TP - ASR7 Programmation Concurrente

—

Producteur-Consommateur

Matthieu Moy, Fabien Rico, Adil Khalfa

Printemps 2018

I Producteur-Consommateur

Le but de ce TP est de modifier le code du TP2 (calcul de fractale de Mandelbrot en parallèle) pour mettre en place un système de producteur-consommateur.

Si vous avez bien réussi le TP2, vous pouvez continuer en utilisant votre code. Sinon, commencez par récupérer l'archive `mandel-threads.zip` sur la page du cours. Compilez et exécutez le code qu'elle contient et vérifiez qu'il s'agit bien du travail demandé au TP2.

Au TP2, nous avions utilisé un mutex pour empêcher les accès concurrents à l'affichage. Cette fois-ci, nous allons dédier un thread à l'affichage (via `draw_rect()`). Ce thread sera le seul à y accéder, donc il n'y aura plus de problème d'exclusion mutuelle. Les threads de calcul n'appelleront plus `draw_rect()`, mais enverront le rectangle d'écran à afficher au thread chargé de l'affichage.

La communication entre les threads de calcul et le thread d'affichage se fera au moyen d'un schema « producteur-consommateur », comme vu en TD. Il s'agit d'une classe C++ (un moniteur), qui expose les méthodes suivantes :

- `void put(element)`, appelée par le ou les producteurs pour envoyer un élément au consommateur. Cette fonction est bloquante si la file est pleine.
- `element get()`, appelée par le ou les producteurs pour récupérer un élément envoyé par un producteur. Cette fonction est bloquante si la file est vide.

Les éléments échangés sont des instances de la structure suivante qui décrit un rectangle à afficher :

```
1  struct rect {
2          int slice_number;
3          int y_start;
4          rect(int sn, int y) : slice_number(sn), y_start(y) {};
5  };
```

Les threads de calculs enverront les rectangles, par exemple avec :

```
1  prod_cons.put(rect(slice_number, y));
```

Le thread chargé de l'affichage récupérera ces valeurs pour appeler `draw_rect()` dessus avec, par exemple :

```
1  rect r = prod_cons.get();
```

I.1 Travail à réaliser

Tout d'abord, implémentons notre producteur-consommateur :

— Créez un nouveau fichier C++ dans lequel vous allez écrire le producteur-consommateur. Créez une classe `ProdCons` contenant les deux méthodes `put()` et `get()` décrites ci-dessus. Si vous êtes à l'aise en C++, écrivez une classe template pour que les arguments de `put()` et `get()` soient génériques.

— Ajoutez le nécessaire pour que cette classe puisse agir comme un moniteur.

— Ajoutez les données nécessaires pour implémenter une file d'attente (FIFO) dans la classe. En TD, nous avons codé un buffer circulaire à la main avec un tableau ; vous pouvez aussi utiliser une structure toute faite de C++ comme `std::list` ou mieux, `std::queue` (vous aurez besoin des méthodes `push()`, `pop()`, `front()` et `size()`).

— Donnez le corps des fonctions `put()` et `get()`.

Utilisez maintenant votre classe dans le code de calcul de Mandelbrot :

— Instanciez votre classe `ProdCons`. Le plus propre est de l'instancier dans la fonction `draw_screen_thread()`, mais vous pouvez aussi le faire en variable globale pour simplifier.

— Écrivez la fonction `x_thread_function()` qui sera exécutée par le thread chargé de l'affichage. Cette fonction doit avoir accès à l'instance de `ProdCons`.

— Dans la fonction `draw_screen_thread()`, lancez ce thread en début de calcul.

— Modifiez la fonction `draw_slice()` pour lui faire envoyer les rectangles au thread chargé de l'affichage.

Testez l'ensemble !

II Gérer la liste des tâches avec un producteur-consommateur

Notre classe `ProdCons` peut aussi être utilisée pour gérer la liste des tâches (la fonction `get_slice()` que nous avons utilisée lors du TP2) : on peut instancier un thread chargé d'appeler `get_slice()` et d'envoyer les tranches d'écrans à calculer aux threads de calcul. De cette manière, le thread producteur sera le seul à appeler `get_slice()` (donc on pourra supprimer le mutex qui protégeait son accès).

Si le temps le permet, mettez en place ce producteur-consommateur en plus de celui de la section précédente.