puis exécute une nouvelle instance de lui-même.
- Si aucune instance de Th1 et de Th3 n'est en cours d'exécution, Th2 exécute une nouvelle instance de lui-même directement.

De manière générale, si un thread avec une échéance intermédiaire (c'est-à-dire qu'il existe des threads avec des échéances plus petites et plus grandes) veut lancer une nouvelle instance de lui-même, il doit préempter l'instance du thread en cours d'exécution si celui-ci possède une échéance plus grande ou bien, laisser l'instance du thread en cours d'exécution se terminer si ce dernier à une période plus petite. Ensuite, si aucun autre thread avec une échéance plus petite ne veut exécuter d'instance, le thread à échéance intermédiaire peut se réactiver.

Th3 veut exécuter une nouvelle instance de lui-même :

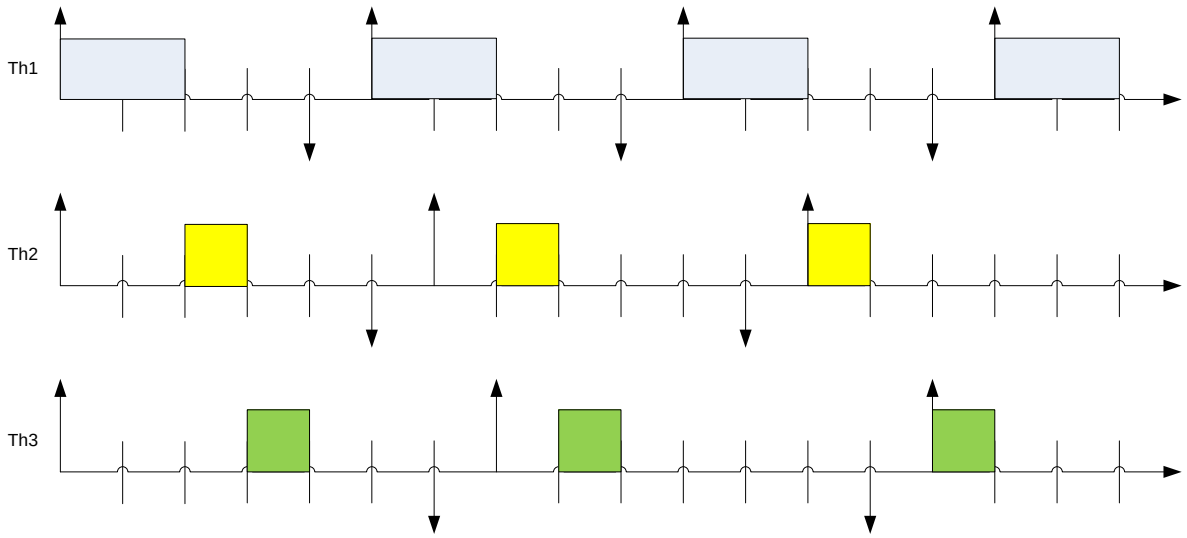
- Si une instance de Th1 est en cours d'exécution, Th3 attend la fin de l'exécution de cette instance de Th1 puis exécute une nouvelle instance de lui-même si une nouvelle instance de Th2 n'est pas lancée. Sinon, Th3 doit aussi attendre la fin de cette dernière.
- Si une instance de Th2 est en cours d'exécution, Th3 attend la fin de l'exécution de cette instance de Th2 puis exécute une nouvelle instance de lui-même si une nouvelle instance de Th1 n'est pas lancée. Sinon, Th3 doit aussi attendre la fin de cette dernière.
- Si aucune instance de Th1 et de Th2 n'est en cours d'exécution, Th3 exécute une nouvelle instance de lui-même directement.

De manière générale, si le thread avec l'échéance la plus grande veut lancer une nouvelle instance de lui-même, il doit laisser toutes les instances des autres threads en cours d'exécution se terminer avant d'exécuter sa nouvelle instance.

Le système est ordonnançable par l'algorithme Deadline Monotonic si toutes les instances de tous les threads se terminent avant leur échéance respective.

Exemples :

	Coût	Période	Échéance
Th1	2	5	4
Th2	1	6	5
Th3	1	7	6



Cas nominal – Avec calcul

Rappelons que chaque tâche possède un coût (C), une période (T) et une échéance (D). Il est possible de vérifier que le système est ordonnançable par l'algorithme Deadline Monotonic à l'aide d'une formule ressemblant à l'analyse de l'utilisation du processeur :

Pour un système à n tâches indépendantes, synchrones, et périodiques à échéances inférieures aux périodes, si

$$\sum_{i=1}^{n} C_i/D_i \leq n(\sqrt[n]{2} - 1)$$

alors le système est ordonnançable par l'algorithme Deadline Monotonic.

Exemple :

	Coût	Période	Échéance
Th1	2	5	4
Th2	1	6	5

$n*(2^{1/n} - 1) = 2 * ((2^{(1 / 2)}) - 1) = 0.828427125$

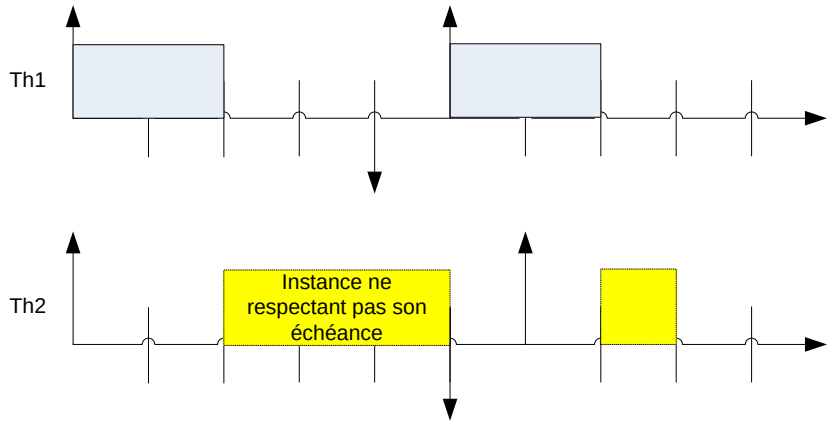
U = 2/4 + 1/5 = 0.7 < 0.828427125 donc le système est ordonnançable par l'algorithme Deadline Monotonic.

Cas d'erreur – Sans calcul

Si en reprenant les cas nominaux précédents sans calcul on se rend compte qu'une des instances des threads ne respectent pas son échéance, le système n'est pas ordonnançable par l'algorithme Deadline Monotonic.

Exemple :

	Coût	Période	Échéance
Th1	2	5	4
Th2	4	6	5



Cas d'erreur – Avec calcul

Dans un système à n threads, si en reprenant la formule précédente :

$$\sum_{i=1}^{n} C_i/D_i$$

le résultat obtenu est supérieur à 1, alors le système n'est pas ordonnançable par Deadline Monotonic et plus encore, il n'est pas faisable. Mais si le résultat obtenu est compris entre 1 et $n*(2^{1/n} - 1)$, alors on ne peut pas conclure sur le fait de savoir si le système est ordonnançable par Deadline Monotonic. Il faut passer au cas nominal sans calcul pour statuer.

Exemple :

	Coût	Période	Échéance
Th1	2	5	4
Th2	4	6	5

$n*(2^{1/n} - 1) = 2 * ((2^(1 / 2)) - 1) = 0.828427125$

U = 2/4 + 4/5 = 1.3 > 0.828427125 donc le système n'est pas ordonnançable par l'algorithme Deadline Monotonic et il n'est pas faisable.

3. Threads périodiques

Le processus doit être réactivé à intervalle fixe et attendre la réactivation lorsque son traitement est terminé. Les Threads périodiques permettent de pouvoir définir des tâches répétées utiles pour les spécifications RTSJ. Il y a des sous classes pour pouvoir paramétrer un thread périodique : PeriodicParameters qui étend ReleaseParameters. Un thread périodique est une tâche qui réalise une boucle à la fin de laquelle il appelle la méthode bloquante waitForNextPeriod().

