Informatique industrielle A7-19571
Systèmes temps-réel

J.F.Peyre

Chapitre 4 : Programmation concurrente en Ada et en C avec la norme Posix

Plan du cours

Pour les deux langages Ada et C (avec la norme Posix) nous étudions successivement

- la définition et ou la création de tâches
- le lancement de tâches
- la terminaison de tâches

Les tâches en Ada sont nommées *Task* et les tâches dans la norme Posix *Thread*

Manipulations de tâches en Ada

Rappel sur la structure d'un programme Ada

```
with Nom_Des_Librairies_Utilisées;

procedure Nom_Du_Programme is

   Partie déclarations

begin

   Partie instructions

end Nom_Du_Programme;
```

Déclarations de constantes, variables, types, sous-programmes (procédure ou fonction) tâches

Séquence d'instructions du programme principal (principalement appels de sous-programmes déclarés dans la partie déclarations ou de sous-programmes de librairies)

Rappel sur la structure d'un programme Ada (suite)

```
1)  with Ada.Text_IO;
2)  with Ada.Integer_Text_IO;
3)
4)  procedure Prog1 is
5)    X: constant Integer := 4;
6)    Y: constant Integer := 5;
7)  begin
8)    Ada.Text_IO.Put( "La somme de " );
9)    Ada.Integer_Text_IO.Put( X );
10)   Ada.Text_IO.Put( " et de " );
11)   Ada.Integer_Text_IO.Put( Y );
12)   Ada.Text_IO.Put( " vaut ");
13)   Ada.Integer_Text_IO.Put( X + Y );
14) end Prog1;
```

Rappel sur la structure d'un programme Ada (suite)

```
1)  with Ada.Text_IO; use Ada.Text_IO;
2)  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3)
4)  procedure Prog1 is
5)    X: constant Integer := 4;
6)    Y: constant Integer := 5;
7)  begin
8)    Put( "La somme de " );
9)    Put( X );
10)   Put( " et de " );
11)   Put( Y );
12)   Put( " vaut ");
13)   Put( X + Y );
14) end Prog1;
```

Définition d'une tâche

- **Une tâche se définit en Ada par deux déclarations :**
  - la déclaration de la "spécification" de la tâche où l'on déclare entre autre les points de synchronisation (appelés entrées) de cette tâche
  - la déclaration du "corps" (ou body) de la tâche qui se décompose en une partie déclaration (entre autre variables locales à la tâche) et une partie instructions (les instructions exécutées par la tâche)
- **Ces déclarations sont faites au niveau de la partie déclaration du programme ou du bloc où elle est définie (nous n'étudions pas dans notre cours l'imbrication de tâches).**

Définition d'une tâche (suite)

- **Exemple de déclaration d'une tâche cyclique (qui ne termine jamais) appelée Tache_A**

```
task Tache_A;
```

Spécification

```
task body Tache_A is
begin
   loop
      Put('A');
   end loop;
end Tache_A;
```

Body

Définition d'un type tâche

- Il est également possible de déclarer un type tâche (i.e. un modèle de tâche) puis de déclarer ensuite des variables de ce type (des instances de tâche)

- Les tâches d'un même type ont le même comportement (celui défini par le type)

- Il peut alors être nécessaire d'inclure des paramètres de lancement afin de pouvoir différencier le comportement de plusieurs tâches d'un même type (il est déconseillé d'utiliser l'identifiant système de la tâche pour réaliser cette différentiation)

Définition d'un type tâche (suite)

```
task type Un_Genre_De_Tache;
```

Spécification du type

```
task body Un_Genre_De_Tache is
begin
  loop
     Put('A');
  end loop;
end Un_Genre_De_Tache;
```

Body du type

```
Tache1 : Un_Genre_De_Tache;
Tache2 : Un_Genre_De_Tache;
```

Variables de ce type

Définition d'un type tâche (suite)

- Il est possible de définir un type tâche avec paramètre (discriminant) qui permet ainsi de particulariser les différentes variables d'un même type tâche

```
task type Un_Genre_De_Tache_Avec_Parametre (Le_Caractere_Personel : Character);

task body Un_Genre_De_Tache_Avec_Parametre is
begin
   loop
      Put(Le_Caractere_Personel);
   end loop;
end Un_Genre_De_Tache_Avec_Parametre;

Tache_B : Un_Genre_De_Tache_Avec_Parametre ('B');
Tache_C : Un_Genre_De_Tache_Avec_Parametre ('C');
```

