SIMULATION D'ALGORITHMES D'ORDONNANCEMENT TEMPS REEL

Années : 2005-2006

Encadrant Maître de conférence:
Frank SINGHOFF

Etudiante en Master1 IL :
Stéphanie EDZANG

Plan

Introduction

Dans le but de compléter la formation en ***Master 1 d'Informatique Logiciel***, un projet de recherche et d'étude (TER) doit être réalisé par les étudiants. C'est dans cette optique que j'ai été amenée à choisir le sujet suivant : ***Simulation d'algorithmes d'ordonnancement temps réel,*** sous la direction de Monsieur Frank SINGHOFF maître de conférence au département informatique.

Il s'agit de faire le point sur les fonctionnalités qu'offrent le logiciel ***Cheddar***, en d'autres termes de faire le tour des différents algorithmes d'ordonnancement que l'on peut traiter avec le dit logiciel ; et d'y ajouter la détection d'interblocage.

Pour réaliser ce travail, il était nécessaire d'apprendre le langage ***Ada*** et de se familiariser avec le logiciel ***Cheddar***. Ce rapport s'articule donc sur trois points à savoir :

- L'apprentissage d'***Ada***
- Le logiciel Cheddar
- Ajout de détection d'interblocage.

I-Apprentissage d'Ada

I.1 Historique

Le département de la défense des Etats-Unis ou encore ***DoD,*** prenant conscience des sommes excessives, dépensées en maintenance pour les applications destinées à des systèmes embarqués, lance un appel au développement d'un cahier des charges qui permette de définir un langage robuste et fiable. C'est ainsi que ***Ada*** voit officiellement jour en 1979.

C'est un langage de programmation orienté objet conçu par une équipe de chercheur dirigé par un français nommé Jean Ichbiah.

Une première version standardisée par l'***ANSI,*** voit le jour en 1983 : ***Ada83***. Cette dernière est alors normalisée en 1987 par L'***ISO*** qui est norme internationale.

Ada95 est une version plus ressente et beaucoup plus complète que ***Ada83***. Cette dernière version est utilisée, de nos jours, dans divers domaines tels que l'automobile, les transports ferroviaires, les technologies aéronautique et les technologies spatiales, d'où son utilité dans les systèmes temps réels.

NB : Dans un système embarqué, le système informatique est intégré à une machine (missile, automobile, lave-linge...).

I.2 Langage

Ada est un langage modulaire, fiable car il est orienté objet, multitâche et bien sûr normalisé. C'est un langage portable sur ***Win32***, ***Linux*** et ***Uni***x. Il est compilé, la détection des erreurs est faite à par le compilateur, ce qui réduit considérablement le débogage.

Type

Ada est un langage fortement typé. En effet, le typage est explicite et très strict :

- Si des objets sont d'un type donné, il n'existe pas de moyen de faire un changement ou un forçage de type.
- Toute opération sur un objet doit appartenir à l'ensemble des opérations connues pour les objets de ce type.
- Un objet possède une valeur et une seule, celle-ci doit appartenir à l'ensemble des valeurs définies par son type.

Exemple :

```
type T_TEMPERATURE is range -78..123;
type T_ETAGE is range -5..12;

A : T_TEMPERATURE ;
B, C : T_ETAGE ;
```

Il est impossible de faire :

A := B + C ; ⇒ génère une erreur de compilation

A et B,C ne sont pas de même type, d'où l'erreur.

Tout comme en ***Java***, ***Ada*** a des types prédéfinis avec lesquels on peut définir d'autres types ou sous types. Ces types prédéfinis sont :

- ***Integer*** avec ses variantes : ***Natural*** (pour zéro ou un nombre entier positif), ***Positive*** (pour des entiers positifs), ***Short_Integer*** (pour entier de 16 bit), ***Short_Short_Integer*** (pour entier de 8 bit), ***Long_Interger*** (pour entier de 32 bit), ***Long_Long_Interger*** (pour entier de 64 bit).
- ***Character*** pour la déclaration d'un caractère.
- ***String*** pour les chaînes de caractères.
- ***Float*** avec ses variantes : ***Short_Float, Long_Float, Long_Long_Float, Fixed***, tous étant des réels plus ou moins de grande taille.
- ***Boolean*** de valeur ***True*** ou ***False***.

Structures de base

- Chaque instruction est terminée par ″ ; ″.
- L'affectation se fait avec ″ := ″.
- Les opérateurs de comparaison sont : égalité ″ = ″ ; différence ″ /= ″ ; inférieure ″ <″; supérieure ″ >″; inférieure ou égale ″ <= ″; supérieur ou égale ″ >= ″; appartenance à un intervalle ″ in ″; non appartenance à un intervalle ″ not in ″.

