Projet Programmation Concurrente – année 2016-17
Polytech'Nice Sophia – SI4
M. Riveill

Etape 3 – mise en œuvre avec des threads Posix et une synchronisation par moniteur

Cette troisième étape consiste à remplacer la synchronisation par sémaphores précédente par une synchronisation à l'aide de moniteurs. Je vous rappelle que vous devrez au cours du dernier TD de l'année préparer un petit exposé de 10' comprenant : présentation comparative de vos solutions et démonstration avec mise en œuvre de la partie graphique.

Comme dans l'étape 1 et 2 vous effectuerez des mesures (option –m) de l'exécution pour les scénarios suivants : -e1, –e2 et –e3 pour-t0, -t1, -t2 pour –p2 (4 personnes), -p4 (16 personnes), -p8 (256 personnes), **soit 27 mesures (9 sur chacune des étapes).**

Vous devrez rendre

- Dans une archive **tar gzip** (fichier tar compressé avec gzip) de nom `projet3-numero-groupe.tgz` se décompressant dans un répertoire de nom `projet3-numero-groupe`
- Après décompression le répertoire projet1-numero-groupe doit contenir
    - Un répertoire `src` contenant vos sources
    - un script shell de nom `compile.sh` permettant de compiler votre code sans inclure la partie graphique et mettre le binaire dans un répertoire `bin`
    - un script `shell` de nom `execute.sh` permettant d'exécuter votre code avec les options `–e3 –t1 –p4 –m`
    - un fichier PDF de nom `numero-groupe.pdf` contenant votre rapport

Le rapport doit être rédigé comme un rapport et donc comporter outre les éléments attendus une introduction et une conclusion. Dans cette seconde étape, le rapport doit compléter le rapport précédent. Il doit insister et décrire :

- l'algorithme utilisé pour déplacer une personne (rappel : un algorithme n'est pas le code C). Pour ceux qui ne savent pas ce qu'est un algorithme vous pouvez lire l'article : https://interstices.info/jcms/c_5776/qu-est-ce-qu-un-algorithme.
- comparer la manipulation des threads en Java (que vous avez vu en cours en SI3) et la manipulation des threads Posix (création, démarrage, arrêt, destruction, passages de paramètres, terminaison) ;
- pour la thread principale (i.e. celle associée au main de l'application), vous devez donner l'algorithme de création des threads filles (option –t1 et –t2) et celui lié à la terminaison de l'application (i.e. attente de création des threads filles précédemment créées) selon les trois solutions mises en oeuvre;
- analyser la mise en oeuvre de chacun des scénarios proposés :
    - étape 1 : est-ce que le simulateur est correct ? pourquoi ?
    - étape 2 : comment avez-vous utilisez les sémaphores pour synchroniser les différentes threads filles entre-elles. Etes-vous capable de modéliser votre programme en FSP ? Etes-vous capable de démontrer que votre programme ne présente pas de risque d'interblocage ? