

Université de Nantes
UFR des Sciences et Techniques
Master ALMA 1

TP Système Temps Réel Embarqué

Robin BONCORPS
Guillaume CHARON
Jérôme PAGES



UNIVERSITÉ DE NANTES

Nantes, le 14 novembre 2012

Sommaire

Introduction	3
1 Cahier des charges	4
1.1 Générateur	4
1.1.1 Choix du mode	4
1.1.2 Génération aléatoire de tâches	5
1.1.3 Génération contrôlée de tâches	5
1.1.4 Écriture dans un fichier	5
1.2 Ordonnanceur	5
1.2.1 Parsage du fichier d'entrée	6
1.2.2 Analyse d'ordonnançabilité	7
1.2.3 Simulation d'ordonnancement	7
1.2.4 Ecriture du fichier de trace	7
2 Analyse du problème	8
2.1 Le générateur de tâches	8
2.2 L'ordonnanceur	8
3 Présentation de la solution	10
3.1 Explication du choix du langage	10
3.2 Générateur	10
3.2.1 Choix du mode	10
3.2.2 Génération Aléatoire	11
3.2.3 Generation Controlée	12
3.2.4 Ecriture du résultat obtenu	13
3.3 Ordonnanceur	14
3.3.1 Conteneur	14
3.3.2 Analyse du fichier d'entrée	14
3.3.3 Analyse d'ordonnançabilité	15
3.3.4 Simulation d'ordonnancement	17
4 Notice utilisateur	24
4.1 Étape 1 : compilation	24
4.2 Étape 2 : génération de tâches	24
4.3 Étape 3 : ordonnanceur	24
5 Jeux d'essai	26

6	Conclusion	27
6.1	Problèmes rencontrés	27
6.1.1	Combiner l'aléatoire et le contrôle des valeurs générées	27
6.2	Améliorations possibles	27
6.2.1	Fonction de génération aléatoire	27
6.2.2	Factoriser les éléments redondants des algorithmes d'ordonnancement	28
6.2.3	Ajout d'algorithmes	28

Introduction

Le but de ce projet, qui nous a été confié dans le cadre des Travaux Pratiques de Systèmes Temps Réel Embarqués (Master ALMA 1), est de développer un simulateur d'ordonnanceur temps réel implémentant les algorithmes suivants :

- Rate Monotonic (RM) ;
- Earliest Deadline First (EDF) ;
- Background Server (BG) ;
- Total Bandwidth Server (TBS).

Deux parties sont à identifier. Étant donné que l'ordonnanceur a besoin d'un jeu de tâches périodiques et/ou apériodiques comme base de travail, la première partie de ce projet sera de créer un générateur de tâches. L'autre partie concernera le simulateur en lui-même. Ces deux parties sont indépendantes, c'est-à-dire que le livrable final est composé de deux exécutables différents.

Le résultat final sera observable sous deux formes : un affichage à l'écran, et un fichier .ktr en sortie écrit selon la syntaxe acceptée par l'outil Kiwi, qui nous permettra de visualiser graphiquement l'ordonnancement des tâches.

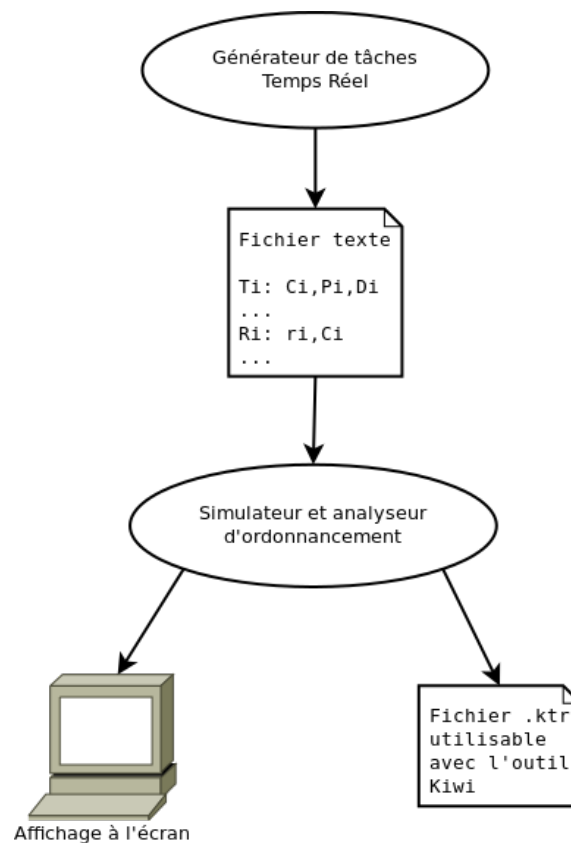


FIGURE 1 – Résumé

1 Cahier des charges

1.1 Générateur

Cette application offre la possibilité de créer un fichier contenant un jeu de tâches (périodiques et apériodiques non critiques). Dans un premier temps, le programme permet de générer manuellement ce jeu de tâche en offrant une interface minimaliste. Dans un deuxième temps, il est possible de générer automatiquement ce fichier en fournissant des contraintes à respecter pour les tâches périodiques et apériodiques.

Lorsque le programme est lancé, l'utilisateur doit choisir le mode de génération. Il rentre les données nécessaires au bon déroulement du mode. Le programme crée un fichier qui contient le résultat de la génération de tâches.