Les structures conditionnelles sont les mêmes : ***if, elsif, et case.***
Quant aux structures itératives on a : ***for, while***. A cette liste s'ajoute une nouvelle boucle : ***loop***. Cette dernière remplit les mêmes fonctions que ***for*** et ***while***, elle s'exécute indéfiniment jusqu'à ce que l'instruction ***exit*** apparaisse.

Les fonctions et les procédures

En ***Ada***, il existe des procédures et des fonctions qui ont des rôles bien différents :

- Les fonctions retournent un résultat et leurs arguments ne sont accessibles qu'en lecture uniquement et ne sont en aucun cas modifiable.
- Les procédures ne retournent jamais de résultats par contre, pour chaque argument, il faut préciser la modalité de passage. C'est-à-dire qu'il faut préciser si l'argument est accessible en lecture seule (in), en écriture uniquement (out) où le mode le plus souple qui autorise de prendre en compte et de modifier la valeur du paramètre (in out).

Structure d'une fonction et d'une procédure

```
Function <nom_function> (arg1:type1 ; arg2:type2.. ) return type_resultat is
  partie déclarative ;
begin
  instructions ;
        return résultat ;
end <nom_function  ;

Procedure <nom_procedure > (arg1: mode1 type1 ; arg2: mode2 type2.. ) is
        partie déclarative ;
begin
        instructions ;
end <nom_procedure > ;
```

NB : Dans la partie déclarative d'une procédure, on peut trouver d'autres procédures et des fonctions.

Organisation générale d'un programme *Ada*

Un programme en ***Ada*** s'articule autour d'un programme principal qui utilise le plus souvent des paquetages.

Un paquetage correspond à des unités cohérentes telles que des regroupements par fonctionnalités (entrées/sorties, bibliothèques mathématiques…) ou encore des types abstraits (nombres rationnels, listes…) et bien d'autres encore.

Structures d'un programme principal

```
With nom_paquetage1 ;
Use nom_paquetage1 ;

        Procedure <nom_procedure> is
                partie déclarative ;
        begin
                corps de la procedure ;
        end <nom_procedure> ;
```

Il est possible de créer soi même un paquetage qui se partage entre deux fichiers : un fichier dont l'extension est ***.ads*** et l'autre ***.adb***. Par comparaison au langage C :

- Le fichier ***.ads*** serait comme le fichier ***.h*** où l'on trouve les spécifications dont des types, les procédures , les fonctions proposés par le paquetage.
- Le fichier ***.adb*** serait comme le fichier ***.c*** où l'on trouve l'implémentation du paquetage et éventuellement des procédures ou fonctions cachées à l'utilisateur.

Généricité

La généricité d'une fonction repose sur son indépendance vis-à-vis du type, et éventuellement du nombre de ses arguments. C'est un concept important pour un langage de haut niveau car il permet d'augmenter le niveau d'abstraction du langage.

En effet, en ***Ada*** il est possible d'utiliser la généricité. Cela consiste à créer des objets de types indéterminés avec lesquels on peut effectuer les opérations spécifiées.

Exemple :

```
…..
generic
  type Element is private;

package Elements is
        procedure add (a, b: In Out Elements );
end Elements;
```

Si l'on veux faire une addition d'entier ou de réel ou autre voici comment on procède:

- a is a new Elements(Integer);

- b is a new Elements(Integer);
- c is a new Elements(Integer);
  c := add(a,b);
- a is a new Elements(Float);
- b is a new Elements(Float);
- c is a new Elements(Float);
  c := add(a,b);

La commande, pour compiler un programme en ***Ada,*** est :

gnatmake –g <nom programme>

le nom du programme correspond au nom du fichier où la procédure principale est implémentée, son extension est ***.adb*** comme vu précédemment.

II-Cheddar

Cheddar est un logiciel conçu dans un but pédagogique à l'Université de Bretagne Occidentale, par l'équipe ***Lisyc***. Il est développé en ***Ada9***5 et en ***GtkAd***a et devrait fonctionner sur toutes les plates-formes supportées par ***GNAT***.

C'est un outil de simulation qui permet d'effectuer un test de faisabilité sur un jeu de tâches donné pour un algorithme d'ordonnancement choisi et dans le cas échéant de représenter graphiquement l'ordonnancement des dites tâches, tout ceci pour les systèmes temps réels.

NB : Un test de faisabilité ou encore test d'ordonnançabilité indique par un calcul si les échéances des tâches peuvent être respectées par une politique et ce sans avoir besoin d'exécuter l'algorithme.

II-1 Systèmes temps réels

Les systèmes informatiques temps réels sont aujourd'hui présents dans de nombreux secteurs d'activités comme :

- L'industrie de production au travers des systèmes de contrôle de procédé.
- L'industrie du transport au travers des systèmes de pilotage embarqués
- Les salles de marché au travers du traitement des données boursières en temps réel.
- La nouvelle économie au travers du besoin toujours croissant du traitement et l'acheminement de l'information (vidéo, pilotage à distance, réalité virtuelle…).

