Université de Monastir
Institut Supérieur d'Informatique et de Mathématiques

Cours: Programmation Temps Réel et Concurrente

Filière: MP2-GL

Chapitre 5: La concurrence en Ada

Réalisé par:
Dr. Sakka Rouis Taoufik

1

I. Introduction

Rappelant que Ada est un langage de programmation fortement recommandé pour le développement des systèmes distribués.
En effet, le langage Ada offre des possibilités importantes vis-à-vis :
- De la programmation structurée : structuration de données (types prédéfinis et constructeurs de types simples et structurés) et structuration de traitements (structures de contrôle et les sous-programmes avec une distinction nette entre procédure et fonction). De plus, Ada est un langage fortement typé : il est plus sévère que Pascal!
− De la programmation modulaire en offrant un concept puissant, appelé package, doté de deux parties : interface (package) et implémentation (package body).

2

I. Introduction

− De la programmation générique en offrant la possibilité de concevoir et de réaliser des unités génériques : sous-programmes et paquetages. Ces unités peuvent être paramétrées sur de types, variables et sous-programmes.

− De la gestion des exceptions: ceci permet d'écrire des logiciels robustes dans divers domaines critiques : temps réel, systèmes embarqués.

- De la traitement de la concurrence: le langage Ada offre des constructions intéressantes permettant d'écrire des programmes concurrents fiables.

- De l'analyse statique: Ada est supporté par plusieurs outils permettant l'analyse statique d'un programme concurrent Ada tels que SPIN, SMV, INCA et FLAVERS. Sachant que l'analyse statique est une technique qui permet d'analyser un programme sans toutefois l'exécuter.

II. Les tâches

La concurrence en Ada est basée sur la synchronisation/communication des tâches.

Les tâches Ada permettent d'avoir des entités qui s'exécutent parallèlement et qui coopèrent selon les besoins pour résoudre les problèmes concurrents (ou non séquentiels).

Une tâche Ada (**task**) est une entité modulaire constituée de deux parties:

- Une spécification décrivant l'interface présentée aux autres tâches: cette partie inclus: le nom, Une partie visible avec déclarations d'entrées et Une partie privée avec déclarations d'entrées.

- Un corps décrivant son comportement dynamique.

Cette partie inclus: le nom (voir spécification), une liste de déclarations et une liste d'instructions

Chapitre 4: la concurrence en Ada

II. Les tâches

Task Interface	**task Server is** **entry take ;** **entry release ;** **end Server;**
Task Body	**task body Server is** **begin** **…** **accept take ;** **…** **accept release ;** **…** **end Server;**
Task Uses	**task Client;** **task body Client is** **begin** **…** **Server.take ;** **…** **Server.release ;** **…** **end Client;**

5

Chapitre 4: la concurrence en Ada

II. Les tâches

Exemple 1: Soit trois activités indépendantes, on peut les exécuter : en séquence ou en parallèle

```
procedure Main is
begin
    Find_Ali_Phone_Nbr;
    Find_Slim_Phone_Nbr;
    Find_Md_Phone_Nbr;
end Main;
```

```
procedure Main is
  task Ali_Phone_Nbr;
  task Slim_Phone_Nbr;
  task body Ali_Phone_Nbr is
  begin
    Find_Ali_Phone_Nbr;
  end Ali_Phone_Nbr;
  task body Slim_Phone_Nbr is
  begin
    Find_Slim_Phone_Nbr;
  end Slim_Phone_Nbr;
begin
  Find_Md_Phone_Nbr;
end Main;
```

6

II. Les tâches

Exemple 2: Soit deux activités indépendantes T1 et T2, on peut les exécuter en parallèle comme suite:

```
with Text_Io;
use Text_Io;

procedure Ord is
  task T1;
  task T2;

  task body T1 is
  begin
      loop
          Put("T1");
      end loop;
  end T1;

  task body T2 is
  begin
    loop
      Put("T2");
    end loop;
  end T2;

  begin
    null ;
  end Ord;
```


II. Les tâches

L'activation d'une tâche est automatique.

Les tâches T1 et T2 deviennent actives quand l'unité parente atteint le **begin** qui suit la déclaration des tâches.
Ainsi, ces deux tâches, dépendantes, sont activées en parallèle avec la tâche principale, qui est le programme principal.

La tâche principale attend que les tâches dépendantes s'achèvent pour terminer. La terminaison comporte donc deux étapes : elle s'achève quand elle atteint le **end** final, puis elle ne deviendra terminée que lorsque les tâches dépendantes, s'il y en a, seront terminées.

III. Les rendez-vous

Les tâches Ada peuvent interagir entre elles (se communiquer entre elles) grâce à des entrées de tâche. Ce mécanisme est appelé rendez-vous.

Une tâche publie ses entrées et leurs signatures dans sa spécification. Une entrée d'une tache correspond à une procédure appelable par une tâche appelante.

L'appelante invoque une entrée de l'appelée et se bloque en attendant que l'appelée accepte
L'appelée accepte les appels sur ses entrées et se bloque en absence de demandes
Lorsque appelante et appelée sont prêtes, l'appel (le rendez-vous) se déroule dans le contexte de l'appelée

III. Les rendez-vous

```
with Text_Io;
use Text_Io;
procedure Ecran is
task T1;
task T2;
task Allocateur is
   entry Prendre;
   entry Relacher;
end Allocateur;
task body Allocateur is
begin
  loop
    select
       accept Prendre;
       accept Relacher;
    or
        terminate;
   end select;
  end loop;
end Allocateur;
```

```
task body T1 is  begin
   Allocateur.Prendre;
   for I in 1 .. 3   loop
        Put("T1");
   end loop;
   Allocateur.Relacher;
end T1;

task body T2 is begin
   Allocateur.Prendre;
   for I in 1 .. 3 loop
      Put("T2");
   end loop ;
   Allocateur.Relacher;
end T2;

begin
    null;
end Ecran;
```

Ce programme affiche aléatoirement T1T1T1T2T2T2 ou T2T2T2T1T1T1

Chapitre 4: la concurrence en Ada

III. Les rendez-vous

```
with Text_Io;
use Text_Io;
procedure Ecran is
task T1;
task T2;
task Allocateur is
   entry Prendre;
   entry Relacher;
end Allocateur;
task body Allocateur is
begin
  loop
   select
      accept Prendre;
      accept Relacher;
   or
       terminate;
   end select;
  end loop;
end Allocateur;
```

```
task body T1 is  begin
loop
Allocateur.Prendre;
   for I in 1 .. 3   loop
        Put("T1");
   end loop;
   Allocateur.Relacher;
end loop ;
end T1;
task body T2 is begin
loop
 Allocateur.Prendre;
   for I in 1 .. 3 loop
      Put("T2");
   end loop ;
   Allocateur.Relacher;
end loop ;
end T2;
begin
    null;
end Ecran;
```

Ce programme affiche aléatoirement une combinaison de séquence de 3 T1 et de 3 T2
Exp:
T1T1T1T2T2T2T1T1
T1T2T2T2…

Chapitre 4: la concurrence en Ada

III. Les rendez-vous

On peut conclure que sur le plan conceptuel, les tâches peuvent être classées en trois catégories :

- **Les tâches serveuses** qui proposent des rendez-vous et n'en demandent pas : par exemple la tache Allocateur.
- **Les tâches actrices** qui ne proposent pas des rendez-vous. Mais elles en demandent : par exemple les tâches T1 et T2.
- **Les tâches actrices/serveuses** qui proposent et demandent des rendez-vous.

III. Les rendez-vous

Le mécanisme de rendez-vous Ada permet à la fois **la synchronisation et la communication**, nécessaires à la résolution des problèmes de compétition et de coopération entre tâches.

Une tâche Ada ne peut traiter qu'un seul rendez-vous à la fois, c'est le principe d'exclusion mutuelle sur les rendez-vous.

Chaque entrée proposée par une tâche serveuse est dotée d'une file d'attente FIFO (Structure de Données First In First Out) gérée par l'exécutif d'Ada. Cette file FIFO permet de stoker les demandes sur cette entrée dans l'ordre d'arrivée.

13

Chapitre 4: la concurrence en Ada

III. Les rendez-vous

Pour gérer le non déterminisme, le langage Ada dispose d'une instruction adéquate appelée l'instruction d'attente sélective **select**.
Cette l'instruction **select** permet à la tâche appelée de sélectionner **arbitrairement** une demande de rendez-vous **parmi les rendez-vous disponibles**.

```
select
        accept entree_A;      …
or
        accept entree_B;      …
or
        accept entree_C; …
end select;
```

14

III. Les rendez-vous

La durée du rendez-vous est la durée nécessaire à l'exécution des instructions attachées à l'entrée :

accept nom_entree do
<Instructions exécutées>
end nom_entree ;

La communication entre tâches est réalisée par la transmission de paramètres typés dans les deux sens (in, out et in out), lors de l'occurrence d'un rendez-vous.

Exemple:

```
task Stream is
    public entry input (data: in String) ;
    public entry output (data: out String) ;
end Stream;
```

L'exemple suivant illustre l'utilisation exclusive d'une ressource critique (variable entière encapsulée par la tâche serveuse **variable_protegee**) par deux tâches actrices t1 et t2.

III. Les rendez-vous

```
procedure LireEcrire is
task variable_protegee is
   entry lire (v : out integer);
   entry ecrire (v : in integer);
end variable_protegee ;
task body variable_protegee is
  x :integer := 0;
 begin
 loop
  select
   accept lire (v : out integer) do
     v :=x ;
    Put("c'est une lecteur à partir de la variable protégée");
   end lire ;
  or
    accept ecrire (v : in integer) do
      x :=v ;
     Put("C'est une écriture dans la variable protégée");
    end ecrire ;
  or
    terminate ;
  end select ;
 end loop ;
end variable_protegee ;
```

```
task t1 ;
task t2 ;
task body t1 is
 a: integer:=5;
 begin
…
variable_protegee.ecrire (a);
…
end t1 ;

task body t2 is
 b: integer:=6;
begin
…
 variable_protegee.lire(b);
…
end t2 ;
 begin
    null;
 end LireEcrire ;
```

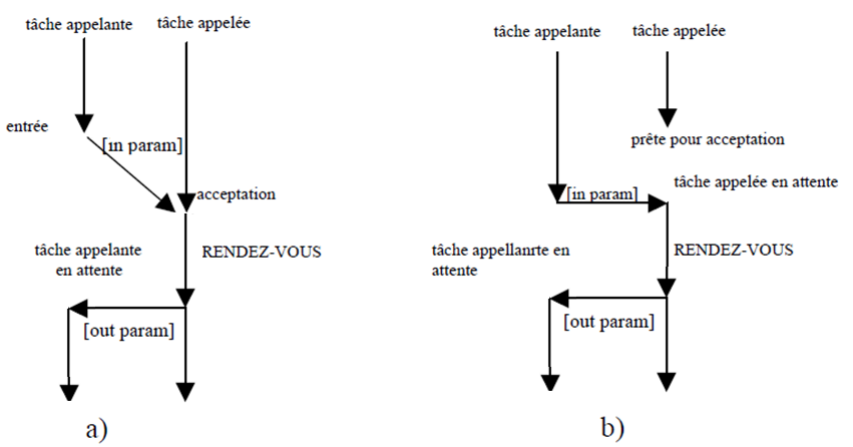
III. Les rendez-vous

L'exemple précédent illustre deux cas.

- **Dans le cas a:** la tâche APPELEE est prête après que la tâche APPELANTE ait effectué une demande de rendez-vous sur une entrée.
- **Dans le cas b:** c'est la tâche APPELEE qui est prête avant la demande de rendez-vous.

Lors d'un appel, si la tâche APPELEE attend sur l'instruction *accept*, le rendez-vous a lieu immédiatement, sinon, la demande de la tâche APPELANTE est mémorisée dans une file FIFO et la tâche est mise en attente passive de l'acceptation du rendez-vous. Ainsi, la forme de communication entre les tâches Ada est donc **bloquante**.

III. Les rendez-vous



IV. Les types tâches

A l'instar des types de données usuels ou des types abstraits, il est possible de déclarer des types de tâches (task type).
Les types tâches en Ada sont considérés comme **limited private** : seule la transmission est permise.
Les affectations et les comparaisons des tâches ne sont pas permises.
Les types tâches permettent de factoriser plusieurs tâches ayant le même comportement. De plus, ils ouvrent la perspective de créer dynamiquement des tâches.

IV. Les types tâches

```
-- type de tâche et non une tâche
task type allocateur is
   entry prendre;
   entry liberer;
end allocateur;
task body allocateur is
begin
  loop
    select
       accept prendre;
       accept liberer;
   or
       terminate;
  end select;
 end loop;
end allocateur;
```

```
-- définition statique des tâches
   ecran : allocateur;
   imprimante : allocateur;
```

Les deux tâches serveuses *ecran* et *imprimante* permettent l'accès exclusif (par des tâches actrices T1, T2, …) respectivement aux deux ressources critiques écran et imprimante.

V. Exercice d'application 1

On souhaite écrire une tâche qui gère une mémoire de calculatrice. Cette mémoire est un entier initialisé à zéro. Les autres tâches consultent et modifient cette mémoire grâce à l'interface suivante:

```
task type memoire is
entry ajouter (num : in integer);
entry soustraire (num : in integer);
        entry consulter (num : out integer);
end memoire;
```

L'entrée ajouter (resp. soustraire) permet d'ajouter (resp. de retrancher) un entier à la mémoire. L'entrée consulter permet de connaître la valeur courante de la mémoire.

1. Écrire le corps de la tâche mémoire. Pour simplifier, dans un premier temps, on suppose que la tâche mémoire accepte les rendez-vous dans cet ordre : ajouter, soustraire, consulter, et ainsi de suite.
2. Modifier le corps de la tâche afin qu'elle puisse accepter les rendez-vous dans un ordre quelconque.
3. Donner une procédure principale illustrant l'utilisation de la tâche mémoire avec une seule tâche de ce type.
4. Modifier le programme précédent en déclarant un tableau de cases mémoire.

V. Exercice d'application 1

```
task body memoire is
   Valeur : integer := 0;
begin
   loop
            accept Ajouter (Num : in integer) do
                     Valeur := Valeur + Num;
            end Ajouter;
            accept Soustraire (Num : in integer) do
                      Valeur := Valeur - Num;
             end Soustraire;
            accept Consulter (Num : out integer) do
                      Num := Valeur;
            end Consulter;
   end loop;
end memoire;
```

Réponse Question1

V. Exercice d'application 1

```
task body memoire is
 Valeur : integer := 0;
begin
   loop
     select
            accept Ajouter (Num : in integer) do
             Valeur := Valeur + Num;
            end Ajouter;
       or
             accept Soustraire (Num : in integer) do
             Valeur := Valeur - Num;
            end Soustraire;
       or
            accept Consulter (Num : out integer) do
             Num := Valeur;
            end Consulter;
     end select;
   end loop;
end memoire;
```

Réponse Question 2

V. Exercice d'application 1

```
procedure Main is
   M : memoire;
   Result : integer;
begin
   M.Ajouter (5);
   M.Soustraire (3);

   M.Consulter (Result);
   Put_Line ("Valeur de la mémoire : " & Integer'Image(Result));

   M.Ajouter (10);

   M.Consulter (Result);
   Put_Line("Valeur de la mémoire : " & Integer'Image(Result));
end Main;
```

Réponse Question 3

V. Exercice d'application 1

```
procedure Main is

type Tableau_Memoire is array (1..3) of memoire;
Memoire_Array : Tableau_Memoire;
Begin
-- Utilisation d'une case mémoire spécifique
Memoire_Array(1).Ajouter(5);
Memoire_Array(1).Consulter(Result);
Put_Line("Valeur de la mémoire 1 : " & Integer'Image(Result));

-- Utilisation d'une autre case mémoire
Memoire_Array(2).Soustraire(3);
Memoire_Array(2).Consulter(Result);
Put_Line("Valeur de la mémoire 2 : " & Integer'Image(Result));

end Main;
```

Réponse Question 4

V. Exercice d'application 2

On cherche dans cet exercice à proposer une solution concurrente au problème producteur-consommateur. On vous demande d'écrire un programme Ada qui garantit que le consommateur lit (affiche à l'écran) chaque donnée (supposant un entier) écrite par (saisie dans) le producteur exactement une fois. Après chaque consommation de la donnée écrite par le producteur, ce dernier refaire de nouveau une deuxième écriture de cette donnée et attend sa consommation. Le cycle de production/consommation ce répète infiniment.