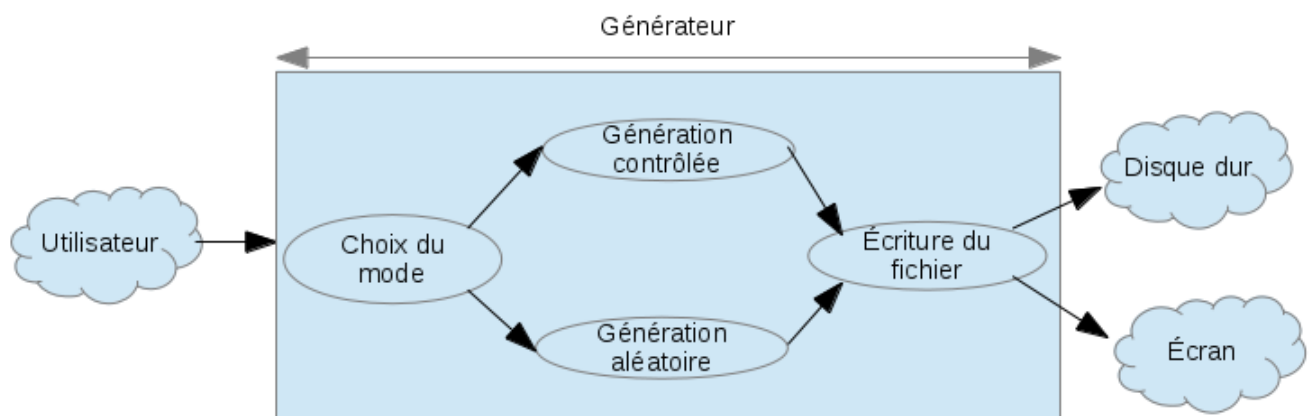


FIGURE 1.1 – Fonctionnement et modules du générateur.

1.1.1 Choix du mode

Dans ce premier module, on choisit le mode d'utilisation du logiciel en fonction de ce que souhaite l'utilisateur. Pour cela, il peut soit renseigner sous forme d'arguments les différentes données, soit en sélectionnant dans le menu ses préférences.

Parmi les choix qui lui sont offerts, on retrouve les cas présentés dans le sujet du projet (génération aléatoire ou contrôlée et tâches périodiques ou non) mais aussi la possibilité de renseigner un fichier contenant déjà toutes les informations sur les tâches. Dans le premier cas, l'utilisateur devra renseigner ensuite le nombre de tâches concernées tandis que dans l'autre cas, le fichier ne sera pas interprété et ce programme s'arrêtera.

1.1.2 Génération aléatoire de tâches

Le premier type de génération permet de générer automatiquement des tâches. L'utilisateur devra renseigner le nombre de tâches périodiques et apériodiques à générer ainsi que le pourcentage d'utilisation du CPU. Ceci permet de générer des fichiers de test aléatoire qui seront visualisable par la suite. Lors de la génération, le système calcule les différentes valeurs caractéristiques des tâches périodiques (C_i, P_i, D_i) et des tâches apériodiques (r_i, C_i) en accord avec le pourcentage d'utilisation du CPU.

1.1.3 Génération contrôlée de tâches

Le deuxième type de génération fait appel à l'utilisateur pour renseigner chacune des valeurs de chaque tâche. De cette manière, l'utilisateur va pouvoir tester plus facilement des cas spécifiques et donc observer de manière plus précise le fonctionnement de l'ordonnanceur par la suite.

Lors de la génération, pour toutes les tâches périodiques il devra renseigner les C_i , P_i et D_i , c'est-à-dire respectivement les durées d'exécution maximales, les périodes d'activation et le délai critique). Dans le cas où il y a aussi des tâches apériodiques, il devra renseigner aussi r_i (la date de réveil) et C_i .

1.1.4 Écriture dans un fichier

Ce module a pour fonction d'écrire dans un fichier les données contenues dans un flux (= stream) et calculées à partir des deux modules précédents de génération. L'intérêt de séparer ces fonctions est double : il est plus pratique de factoriser le code à travers une seule fonction d'écriture appelée par les différentes générations et cela facilite la maintenance.

1.2 Ordonnanceur

Cette application s'incrit à la suite du générateur de tâches. Elle offre, après récupération des informations du fichier issus du générateur, la possibilité de vérifier l'ordonnançabilité d'un jeu de tâches fournie. Ceci nous permet de savoir si les contraintes temporelles sont respectées.

La deuxième fonctionnalité de ce programme est d'offrir un environnement de simulation selon plusieurs politiques d'ordonnancements :

- pour les tâches périodiques : **RM** et **EDF**
- pour les tâches apériodiques non critiques : **BG** et **TBS**

Cet environnement, après simulation, fourni à l'utilisateur des résultats de performances relatifs à l'ordonnement :

- le nombre de commutations de contexte,
- le nombre de préemptions
- les temps de réponse des tâches apériodiques.

À la suite de la simulation, le programme doit fournir un fichier de trace de la séquence d'ordonnement. Ce fichier est au format « ktr » et est exploitable par l'outil Kiwi. Kiwi est un outil graphique développé à l'Université polytechnique de Valence en Espagne. Il permet, à partir d'un fichier texte normé, d'afficher un graphe d'ordonnement. Nous détaillerons par la suite le contenu du fichier de trace mais il est important pour la suite de savoir que le fichier doit être rempli de manière chronologique.

Le programme final devra pouvoir être exécutable en ligne de commande.

En entrée : le nom du fichier contenant le jeu de tâches périodiques et apériodiques pour lesquelles on veut simuler l'ordonnancement.

En sortie : un ou plusieurs fichiers .ktr utilisable avec l'outil Kiwi et contenant la séquence d'ordonnancement. Le programme affichera également quelques résultats via la sortie standard.

Afin de faciliter la prise en main du programme, on proposera un menu succinct présentant les différentes fonctionnalités qui s'offrent à l'utilisateur.