Ce qui est primordiale de retenir en systèmes informatiques temps réel, c'est la notion de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. En effet, les dits systèmes doivent délivrer des résultats exacts dans les délais imposés.
Pour garantir le respect des limites ou contraintes temporelles, il est nécessaire que :

- Les différents services et algorithmes utilisés s'exécutent en temps borné.
- Les différents enchaînements possibles des traitements garantissent que chacun des services et algorithmes ne dépassent pas leurs limites temporelles. Pour cela, il est nécessaire de faire appel à la théorie de l'ordonnancement.

L'ordonnancement consiste à organiser dans le temps la réalisation de tâches, compte tenu de contraintes temporelles (délais, contraintes d'enchaînement) et de contraintes portant sur la disponibilité des ressources requises.

Il existe plusieurs algorithmes d'ordonnancement dont :

- ***Rate Monotonic*** ou encore ***RM*** : C'est un algorithme à priorité fixe, utilisé uniquement pour les tâches périodiques. La tâche qui a la priorité la plus forte, est élue pour être exécutée. La priorité est l'inverse de la périodicité : la tâche ayant la période la plus petite, dans un jeu de tâches, est la plus prioritaire. C'est un algorithme optimal dans la classe des algorithmes à priorité fixe, facile à implanter dans un système d'exploitation. Les tâches sont ordonnançables en mode préemptif ou non préemptif.

- ***Earliest Deadline First*** ou encore ***EDF*** : C'est un algorithme à priorité dynamique, pour les tâches périodiques et apériodiques. Comme ***RM***, les tâches sont ordonnançeables en mode préemptif et non préemptif. C'est également un algorithme optimal, difficile à implanter dans un système d'exploitation. Il est instable en surcharge et est moins déterministe que ***RM***. Pour un jeu de tâches donné, on calcule l'échéance pour chacune d'elle et celle ayant la plus courte échéance est élue.

- ***Deadline Monotonic*** ou encore ***DM*** : c'est un algorithme à priorité statique. La priorité d'une tâche est inversement proportionnelle à son échéance. Cet algorithme équivaut à R M quand l'échéance est égale à la période (échéance sur requête).

.

- ***Least Laxity First*** ou encore ***LLF*** : C'est un algorithme basé sur des priorités dynamiques. La priorité d'une tâche est inversement proportionnelle à sa laxité

dynamique. Autrement dit, la tâche de plus haute priorité à l'instant ***t*** dispose de la plus petite laxité.

- ***Ordonnancement POSIX 1003b*** : C'est une extension temps réel du standard ***ISO / ANSI POSIX*** définissant une interface portable de systèmes d'exploitation. C'est un algorithme à priorité fixe, s'exécute uniquement en mode préemptif (***RM*** facile). Les tâches sont élue grâce à une file d'attente par priorité et une politique de gestion de la file parmi: ***SCHED_FIFO***, ***SCHED_RR*** et ***SCHED_OTHERS***.

SCHED_FIFO, la tâche quitte la tête de file :

- Si elle est terminée.
- Si elle est bloquée, dans ce cas, il y a un délai à respecter avant d'être remise en queue.
- Et enfin s'il y a une libération explicite, on la remet en queue.

SCHED_RR remplit les mêmes conditions que ***SCHED_FIFO***, mais en plus, la tâche en tête de file est placée en queue après expiration d'un quantum.

SCHED_OTHERS, pour un fonctionnement non normalisé.

Tous les algorithmes précédemment cités, sont implantés dans le logiciel Cheddar.

Quelques définitions :

Mode préemptif : une tâche plus prioritaire peut interrompre une autre moins prioritaire en cours qui est d'exécution.

Mode non préemptif : aucune interruption n'est admise, quand une tâche est en cours d'exécution et qu'une autre plus prioritaire arrive, elle doit attendre que celle en cours se termine.

II-2 Fonctionnalités actuelles de Cheddar

Cheddar comporte deux composants logiciels :

- Une interface qui permet à l'utilisateur d'enregistrer le jeu de tâches qu'il souhaite faire analyser. Il peut ainsi, choisir un algorithme d'ordonnancement, un protocole, lancer ou pas la représentation graphique des tâches. Les résultats des simulations seront présentés sur la même interface.

- Une bibliothèque comportant les principaux résultats de la théorie de l'ordonnancement temps réel ainsi que quelques outils de files d'attente.

Comme précédemment dit, ***RM***, ***EDF***, ***DM***, ***LLF*** et ***POSIX 1003b*** sont des algorithmes d'ordonnancement que ***Cheddar*** propose.

Lors d'une simulation où des taches utilisent des ressources, il est possible d'obtenir les informations suivantes :

- Les temps de blocage sur ressources partagées (bornes maximales, minimales, délais moyens).

- Les temps de réponse des tâches (bornes maximales, minimales, délais moyens).

- Nombre de commutations de contexte, de préemption.

- Recherche d'interblocage, d'inversion de priorité.

