Université de Nantes
UFR des Sciences et Techniques

TP Système Temps Réel Embarqué

Robin BONCORPS
Guillaume CHARON
Jérôme PAGES

Nantes, le 6 novembre 2012

Sommaire

1 Générateur de tâches

1.1 Spécification de l'environnement

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.

1.2 Spécification du logiciel

1.2.1 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.
- Les langages C/C++ sont très utilisés dans le monde de la programmation système (par exemple le noyau Linux) ou de l'embarqué (comme le système d'exploitation Trampoline par exemple). En effet, ceux-ci étant bas niveau, il permet de gérer de manière assez précise la mémoire (pointeurs, allocations, etc.).

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

1.2.2 Fonctionnalités

A mettre : un petit schéma comme on a fait au premier TP qui montre le lien entre les différents modules

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.

Génération aléatoire de tâches

TODO

Génération contrôlée de tâches

Le deuxième type de génération fait appel à l'utilisateur pour rensigner 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 Ci, Pi et Di, 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 ri (la date de réveil) et Ci.

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.

1.2.3 Décomposition fonctionnelle

Une classe par type de génération

1.3 Conception fonctionnelle

1.3.1 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 controlée.

Algo :

```
Entrée : Int choix
Sortie :  -

Si (choix == 1 ) Alors
   Lancer la génération controlée
Sinon
```

```
   Si (choix == 2) Alors
  Lancer la génération aléatoire
   Sinon
  Si (choix == 3) Alors
     Quitter le programme
  Sinon
     Afficher erreur de choix
  FinSi
   FinSi
FinSi
```

1.3.2 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 (Ci,Pi,Di) d'un nombre de tâches (determiné par l'utilisateur) en accord avec un facteur d'utilisation du processeur (déterminé lui aussi par l'utilisateur). Ex : pour trois taches et facteur = 75%

Tirage aléatoire de trois nombres (35, 25, 15) dont la somme vaut 75.
Ces nombres représentent le rapport Ci/Pi dans la formule
Il faut ensuite donner une valeur à Ci et Pi. Pour cela, on trouve le pgcd du nombre et de 100. Pour C1/P1 = 35, on obtient 5. on affecte à C1 $\leftarrow$ 35/5 et à P1 $\leftarrow$ 100/5 , on obtient donc C1 = 7 et P1 = 20. On considère, dans ce programme que Pi = Di.

On obtient donc dans cette exemple :
T1(7,20,20)
T2(1,4,4)
T3(3,20,20)

Toutes les données sont stockées dans 3 tableaux (un pour les Ci, un pour Pi, un pour Di).

Algo :

```
Entrée: Int nbTaches, Int factUtProcesseur
Sortie : 3 Tableaux d'Int (pour les valeurs de Ci, Pi, Di).
indice du tableau = numéro 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 tiré aléatoirement :
maximum = Up - somme des Ci/Pi précédents - nombre de taches restantes
Ensuite, on tire au hasard une valeur allant de 1 à 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 résumer, on obtient ici les Ci/Pi : 32, 18 et 25
*/
```

```
Pour i allant de 0 à (nbTaches - 1)
   nbMaxLimite = nbMax - nbTachesRestantes
   // le nombre aléatoire correspond à (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)
   tabCi[i] = 100 / pgcd(nb\_genere,100)

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

// Pour la dernière tâche, on ne génère 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)
tabCi[ nbTaches - 1 ] = 100 / pgcd(nb\_genere,100)
```

1.3.3 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 (Ci, Pi, Di).

Algo :

```
Entrée : Int nbTaches
Sortie :  3 Tableaux d'Int (pour les valeurs de Ci, Pi, Di).
indice du tableau = numéro de la tâche - 1 .


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

Pour i allant de 0 à (nbTaches - 1)
// on demande à l'utilisateur de renseigner les différentes valeurs ...
lire(Ci)
lire(Pi)
```

```
lire(Di)
// ... et on les insére dans les tableaux respectifs
tabCi[ i ] = Ci
tabPi[ i ] = Pi
tabDi[ i ] = Di
fin pour
```

1.3.4 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 : TODO

1.4 Définition de la réalisation

TODO

1.5 Réalisation

1.5.1 Répartition des tâches

1.6 Tests

2 Simulateur d'ordonnanceur

2.1 Spécification de l'environnement

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 [1] 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 succint présentant les différentes fonctionnalités qui s'offrent à l'utilisateur.