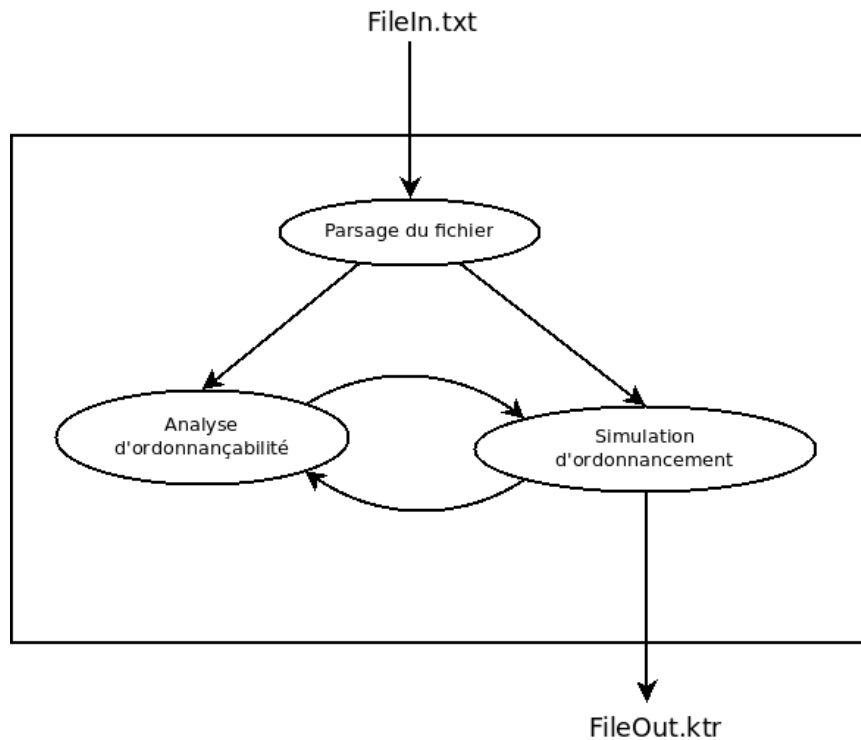


FIGURE 1.2 – Diagramme des fonctionnalités

1.2.1 Parsage du fichier d'entrée

Bien que le fichier en entrée soit, dans le cas général, un fichier écrit par notre générateur de tâche et donc normalement correct, nous ne sommes pas à l'abris de voir arriver en entrée un fichier corrompu. Soit par l'utilisateur lui-même, soit de n'importe quel autre moyen que ce soit. De plus, la syntaxe du fichier en entrée est relativement simple, ce qui permet à un utilisateur lambda de créer son propre fichier sans passer par notre générateur de tâches.

Sachant que le fichier sera parsé suivant le *pattern* de spécification des tâches utilisé par notre générateur, il est indispensable de vérifier la bonne syntaxe du fichier avant de récupérer quoi que ce soit. Cette vérification est effectuée grâce à une expression régulière représentant toutes les syntaxes possible pour une ligne du fichier. Dans les faits, cette *Regex* est divisée en deux :

- Pour les tâches périodiques : $\text{\textasciitilde}T[0-9]^+ : [0-9]^+, [0-9]^+, [0-9]^+ \$$;
- Pour les tâches apériodiques : $\text{\textasciitilde}R[0-9]^+ : [0-9]^+, [0-9]^+ \$$.

Chacune des lignes du fichier doit donc respecter soit l'une, soit l'autre expression régulière.

Une fois la syntaxe vérifiée, il est très aisé de travailler le fichier ligne par ligne pour en extraire

les informations nécessaires grâce aux fonctions sur les *string*. Chacune des tâches sera ajouter à un Conteneur de tâches qui sera ensuite passé à l'ordonnanceur pour que celui-ci puisse travailler dessus.

1.2.2 Analyse d'ordonnançabilité

On fait tous les calculs en même temps :

- Condition nécessaire pour l'ordonnançabilité du jeu de tâches avec RM ;
- Condition suffisante avec RM ;
- Condition suffisante et nécessaire avec EDF.

1.2.3 Simulation d'ordonnancement

1.2.4 Ecriture du fichier de trace

Pendant la simulation de l'ordonnancement, l'application génère les différentes lignes qui composent le fichier de trace exploitable par Kiwi. Ce fichier se décompose en 2 parties :

- Une entête contenant la déclaration des tâches et la durée d'ordonnancement (hyperpériode)
- Les événements concernant les tâches dans l'ordre chronologique.

2 Analyse du problème

2.1 Le générateur de tâches

Dans le cadre de ce mini-projet, les principales difficultés concernent la génération pseudo-aléatoire de fonctions. En effet, il ne s'agit pas réellement d'une génération aléatoire puisqu'en plus de l'incapacité des ordinateurs de générer du hasard, le résultat attendu doit être borné et facilement manipulable. Par exemple, nous ne souhaitons pas ce genre de fraction : car cela donnera des hyperpériodes trop grandes et difficilement manipulables avec kiwi mais plutôt de ce genre :

2.2 L'ordonnanceur

Après que les tâches aient été créées, un autre problème se pose : comment peut-on ordonner les tâches de manière automatique afin de simuler un véritable ordonnanceur ? Pour cela, on identifie deux types d'algorithmes possibles : ceux qui prennent en compte uniquement les tâches périodiques et ceux qui permettent de gérer aussi les tâches apériodiques. Ici, nous allons nous intéresser uniquement aux algorithmes RM, EDF, BG et TBS.

Pour unifier le fonctionnement de tous les algorithmes, il convient alors de fédérer les manières de représenter les tâches en mémoire afin de ne pas reparser le fichier à chaque ordonnancement. Pour pouvoir appeler à la suite les algorithmes, nous avons opté pour la solution de créer un objet conteneur qui sauvegardera des objets tâche apériodique et périodique dans des vecteurs afin d'y accéder facilement. Les objets tâches comportent les attributs spécifiques C_i , P_i et D_i ou r_i et C_i en fonction de leur type. De cette manière, en utilisant des accesseurs, il sera facile de récupérer les diverses informations sur les tâches pour construire la politique d'ordonnement et effectuer les tests de conditions d'ordonnabilité.

Avant même de commencer à réfléchir sur les algorithmes d'ordonnement, nous avons réfléchi à la structure de ces algorithmes. Nous verrons dans la conclusion que ça n'était pas forcément la bonne solution, et qu'il aurait mieux valu extraire cette structure de l'algorithme d'ordonnement.

Premièrement, un ordonnanceur a besoin d'avoir plusieurs informations à propos des tâches qui lui demandent du temps d'exécution. Par exemple, l'algorithme EDF, qui élit une tâche selon son D_i , doit bien entendu connaître le D_i des différentes tâches qui attendent leur exécution. Toutes ces informations dont ont besoin les algorithmes, et qui seront stockées dans différentes structures, devront être initialisées en tout début de fonction afin que l'ordonnanceur puisse s'en servir à sa guise plus tard. Nous appellerons toutes ces informations le « contexte d'exécution ».