- Echéances manquées.

Cheddar offre la possibilité d'appliquer des tests de faisabilité dans le cas préemptif ou non préemptif :

- Borne sur les temps de réponse (pour EDF, LL, DM et RM).

- Borne sur les temps de blocage sur ressources partagées (avec PCP et PIP),

- Test sur le taux d'utilisation processeur (avec EDF, LLF, RM et DM).

Il y a également l'expression et ordonnancement de tâches avec contraintes de précédence :

- Temps de réponse de bout en bout (test de faisabilité).

- Modification de paramètres de tâches (test de faisabilité et simulation).

- Ordonnancement de tâches avec graphe de précédences pour système multi-processeurs.

Cheddar propose aussi :

- Outils de partitionnement pour système multi-processeurs.

- Outils pour analyser l'utilisation de tampons :

- Test de faisabilité pour calculer une borne sur la taille des tampons.
- Outils de simulation pour analyser le taux d'occupation des tampons à partir d'une simulation.
- Ordonnancement de messages.

Exemple :

Soient deux tâches : T1, T2 et deux ressources : R1, R2 telles que :

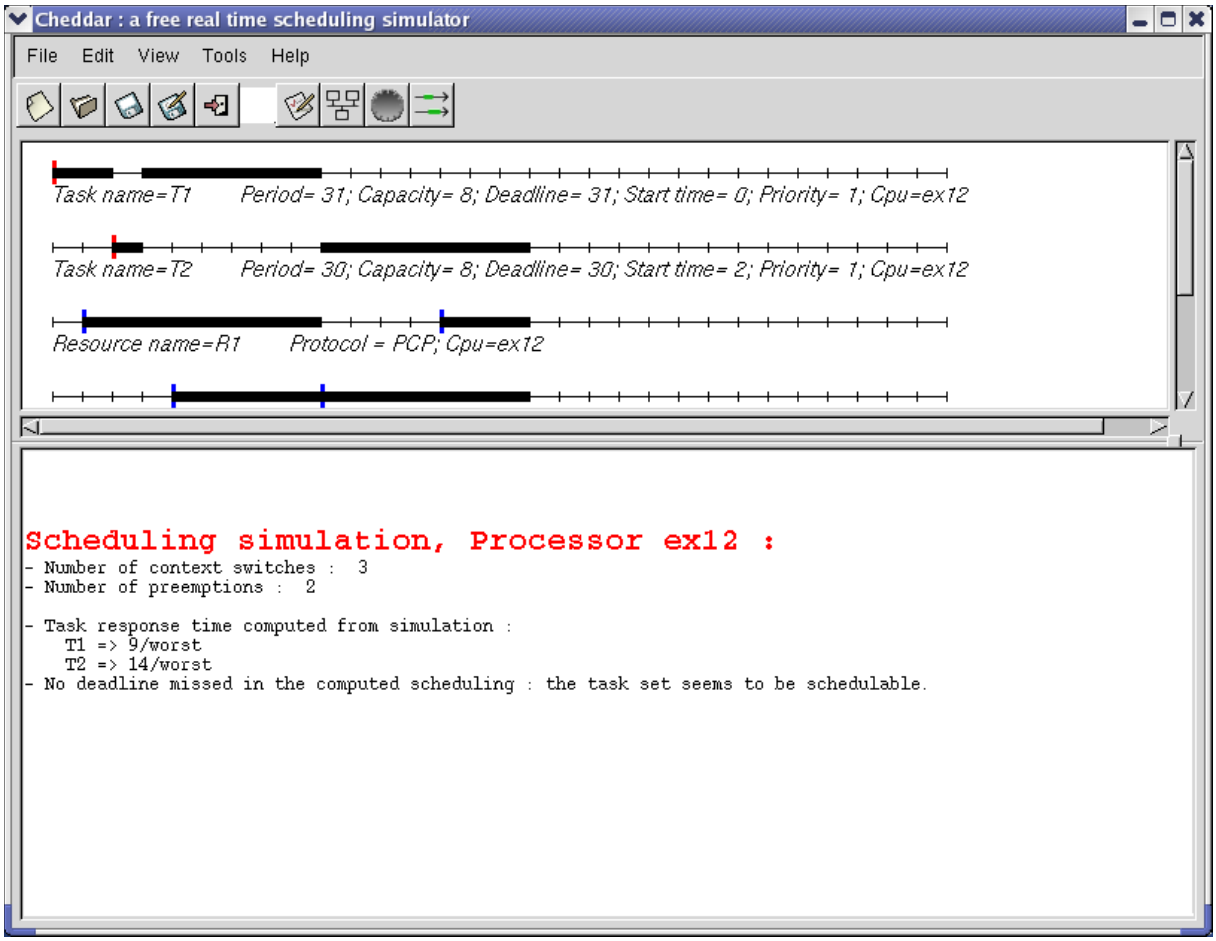
	T1	T2
Si	0	2
Pi	31	30
Ci	8	8

- Si : Instant d'arrivée de la tâche i.
- Pi : Période de la tâche i.
- Ci : capacité de la tâche i.
- i allant de 1 à 2.

