Cours Programmation Concurrente

Master MIAGE M1

Jean-François Pradat-Peyre,
Université Paris Nanterre - UFR SEGMI

2023-2024

2 : Créer des entités concurrentes, sémantique et mise en pratique

Programmation concurrente : Quelques principes

- Programme concurrent = programme qui fait intervenir plusieurs entités (*processus* ou *tâches*) qui coopèrent pour atteindre un objectif commun

- Un programme concurrent peut s'exécuter sur différentes machines (modèle réparti et vrai parallélisme), sur une machine parallèle (modèle parallèle et vrai parallélisme) ou sur une seule machine (modèle centralisé et temps partagé)

- Chaque entité concurrente a un rôle spécifique (code ou données)

Programmation concurrente : Quelques principes (suite)

- Une entité concurrente peut être une tâche ou un processus ou une coroutine
- Par défaut un processus a son propre espace d'adressage et donc ne partage pas de données avec les autres processus (données privées)
- Par défaut une tâche partage un espace d'adressage commun à toutes les tâches mais chacune possède un espace privé : pile d'exécution, variables locales (donc, des données partagées et des données privées)
- Sur un même processeur, les entités concurrentes sont exécutées à tour de rôle selon le choix de **l'ordonnanceur** du système hôte
- Le choix de l'ordonnanceur est souvent imprévisible (paramètres système complexes) sauf dans les système temps réel (ce qui n'est pas notre cas)

Suite du chapitre

Pour les deux langages, C (avec la norme Posix) puis Java nous étudions successivement

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

Les tâches en Java et dans la norme Posix *sont nommées Thread*

2.1 Manipulations de tâche avec le langage C et 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 : interface (API) vers le système sous-jacent

- Une tâche Posix n'est définie du point de vue du programmeur que par son identifiant système de type « `pthread_t` »

- Une 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 une nouvelle tâche Posix s'exécutant concurremment avec le tâche Posix appelant la primitive (son frère et non son père).

- La nouvelle tâche Posix exécute la fonction `start_routine` en lui passant `arg` comme unique argument.

Création d'un tâche Posix (suite)

- L'argument `attr` indique les attributs de la nouvelle tâche Posix. Cet argument peut être `NULL`, auquel cas, les attributs par défaut sont utilisés, i.e :
  - 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 réussit) 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.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 créateur avant la création de la tâche Posix

- Ceci explique dans le code précédent les instructions

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

qui permettent à la tâche Posix créée d'avoir un espace propre pour désigner le caractère passé en paramètre; il se pourrait sinon que le créateur modifie la valeur de ce caractère avant que la tâche créée n'ait eu le temps de sauvegarder la valeur initiale.

Exemple de programme multi-tâche Posix (suite)

Le programme suivant est donc incorrect (**Dire pourquoi !**)

```
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 procéderait par exemple 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

- La nouvelle 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 d'autres sémantiques, si le programme principal termine, 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 `pthread_cancel(pthread_t thread)`)

- La primitive "`pthread_join`" de synopsis

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

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

Exemple de terminaison : le code des threads

```
#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);
}
```

2.2 Manipulations de tâche avec le langage Java

Définition de tâche en Java

- Une tâche en Java est un objet qui hérite de la classe Thread et qui implémente la méthode run() de signature *public void run() ;*

- Le code de la thread doit se trouver dans cette méthode.
- Exemple :

```
public class MonThread extends Thread {
    public void run() {
        // code de la thread
    }
}
```

- La méthode run() ne peut pas prendre de paramètre et ne peut pas retourner de valeur. S'il faut des paramètres, ils seront transmis au constructeur qui se chargera de les stocker en variable d'instance. Un accesseur pourra être mis en place pour récupérer une valeur à la fin du traitement.

Création de tâche en Java

- Une tâche est un objet qui est créée comme un objet (new)

  - ✓ `MonThread uneTache = new MonThread();`

- Comme pour un objet « normal », le constructeur sera choisi en fonction des paramètres liés à la création

- La création NE démarre PAS la tâche ; il faudra la lancer « à la main »

Lancement d'une tâche en Java

- Pour lancer/démarrer une tâche (qui a déjà été créée) il faut invoquer la méthode start() sur l'objet correspondant

- La méthode start invoquera automatiquement la méthode run de l'objet (l'équivalant sui « main » de la tâche)

  - ✓ `uneTache.start();`

- Il est également possible de combiner lancement et création de tâche

  - ✓ `MonThread uneTache = new MonThread().start();`

Terminaison d'une tâche en Java

- Une tâche termine lorsqu'elle a terminé toutes les instructions de sa méthode « `run` » (fin normale ou fin en exception)

- Une tâche peut également être stoppée pendant un temps donné (approximatif) avec la méthode `sleep`() ;

- On attend la fin d'une tâche avec la méthode « `join` »

```
monThread.join();
```

- **Le programme ne termine que lorsque toutes les tâches qu'il a créées ont terminé**

Autre façon de créer des tâches : implémenter Runnable

- L'interface `java.lang.Runnable` déclare la méthode `public abstract void run`
- La classe Thread définit un constructeur recevant un argument de type Runnable
- Si l'on suppose que `r` soit une instance d'une classe qui implémente Runnable
- On peut écrire :

```
Thread t = new Thread(r);
```

- Qui crée une thread `t` qui peut être démarrée par l'instruction `t.start();`
- Cette construction est plus efficace que la précédente car le code est partagé entre toutes les thread crées à partir d'une implémentation de Runnable; dans l'autre construction chaque objet possède son propre code.

Autre façon de créer des tâches : implémenter Runnable

```
class Repetiteur implements Runnable {

    String chaine;

    Repetiteur(String chaine) {
        this.chaine = chaine;
    }

    public void run() {
        System.out.println(chaine);
        System.out.println(chaine);
    }
}
```

```
class Ecrivain {
    public static void main(String[] argv) {
        new Thread(new Repetiteur("soleil")).start();
        new Thread(new Repetiteur("neige")).start();
        new Thread(new Repetiteur("ski")).start();
        System.out.println("A la montagne");
    }
}
```

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

Les prochaines séances

- S3 : Synchronisation entre entités concurrentes, sémantique et illustrations en Java et C/Posix
- S4 : Paradigmes de la concurrence 1
- S5 : Paradigmes de la concurrence 2
- S6 : Programmation concurrente en Python