Bien entendu, ça n'est pas à l'ordonnanceur de mettre à jour le contexte d'exécution. On réservera donc la fin de chaque fonction d'ordonnement à la mise à jour de ces structures.

Enfin, l'ordonnanceur ne doit pas être appelé à chaque unité de temps écoulée mais seulement : au

réveil d'une tâche, à la fin de l'exécution d'une tâche, et au début d'une nouvelle période d'une tâche périodique.

Voici la structure finale des algorithmes d'ordonnancement :

1. Initialisation du contexte d'exécution ;
2. Jusqu'à la fin de l'hyperpériode :
 - (a) Vérifier si l'ordonnanceur doit être appelé ;
 - (b) Si oui, exécuter l'algorithme d'ordonnancement ;
 - (c) Affichage du résultat à l'écran ;
 - (d) Mettre à jour le contexte d'exécution .
3. Affichage des statistiques sur l'ordonnancement (nombre de changements de contexte, de préemptions, temps de réponse des tâches apériodiques) .

Une fois cette structure mise en place, il suffit simplement d'implémenter un algorithme d'ordonnancement qui va tout simplement retourner le numéro de la tâche qu'il a élue en utilisant les informations qui sont mises à disposition par le contexte d'exécution.

3 Présentation de la solution

3.1 Explication du choix du langage

Nous avons opté pour le langage C++ pour réaliser ce projet car celui-ci possède un certain nombre d'avantages. Il s'agit d'un langage orienté objet très utilisé dans le monde et de nombreux outils libres sont disponibles pour l'utiliser (comme GCC pour la compilation par exemple). De plus, la documentation correspondante est très facilement accessible dans la littérature ou sur le web.

Bien que le travail qui nous a été demandé ne requiert pas la maîtrise d'une programmation bas niveau, le langage C++ nous apparait en cohérence avec le thème abordé par le projet (les systèmes temps réel embarqués).

3.2 Générateur

3.2.1 Choix du mode

Pré-condition : Le programme vient d'être lancé.

Post-condition : Un choix valide a été effectué.

Objectif : Permettre de choisir entre la génération aléatoire et la génération contrôlée.

Algo :

```
Entree : Int choix
Sortie : -
```

```
Si (choix == 1 ) Alors
    Lancer la generation controlee
Sinon
    Si (choix == 2) Alors
        Lancer la generation aleatoire
    Sinon
        Si (choix == 3) Alors
            Quitter le programme
        Sinon
            Afficher erreur de choix
    Finsi
Finsi
Finsi
```

3.2.2 Génération Aléatoire

Pré-condition : La génération aléatoire a été choisie.

Post-condition : Toutes les valeurs caractéristiques des tâches ont été générées en accord avec le paramétrage de l'utilisateur.

Objectif : Générer aléatoirement les caractéristiques (C_i, P_i, D_i) d'un nombre de tâches (déterminé par l'utilisateur) en accord avec un facteur d'utilisation du processeur (déterminé lui aussi par l'utilisateur). Ex : pour trois tâches et facteur = 75%

Tirage aléatoire de trois nombres (35, 25, 15) dont la somme vaut 75.

Ces nombres représentent le rapport C_i/P_i dans la formule

Il faut ensuite donner une valeur à C_i et P_i . Pour cela, on trouve le pgcd du nombre et de 100. Pour $C_1/P_1 = 35$, on obtient 5. on affecte à $C_1 \leftarrow 35/5$ et à $P_1 \leftarrow 100/5$, on obtient donc $C_1 = 7$ et $P_1 = 20$. On considère, dans ce programme que $P_i = D_i$.

On obtient donc dans cet exemple :

$T_1(7, 20, 20)$

$T_2(1, 4, 4)$

$T_3(3, 20, 20)$

Toutes les données sont stockées dans 3 tableaux (un pour les C_i , un pour P_i , un pour D_i).

Algo :

Entree : Int nbTaches, Int factUtProcesseur

Sortie : 3 Tableaux d Int (pour les valeurs de C_i , P_i , D_i).

indice du tableau = numero de la tâche - 1..

TabCi[nbTaches], TabPi[nbTaches], TabDi[nbTaches]

nbTachesRestantes = nbTaches - 1

nbMax = factUtProcessus

/*

On calcule la valeur maximale que peut prendre le nombre tire

aléatoirement :

maximum = Up - somme des C_i/P_i precedents - nombre de tâches restantes

Ensuite, on tire au hasard une valeur allantDe 1 a cette valeur maximale.

Exemple : maxT1 = 75 - 0 - 2

randT1 = 32 (valeur calculée avec le pseudo-hasard)

maxT2 = 75 - 32 - 1

randT2 = 18

maxT3 = 75 - (32 + 18) - 0

randT3 = 25

Pour resumer, on obtient ici les C_i/P_i : 32, 18 et 25

*/

```

Pour i allantDe 0 a (nbTaches - 1)
    nbMaxLimite = nbMax - nbTachesRestantes
    // le nombre aleatoire correspond a (Ci/Pi)
    nbAleatoire = nombre Aleatoire entre 1 et ce nombre maximum

    // On calcule et stocke la valeur des Ci, Pi et Di dans 3 tableaux
    distincts
    tabCi[i] = nb\_genere / pgcd(nb\_genere,100)
    tabPi[i] = 100 / pgcd(nb\_genere,100)
    tabDi[i] = 100 / pgcd(nb\_genere,100)

    /* On abaisse ensuite la limite pour le prochain tirage aleatoire afin
       de ne jamais dépasser la valeur de Up. */
    limite\_maj = limite\_maj - nb\_genere
Fin du pour

// Pour la derniere tache, on ne genere pas de valeur. On prend ce qu'il
reste.
nb\_genere = limite\_maj
tabCi[ nbTaches - 1 ] = nb\_genere / pgcd(nb\_genere,100)
tabPi[ nbTaches - 1 ] = 100 / pgcd(nb\_genere,100)
tabDi[ nbTaches - 1 ] = 100 / pgcd(nb\_genere,100)

```

3.2.3 Generation Controlée

Pré-condition : La génération controlée a été choisie.