- R1 peut être allouées à T1 à partir de sa 2ème unité de temps jusqu'à la fin de sa capacité.
- R2 peut être allouées à T1 à partir de sa 4ème unité de temps jusqu'à la fin de sa capacité.
- R1 peut être allouées à T2 à partir de sa 6ème unité de temps jusqu'à la fin de sa capacité.
- R2 peut être allouées à T2 à partir de sa 2ème unité de temps jusqu'à la fin de sa capacité.

NB : Nous sommes en exclusion mutuelle.
- Mode préemptif.
- Protocole PCP.

Voir simulation sur ***Cheddar*** page suivante.


Cheddar : a free real time scheduling simulator
File Edit View Tools Help
Task name=T1 Period= 31; Capacity= 8; Deadline= 31; Start time= 0; Priority= 1; Cpu=ex12
Task name=T2 Period= 30; Capacity= 8; Deadline= 30; Start time= 2; Priority= 1; Cpu=ex12
Resource name=R1 Protocol = PCP; Cpu=ex12
Scheduling simulation, Processor ex12 :
- Number of context switches : 3
- Number of preemptions : 2
- Task response time computed from simulation :
T1 => 9/worst
T2 => 14/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

III- Ajout de détection d'interblocage

Lors d'une simulation, ***Cheddar*** propose deux protocoles pour le partage des ressources :

- ***Priority Inhéritance Protocol*** ou encore ***PIP*** : une tâche qui bloque une autre plus prioritaire qu'elle, exécute la section critique (portion de temps où la tâche utilise la ressource) avec la priorité de la tâche bloquée. Ce protocole est utilisé pour le partage de ressources en exclusion mutuelle. C'est-à-dire qu'une ressource donnée ne peut être utilisée que par une unique tâche en même temps (deux tâches ne peuvent accéder en même temps à une ressource). Il y a possibilité d'avoir un interblocage.

- ***Priority Ceiling Protocol*** ou encore ***PCP*** : une variable (plafond) stocke le plus haut niveau de priorité de toutes les sections critiques actuellement acquises. Ainsi, une tâche accède à une ressource et par la même occasion bloque les autres tâches, si sa priorité est inférieure ou égale au plafond. Avec ce protocole, plusieurs tâches peuvent accéder à une ressource en même temps. Ceci selon le nombre de tâches autorisées à utiliser la ressource en concurrence. Il n'y a aucune possibilité d'être en présence d'interblocage.

Il se trouve, lors d'une simulation, en choisissant le protocole ***PIP***, que les interblocages sont visibles sur le graphe, mais qu'ils ne sont pas détectés par ***Cheddar***.

En effet, lorsqu'on demande le rapport de recherche d'interblocages, on ne trouve pas du tout de trace sur ces derniers. Il faut préciser ici que pour la suite, je ne tiendrai compte du partage de ressources qu'en exclusion mutuelle.

Il fallait donc ajouter le code permettant la détection des interblocages au logiciel ***Cheddar***. Pour ce faire, il y a deux étapes importantes à réaliser avant de passer au codage de cette fonction :

- Savoir reconnaître une situation d'interblocage.

- Chercher une méthode de détection reconnue et le plus souvent utilisée.

III-1 Situation d'interblocage

Pour provoquer un interblocage les conditions suivantes doivent être respectées :

- ***Condition d'exclusion mutuelle*** : chaque ressource est soit attribuée à une tâche, soit disponible.
- ***Condition de détention et d'attente*** : les tâches ayant déjà obtenues des ressources peuvent en demander de nouvelles.
- ***Pas de réquisition*** : les ressources déjà détenues ne peuvent être retirées de force à une tâche. Elles doivent être explicitement libérées par les tâches qui les détiennent.
- ***Condition d'attente circulaire*** : il doit y avoir un cycle d'au moins deux tâches, chacune attendant une ressource détenue par l'autre.

Pour résumer : un ensemble de tâches est en interblocage si chaque tâche attend une ressource que seule une autre tâche de l'ensemble détient. Comme toutes les tâches attendent, aucune d'elles ne peut jamais relâcher la ressource qu'elle détient, par conséquent toutes les tâches attendent indéfiniment.

Pour modéliser une situation d'interblocage, les réseaux de Pétri ou encore RdP sont d'une aide précieuse.

En effet, les RdP sont un outil graphique et mathématique permettant la conception, la spécification, la simulation et la vérification de systèmes.