Cas nominal

- Le programmeur crée un thread périodique et le configure.
- Le paramètre start est atteint et la tâche prévue commence.
- Le traitement se termine, la méthode waitForNextPeriod() s'active.
- La période se termine.
- Une nouvelle période commence et la tâche 2 est recommencée tant que le programme s'exécute.

4. Module de faisabilité

L'étude de faisabilité permet au développeur de savoir si son application temps-réel peut se dérouler correctement, si toutes les contraintes peuvent être respectées. Pour cela, le développeur crée ses threads, les ajoute au module d'étude de faisabilité qui renvoie si le système est faisable ou non.

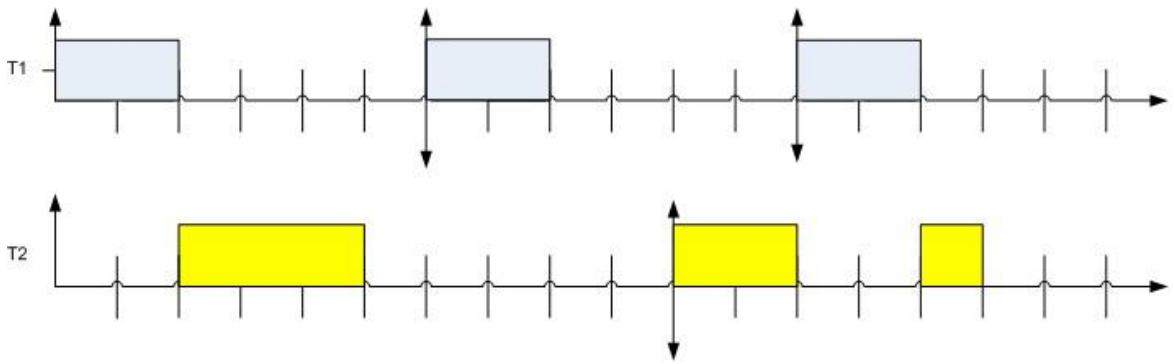
L'algorithme appliqué dans le module de faisabilité dépend des choix du développeur (Rate Monotonic, Deadline Monotonic, Earliest Deadline First, ...). Le déroulement de ces algorithmes ne sera donc pas développé dans cette partie.

Cas nominal

- Le programmeur crée deux threads :

Nom	Temps de traitement	Période	Priorité
T1	20	60	Maximale
T2	30	100	Normale

- Il indique que l'algorithme utilisé est "Rate Monotonic"
- Il appelle le module de faisabilité qui va appliquer des algorithmes pour voir si le système est faisable :



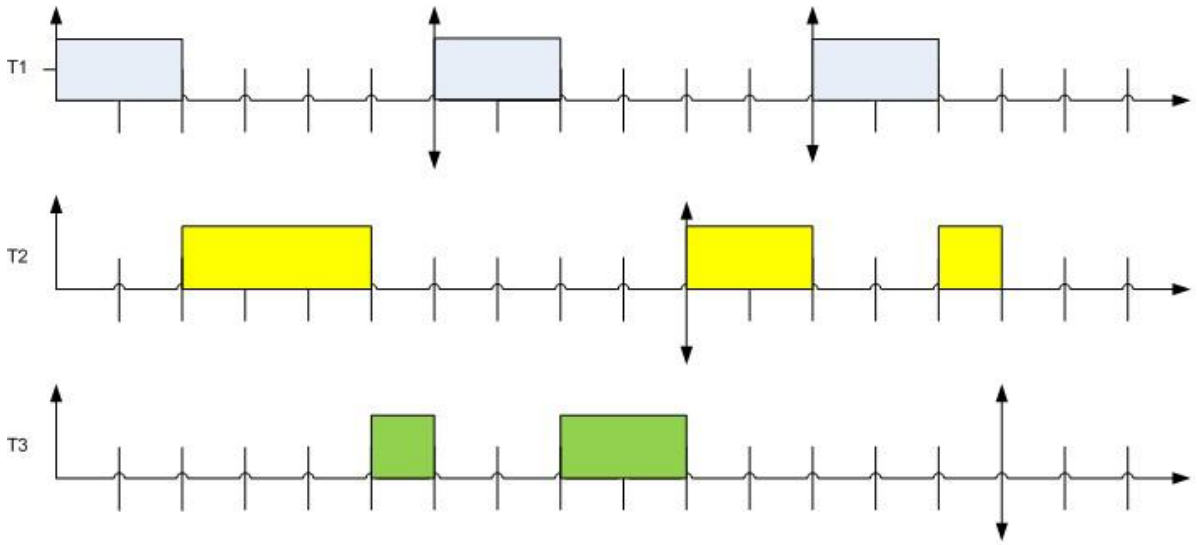
- Le module renvoie vrai, le système est ordonnançable de manière à respecter les échéances fixées.

Cas d'erreur

- Le programmeur décide de rajouter un thread au système précédent. Ce nouveau thread a une priorité moins importante que le thread T2 :

Nom	Temps de traitement	Période	Priorité
T1	20	60	Maximale
T2	30	100	Normale
T3	70	150	Faible

- Il indique que l'algorithme utilisé est "Rate Monotonic"
- Il appelle le module de faisabilité :



- Le module renvoie faux, le système ne peut être ordonnancé de manière à respecter toutes les échéances.

5. PIP

PIP (Priority Inheritance Protocol) est un protocole de synchronisation qui permet de gérer les inter-blocages entre les threads de priorités différentes. Il permet d'éliminer les problèmes d'inversion de priorité. Le principe de PIP est que quand un thread bloque des threads de plus haute priorité, sa priorité est changée temporairement et devient la priorité la plus élevée des threads bloqués. A la fin de l'utilisation de la ressource, le thread revient à sa priorité initiale.

L'héritage de priorité est transitif, lorsqu'un thread t1 bloque un thread t2 et que le thread t2 bloque un thread t3, alors t1 hérite de la priorité de t3 par t2.

On souhaite vérifier que les threads de priorités inférieurs héritent des priorités des autres threads au bon moment.

Cas nominal

Scénario à deux threads

Création de deux threads : un thread de priorité H (Ht1) et un thread de priorité L (Lt2).

- Lt2 prend la ressource A (ressource qui sera demandée par Ht1)
- Lt2 est préempté par Ht1 alors qu'il n'a pas libéré la ressource A
- Ht1 s'exécute et demande la ressource A
- Ht1 est préempté par Lt2 et Lt2 change de priorité, il reçoit la priorité de Ht1
- Lt2 s'exécute jusqu'à libérer la ressource A
- Lt2 reprend sa priorité initiale
- Ht1 préempte Lt2 et finit de s'exécuter.

Scénarii à trois threads

Scénario 1 :

Création de trois threads : un thread de priorité H (Ht1), un thread de priorité M (Mt2) qui devrait se lancer pendant l'héritage de priorité et un thread de priorité L (Lt3)

- Lt3 prend la ressource A (ressource qui sera demandée par Ht1)
- Lt3 est préempté par Ht1 alors qu'il n'a pas libéré la ressource A
- Ht1 s'exécute et demande la ressource A
- Ht1 est préempté par Lt3 et Lt3 change de priorité, il reçoit la priorité de Ht1
- Lt3 s'exécute, Mt2 essaye de se lancer mais ne peut pas car Lt3 a temporairement une priorité plus importante puis Lt3 libère la ressource A
- Lt3 reprend sa priorité initiale
- Ht1 préempte Lt3 et fini de s'exécuter
- Mt2 s'exécute.

