TP

IN2

Claire Pagetti
Mars-Avril 2016

1 TP 1 : Lustre

Objectifs

L'objectif de cette séance de travaux pratiques est d'utiliser le langage Lustre pour programmer la spécification du robot Lego et d'utiliser la boîte à outils pour simuler et valider ces programmes.

1.1 Environnement de travail

La boîte à outil Lustre est installée sur les stations Windows et également sur les stations Linux.

1.1.1 L'environnement Cygwin

Lancer l'environnement Cygwin Durant la session, nous allons utiliser l'environnement Cygwin. Cygwin crée un environnement UNIX sous Windows. Pour lancer Cygwin, allez dans le menu *Demarrer*, choisissez *N7* et *Lejos*. Un terminal de travail apparaît : il est alors possible d'exécuter les commandes UNIX standard (ls, cat ...).

Plusieurs éditeurs graphiques sont disponibles, vous pouvez par exemple utiliser l'éditeur Crimson en suivant le chemin *Demarrer*, puis *Bureautique* et l'éditeur Crimson.

Espace de travail Vous devez travailler dans un espace local partagé, plus précisément dans `/tmp`. Vous devez d'abord créer un dossier portant votre nom pour stocker votre travail :

```
> mkdir /tmp/votre_nom
```

Attention ! Le répertoire `/tmp` est un espace partagé, local à l'ordinateur auquel vous êtes connecté. Afin d'éviter de mauvaise surprise, n'oubliez pas à la fin de la séance de copier le continu dans votre compte personnel et d'effacer le dossier d'aujourd'hui :

```
> cp -r /tmp/your_name /cygdrive/g
```

La boîte à outils LUSTRE se trouve dans le répertoire `/usr/local/lustre`. Pour pouvoir l'utiliser, vous devez ajouter le chemin :

```
> source /usr/local/lustre/lv400.sh
```

1.1.2 Salles Linux

Si vous souhaitez programmer du LUSTRE dans les salles Linux. La boîte à outil LUSTRE se trouve dans le répertoire `/usr/local/tools/lustre`. Dans le terminal, modifiez le chemin :
`setenv PATH "${PATH}:/usr/local/tools/lustre/bin"`
`setenv LUSTRE_INSTALL "/usr/local/tools/lustre".`
Vous pouvez l'ajouter de manière permanent dans le .cshrc. Il faut ajouter les lignes :
`setenv PATH "${PATH}:/usr/local/tools/lustre/bin"`
`setenv LUSTRE_INSTALL "/usr/local/tools/lustre"`
Après sauvegarde, n'oubliez pas de mettre à jour le terminal `source .cshrc`

1.2 La boîte à outils LUSTRE

Premier programme LUSTRE Ecrivez le code du front montant d'horloge `rising_edge` dans l'éditeur de texte, sauvez-le sous un nom de type `mon_tp.lus`.

Compilation d'un programme La commande pour compiler un programme LUSTRE est :

lustre $< nom.lus > < node > [< options >]$

Vérifiez la syntaxe de votre nœud `rising_edge` avec le compilateur.

Simulation Il est possible de simuler le comportement d'un programme avec l'outil LUCIOLE :

luciole $< nom.lus > < node >$

Dans le menu sur les horloges, il y a deux manières de faire évoluer le système :
— **auto-step** : une entrée unique est modifiée à chaque top d'horloge ;
— **compose** : plusieurs entrées peuvent être modifiées à chaque pas d'horloge (il faut alors clicker sur step pour faire avancer le temps).
Simulez le nœud `rising_edge` avec LUCIOLE.

Vérification fonctionnelle Pour vérifier une propriété, on introduit un nœud particulier dont la sortie est un booléén. Ce booléen correspond au prédicat *est-ce que la propriété est satisfaite ?* L'outil de vérification s'appelle ainsi :

lesar $< nom.lus > < node > [< options >]$

Les options principales sont :
— **-diag** : affiche le diagnostique de vérification ;
— **-enum** : utilise une méthode énumérative.

Verification d'une propriété Ajouter dans votre fichier `mon_tp.lus`, le code du nœud `falling_edge` qui détecte les fronts descendants d'un flot booléen. Pour un flot d'entrée booléen X, vérifiez les propriétés suivantes :
— $P1 = (rising\_edge(X) = falling\_edge(not(X)))$.

— $P2 = (rising\_edge(not(X)) = falling\_edge(X))$.

Pour cela, inroduisez un nouveau nœud qui appelle `rising_edge` et `falling_edge`, et qui renvoie une unique sortie $Prop$ de type booléen qui vaut vrai ssi $P1$ et $P2$ sont vraies.

1.3 Travail à faire

Exercice 1 *Programmez et simulez un compteur :*

— `node counter (const N : int ; RAZ : bool) returns (Top : bool)`

— *Top=true si N tops se sont écoulés, le signal Raz remet le compteur à 0.*

Exercice 2 *Programmez et simulez le nœud :*

— `node keep (x : bool ; RAZ : bool) returns (y : bool)`

— *y=true quand x=true jusqu'à la réception de RAZ=true sinon y=false.*

t=	*0*	*1*	*2*	*3*	*4*
x	*false*	**true**	*false*	*false*	**true**
RAZ	*false*	*false*	*false*	**true**	**true**
y	*false*	**true**	**true**	*false*	**true**

Exercice 3 [1] *On souhaite modéliser un métronome battant la mesure pour un rythme donné. Il dispose d'une entrée rythme. Le métronome pulsera différemment selon que le rythme est unaire (rythme= 1), binaire, ternaire ou quaternaire.*

— *Pulsation unaire : tic, tic, tic, ...*

— *Pulsation binaire : tic, tac, tic, tac, ...*

— *Pulsation ternaire : tic, tac, toc, tic, tac, toc, ...*

— *Pulsation quaternaire : tic, tac, toc, tut, tic, tac, ...*

Le métronome produit donc un signal de sortie de type énuméré $\in \{tic, tac, toc, tut\}$. *Pour chaque rythme, deux occurrences de signaux consécutifs sont séparées par 1 top de l'horloge de base. L'interface du nœud métronome est la suivante :*

```
node metronome_variable(rythme : int)
returns (son : type_son);
```

Ecrivez deux versions du nœud (une avec une variable intermédiaire entière et une sans). On souhaiterait vérifier plusieurs propriétés sur le métronome :

— *2 sorties ne peuvent être vrai en même temps,*

— *si* `rythme`$\notin [1, 4]$, *alors aucune sortie n'est vraie.*

Pour cela, écrire un nœud `verif_i` *pour chacune des propriétés, simulez le nœud et vérifier la.*

On complexifie ensuite le métronome : les pulsations sont données par l'entrée `freq`. *Ecrivez deux versions du nœud : une avec des if et une avec des condact.*

1. Libre adaptation de http ://www-soc.lip6.fr/ ema/enseignement/2011_LS/cours/exam_10-11_rep.pdf

2 TP 2 : Le robot Lego Mindstorm NXT

2.1 Aperçu général

Le robot Lego Mindstorm NXT se compose d'une « brique » Lego NXT qui contrôle 4 capteurs et 3 actionneurs. La brique NXT est composée d'un processeur central de type ARM7 contrôlant les applications du robot et les entrées/sorties numériques. C'est un processeur 32 bits fonctionnant à 48 MHz. Il possède aussi une mémoire Flash de 256 kO et une mémoire RAM de 64 kO. Le processeur central est associé à un coprocesseur d'entrée/sortie qui gère les capteurs et les actionneurs analogiques.

2.2 Programmation du robot avec LUSTRE

Le nœud du robot doit être spécifié de la manière suivante :

```
node robot_your_name (
                capteur_1: bool; -- sensor of touch on port 1
                capteur_2: bool; -- sensor of touch on port 2
                capteur_3: bool; -- sensor of touch on port 3
                capteur_4: bool; -- sensor of touch on port 4
 )
returns (
Avant_A, Arriere_A :bool ;     -- go forward or backward engine A
Vitesse_A : int; -- percentage of speed for engine A
Avant_B, Arriere_B :bool ;     -- go forward or backward engine A
Vitesse_B : int; -- percentage of speed for engine A
Avant_C, Arriere_C :bool ;     -- go forward or backward engine A
Vitesse_C : int; -- percentage of speed for engine A
);
```

Remarque 1 *L'utilisation d'un unique programme* LUSTRE *pour le robot créera une unique tâche temps réel pour contrôler le robot. Cela pourrait être étendu puisque OSEK gère des systèmes multitâches à priorité fixe.*

2.3 Compilation en C pour le robot

La commande `poc` génère un code C ANSI classique. Ce code doit s'interfacer avec le robot et doit utiliser la librairie de gestion des signaux d'acquisition et de contrôle. Cette étape est réalisée par le programme `nxtlus.pl`.

```
> ./nxtlus.pl file_name robot_your_name
```