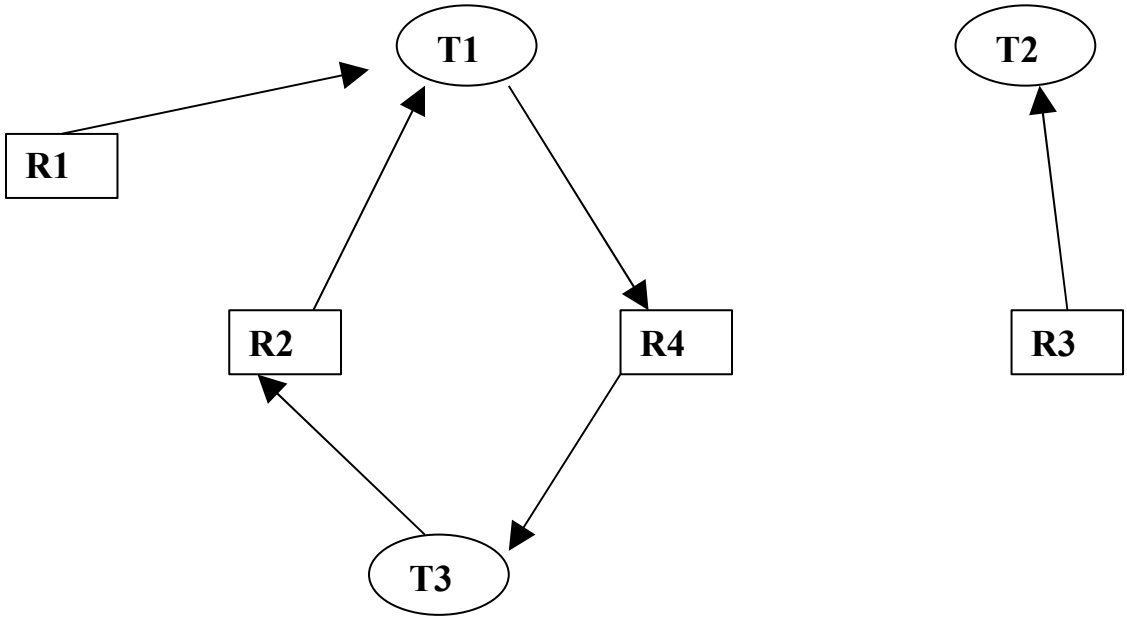
L'exemple suivant modélise un interblocage.

Soient quatre ressources R1, R2, R3, R4 et trois tâches T1, T2,T3 telles que :

- R1 et R2 sont allouées à T1. Ce dernier est en attente de R4.
- R 3 est allouée T2.
- R4 est allouée à T3. Ce dernier attend R2

Pour le RdP qui va suivre :

- Un cercle représente une tâche.
- Un carré représente une ressource.
- Un arc allant d'un cercle à un carré signifie que la tâche est bloquée en attente de la ressource que pointe l'arc.
- Un arc allant d'un carré à un cercle signifie que la ressource est attribuée à la tâche que pointe l'arc.