Scénario 2 :

Création de trois threads : un thread de priorité H (Ht1), un thread de priorité M (Mt2) et un thread de priorité L (Lt3)

- Lt3 prend la ressource A (ressource qui sera demandé par Mt2)
- Lt3 est préempté par Mt2 alors qu'il n'a pas libéré la ressource A
- Mt2 s'exécute et demande la ressource A
- Mt2 est préempté par Lt3 et Lt3 change de priorité, il reçoit la priorité de Mt2
- Lt3 s'exécute,
- Ht1 préempte Lt3 et s'exécute jusqu'à la fin
- Lt3 fini de s'exécuter et libère la ressource A
- Lt3 reprend sa priorité initiale
- Mt2 préempte Lt3 et fini de s'exécuter

Scénario à quatre threads – cas général avec transitivité

Soient :

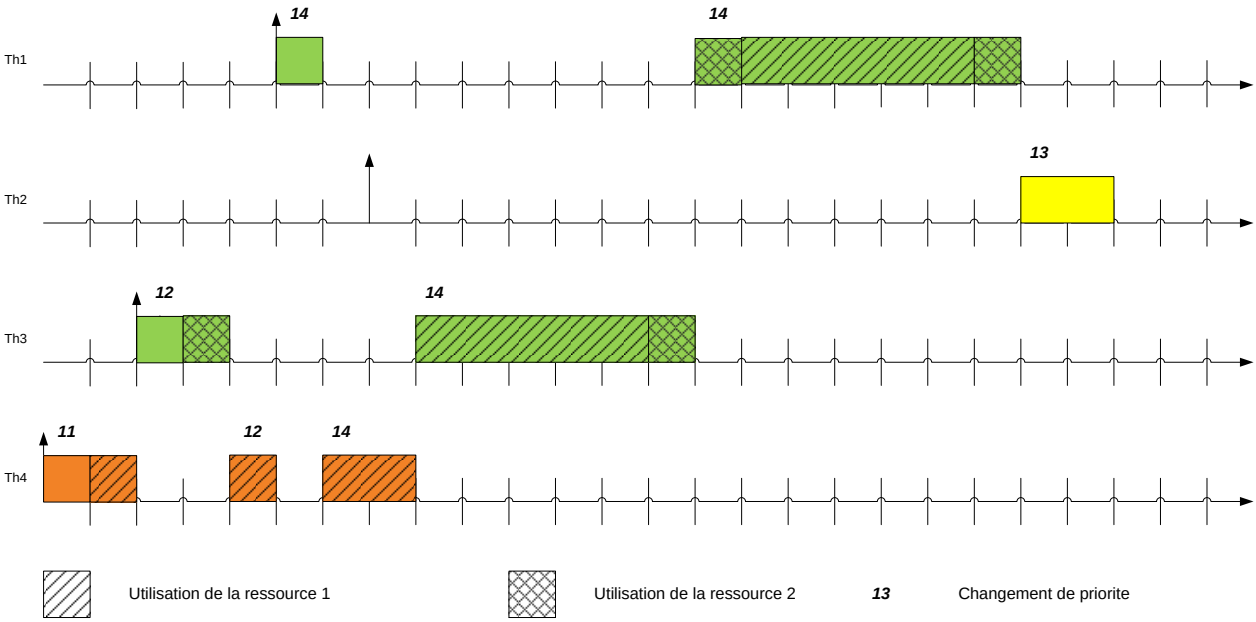
t1, t2, t3 et t4 quatre threads. t1 possède la priorité la plus élevée et t4 la plus faible.

Soient R1 et R2 deux ressources telles que la ressource R2 utilise la ressource R1 pendant 5 secondes.

Le développeur crée les 4 threads avec les paramètres suivants :

Nom	Temps de traitement	Période	Priorité
T1	90	50	14
T2	20	70	13
T3	80	20	12
T4	50	0	11

Le scheduler préempte les threads de la manière suivante :



- A T0, le thread t4 s'exécute, il attend une seconde puis demande la ressource R1.
- A T2, le thread t4 est préempté par le thread t3, il attend une seconde puis demande la ressource R2.
- A T3, la ressource R2 demande la ressource R1, celle-ci étant utilisée par le thread t4, la priorité de t4 devient la même que t3.
- A T5, le thread t1 s'exécute, il attend une seconde puis demande la ressource R2. Celle-ci est utilisée par le thread t3 qui est bloqué par le thread t4, la priorité du thread t4 devient donc la même que t1.
- A T6, t2 souhaite s'exécuter mais sa priorité est inférieure à celle de t4 (avec PIP) donc il ne préempte pas t4.
- A T8, t4 finit de s'exécuter, t3 a alors la même priorité que t1 et peut s'exécuter.
- A T14, t4 finit de s'exécuter. t1 peut s'exécuter.
- A T22, t1 finit de s'exécuter. t2 peut alors commencer.
- A T24, t2 finit de s'exécuter.

6. Mémoire immortelle

La mémoire immortelle est allouée à la création d'un objet et n'est jamais supprimée tant que la machine virtuelle est en fonctionnement. Le Garbage Collector ne doit pas intervenir sur cette mémoire. Pour vérifier le bon fonctionnement de la mémoire immortelle, il faut créer des objets dans celle-ci et vérifier que les objets ne sont jamais détruits (il n'y a pas d'instructions « delete » en Java) et que la mémoire est toujours occupée par ces objets crées.

7. Détection des fautes

Le but de cette partie est de réussir à détecter les fautes dans un système Temps Réel.

Les fautes peuvent être de différents types : un dépassement d'échéance ou un dépassement de coût. Une fois qu'une faute est détectée, il faut prévenir le programme de l'erreur et éventuellement lancer un évènement par défaut (par exemple, enlever la tâche de l'ordonnanceur) ou défini par le développeur.

Gestion des dépassements :

Le dépassement peut être strict : si un thread dépasse son coût alors on lance le handler.

Le dépassement peut être également souple c'est à dire que l'on définit une sensibilité : si un thread dépasse son coût, on attend un délai supplémentaire et si le thread ne s'est toujours pas lancé alors on lance le handler.

Cette sensibilité peut être déclarée dans la JVM en dure (la même pour tous les threads (efficacité)) ou faire une déclaration personnalisée pour chaque thread (flexible pour le développeur).

Scénarii :

- Dépassement d'échéance :

* Système strict avec aucune indulgence de retard :

¤ avec un handler défini par le développeur

- définition du système :

Tâches	Départ	Coût	Echéance	Période	Priorité
T1	0	2	6	6	14
T2	0	3	4	4	13

- résultat :

- T1 démarre

- T1 consomme 2 temps

- T1 se termine

- T2 démarre

- T2 consomme 2 temps

- lancement du handler du dépassement d'échéance défini par le développeur pour T2

¤ avec aucun handler :

- définition du système :

Tâches	Départ	Coût	Echéance	Période	Priorité
T1	0	2	6	6	14
T2	0	3	4	4	13

- résultat :

- T1 démarre

- T1 consomme 2 temps

- T1 se termine

- T2 démarre

- T2 consomme 2 temps

- lancement du handler du dépassement d'échéance défini par défaut.

* Système souple avec sensibilité imposée dans la JVM et définie pour tous les threads :

¤ avec un handler défini par le développeur

- définition du système :

Sensibilité de la JVM pour le dépassement d'échéance : 2					
Tâches	Départ	Coût	Echéance	Période	Priorité
T1	0	1	5	5	15
T2	0	3	3	9	14
T3	0	2	3	9	13
T4	0	2	5	9	12

Ajout d'un handler pour un dépassement d'échéance pour T3.

Laisse le handler par défaut pour T4.

- résultat :

- T1 démarre

- T1 consomme 1 temps

- T1 se termine

- T2 démarre

- T2 consomme 2 temps

- le système remarque le dépassement d'échéance pour T2 et T3 et attend le délai.

- T2 consomme 1 temps et se termine (le handler n'a pas été lancé)

- T3 démarre

- T3 consomme 1 temps

- lancement du handler du dépassement d'échéance défini par le développeur pour T3 et détection du dépassement d'échéance de T4

- T1 démarre, consomme 1 temps puis se termine

- T4 démarre

- T4 consomme 1 temps

- lancement du handler du dépassement d'échéance défini par défaut.