Cette opération crée un fichier `robot_your_name.c` qui doit ensuite être compilé pour la cible processeur ARM. Pour cela, un fichier `Makefile` est fourni que vous devez modifier en mettant le nom du .c.

```
> make all
```

Le fichier généré `robot_your_name_samba_ram.bin` doit ensuite être téléchargé sur la RAM du robot. Appliquez la commande (pendant que le robot est connecté par port USB) :

```
> sh ./rxeflash.sh
```

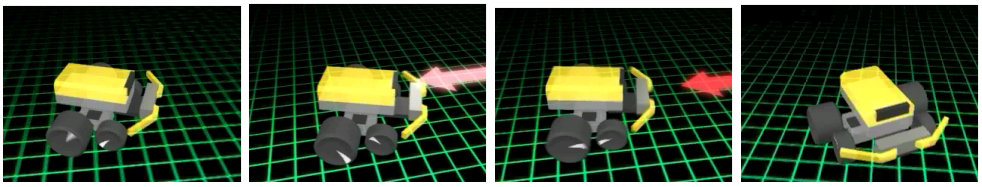
Vous pouvez mettre en route le robot et vous assurer que votre code est correct.

2.4 Travail à faire

Il s'agit de contrôler un *Roverbot* possédant :
— 2 roues reliées aux moteurs pour avancer, reculer et tourner
— 2 parechocs reliés à des capteurs de toucher

Vous devez contrôler le *Roverbot* de façon à répondre aux spécifications suivantes. Comme le montre la figure 1, le comportement du *Roverbot* est le suivant :

1. le *Roverbot* avance
2. un des parechocs touche un obstacle
3. le *Roverbot* recule pendant 50 unités de temps, puis
4. tourne pendant 30 unités de temps



(a) Go forward (b) Obstacle (c) Go back (d) Turn

Figure 1 – Behaviour of the *Roverbot*

Exercice 4 *1. Définissez le nœud* Lustre *associé au robot qui respecte le comportement attendu ;*
2. Simulez votre programme et corrigez les bugs ;
3. Vérifiez les propriétés suivantes : quand le robot touche un obstacle alors il recule, si le robot s'arrête de reculer, alors il tourne . . .
4. Compilez et téléchargez le code sur le robot Lego.

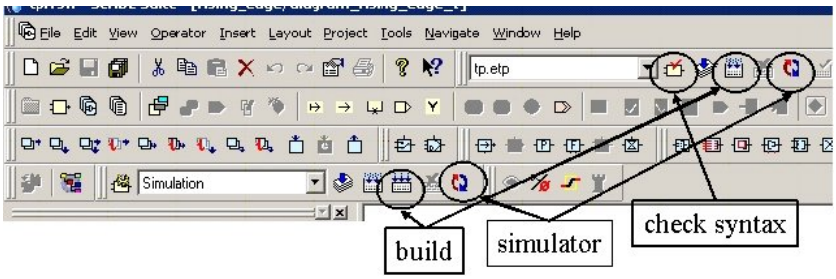
3 TP 3 : Scade

L'objectif de ce TP est de se familiariser avec l'outil Scade Suite. Nous allons écrire plusieurs programmes, les simuler et vérifier des propriétés.

Configuration de votre environnement Dans le menu *Demarrer*, puis *tous les programmes*, puis *Menu N7*, puis *Scade*, lancer l'exécutable *VCS*. L'interface graphique de Scade se lancera. Créez un nouveau projet : *File*, *New*, donnez le nom *tp_scade* dans le champ *Project name* puis choisissez l'emplacement dans *Location*. Cliquez ensuite sur *ok*, puis cliquez sur *terminer*.

Pour faire apparaître les icônes de simulation, cliquez dans *View* → *Toolbars* → *Code generator*.

Premier programme SCADE Dans la hiérarchie qui apparaît à gauche, sélectionnez *tp_scade* puis cliquez à droite, sélectionnez *insert*, puis *operator*. Regardez ensuite dans les images pour retrouver le raccourci *new operator*. Dans la suite, vous pourrez utiliser l'une ou l'autre des solutions. Appelez ce premier opérateur `rising_edge`. Ecrivez la version graphique de ce nœud : ajoutez dans l'interface les entrée/sortie, double cliquez sur le diagramme pour l'éditer.
Simulez le nœud `rising_edge` avec le simulateur SCADE Suite.