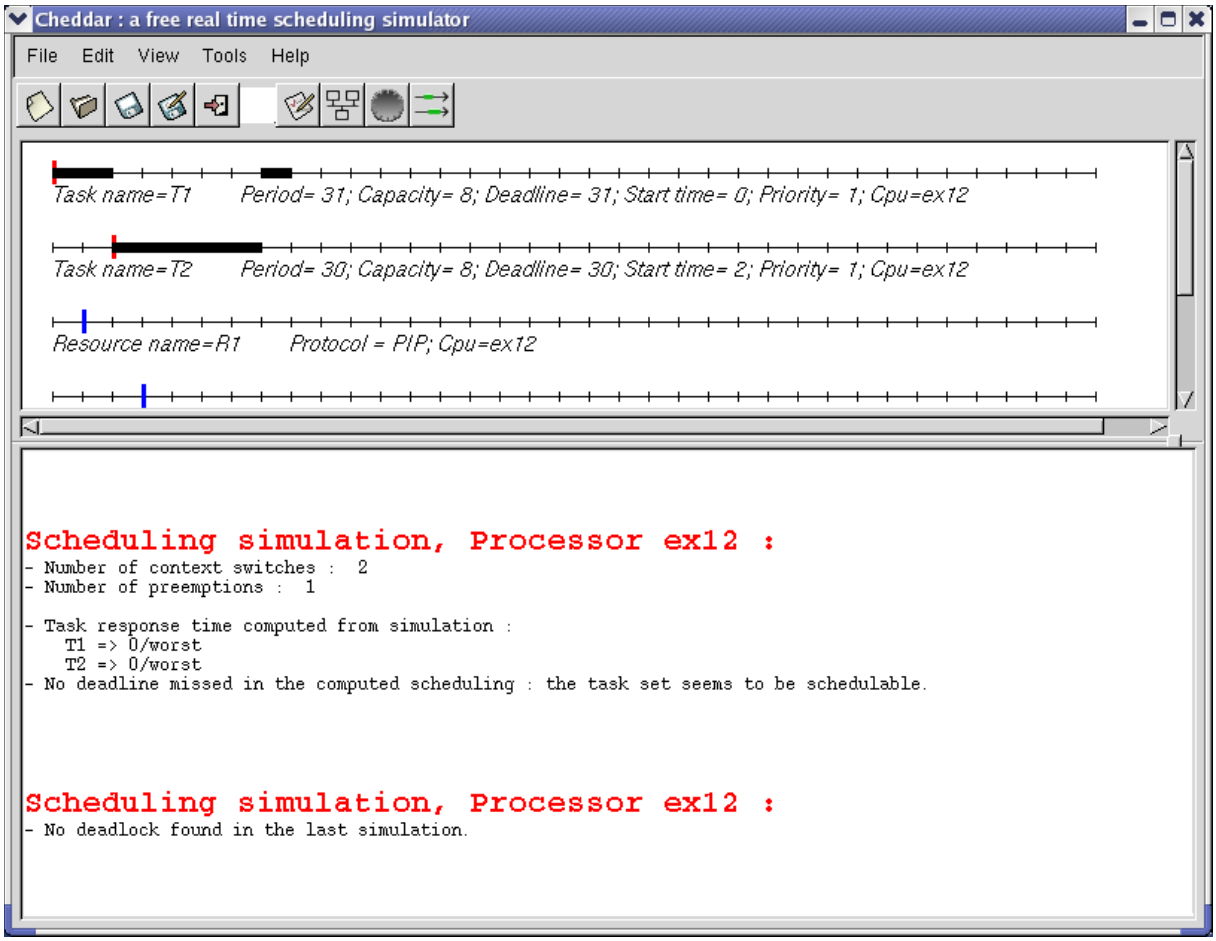


Dans ce schéma, on se retrouve avec un cycle entre T1, T3, R2 et R4, il n'y a aucun moyen de sortir de ce cycle : on est en présence d'un interblocage.

Bien qu'il soit relativement aisé de repérer visuellement, à partir d'un graphe, une situation d'interblocage, ***Cheddar*** nécessite un algorithme de détection des interblocages pour pouvoir générer un rapport.

Pour le même exemple qu'en page 12, mais avec le protocole PIP, voici le résultat de la simulation (page suivante).

On remarque bien sue le schéma, qu'il n'y a pas d'évolution dans le temps des tâches et même des ressources : ce qui démontre bien qu'il y a un interblocage.
Pourtant le rapport dit qu'il n'y a aucun interblocage.


Cheddar : a free real time scheduling simulator
File Edit View Tools Help
Task name=T1 Period= 31; Capacity= 8; Deadline= 31; Start time= 0; Priority= 1; Cpu=ex12
Task name=T2 Period= 30; Capacity= 8; Deadline= 30; Start time= 2; Priority= 1; Cpu=ex12
Resource name=R1 Protocol = PIP; Cpu=ex12
Scheduling simulation, Processor ex12 :
- Number of context switches : 2
- Number of preemptions : 1
- Task response time computed from simulation :
T1 => 0/worst
T2 => 0/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.
Scheduling simulation, Processor ex12 :
- No deadlock found in the last simulation.

III-2 Méthodes de détection d'interblocage

=

Parmi les différentes méthodes de détection d'interblocage, il y a une qui revient le plus souvent en ce qui concerne le partage de ressources en exclusion mutuelle : c'est l'algorithme réparti de détection d'interblocage.

Voici comment fonctionne cet algorithme :

1. Pour chaque nœud, N du graphe, suivre les cinq étapes suivantes :
2. Initialiser une liste L vide et désigner tous les arcs comme non marqués.
3. Ajoutez le nœud en cours à la fin de la liste L et vérifier si le nœud apparaît deux fois dans L. Si tel est le cas, le graphe comporte un cycle et l'algorithme prend fin. Sinon continuer l'algorithme.
4. Pour un nœud donné, voir s'il y a des arcs marqués sortants. Si tel est le cas rendez-vous à l'étape 5, dans ce cas contraire se rendre à l'étape 6.
5. Choisir au hasard un arc sortant non marqué et marquez-le. Suivez-le jusqu'au prochain nouveau nœud en cours et se rendre à l'étape 3.
6. Situation d'impasse : supprimer l'arc et revenez au nœud précédent, c'est-à-dire à celui qui était actif juste avant celui-ci. Faites en le nœud en cours et reprendre l'étape 3. Si ce nœud est le nœud initial, le graphe ne contient pas de cycle et l'algorithme prends.

Le fonctionnement de l'algorithme réparti précédent, n'est qu'une explication générale, il faut pouvoir l'adapter au langage et à la situation présente.

Après l'étude du code se trouvant dans le répertoire ***framework*** plus précisément du package ***scheduling_sequence.adb*** de ***Cheddar,*** il est possible de connaître à chaque unité de temps quelles sont les ressources allouées, attendues, relâchées et quelles sont les tâches auxquelles ces ressources sont rattachées.