Post-condition : Toutes les valeurs caractéristiques des tâches ont été générées en accord avec le paramétrage de l'utilisateur.

Objectif : Permettre à l'utilisateur de générer des tâches périodiques ou apériodiques en renseignant les différentes caractéristiques de chacune des tâches (C_i , P_i , D_i) ou, respectivement, (r_i , C_i).

Algo :

Entree : Int nbTaches

Sortie : 3 Tableaux d Int (pour les valeurs de Ci, Pi, Di).

indice du tableau = numero de la tache - 1 .

```

//generation des taches periodiques
lire(nbTachesPeriodiques)

```

```

TabCiP[ nbTachesPeriodiques ], TabPiP[ nbTachesPeriodiques ], TabDiP[
    nbTachesPeriodiques ]

```

```

Pour i allantDe 0 a (nbTachesPeriodiques - 1) Faire
    // on demande a l utilisateur de renseigner les differentes valeurs
    ...
    lire(CiP)
    lire(PiP)
    lire(DiP)
    // ... et on les insere dans les tableaux respectifs
    tabCiP[ i ] = CiP
    tabPiP[ i ] = PiP
    tabDiP[ i ] = DiP

```

Finpour

```

//generation des taches Aperiodiques
lire(nbTachesAperiodiques)

```

```

TabriA[ nbTachesAperiodiques ], TabCiA[ nbTachesAperiodiques ]

```

```

Pour i allantDe 0 a (nbTachesAperiodiques - 1) Faire
    // on demande a l utilisateur de renseigner les differentes valeurs
    ...
    lire(riA)
    lire(CiA)
    // ... et on les insere dans les tableaux respectifs
    tabriA[ i ] = riA
    tabCiA[ i ] = CiA

```

Finpour

3.2.4 Ecriture du résultat obtenu

Pré-condition : La génération (qu'elle soit contrôlée ou non) envoie un outputstream contenant les chaînes de caractère à écrire dans un fichier.

Post-condition : L'écriture s'est bien déroulée : le fichier contient bien la chaîne.

Objectif : Enregistrer les données générées par le module précédent de manière pérenne dans un fichier.

Algo :

creation du fichier

```

    outputstream ops = ""
Pour i allantDe 0 a (nbTachesPeriodiques - 1) Faire
    ops << "T" << (i+1) << ": " << tabCiP[i] << "," << tabPiP[i] << "," <<
        tabDiP[i] << endl

```

Finpour

```

Pour i allantDe 0 a (nbTachesAperiodiques - 1) Faire

```

```
ops << R" << (i+1) << ": " << tabriA[i] << "," << tabCiA[i] << endl
Finpour
```

écriture dans le fichier de ops

3.3 Ordonnanceur

3.3.1 Conteneur

Comme nous l'avons vu, les tâches qui sont spécifiées dans le fichier en entrée doivent être stockées dans un conteneur pour que l'ordonnanceur puisse travailler dessus. Voici donc la manière dont nous avons décidé de concevoir notre conteneur de tâches :

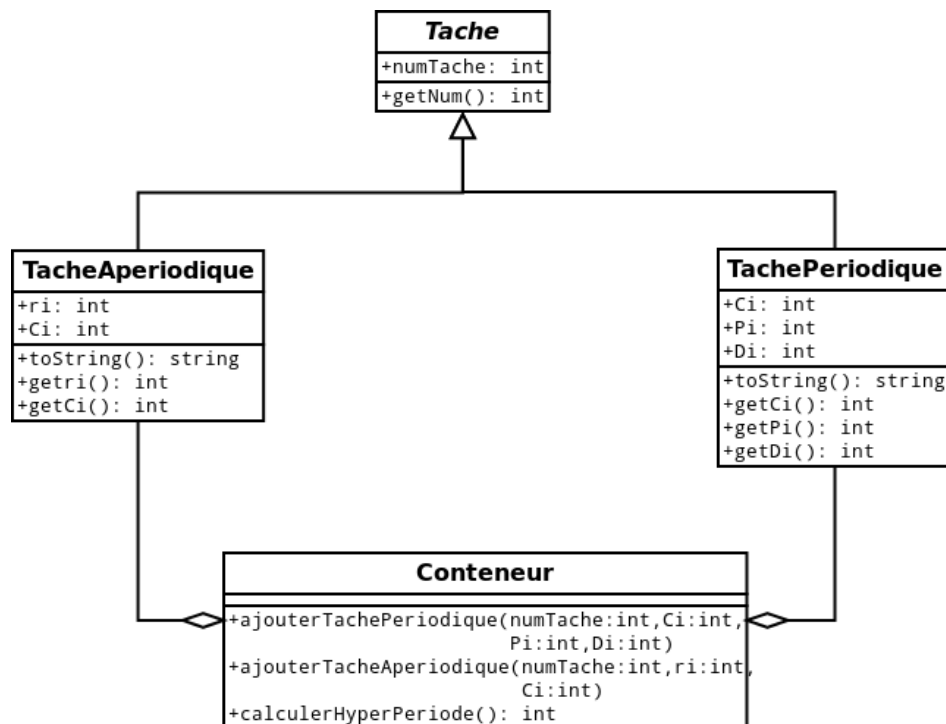


FIGURE 3.1 – Diagramme de classes du conteneur de tâches

3.3.2 Analyse du fichier d'entrée

Pré-condition : Le fichier spécifié existe et est lisible

Post-condition : Le fichier est syntaxiquement valide et toutes les tâches ont été ajoutées au conteneur

Objectif : Ajouter toutes les tâches spécifiées par le fichier dans le conteneur

Algo :

Entree : file fileIn , Conteneur conteneur

Sortie : conteneur

```
Tantque line = nextLine(fileIn) Faire
    verifierSyntaxe(line);
```

```

elements = eclater(line , ',')
Si commencePar(line , 'T') Alors
    ajouterTachePeriodique(conteneur , elements[0] , elements[1] ,
        elements[2])
Sinon si commencePar(line , 'R') Alors
    ajouterTacheAperiodique(conteneur , elements[0] , elements[1])
Finsi
Fintantque

```

3.3.3 Analyse d'ordonnancabilité

Condition nécessaire RM

Pré-condition : n représente le nombre de tâches périodiques.