Exercice 5 *Ecrivez le nœud* `falling_edge` *qui détecte les fronts descendants d'un flot booléen. Ecrivez ensuite un nœud* `verif` *afin de vérifier les propriétés suivantes :*

— $P1 = (rising\_edge(X) = falling\_edge(not(X)))$.
— $P2 = (rising\_edge(not(X)) = falling\_edge(X))$.

Sélectionnez la variable de sortie P1*, cliquez droit sur* insert → proof objectif. *Donnez un nom à la preuve, par exemple* verif1. *Cliquez droit sur* analyze. *Si la réponse est* falsifiable, *cela veut dire que la propriété n'est pas vérifiée. Cliquez sur le* scenario *qui vous montrera le contre-exemple. Vérifiez ensuite* p2 *et* p3.

Exercice 6 *Re-écrire le nœud* counter3 *du cours p121 en utilisant les opérateurs* fby *et* condact.

Exercice 7 *Déclarez une constante* `N` *de valeur 10. Programmez un compteur qui prend en entrée un booléen* `RAZ` *et renvoie un booléen* `top`. *Le signal RAZ remet le compteur à 0 et* `top`*=true si N tops se sont écoulés depuis le début de l'exécution ou depuis le dernier* `RAZ`.

Simulez le nœud pour le scénario suivant : `RAZ`$= f^{12}tf^{12}$.

Exercice 8 *Créez un nouvel opérateur* metronome. *Ajoutez les interfaces du métronome codé dans l'exercice 3 (avec les entiers). Sélectionnez le diagramme et cliquez droit sur* Convert to textual. *Recopiez ensuite le code* LUSTRE *de votre métronome. Créez un nouvel opérateur* verif *(en version graphique) qui devra vérifier les deux propriétés de l'exercice 3 :*

— *p1 : si rythme est dans* $[1, 4]$ *alors au moins une des sorties est vraie, sinon aucune sortie n'est vraie*
— *p2 : deux sorties ne peuvent être vraies simultanément*
— *p3 : vérifiez que les deux versions de vos métronomes sont équivalents*

Sélectionnez la variable de sortie p1*, cliquez droit sur* insert → proof objectif. *Donnez un nom à la preuv, par exemple* verif1.*Cliquez droit sur* analyze. *Si la réponse est* falsifiable, *cela veut dire que la propriété n'est pas vérifiée. Cliquez sur le* scenario *qui vous montrera le contre-exemple. Vérifiez ensuite* p2 *et* p3.

Exercice 9 *Programmez un opérateur* `VitesseTemp` *ayant la spécification suivante :*

— *entrée :* `x` *de type réel indiquant la température observée,*

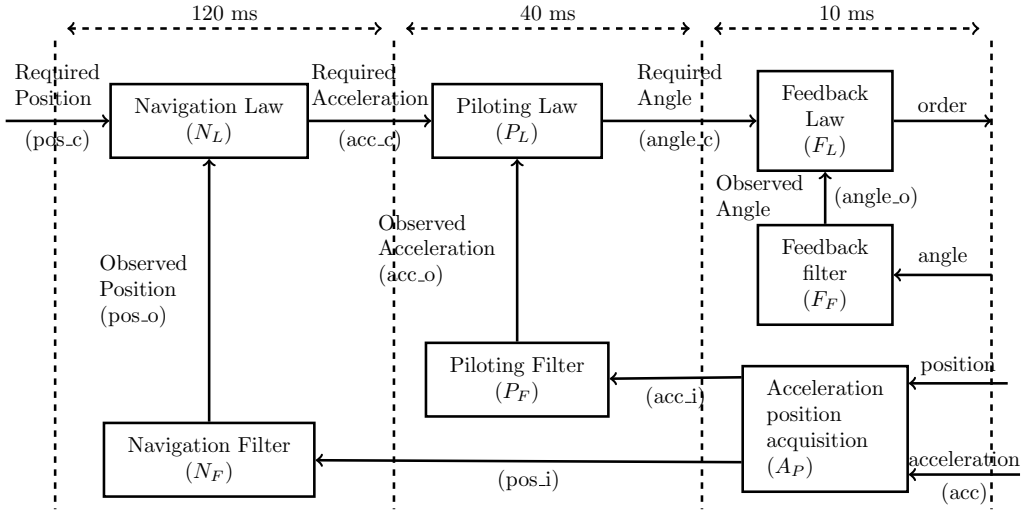
— *sortie :* `alarme` *de type booléen*
— *fonction :* `alarme`$_k$ *est vrai ssi la différence* `x`$_k$ - `x`$_{k-1}$ *n'est pas dans l'intervalle* $[-2, 2]$.
On souhaite vérifier que lorsqu'on applique le nœud `VitesseTemp` *au flot* `x` *défini par :*
— `x`$_0$ = *0*
— `x`$_{2n}$ = `x`$_{2n-1}$ + *1.5*
— `x`$_{2n+1}$ = `x`$_{2n}$
l'alarme ne se déclenche jamais. Ecrivez le nœud `verif` *correspondant et utilisez un* New Boolean Activate *pour spécifier* `x`.

Exercice 10 *Modélisez un automate à deux états dont les entrées sont* `on` *et* `off` *(booléens) et la sortie* `etat` *(booléen) qui passe de faux à vrai sur la commande* `on`*, de vrai à faux sur la commande* `off`.

4 TP4 : ordonnancement avec STORM

4.1 Etude de cas

On considère un exemple simplifié d'un système de commande de vol représenté dans la figure ci-dessous. Ce système contrôle l'attitude, la trajectoire et la vitesse d'un aéronef. L'exemple est constitué de 7 tâches qui se répètent à des rythmes périodiques. Le sous-système le plus rapide s'exécute à 10 ms et est constitué de 2 tâches. La première acquière les données relatives à l'état de l'aéronef (angles, position, accélération) et la deuxième calcule les lois d'asservissement des gouvernes ce qui génère un ordre envoyé vers ces actuateurs. Le sous-système intermédiaire est la boucle de pilotage qui s'exécute à 40ms et qui détermine l'accélération à appliquer. Le sous-système le plus lent est la boucle de navigation qui s'exécute à 120ms. L'objectif est de déterminer la position à atteindre. La position souhaitée est reçue au rythme le plus lent.



Tâche	période	WCET	date de réveil	deadline	priorité
NL	120	40	10	120	1
NF	120	20	0	120	2
PL	40	10	0	40	3
PF	40	15	0	40	4
FL	10	2	0	10	5
FF	10	1	0	10	6
AP	10	1	0	10	7

Exercice 11 *Est-ce que le système des commandes de vol est ordonnançable sur un monoprocesseur pour RM, EDF ? Jusitfiez vos réponses.*

Si les réponses précédentes sont négatives, à partir de combien de réponse peut-on espérer trouver un oraonnancement faisable ?

4.2 STORM

STORM est un outil de simulation d'ordonnancement temps réel multiprocesseurs. A partir d'une spécification des tâches temps réel et du choix de la politique d'ordonnancement (ces données sont précisées dans un fichier .xml), l'outil simule l'exécution de ces tâches. Tous les documents et les sources sont disponibles sur les pages web de STORM : `http://storm.rts-software.org`

Archive Téléchargez l'archive complète sur `www-tr/supports/ermont` dans la rubrique Enseignements 2IN → Systèmes Temps Réel. Elle contient :

1. le manuel d'utilisateur `User-guide-V3.2.pdf`
2. le manuel du concepteur `Designer-guideV3.3_version1.pdf`
3. l'exécutable `storm-3-2.jar`
4. un exemple de tâche `tache_EDF.xml`

4.2.1 Simulation d'ordonnancement multiprocesseur

Dans un premier temps, nous allons manipuler l'outil de simulation en utilisant des politiques déjà intégrées dans STORM. Lancer l'outil :

1. soit en double cliquant sur le .jar.
2. soit en tapant la commande `java -cp .:./storm-3-2.jar programme.programme` dans le terminal dans le dossier contenant le .jar

L'interface graphique se lance. Pour effectuer une simulation, tapez dans la fenêtre de STORM :

1. `exec tache_EDF.xml` L'outil exécutera la simulation de l'exemple.
2. `pa` pour afficher le résultat.

La description détaillée du fichier .xml est donnée dans le manuel d'utilisation. Ouvrez l'exemple `tache_EDF.xml` dans votre éditeur favori.

1. Il faut préciser le temps de la simulation : `<SIMULATION duration="50">`. Modifiez le fichier `tache_EDF.xml` pour mettre une simulation de 120 et relancez la simulation. Pour visulaliser correctement la simulation, tapez `shift+->` dans les graphiques ;

2. il faut préciser les CPU. Rajouter un deuxième CPU
   `<CPU className="storm.Processors.CT11MPCore" name ="CPU B" id="2"></CPU>` et relancer la simulation ;
3. il faut préciser les informations sur les tâches. Rajoutez une tâche T_4 telle que $T(T_4) = 10$, $C(T_4) = 2$, $r(T_4) = 0$ et $D(T_4) = 10$. Puis relancez la simulation ;
4. il faut préciser la politique d'ordonnancement :
   `<SCHED className="storm.Schedulers.EDF_P_Scheduler"> </SCHED>` Changez de politique prédéfinie en regardant la liste dans le guide utilisateur dans les annexes.