2.2 Specification du logiciel

Pour les mêmes raisons que celles décrites dans le Chapitre 1.2.1, le langage utilisé est le C++.

2.2.1 Fonctionnalité

1. TODO présenter kiwi vite fait

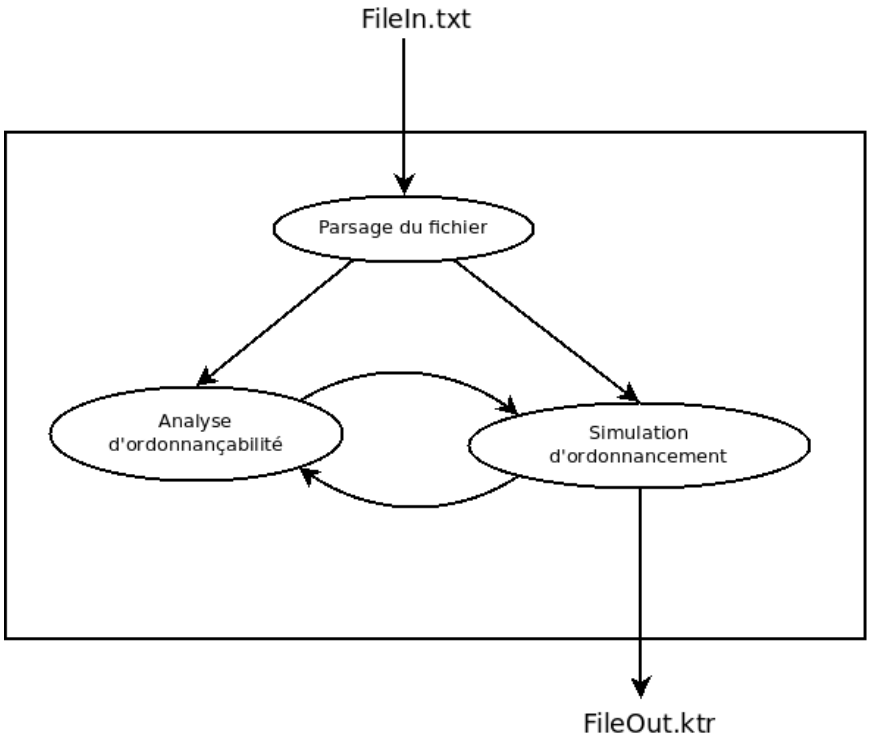

FIGURE 2.1 – Diagramme des fonctionnalités

Parsage du fichier d'entrée

Vérification de la syntaxe avec expression régulière + enregistrement des tâches et de leurs paramètres dans un modèle.

Analyse d'ordonnançabilité

On fait tous les calculs en même temps :
- Condition nécessaire pour l'ordonnançabilité du jeu de tâche avec RM ;
- Condition suffisante avec RM ;
- Condition suffisante et nécessaire avec EDF.

Simulation d'ordonnancement

2.2.2 Decomposition fonctionnelle

2.3 Conception fonctionnelle

2.3.1 Parsage du fichier d'entrée

post-condition :

Objectif :

Algo :

```
ALGO TODO
```

2.3.2 Analyse d'ordonnançabilité

Condition nécessaire RM

pré-condition :

post-condition :

Objectif :

Algo :

```
ALGO TODO
```

Condition suffisante RM

post-condition :

Objectif :

Algo :

```
ALGO TODO
```

Condition nécessaire et suffisante EDF

post-condition :

Objectif :

Algo :

```
ALGO TODO
```

2.3.3 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 périodiques dans le but ensuite de les "patcher" avec nos algorithmes d'ordonnancement des tâches apériodiques (BG et TBS).

Rate Monotonic

pré-condition :

post-condition :

Objectif :

Algo :

```
Entrée :
Sortie :


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

tant que (t < getHyperPeriode())

    Booleen need_to_poll = poll_needed()

    si(need_to_poll)

        TODO : ALGO RM

    fsi

    maj_context()
    t++

fin tant que
```

Earliest Deadline First

pré-condition :

post-condition :

Objectif :

Algo :

```
Entrée :
Sortie :


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

tant que (t < getHyperPeriode())

Booleen need_to_poll = poll_needed()
```

```
si(need_to_poll)

TODO : ALGO EDF

fsi

maj_context()
t++

fin tant que
```

2.4 Définition de la réalisation

2.5 Réalisation

2.6 Tests