Post-condition : Soit c'est non-ordonnancable, soit on ne peut rien conclure.

Objectif : Vérifier les conditions d'ordonnancabilité nécessaires pour l'algorithme RM.

Algo :

$U = 0$

Pour i de 0 à n Faire

$U = U + C_i/P_i$

Finpour

Si $U \leq 1.0$ Alors

afficher("On ne peut rien conclure")

Sinon

afficher("Non-ordonnancable")

Finsi

Condition suffisante RM

Pré-condition : n représente le nombre de tâches périodiques.

Post-condition : Soit c'est ordonnancable, soit on ne peut rien conclure.

Objectif : Vérifier les conditions d'ordonnancabilité suffisantes pour l'algorithme RM.

Algo :

$U = 0$

$UBoundRM = n * (2^{(1.0/n)} - 1)$

Pour i de 0 à n Faire

$U = U + C_i/P_i$

Finpour

Si $U \leq UBoundRM$ Alors

afficher("Ordonnancable")


```

Sinon
    afficher("On ne peut rien conclure")
Finsi

```

Condition nécessaire et suffisante EDF

Pré-condition : n représente le nombre de tâches périodiques et il n'y a pas de tâches apériodiques.

Post-condition : On obtient une réponse binaire : le système est ordonnancable ou non.

Objectif : Vérifier les conditions d'ordonnancabilité nécessaires et suffisantes pour l'algorithme EDF.

Algo :

```

egaliteDiPi = vrai
superioritePiDi = vrai
U = 0

Pour i de 0 a n Faire
    // Verifie si tous les Pi sont bien egaux aux Di
    egaliteDiPi &= (Pi == Di)
    // idem pour Pi > Di
    superioritePiDi &= (Pi > Di)
Finpour

Si egaliteDiPi Alors
    afficher "Test de condition necessaire et suffisante pour EDF : "

    Pour i de 0 a n Faire
        U = U + Ci/Pi
    Finpour

    Si (U <= 1.0) Alors
        afficher "ordonnancable"
    Sinon
        afficher "non-ordonnancable"
    Finsi

Sinon Si superioritePiDi Alors
    afficher "Test de condition suffisante pour EDF : "

    Pour i de 0 a n Faire
        U = U + Ci/Di
    Finpour

    Si (U <= 1.0) Alors
        afficher "ordonnancable"

```

```

    Sinon
        afficher "on ne peut rien conclure"
    Finsi
Sinon
    afficher "Erreur : au moins un  $P_i < D_i$ "
Finsi

```

Condition nécessaire et suffisante EDF-TBS

Pré-condition : n représente le nombre de tâches périodiques et U_s est .

Post-condition : On obtient une réponse binaire : le système est ordonnançable ou non.

Objectif : Vérifier les conditions d'ordonnançabilité nécessaires et suffisantes pour l'algorithme EDF.

Algo :

$U_p = 0$

Pour i de 0 à n Faire

$U_p = U_p + C_i/P_i$

Finpour

afficher "Test de condition nécessaire et suffisante pour EDF-TBS : "

Si $(U_p + U_s \leq 1.0)$ Alors

afficher "ordonnançable"

Sinon

afficher "non-ordonnançable"

Finsi

3.3.4 Simulation d'ordonnancement

Le simulateur d'ordonnancement est découpé en deux algorithmes principaux :

- RM;
- EDF.

Nous avons en premier lieu commencé par développer ces deux algorithmes sans tenir compte des tâches apériodiques dans le but ensuite de les « patcher » avec nos algorithmes d'ordonnancement des tâches apériodiques (BG et TBS).

poll_needed

Pré-condition : un algo d'ordonnancement a été lancé

Post-condition : retourne un booléen qui informe du besoin du besoin d'élire une nouvelle tâche à exécuter

Objectif : Savoir si on doit élire une nouvelle tâche ou non

Algo :

`bool need_to_poll = false;`

```

//verification du reveil d'une tache Periodique
  Pour tachePeriodique dans context Faire
    Si time = (date Reveil de tachePeriodique) Alors
      ajout de tachePeriodique dans listePeriodiquesPretes
      need_to_poll = true
    Finsi
  Finpour

//verification du reveil d'une tache nAperiodiques
Si (un serveur de taches a ete specifie) Alors
  Pour tacheAperiodique dans context Faire
    Si time = (date Reveil de tacheAperiodique) Alors
      ajout de tacheAperiodique dans listeAperiodiquePretes
      need_to_poll = true
      Si (le serveur de tache aperiodique choisi est TBS) Alors
        calcul de dk pour tacheAperiodique
        stockage dans context
      Finsi
    Finsi
  Finpour
Finsi

//verification de la terminaison de la tache courante executee
Si (la tache courante executee n'est PAS un temps creux) Alors
  Si (la capacite restante de tache courante executee = 0) Alors
    need_to_poll = true
    Si (la tache courante executee est une tache Aperiodique)
      Alors
        suppression de la tache de listeAperiodiquesPretes
      Sinon
        suppression de la tache de listePeriodiquesPretes
      Finsi
    Finsi
  Finsi
Finsi

retourne need_to_poll

```

init_context

Pré-condition : un algo d'ordonnancement a été lancé

Post-condition : le context d'exécution des taches a été initialisé

Objectif : initialiser le contexte d'exécution des taches (prochain reveil de tache, prochaine deadline, capacite restante)

- creation d'un tableau contenant toutes les taches (taches periodiques puis taches aperiodiques)
- le tableau context est un tableau de 3 case de large et nbTaches de long
- pour les taches periodiques :
indice -> Capacite restant | prochaine date de reveil | prochaine date echeance
- pour les taches aperiodiques :
indice -> Capacite restante | date de reveil | date echeance (si TBS sinon 0)

