INF5171: Cours–Laboratoire #9
Parallélisme et concurrence avec les *threads* Posix

Mardi, 21 novembre 2017
13h30–16h30
PK-S1560

Le but de ce laboratoire est de vous familiariser avec l'utilisation des constructions `C` pour le parallélisme *fork/join* et la gestion de la concurrence à l'aide de moniteurs.

Obtenir le code à compléter

Obtenez une copie du code en exécutant la commande suivante sur `japet` :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/LaboPosix.git
```

1 Fonction `count` avec parallélisme récursif

Le code qui vous est fourni dans le fichier `count.c` définit trois (3) versions d'une fonction `count_*` qui trouve le nombre d'éléments d'un tableau qui satisfont un prédicat :

- `count_seq` : Version séquentielle itérative ;
- `count_rec` : Version séquentielle récursive ;
- `count_rec_par2` : Version parallèle récursive.

Dans le cas de `count_rec_par2`, il s'agit de la solution «simple» qui génère deux (2) nouveaux *threads*, un pour chaque sous-problème. Toutefois, comme on l'a vu précédemment (`PRuby`), le défaut d'une telle approche est que le *thread* parent **ne fait rien** pendant que les deux enfants s'exécutent, créant donc inutilement des *threads*.

En vous inspirant (copier/coller/modifier) de `count_rec_par2`, définissez une version parallèle récursive `count_rec_par1`, **qui genère un seul nouveau *thread*** pour traiter un des sous-problèmes, le *thread* parent prenant en charge l'autre sous-problème.

Cibles du `makefile` : `count` (compilation) et `run_count` (exécution).

Indice : La fonction `threaded_count1` est une fonction «comme les autres», donc elle peut être appelée sans nécessairement passer par la création d'un *thread* — il faut simplement pouvoir lui passer les arguments sous la forme appropriée, et lui indiquer le «mode d'appel» (séquentiel ou parallèle)!

2 Type abstrait `SVar`

Le fichier `svar.h` définit l'interface d'un type abstrait `SVar` pour des variables synchronisées... mais **plus simples** que celles du devoir #2.

Vous devez compléter le code, dans le fichier `svar.c`, pour quatre (4) opérations de ce type abstrait, et ce de façon à ce que les tests unitaires s'exécutent avec succès :

- `SVar new_SVar();`
- `Value value( SVar );`
- `void set_value( SVar, Value );`
- `SVar then( SVar, Value (*proc)(Value) );`

Cibles du `makefile` : `svar` (compilation) et `run_svar_{1,2,3}` (exécution de différents groupes de tests, des plus simples au plus complexes) ou `run_svar` (exécution de tous les tests).