Création d'une tâche

- En Ada il n'y a aucune instruction ou construction particulière à utiliser (nous ne considérons pas dans ce cours la notion de pointeur sur tâche)
- Une tâche est une variable et sa création se fait lors de l'élaboration de la partie où est déclarée la variable tâche (de la même manière que la création de toute variable)
- La création peut néanmoins échouer du fait d'une erreur survenant lors de l'élaboration; la tâche ne pourra alors pas s'exécuter.

Lancement d'une tâche

- De nouveau, il n'y a dans le cas général aucune instruction particulière à utiliser
- Le lancement des tâches déclarées dans un programme (plus précisément dans un bloc) est automatique et se fait en même temps que débute l'exécution du programme (ou du bloc)
- Les tâches s'exécutent donc en concurrence avec le programme principal (mais également les unes par rapport aux autres)

Exemple de programme multi-tâches

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Prog2 is
   -- ================================================================
   task type Un_Genre_De_Tache_Avec_Parametre (Le_Caractere_Personel : Character);
   task body Un_Genre_De_Tache_Avec_Parametre is
   begin
      loop
         Put(Le_Caractere_Personel);
      end loop;
   end Un_Genre_De_Tache_Avec_Parametre;
   -- ================================================================

   Tache_B : Un_Genre_De_Tache_Avec_Parametre ('B');
   Tache_C : Un_Genre_De_Tache_Avec_Parametre ('C');

begin
   loop
      Put('Z');
   end loop;
end Prog2;
```

Exemple de programme multi-tâches

- **L'exécution du programme précédent produit à l'écran des suites de caractères B, C et Z.**

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC....CCCCCCZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZ....ZZZZZZBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB....BBBBBBCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCC....CCCCCC....

- **L'alternance provient :**

1) du fait que les deux tâches Tache_B et Tache_C s'exécutent en parallèle avec le programme principal (qui affiche sans fin le caractère `Z`)

2) du fait que l'écran est une ressource partagée et ne peut être utilisé que par une tâche à la fois (les caractères sont donc affichés un par un)

3) du fait que les deux tâches et le programme principal ont la même priorité et sont donc exécutés équitablement à tour de rôle (partage de temps)

Terminaison d'une tâche

- **Une tâche se termine de différentes façons (en simplifiant légèrement) :**
  - soit lorsqu'elle a fini d'exécuter les instructions de son "body" (et que les tâches dépendant d'elle ont aussi terminées)
  - soit lorsqu'elle lève une exception non récupérée (terminaison en erreur)
  - soit lorsqu'elle est avortée par une instruction « `abort Nom_De_La_Tache;` » ou par le système sur lequel elle s'exécute

Terminaison normale

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Prog2_Fin_Normale is
   -- ================================================================
   task type Un_Genre_De_Tache_Avec_Parametre (Le_Caractere_Personel : Character);
   task body Un_Genre_De_Tache_Avec_Parametre is
   begin
      for I in 1..100_000 loop
         Put (Le_Caractere_Personel);
      end loop;
   end Un_Genre_De_Tache_Avec_Parametre;
   -- ================================================================
   Tache_B : Un_Genre_De_Tache_Avec_Parametre ('B');
   Tache_C : Un_Genre_De_Tache_Avec_Parametre ('C');

begin
   for I in 1..100_000 loop
      Put ('Z');
   end loop;
end Prog2_Fin_Normale;
```

s'arrête après 100 000 tours

Terminaison (suite)

- **Le transparent suivant montre un exemple de terminaison forcée : le principal termine l'exécution des deux tâches internes par un appel explicite à l'instruction « `abort` »**
  - Les tâches `Tache_B` et `Tache_C` terminent car le programme principal les tue
  - Le principal termine car 1) il termine son body et 2) les tâches qui dépendent de lui terminent également.