Algo :

```
context [ nbTaches ] [ 3 ]
```

```
Pour i allantDe 0 a nbTachePeriodiques Faire
```

```
    context[i][0] = Ci de tachePeriodique_i
```

```
    context[i][1] = Pi de tachePeriodique_i
```

```
    context[i][2] = Di de tachePeriodique_i
```

```
Finpour
```

```
Si (il y a un serveur de precise) Alors
```

```
    Pour i allantDe nbTachesPeriodiques a nbTaches Faire
```

```
        context[i][0] = Ci de tacheAperiodique_i
```

```
        context[i][1] = ri de tacheAperiodique_i
```

```
        context[i][2] = 0
```

```
    Finpour
```

```
Finsi
```

MAJ_context

Pré-condition : le contexte à été initialisé

Post-condition : le contexte a été mis a jour

Objectif : mettre a jour le contexte pour faire état des reveils de taches et vérifier le dépassement d'échéance

Algo :

```
Pour tachePeriodique dans context Faire
```

```
    //verification du depassement d'echeance
```

```
    Si (time = (date echeance tachePeriodique) ) Alors
```

```
        Si ( (capacite restante de tachePeriodique) > 0 ) Alors
```

```
            Message("Depassement d'echeance pour la tache : T" +  
                    tachePeriodique)
```

```
            quitter
```

```
        Finsi
```

```
    Finsi
```

```
Si (time = (date reveil de tachPeriodique) ) Alors
```

```
    capacite restante tachePeriodique = Ci de tachePeriodique
```

```

        date de reveil tachePeriodique = date de reveil tachePeriodique +
            Pi de tachePeriodique
        date echeance tachePeriodique = date echeance tachePeriodique + Di
            de tachePeriodique
    Finsi
Finpour

//Verification de depassement d'echeance pour les taches aperiodiques
Si (serveur = TBS) Alors
    Pour tacheAperiodique dans context Faire
        Si (time = (deadline de tacheAperiodique) ) Alors
            Si ( (capacite restante tacheAperiodique) > 0 ) Alors
                Message("Depassement d'echeance pour la tache : R" +
                    tacheAperiodique)
                quitter
            Finsi
        Finsi
    Finpour
Finsi

```

Rate Monotonic + Serveur

Pré-condition : le conteneur contient des taches & RM a été choisi

Post-condition : une séquence d'ordonnancement est délivrée

Objectif : Simuler un ordonnancement temps reel selon RM

Algo :

Entree : Le conteneur de tache & un serveur si il y a des taches aperiodiques

Sortie : Affichage de l'ordonnancement

```

//Initialisation du context :
    - Creation d'un tableau de taches periodiques par ordre de prio :
        tabPrioPeriodique
    - Creation d'un tableau de taches aperiodiques par ordre de prio :
        tabPrioAperiodique
    - Creation d'un tableau qui contient la capacite restante de chaque
        tache : tabTpsRestant

int t := 0
int task_executed := -1
bool tache_elue = false;

Tantque (t < getHyperPeriode()) Faire
    Booleen need_to_poll = poll_needed()

```

```

Si(need_to_poll) Alors
    tache_elue = false
    Pour taches dans tabPrioPeriodique Faire
        Si (tabTpsRestant[tache] >0 && tache_elue == false) Alors
            task_executed = tache;
            tache_elue = true;
        Finsi
    Finpour
    Si tache_elue == false Alors
        //temps_creux ou selection d'une tache aperiodique
    Finsi
finsi

tabTpsRestant[task_executed]--;
maj_context()
t++

```

Fintantque

Earliest Deadline First + Serveur

Pré-condition : Le conteneur contient des taches & EDF a été choisi

Post-condition : une séquence d'ordonnancement est délivrée

Objectif : Simuler un ordonnancement temps reel selon EDF

Algo :

Entree : le conteneur de tache & un serveur si il y a des taches aperiodiques

Sortie : Affichage de l'ordonnancement

```

init_context()
int t := 0
int task_executed := -1

```

```

Tantque (t < getHyperPeriode()) Faire
    Booleen need_to_poll = poll_needed()

```

```

    Si(need_to_poll) Alors
        Si (aucune tache periodique prete) Alors
            Si (aucune tache aperiodique prete) Alors
                temps_creux
            Sinon
                task_executed = Selection de la tache Aperiodique prete
                dont le ri est le plus petit

```

```

Finsi
Sinon
    deadlineProche = hyperperiode
    task_executed = temps_creux

    //election de la tache Periodique prete la plus prioritaire
    Pour tachePeriodiquePrete dans tabPeriodiquesPretes Faire
        Si ( deadlineProche > (prochaine date echeance
            tachePeriodiquePrete) ) Alors
            deadlineProche = prochaine date echeance
            tachePeridiodiquePrete
            task_executed = tachePeriodiquePrete
        Sinon
            Si ( deadlineProche = (prochaine date echeance
                tachePeriodiquePrete) ) Alors
                // Conflit —> on compare les dates de reveil
                Si ( ri_task_executed > ri_tacheAperiodiquePrete)
                    Alors
                        task_executed = tachePeriodiquePrete
                Sinon
                    Si (ri_task_executed =
                        ri_tacheAperiodiquePrete) Alors
                        // Conflit —> on compare les numeros de
                        tache
                        Si (num_task_executed >
                            num_tachePeriodiquePrete) Alors
                            task_executed = tachePeriodiquePrete
                    Finsi
                Finsi
            Finsi
        Finsi
    Finsi
Finsi
Finsi
Finspour

//election de la tache Aperiodique prete la plus prioritaire
Si (serveur == TBS) Alors
    Pour tacheAperiodiquePrete dans tabAperiodiquesPretes
        Faire
            Si ( (dk de tacheAperiodiquePrete) < deadlineProche )
                Alors
                    task_executed = tacheAperiodiquePrete
            Sinon
                Si ( (dk de tacheAperiodiquePrete) =
                    deadlineProche ) Alors

```

```

        Si ( (ri de tacheAperiodiquePrete) < (ri de
            task_executed) ) Alors
            task_executed = tacheAperiodiquePrete
        Finsi
    Finsi
Finsipour
Finsi
Finsi
Finsi
Finsi
    Capacite_restante de task_executed--
    t++

    MAJ_context()

fin tant que

```


4 Notice utilisateur

Vous disposez normalement des sources de l'application ainsi que des Makefile répartis dans deux répertoires : *Generateur* et *Ordonnanceur*. L'application est en effet divisée en deux parties, l'une générant un jeu de tâches périodiques et apériodiques dans un fichier texte, l'autre simulant l'ordonnement des tâches générées.