* Système souple avec sensibilité définie pour chaque thread :

- définition du système :

Tâches	Départ	Coût	Echéance	Période	Priorité	Sensibilité
T1	0	1	5	5	15	0
T2	0	3	3	9	14	2
T3	0	2	3	9	13	2
T4	0	2	5	9	12	3

Ajout d'un handler pour un dépassement d'échéance pour T3

- résultat :

- T1 démarre

- T1 consomme 1 temps

- T1 se termine

- T2 démarre

- T2 consomme 2 temps

- le système remarque le dépassement d'échéance pour T2 et T3 et attend le délai.

- T2 consomme 1 temps et se termine (le handler n'a pas été lancé)

- T3 démarre

- T3 consomme 1 temps

- lancement du handler du dépassement d'échéance défini par le développeur pour T3 et détection du dépassement d'échéance de T4 et attente du délai

- T1 démarre, consomme 1 temps puis se termine

- T4 démarre

- T4 consomme 2 temps et se termine (le handler n'a pas été lancé).

- Dépassement de coût :

* Système strict avec aucune indulgence de retard :

¤ avec un handler défini par le développeur

- définition du système :

Tâches	Départ	Coût	Echéance	Période	Priorité
T1	0	2	8	8	14
T2	0	3	8	8	13

T2 est en fait une tâche qui dure 4 temps

- résultat :

- T1 démarre

- T1 consomme 2 temps

- T1 se termine

- T2 démarre

- T2 consomme 3 temps

- lancement du handler du dépassement de coût défini par le développeur pour T2.

¤ avec aucun handler :

- définition du système :

Tâches	Départ	Coût	Echéance	Période	Priorité
T1	0	2	6	6	14
T2	0	3	4	4	13

- résultat :

- T1 démarre

- T1 consomme 2 temps

- T1 se termine

- T2 démarre

- T2 consomme 3 temps

- lancement du handler du dépassement de coût défini par défaut.

* Système souple avec sensibilité imposée dans la JVM et définie pour tous les threads :

¤ avec un handler défini par le développeur

-définition du système :

Sensibilité de la JVM pour le dépassement de coût : 2					
Tâches	Départ	Coût	Echéance	Période	Priorité
T1	0	1	20	20	15
T2	0	3	20	20	14
T3	0	2	20	20	13
T4	0	2	20	20	12

T2 est en fait une tâche de coût 4.

T3 est en fait une tâche de coût 5.

T4 est en fait une tâche de coût 5.

Ajout d'un handler pour un dépassement de coût pour T3.

Laisse le handler par défaut pour T4.

-résultat :

- T1 démarre
- T1 consomme 1 temps
- T1 se termine
- T2 démarre
- T2 consomme 3 temps
- le système remarque le dépassement de coût pour T2 et attend le délai.
- T2 consomme 1 temps et se termine (le handler n'a pas été lancé)
- T3 démarre
- T3 consomme 2 temps
- détection du dépassement de coût de T3
- T3 consomme 2 temps
- lancement du handler du dépassement de coût défini par le développeur pour T3
- T4 démarre
- T4 consomme 2 temps
- détection du dépassement de coût de T4
- T4 consomme 2 temps
- lancement du handler du dépassement de coût défini par défaut.

* Système souple avec sensibilité définie pour chaque thread :

- définition du système :

Tâches	Départ	Coût	Echéance	Période	Priorité	Sensibilité
T1	0	1	20	20	15	0
T2	0	3	20	20	14	2
T3	0	2	20	20	13	2
T4	0	2	20	20	12	4

Ajout d'un handler pour un dépassement de coût pour T3

T2 est en fait une tâche de coût 4.

T3 est en fait une tâche de coût 5.

T4 est en fait une tâche de coût 5.

- résultat :

  - T1 démarre
  - T1 consomme 1 temps
  - T1 se termine
  - T2 démarre
  - T2 consomme 3 temps
  - le système remarque le dépassement de coût pour T2 et attend le délai
  - T2 consomme 1 temps et se termine (le handler n'a pas été lancé)
  - T3 démarre
  - T3 consomme 2 temps
  - détection du dépassement de coût de T3
  - T3 consomme 2 temps
  - lancement du handler du dépassement de coût défini par le développeur pour T3
  - T4 démarre
  - T4 consomme 2 temps
  - détection du dépassement de cout de T4
  - T4 consomme 3 temps
  - T4 fini (le handler n'a pas été lancé).

Scénarii de tests optionnels

Cette partie décrit les scénarii de tests concernant les améliorations optionnelles à apporter dans le projet LEJOS. Cela concerne l'implémentation des algorithmes PCE et PCP, ainsi que l'algorithme EDF.

1. PCE

PCE (Priority Ceiling Emulation, appelé aussi highest locker priority) est un algorithme d'ordonnancement temps réel à priorité fixe. Il permet de changer la priorité des tâches en fonction des accès à des ressources partagées : il attribue temporairement la priorité la plus élevée à une tâche lorsque celle-ci accède à une ressource partagée, puis celle-ci retrouve sa priorité initiale une fois que la ressource a été libérée.

Une seule instance d'une tâche peut être exécutée à la fois dans ce genre de systèmes. Chaque tâche possède un coût (C), une période (T), une échéance (D) et une priorité (P).

On souhaite savoir comment un système respecte l'algorithme PCE.

Cas nominal

Scénario à deux threads

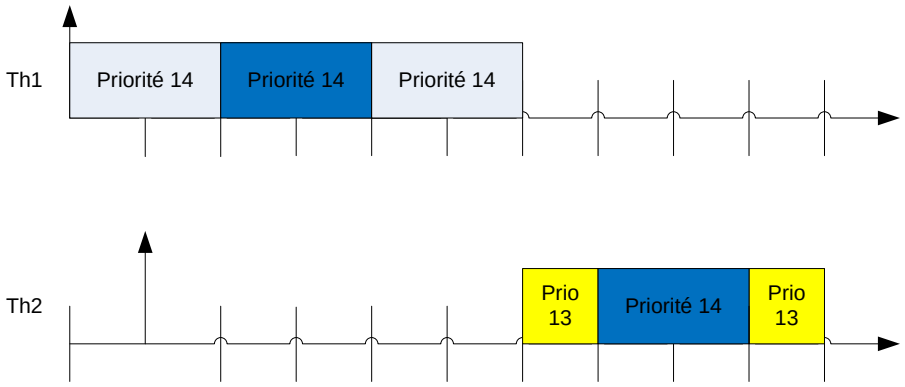
Soit deux threads Th1 et Th2. Th1 possède la priorité la plus haute. Les deux threads veulent accéder à la même ressource à des instants différents.

Th1 accède en premier à la ressource partagée

- Th1 exécute une nouvelle instance de lui-même (si une instance de Th2 est en cours d'exécution, il la préempte) et accède à la ressource partagée. Etant donné que Th1 possède déjà la priorité la plus importante, celle-ci ne change pas.
- Quelque soit l'instant d'activation d'une nouvelle instance de Th2, celui-ci attend la fin de l'exécution de l'instance de Th1 avant de lancer son instance.

Exemple :

	Coût	Période	Échéance	Priorité initiale
Th1	6 (dont 2 pour la ressource)	20	20	14
Th2	4 (dont 2 pour la ressource)	20	20	13
Accès à la ressource				

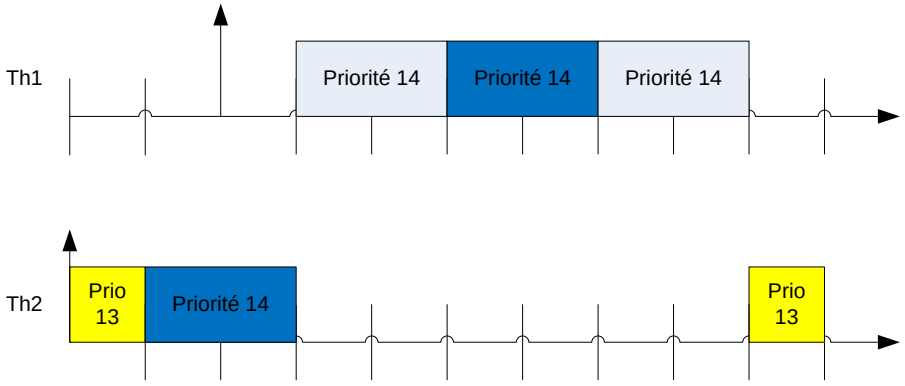


Th2 accède en premier à la ressource partagée

- On suppose qu'aucune instance de Th1 n'est en cours d'exécution. Th2 exécute une nouvelle instance de lui-même et accède à la ressource partagée. A cet instant précis, elle hérite de la priorité la plus haute du système (ici, celle de Th1) et reprend sa priorité initiale une fois qu'elle n'utilise plus la ressource partagée et peut de nouveau être préempté par Th1.
- Si Th1 veut exécuter une nouvelle instance de lui-même, il doit attendre que Th2 ait libéré la ressource partagée avant de pouvoir s'exécuter. Dans le cas où Th2 n'a pas encore accédé à la ressource ou qu'il l'a libéré, Th1 peut préempter Th2 et s'exécuter.