- **Ce genre de terminaison est dangereux; en effet, il se peut qu'une des tâches `Tache_B` ou `Tache_C` (ou même les deux) n'ait pas le temps de s'exécuter; elles peuvent être tuées avant.**

Terminaison forcée

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Prog2_Fin_Forcee is
   -- ================================================================
   task type Un_Genre_De_Tache_Avec_Parametre(Le_Caractere_Personel : Character);
   task body Un_Genre_De_Tache_Avec_Parametre is
   begin
      loop
         Put(Le_Caractere_Personel);
      end loop;
   end Un_Genre_De_Tache_Avec_Parametre;
   -- ================================================================
   Tache_B : Un_Genre_De_Tache_Avec_Parametre('B');
   Tache_C : Un_Genre_De_Tache_Avec_Parametre('C');

begin
   for I in 1..100_000 loop
      Put('Z');
   end loop;
   abort Tache_B;
   abort Tache_C;
end Prog2_Fin_Forcee;
```

Manipulations de tâche avec la norme Posix

Définition d'une tâche Posix

- La norme Posix ne définit pas explicitement la notion de tâche en terme de construction de langage
- En fait, un tâche Posix n'est défini du point de vue du programmeur que par son identifiant système de type « `pthread_t` »
- Un tâche Posix n'est pas définie mais créée par l'appel d'une primitive système

Création d'un tâche Posix

- Un tâche Posix est crée grâce à l'appel système « `pthread_create` »
- Le synopsis de cette fonction est :

```
int  pthread_create( pthread_t       * pthread,
                     pthread_attr_t  * attr,
                     void            * (*start_routine)(void *),
                     void            * arg);
```

- L'appel de `pthread_create` crée un nouveau tâche Posix s'exécutant concurremment avec le tâche Posix appelant.
- Le nouveau tâche Posix exécute la fonction `start_routine` en lui passant `arg` comme premier argument.

Création d'un tâche Posix (suite)

- **L'argument `attr` indique les attributs du nouveau tâche Posix. Cet argument peut être `NULL`, auquel cas, les attributs par défaut sont utilisés:**
  - la tâche Posix créée est joignable (non détachée)
  - elle utilise la politique d'ordonnancement usuelle (pas temps-réel).

- **La valeur retournée est 0 en cas de succès et un code d'erreur non nul en cas d'erreur (par exemple `EAGAIN` lorsqu'il n'y a pas assez de ressources système pour créer une nouvelle tâche Posix)**

- **En cas de succès, l'identifiant système de la nouvelle tâche Posix est stocké à l'emplacement mémoire pointé par le premier l'argument (pthread_t * pthread) de l'appel à `pthread_create`**

Lancement d'un tâche Posix

- Il n'y a aucune action spécifique à mener

- La création d'un tâche Posix (si elle réussie) lance également l'exécution du tâche Posix qui s'exécute concurremment avec son créateur

Exemple de programme multi-tâche Posix

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread Posix.h>


void *affCar(void *arg){
   char c;

   c = * (char *)arg;

   while(1){
      putchar( c);
   }
}
```

./...

Exemple de programme multi-tâche Posix (suite)

```
int main(void)
{
   char *leCar;
   pthread_t tache_Posix_B;
   pthread_t tache_Posix_C;

   leCar = (char*) malloc(1*sizeof(char)); *leCar = 'B';
   pthread_create( &tache_Posix_B, NULL, affCar, (void*) leCar);

   leCar = (char*) malloc(1*sizeof(char)); *leCar = 'C';
   pthread_create( &tache_Posix_C, NULL, affCar, (void*) leCar);

   while(1){
      putchar('Z');
   }
}
```

Exemple de programme multi-tâche Posix (suite)

- Le tâche Posix crée s'exécute concurremment avec son créateur mais partage son espace d'adressage

- Tout ce qui doit être propre au tâche Posix doit donc être dupliqué par le père avant la création du tâche Posix

- Ceci explique dans le code précédent les instructions

```
leCar = (char*) malloc(1*sizeof(char)); *leCar = 'C';
```

qui permettent au tâche Posix créée d'avoir un espace propre pour désigner le caractère passé en paramètre; il se pourrait sinon que le père modifie la valeur de ce caractère (par exemple en le passant à un autre tâche Posix) avant que le premier n'ait eu le temps de sauvegarder la valeur initiale.