4.1 Étape 1 : compilation

```
$ cd Generateur
$ make clean && make
$ cd ../Ordonnanceur
$ make clean && make
```

4.2 Étape 2 : génération de tâches

Placez-vous dans le répertoire *Generateur* et lancez l'exécutable :

```
./Generateur [-f fichier.txt|-c nbTaches|-a nbTaches utilisationCPU].
```

Les arguments ne sont pas obligatoires mais vous permettent de spécifier un certain nombre d'options qui vous seront sinon demandées à l'exécution. Si le fichier de sortie n'est pas spécifié, les tâches seront créées dans le fichier *jeuDeTaches.txt*.

Il suffit ensuite de suivre l'exécution du programme. La génération aléatoire vous permet de générer un certain nombre de tâches sans vous soucier de leurs valeurs. Il vous faudra tout de même spécifier le nombre de tâches que vous souhaitez et le facteur d'utilisation processeur indu. Quant à la génération contrôlée, elle vous propose de spécifier vous-même les paramètres des tâches à créer.

Une fois l'exécution terminée, vous pouvez ouvrir le fichier de sortie et le modifier. Attention toutefois, la syntaxe doit respecter la forme suivante, ou le fichier sera refusé par l'ordonnanceur : $R_i : r_i, D_i$ ou $T_i : C_i, P_i, D_i$ avec une seule tâche par ligne. Vous pouvez également proposer votre propre fichier sans passer par le générateur.

4.3 Étape 3 : ordonnanceur

Placez-vous dans le répertoire contenant l'exécutable de l'ordonnanceur et lancez l'exécutable :

`./Ordonnanceur file.txt`. *file.txt* est un argument obligatoire et correspond au fichier contenant les tâches que vous souhaitez ordonnancer. Il peut donc s'agir d'un fichier créé par le générateur, ou de

votre propre fichier respectant la même syntaxe.

Si votre fichier est correct, un menu s'affiche et vous propose d'effectuer plusieurs action. Il vous suffit de vous laisser guider. Les résultats de vos actions s'afficheront directement à l'écran, et un fichier .ktr sera créé à chaque fois que vous lancerez un algorithme d'ordonnancement. Ce fichier décrit l'ordonnancement des tâches et peut être ouvert avec l'outil Kiwi, permettant de visualiser graphiquement le résultat.

5 Jeux d'essai

TODO

6 Conclusion

6.1 Problèmes rencontrés

6.1.1 Combiner l'aléatoire et le contrôle des valeurs générées

Nous avons rencontré un problème purement algorithmique à propos de la génération aléatoire. En effet, notre objectif était de générer des nombre le plus aléatoirement possible, tout en contrôlant l'hyperperiode qui découle de ces nombres. Or, ces deux objectifs ne nous paraissaient pas totalement compatibles.

Nous avons implémenté deux solutions. La première, celle qui est présenté précédemment, permet de contrôler l'hyperperiode. Le soucis de cette solution est que les P_i générés ne sont pas totalement aléatoires, puisque ce sont uniquement des diviseurs de 100.

Nous avons donc cherché à implémenté une autre solution permettant de générer des C_i et des P_i totalement aléatoire. Après réflexion algorithmique, nous obtenions des chiffres très aléatoires et qui restent cohérents. Cependant les hyperperiodes indues étaient totalement irréalistes (souvent plusieurs centaines de milliers d'unités de temps). Nous avons donc décidé de retenir la première solution, tout en ayant à l'esprit qu'elle doit être amélioré dans une version future.

Exemple de résultat obtenu avec la deuxième solution pour la génération de 5 tâches périodiques avec 75% d'utilisation processeur, sous la forme $T_i : C_i, P_i, D_i$:

T1: 7,78,78
T2: 6,19,19
T3: 17,123,123
T4: 4,83,83
T5: 12,63,63

Ce jeu de tâche nous donne une hyperpériode de 105 908 166 unités de temps, ce qui est absolument disproportionné.

6.2 Améliorations possibles

6.2.1 Fonction de génération aléatoire

Comme nous l'avons vu dans la partie 6.1.1, nous avons rencontré un problème pour développer une fonction de génération aléatoire qui soit optimale. Une amélioration possible pour la suite serait de trouver une solution algorithmique permettant de combiner les avantages des deux méthodes :

un résultat vraiment aléatoire, et une hyperpériode *contrôlée*, c'est-à-dire raisonnable. Nous pensons qu'une hyperpériode raisonnable se définit approximativement entre 0 et 1000 au vu de l'objectif de l'application.

6.2.2 Factoriser les éléments redondants des algorithmes d'ordonnancement

Nos fonctions d'ordonnancement ont un défaut : elles contiennent bien plus que l'algorithme d'ordonnancement qui leur correspond. Par exemple, dans la fonction `RM()`, il y a tout un tas d'initialisation et de calculs avant d'arriver au coeur de l'algorithme. De plus, les éléments algorithmiques extérieurs à l'ordonnancement sont similaires pour chacune des deux fonctions. Il serait donc intéressant de factoriser ces éléments en créant par exemple une fonction `init_context()` qui s'occupe d'initialiser toutes les variables et structures représentant le contexte d'exécution, une fonction `maj_context()` pour mettre à jour ce contexte, ainsi qu'une fonction `need_to_poll()`.

Pour des raisons de lisibilité, nous avons d'ailleurs choisi de représenter ce découpage dans la description des algorithmes d'ordonnancement (partie 3.3.4).

6.2.3 Ajout d'algorithmes

Lorsque nous étions à mi-chemin dans l'avancement du projet, nous avions l'ambition d'ajouter quelques autres algorithmes d'ordonnancement afin d'enrichir notre application. Malheureusement, nous n'avons pas eu le temps d'aller plus loin. Nous proposons donc l'ajout de ces nouveaux algorithmes à la liste des améliorations possibles.