Exemple :

	Coût	Période	Échéance	Priorité initiale
Th1	6 (dont 2 pour la ressource)	20	20	14
Th2	4 (dont 2 pour la ressource)	20	20	13
Accès à la ressource				



Scénario à trois threads – Cas général

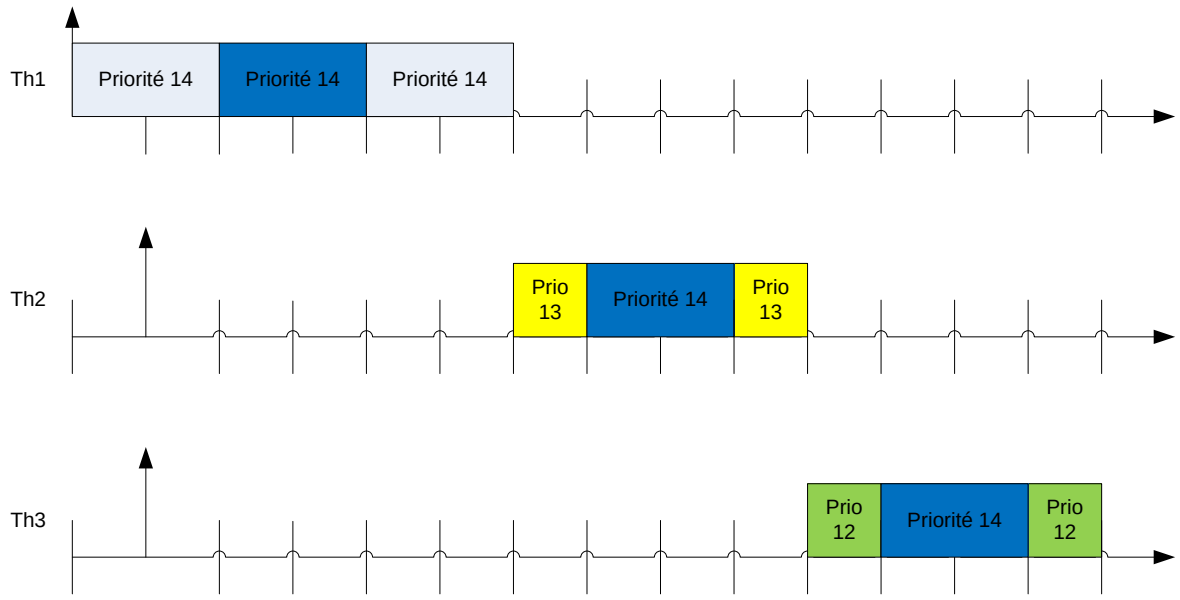
Soit trois threads Th1, Th2 et Th3. Th1 possède la priorité la plus haute, Th2, une priorité intermédiaire et Th3, la priorité la plus basse. Les trois threads veulent accéder à la même ressource à des instants différents.

Th1 accède en premier à la ressource partagée

- Th1 exécute une nouvelle instance de lui-même (si une instance de Th2 ou de Th3 est en cours d'exécution, il la préempte) et accède à la ressource partagée. Etant donné que Th1 possède déjà la priorité la plus importante, celle-ci ne change pas.
- Quelque soit l'instant d'activation d'une nouvelle instance de Th2 ou de Th3, celui-ci attend la fin de l'exécution de l'instance de Th1 avant de lancer son instance.

Exemple :

	Coût	Période	Échéance	Priorité initiale
Th1	6 (dont 2 pour la ressource)	20	20	14
Th2	4 (dont 2 pour la ressource)	20	20	13
Th3	4 (dont 2 pour la ressource)	20	20	12
Accès à la ressource				

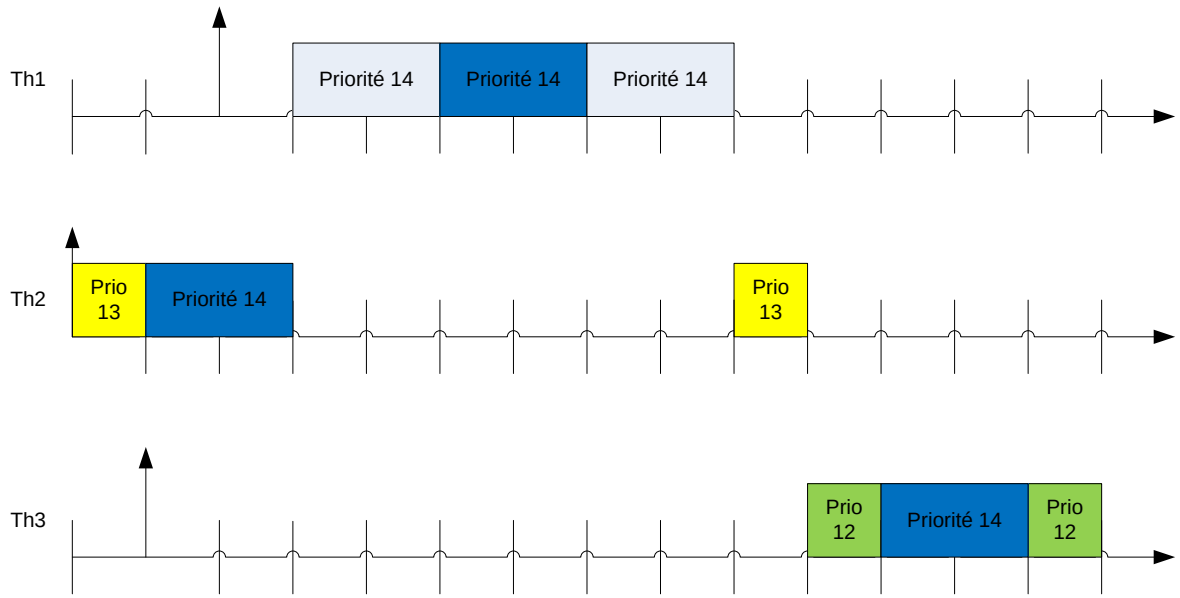


Th2 accède en premier à la ressource partagée

- On suppose qu'aucune instance de Th1 n'est en cours d'exécution. Th2 exécute une nouvelle instance de lui-même (si une instance de Th3 est cours d'exécution, il la préempte) et accède à la ressource partagée. A cet instant précis, elle hérite de la priorité la plus haute du système (ici, celle de Th1) et reprend sa priorité initiale une fois qu'elle n'utilise plus la ressource partagée et peut de nouveau être préempté par Th1.
- Si Th1 veut exécuter une nouvelle instance de lui-même, il doit attendre que Th2 ait libéré la ressource partagée avant de pouvoir s'exécuter. Dans le cas où Th2 n'a pas encore accédé à la ressource ou qu'il l'a libéré, Th1 peut préempter Th2 et s'exécuter.
- Quelque soit l'instant d'activation d'une nouvelle instance de Th3, celui-ci attend la fin de l'exécution de l'instance de Th2 avant de lancer son instance.