Exemple de programme multi-tâche Posix (suite)

Le programme suivant est donc incorrect

```
int main(void)
{
   char leCar;

   pthread_t tache_Posix_B;
   pthread_t tache_Posix_C;

   leCar = 'B';
   pthread_create( &tache_Posix_B, NULL, affCar, (void*) &leCar);

   leCar = 'C';
   pthread_create( &tache_Posix_C, NULL, affCar, (void*) &leCar);

   while(1){
      putchar('Z');
   }
}
```

Encapsulation de l'appel `pthread_create`

- Il peut être commode d'encapsuler les création de tâche Posix par une fonction unique (ceci évite par exemple de traiter systématiquement le code de retour de l'appel de `pthread_create`)

- On procéderais alors de la façon suivante

```
pthread_t cree_tache(void * (*start_routine)(void *), void * arg){
   pthread_t id;
   int erreur;

   erreur = pthread_create( &id, NULL, start_routine, arg);
   if (erreur != 0){
      perror( "Echec creation de tâche Posix" );
      exit(-1);
   }
   return id;
}
```

Terminaison

- **Le nouveau tâche Posix s'achève soit explicitement en appelant « `pthread_exit()` », ou implicitement lorsque la fonction `start_routine` s'achève (*ce dernier cas est équivalent à appeler `pthread_exit` avec la valeur renvoyée par `start_routine` comme code de sortie*).**

- **Contrairement à Ada, si le programme principal termine normalement, les tâches Posix créées par celui-ci seront terminées par un appel implicite à l'appel système "exit" qui termine le processus englobant**

- **Si l'on supprime dans l'exemple précédent la boucle du programme principal, les tâche Posix tache_Posix_B et tache_Posix_C sont terminées à peine créés (elles n'ont en fait pas le temps de s'exécuter); en Ada, le programme n'aurait pas terminé**

Terminaison (suite)

- Si l'on veut terminer "proprement" un programme (ou un sous-programme) créant des tâche Posix, il est nécessaire d'attendre la terminaison de celles-ci (ou de les détruire délibérément)

- La primitive "`pthread_join`" de synopsis

```
int pthread_join(pthread_t th, void **tache_return);
```

suspend l'exécution de tâche Posix appelant jusqu'à ce que le tâche Posix identifié par `th` termine elle-même son exécution (soit en terminant ses instructions soit par un appel de "`pthread_exit`")

Exemple de terminaison

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread Posix.h>


void *affCar2(void *arg){
   char c;
   int i;

   c = * (char *)arg;

   for (i=0; i<1000; i++){
      putchar( c );
   }
}
```

./...

Exemple de terminaison (non contrôlée)

```
int main(void)
{
   char *leCar; int i;
   pthread_t tache_Posix_B;
   pthread_t tache_Posix_C;

   leCar = (char*) malloc(1*sizeof(char)); *leCar = 'B';
   tache_Posix_B = cree_tache( affCar2, (void*) leCar);

   leCar = (char*) malloc(1*sizeof(char)); *leCar = 'C';
   tache_Posix_C = cree_tache( affCar2, (void*) leCar);

   for (i=0; i<1000; i++){
      putchar('Z');
   }
}
```

Exemple de terminaison (contrôlée)

```
int main(void)
{
   char *leCar; int i;
   pthread_t tache_Posix_B;
   pthread_t tache_Posix_C;

   leCar = (char*) malloc(1*sizeof(char)); *leCar = 'B';
   tache_Posix_B = cree_tache( affCar2, (void*) leCar);

   leCar = (char*) malloc(1*sizeof(char)); *leCar = 'C';
   tache_Posix_C = cree_tache( affCar2, (void*) leCar);

   for (i=0; i<1000; i++){
      putchar('Z');
   }
   pthread_join( tache_Posix_B, NULL);
   pthread_join( tache_Posix_C, NULL);
}
```

Conclusion

- **Afin de définir une application concurrente il faut pouvoir :**
  - définir et si nécessaire créer les taches de l'application
  - lancer (démarrer) les taches afin de les activer
  - les terminer (ou prévoir leur terminaison)

- **Ces notions sont mises en oeuvre différemment selon les normes, les langages, les systèmes ou les exécutifs**