Coder l'étude de cas des commandes de vol en STORM.

4.2.2 Codage d'ordonnanceurs personnalisés

Ouvrez le guide du designer p 12 et ouvrez le fichier `EDF_mien_Scheduler.java`. Il contient le code d'un ordonnanceur gEDF avec préemption. Le modèle de tâches et les interactions proposées avec l'ordonnanceur est décrit dans la Fig. 2. En noir, apparaissent les actions de la tâche et en bleu celles sur lesquelles l'ordonnanceur agit.

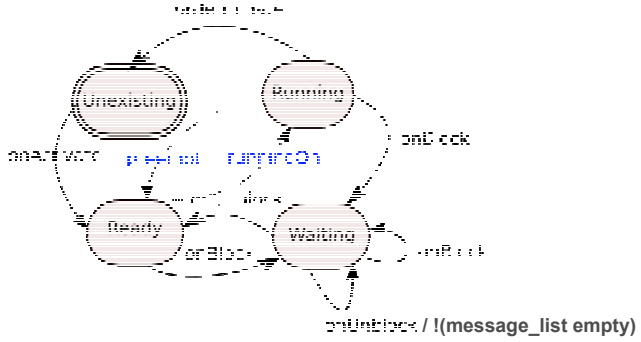


FIGURE 2 – Modèle de tâches en STORM

La class `EDF_mien_Scheduler` étend `Scheduler`. Chacune des classes doivent contenir :

1. une file globale `private LReady list_ready;` dans le cas d'un ordonnanceur global ou m files dans le cas d'un ordonnanceur partitionné ;
2. un booléen `private boolean todo = false;` qui activera la simulation ;
3. la méthode `init` qui initialise la ou les files ;
4. les méthodes `onActivate`, `onUnBlock`, `onBlock` et `onTerminated`. Les deux premières méthodes correspondent à une activation ou au réveil d'une nouvelle tâche. Il faut alors ajouter cette nouvelle tâche à la file et mettre le booléen `todo=true`. Les deux dernières méthodes correspondent à la terminaison ou au blocage d'une tâche active. Il faut alors retirer cette tâche de la file et mettre le booléen `todo=true`.

(a) ces méthodes prennent en entrée `EvtContext c`. Pour récupérer la tâche qui est modifiée, il faut utiliser la méthode `(Task)c.getCible();` sauf pour `onUnBlock` qui utilise la méthode `(Task)c.getSource();`

(b) on peut ajouter et lire un nouveau champ dans le fichier .xml.

5. une méthode `sched` qui correspond à l'ordonnancement et qui est activée quand `todo` est vrai ;
6. une méthode `select` qui contient le cœur de la simulation. D'abord il faut trier la file ou les files selon le critère de la politique d'ordonnancement. Dans le cas d'EDF, on trie selon la deadline absolue. Il faut ensuite préempter les tâches qui sont les moins prioritaires. Dans le cas d'EDF, on préempte les tâches dont la position est $\geq m$. Puis il faut activer les m tâches les plus prioritaires.

Pour tester le nouvel ordonnanceur :

1. compiler la class `javac -cp ./storm-3-2.jar EDF_mien_Scheduler.java`
2. lancer l'exécutable `java -cp .:./storm-3-2.jar programme.programme`
3. modifier le fichier `tache_EDF.xml` en commentant la ligne
   `<SCHED className="storm.Schedulers.EDF_P_Scheduler"> </SCHED>` et en décommentant la ligne
   `<!--<SCHED className="EDF_mien_Scheduler"> </SCHED> -->`

Exercice 12 *Programmer les ordonnanceurs suivants :*

1. *écrire un ordonnanceur global RM* `RM_Scheduler` *;*
2. *écrire un ordonnanceur partitionné - RM. Dans la description des tâches, rajouter le champ* `<TASK className="storm.Ta` *qui spécifie le numéro du processeur sur lequel est alloué la tâche.*
3. *s'il vous reste encore du temps, implanter l'heuristique first fit au lieu de lire le champ alloc.*