Exemple :

	Coût	Période	Échéance	Priorité initiale
Th1	6 (dont 2 pour la ressource)	20	20	14
Th2	4 (dont 2 pour la ressource)	20	20	13
Th3	4 (dont 2 pour la ressource)	20	20	12
Accès à la ressource				

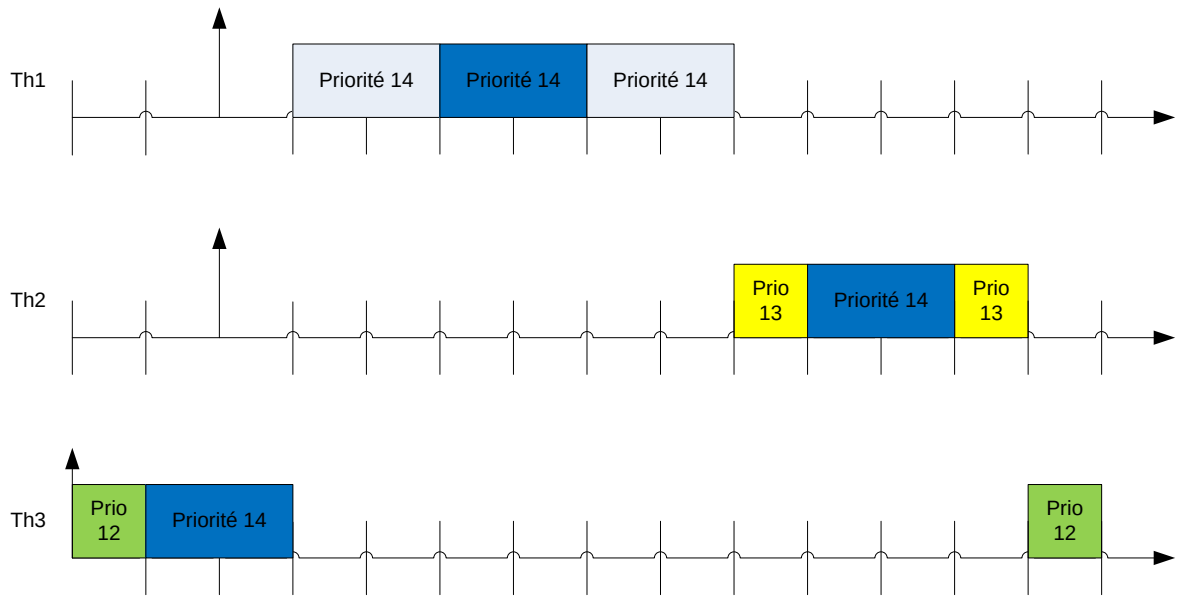


Th3 accède en premier à la ressource partagée

- On suppose qu'aucune instance de Th1 ou de Th2 n'est en cours d'exécution. Th3 exécute une nouvelle instance de lui-même et accède à la ressource partagée. A cet instant précis, elle hérite de la priorité la plus haute du système (ici, celle de Th1) et reprend sa priorité initiale une fois qu'elle n'utilise plus la ressource partagée et peut de nouveau être préempté par Th1 ou Th2.
- Si Th1 ou Th2 veulent exécuter une nouvelle instance d'eux-mêmes, ils doivent attendre que Th3 ait libéré la ressource partagée avant de pouvoir s'exécuter. Dans le cas où Th2 n'a pas encore accédé à la ressource ou qu'il l'a libéré, Th1 peut préempter Th2 et s'exécuter.

Exemple :

	Coût	Période	Échéance	Priorité initiale
Th1	6 (dont 2 pour la ressource)	20	20	14
Th2	4 (dont 2 pour la ressource)	20	20	13
Th3	4 (dont 2 pour la ressource)	20	20	12
Accès à la ressource				



De manière générale, lorsqu'une tâche accède à une ressource partagée, elle hérite de la priorité la plus haute du système et reprend sa priorité initiale une fois qu'elle a libéré la ressource.

2. PCP

PCP (Priority Ceiling Protocol) est un protocole de synchronisation de ressources partagées utilisé en développement temps réel et qui permet d'éviter les inversions non bornées et les inter-blocages dus à une mauvaise gestion des sections critiques. Le principe de ce protocole est d'attribuer une priorité à chaque section critique du système. Cette priorité est calculée en fonction des tâches qui utilise la section critique : elle obtient la priorité de la tâche à plus haute priorité qui l'utilise. Cependant, la priorité des sections critiques n'est à prendre en compte que lorsque plusieurs tâches différentes accèdent à des sections critiques différentes ou non.

Cas nominal

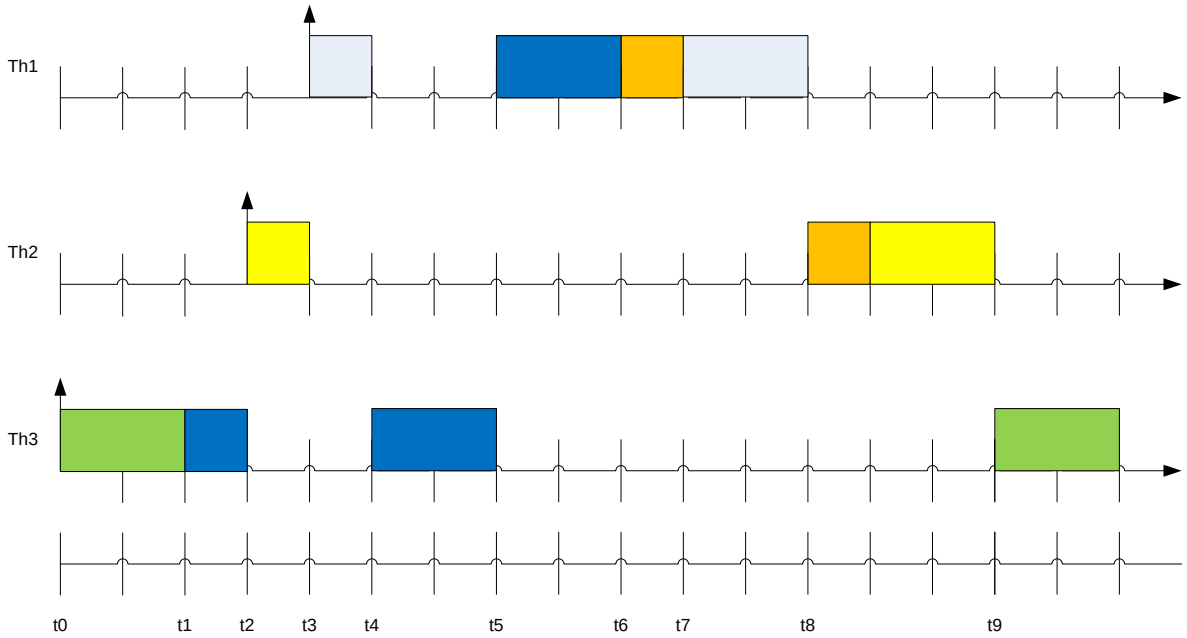
Scénario à trois threads – cas général

Soit trois threads Th1, Th2 et Th3. Th1 possède la priorité la plus haute, Th2, une priorité intermédiaire et Th3, la priorité la plus basse. Le système possède aussi des sections critiques. Nous devons vérifier que celles-ci héritent de la priorité de la tâche la plus importante les utilisant et que ces priorités affectées n'interviennent que lorsque plusieurs tâches différentes accèdent à des sections critiques différentes ou non.

Exemple 1 : deux ressources partagées

	Coût	Période	Échéance	Priorité
Th1	6 (dont 2 pour la ressource 1 et 1 pour la ressource 2)	20	20	14
Th2	4 (dont 1 pour la ressource 2)	20	20	13
Th3	7 (dont 3 pour la ressource 1)	20	20	12
Accès à la ressource 1				14
Accès à la ressource 2				14

- La ressource 1 est utilisée par Th1 et Th3 : elle a une priorité de 14.
- La ressource 2 est utilisée par Th1 et Th2 : elle a une priorité de 14.



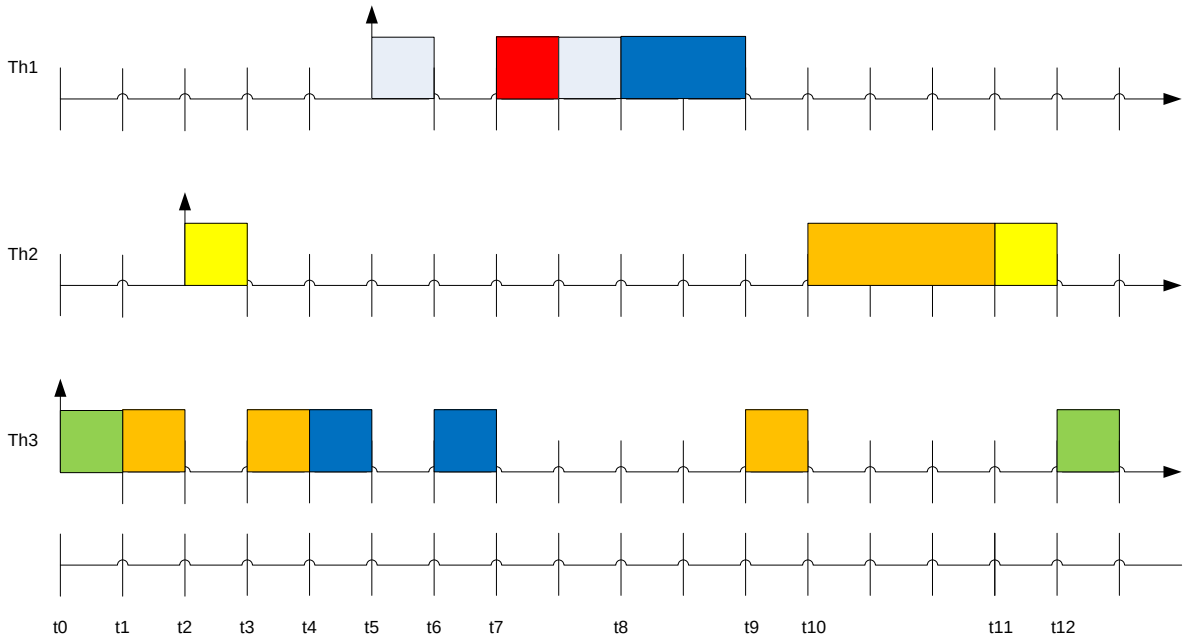
- A t0, Th3 lance une nouvelle instance de lui-même.
- A t1, Th3 veut accéder à la ressource 1 : aucun autre thread ne veut s'exécuter et ne veut utiliser la ressource 1 alors Th3 y accède.
- A t2, Th2 exécute une nouvelle instance de lui-même et préempte Th3.
- A t3, Th2 veut accéder à la ressource 2 mais est préempté par Th1 qui lance une nouvelle instance de lui-même.
- A t4, Th1 veut accéder à la ressource 1 mais est déjà utilisée par Th3, qui reprend la main et continue ses opérations sur la ressource 1.
- A t5, Th3 libère la ressource 1. Th2 veut accéder à la ressource 2 et Th1 veut accéder à la ressource 1 : c'est Th1 qui reprend la main car sachant que les deux ressources ont la même priorité, ce n'est pas le cas des deux threads : Th1 est plus prioritaire que Th2.
- A t6, Th1 libère la ressource 1 et accède à la ressource 2 qui n'est utilisée par personne.
- A t7, Th1 libère la ressource 2 et continue son exécution.
- A t8, Th1 termine son exécution et Th2 reprend la main en accédant à la ressource 2 qui est libre.
- A t9, Th2 termine son exécution et Th3 reprend la main et continue son exécution

Exemple 2 : trois ressources partagées + sections critiques imbriquées

	Coût	Période	Échéance	Priorité
Th1	5 (dont 2 pour la ressource 1 et 1 pour la ressource 3)	20	20	14

Th2	5 (dont 3 pour la ressource 2)	20	20	13
Th3	7 (dont 3 pour la ressource 2 et 1 pour la ressource 1)	20	20	12
Accès à la ressource 1				14
Accès à la ressource 2				13
Accès à la ressource 3				14

- La ressource 1 est utilisée par Th1 et Th3 : elle a une priorité de 14.
- La ressource 2 est utilisée par Th2 et Th3 : elle a une priorité de 13.
- La ressource 3 est utilisée par Th1 : elle a une priorité de 14.



- A t0, Th3 lance une nouvelle instance.
- A t1, Th3 accède à la ressource 2 : pas de problèmes car aucun thread prioritaire n'est activé.
- A t2, Th2 préempte Th3 en s'exécutant.
- A t3, Th2 veut accéder à la ressource 2 mais ne peut pas car elle est déjà utilisée par Th3, donc ce dernier se réactive.
- A t4, Th3 veut accéder à la ressource 1 qui est libre sans avoir libéré la ressource 2 (sections critiques imbriquées).

- A t5, Th1 préempte Th3 en s'exécutant.
- A t6, Th1 veut accéder à la ressource 3 mais n'y accède pas car même si elle est libre, Th3 est déjà dans une section critique de priorité égale à la section critique de la ressource 3 : Th3 reprend la main pour utiliser la ressource 1.
- A t7, Th3 libère la ressource 1 et est préempté par Th1 qui accède à la ressource 3.
- A t8, Th1 veut accéder à la ressource 1 : elle n'est utilisée par personne donc Th1 peut y accéder.
- A t9, Th3 reprend la main et continue son exécution dans la section critique de la ressource 2.
- A t10, Th3 libère la ressource 2 et Th2 peut enfin accéder à cette dernière.
- A t11, Th2 libère la ressource 2 et continue son exécution.
- A t12, Th3 reprend son exécution.

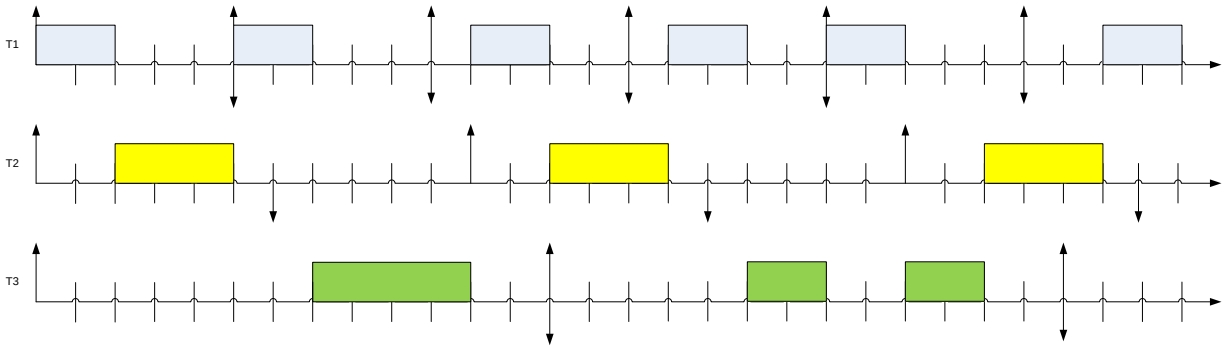
3. EDF

EDF (Earliest Deadline First) est un algorithme d'ordonnancement préemptif à priorités dynamiques. L'échéance de la tâche permet de définir sa priorité par rapport aux autres tâches. Plus l'échéance est proche, plus la priorité de la tâche est grande. L'affectation des priorités est faite à chaque fois qu'une échéance se termine ou lorsqu'une nouvelle période démarre. Cet algorithme est optimal pour l'ensemble des types de systèmes de tâches mais est difficile à mettre en œuvre et ne prend pas en compte les surcharges d'utilisation du CPU.

Pour en comprendre le fonctionnement, considérons les trois tâches ci-dessous :

Nom de la tâche	Coût	Période	Echéance
T1	2	5	5
T2	3	11	6
T3	4	13	13

Maintenant, voyons étape par étape comment sont réparties les priorités.