Partant de ce fait, l'algorithme mis en place respecte les étapes suivantes :

1. Déclarer deux tableaux (ou matrices), à deux dimensions, d'entier :
   - Le nombre de ligne correspond au nombre total de tâches de la simulation.
   - Le nombre de colonne correspond au nombre total de ressources de la simulation.
   - Un tableau pour les ressources allouées : ***Matrice_Alloue***.
   - Un tableau pour les ressources attendues : ***Matrice_Attendu***.
   - Les valeurs contenues dans ces tableaux ne peuvent être que 0 ou 1.
   - Initialiser les tableaux à 0.

2. Déclarer une variable pour l'analyse des matrices, de type entier : Analyse_Matrice.

3. Déclarer une liste L pour stocker les ressources et les tâches impliquées dans l'interblocage.

4. Pour une unité de temps donnée, remplir les matrices :

   - Si l'événement en cours correspond à une ressource allouée alors affecter la valeur 1 à ***Matrice_Alloue*** : à la ligne correspondant à la tâche qui détient la ressource et à la colonne correspondant à la ressource détenue.

   - Si l'événement en cours correspond à une ressource attendue alors affecter la valeur 1 à ***Matrice_Attendu*** : à la ligne correspondant à la tâche qui demande la ressource et à la colonne correspondant à la ressource demandée.

   - Si l'événement en cours correspond à une ressource relâchée alors affecter la valeur 0 à ***Matrice_Alloue*** : à la ligne correspondant à la tâche qui relâche la ressource et à la colonne correspondant à la ressource relâchée.

   - Analyser les deux matrices : avant tout initialiser à 0 la variable ***Analyse_Matrice*** ; Parcourir les deux matrices jusqu'à la fin de leur taille, si une même ressource est allouée, attendue et qu'en plus la tâche qui la détient et celle qui la demande sont différentes alors incrémenter de une unité ***Analyse_Matrice***.

   - Si la valeur de ***Analyse_Matrice*** est supérieure à deux, alors détection d'interblocage, insérer les ressources et les tâches concernées dans la liste L.

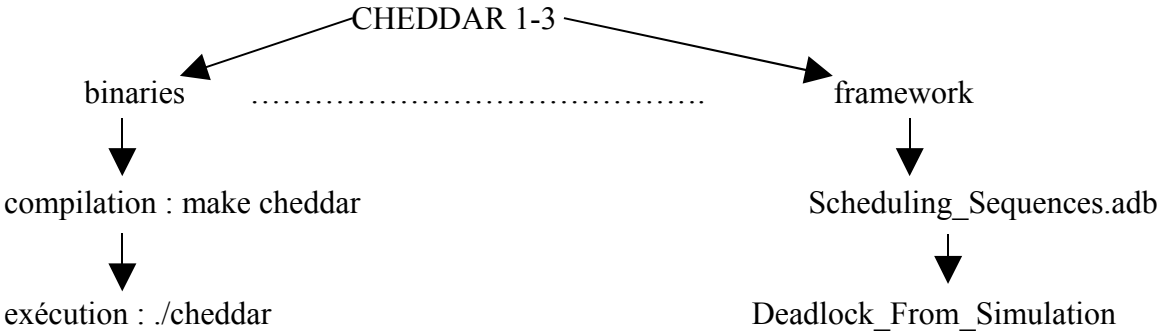
   - Recommencer l'étape 4 jusqu'au temps d'arrêts de la simulation.

5. Retoutnez L.

Cet algorithme à été implanté :

- En ***Ada95***.
- Sous le système d'exploitation ***Linux.***.

Schéma des répertoires et fichiers utilisés :



Conclusion

La fonctionnalité de détection d'interblocage n'est pas encore opérationnelle.
En effet, le code a été réalisé, mais il comporte encore des erreurs de compilation.

Il faut reconnaitre que ce n'est pas facile de coder en ***Ada***, car c'est un langage qui ne tolère aucune souplesse au niveau du typage.

Etant habituer à programmer avec des langages plus souple à ce niveau comme ***C***, ***Java*** et même ***Smalltalk*** (qui soit dit en passant ne possède pas de typage), j'ai eu tendance à ne pas tenir compte de la spécificité du langage ***Ada***.

Bibliographie

Langage Ada et Algorithmique

Auteurs : **Robert Ogor**
Robert Rannou

Editeur : HERMES

Systèmes d'exploitation

Auteurs : **Endrew Tanenbaum**

Editeur : **Pearson Education**

Cheddar : a flexible Real Time Scheduling framework

Auteurs : **F. Singhoff**
J. Legrand
L. Nana
L. Marcé

Fascicule

http://beru.univ-brest.fr/~singhoff/

TP,TD et cours d'*Ada*.

Documentation *Cheddar*.

http://beru.univ-brest.fr/~singhoff/DOC/LANG/ADA/BOOK/book.html

Documentation sur *Ada.*