- t = 0 :

La tâche T1 a l'échéance la plus proche, elle prend donc la priorité la plus forte. La tâche T2 prend la priorité moyenne et la tâche T3 la priorité la plus faible.

- t = 2 :

La tâche T1 vient de se terminer, on recalcule donc les priorités. L'échéance de T2 intervient dans 4 unités de temps et celle de T3 dans 11 unités de temps. T2 prend donc la plus forte priorité.

- t = 5 :

T2 vient de se terminer et T1 est réactivé. L'échéance de T1 est dans 5 UT tandis que celle de T3 est dans 8 UT. T1 prend donc la plus forte priorité.

- t = 7 :

T1 se termine. T3 prend la plus forte priorité.

- t = 10 :

T3 n'est pas finie, mais T1 est réactivé. L'échéance de T1 est dans 5 UT et celle de T3 dans 3 UT. T3 garde donc la plus forte priorité.

- t = 11 :

T3 est finie. T2 est réactivée. L'échéance de T1 est dans 4 UT, celle de T2 dans 6 UT. T1 prend la plus forte priorité.

- t = 13 :

T1 est finie. T3 se réactive. T2 a l'échéance la plus proche et prend donc la priorité la plus élevée.

- t = 15 :

T2 n'est pas finie et T1 se réactive. L'échéance de T1 est dans 5 UT, celle de T2 dans 2 UT et celle de T3 dans 11 UT. La tâche T2 garde donc la priorité la plus élevée.

- t = 16 :

T2 se termine. L'échéance de T1 est dans 4 UT et celle de T3 dans 10 UT. T1 prend la plus forte priorité.

- t = 18 :

T1 se termine. T3 a l'échéance la plus proche.

- t = 20 :

T1 est réactivée alors que T3 n'est pas finie. L'échéance de T3 intervient dans 6 UT tandis que celle de T1 n'est que dans 5 UT. T1 prend la plus forte priorité.

- t = 22 :

T1 se termine, T2 est réactivée. L'échéance de cette dernière est dans 6 UT, celle de T3 dans 4 UT, T3 prend la plus forte priorité et peut se poursuivre.

- t = 24 :

T3 s'achève. T2 prend la plus forte priorité.

- t = 25 :

T1 se réactive alors que T2 n'est pas finie. L'échéance de T1 est dans 5 UT, celle de T2 dans 3 UT. T2 garde sa priorité élevée.

- t = 26 :

T3 est réactivée, T2 n'a pas terminé son traitement. L'échéance de T1 intervient dans 4 UT, celle de T2 dans 2 UT et celle de T3 dans 13 UT. T2 garde sa plus forte priorité et se poursuit.

- t = 27 :

T2 se termine. L'échéance de T1 est dans 3 UT et celle de T3 dans 12 UT. La plus forte priorité est attribuée à T1.

- t = 29 :

T1 se termine. T3 prend la plus forte priorité.

Glossaire

AsyncEventHandler : Objet JAVA RT Schedulable qui permet de gérer des actions ponctuelles.

Cost enforcement : Lorsque l'on utilise le CPU plus que ce qui est attendu, il faut modifier le "budget" temps alloué pour la tâche concernée. On parle donc de cost enforcement.

Cost overrun detection : détection du dépassement de coût : permet de savoir si l'utilisation du CPU dépasse le temps d'utilisation imparti et ainsi de savoir si le système est faisable.

Deadline Monotonic : Un algorithme d'ordonnancement temps réel à priorité fixe. Il attribue la priorité la plus forte à la tâche qui possède la plus petite échéance.

Déterministe : Qui s'assure que l'on respecte les contraintes.

Échéance : Temps avant lequel l'instance d'un thread doit être traitée.

EDF : *Earliest Deadline First scheduling* est un algorithme d'ordonnancement préemptif à priorité dynamique utilisé dans les systèmes temps réel. Il attribue une priorité à chaque requête en fonction de l'échéance de cette dernière.

Exclusion mutuelle : Principe d'utilisation d'une ressource partagée par un processus à la fois.

Faisabilité : Existence d'un ordonnancement de l'ensemble des tâches qui respecte les contraintes temporelles associées à ces tâches.

Garbage Collector : Sous-système informatique de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.

Héritage de priorité : La priorité d'une tâche devient celle d'une autre tâche de priorité supérieure si cette dernière requiert des ressources de la première tâche.

Immortal memory : Type de mémoire utilisé par les NoHeapRealTimeThread. Elle est libérée à la fin de l'application.

Inversion de priorités : Cas ou une tâche de plus haute priorité est exécutée après une tâche de priorité moindre pour des raisons d'accès aux ressources.

Inversion non bornée : Inversion de priorité pour laquelle on ne peut pas évaluer le pire temps de blocage et on ne peut donc plus garantir l'échéance.

JVM : La « Java Virtual Machine » (abrégé JVM, en français machine virtuelle Java) est une machine virtuelle permettant d'interpréter et d'exécuter le bytecode JAVA. Certaines machines virtuelles sont spécialisées pour respecter certaines contraintes, telles que les contraintes temps-réel des applications.

Moniteur : Les moniteurs permettent de gérer les éléments partagés. Ce sont des jetons, qu'il faut impérativement avoir avant de pouvoir prendre une ressource commune, et le relâcher ensuite.

NoHeapRealtimeThread : RealTimeThread dont les objets ne sont plus alloués sur le tas.

Ordonnancement : Manière dont les threads sont ordonnés pour leur exécution.

Ordonnanceur : Gestionnaire des priorités des threads.

Ordonnanceur optimal : Qui permet de trouver un ordonnancement faisable s'il existe.

PCP : *Priority Ceiling Protocol* : protocole de synchronisation de ressources partagées qui permet d'éviter les inversions non bornées et les inter-blocages dus à une mauvaise gestion des sections critiques.

PIP : *Priority Inheritance Protocol* : protocole de synchronisation qui permet de gérer les inter-blocages entre les threads de priorités différentes.

Préemption de thread : Le thread est mis en attente lorsqu'un thread de priorité supérieure demande à être activée.

Priorité : Priorité que l'on applique à un thread pour son ordonnancement.

Priorités dynamiques : Priorités qui ne sont plus établies une bonne fois pour toutes mais qui peuvent évoluer selon les nouvelles informations fournies.

Priorités fixes : Les priorités sont établies au préalable et ne changent plus.

Rate Monotonic : Un algorithme d'ordonnancement temps réel en ligne à priorité fixe. Il attribue la priorité la plus forte à la tâche qui possède la plus petite période.

RealTimeThread : Objet Schedulable qui permet de gérer les priorités afin de respecter les contraintes temps réel.

Scoped memory : Type de mémoire utilisé par les NoHeapRealTimeThread. Elle est libérée quand les threads temps réel qui l'occupent se terminent.

Sensibilité : marge d'erreur que l'on peut accepter pour que le processus se déroule normalement. La sensibilité peut concerner le temps d'exécution ou le nombre d'échéances réussies.

Tâches apériodiques : Tâches activées à des instants qui ne sont pas périodiques.

Tâches sporadiques : Tâches à échéances fortes qui sont activées à des instants irréguliers avec un temps minimal entre deux activations d'instances successives.

Temps Réel Dur : On parle de temps réel dur quand les événements traités trop tardivement ou perdus provoquent des conséquences catastrophiques pour la bonne marche du système (perte d'informations cruciales, plantage...).

Temps Réel Souple : On parle de temps réel souple quand les événements traités trop tardivement ou perdus sont sans conséquence catastrophique pour la bonne marche du système.

Thread : Processus léger qui partage la mémoire virtuelle de la machine. Les threads peuvent être exécutés en parallèle et chaque thread est indépendant mais les ressources utilisées dans un thread peuvent être partagées.

Transitivité : Mécanisme qui permet que l'ensemble des tâches qui utilisent une même ressource utilisent la plus haute priorité pour ne pas être bloquée par une tâche dont la priorité se situe entre les priorités de la tâche activée et celle bloquée pour la ressource.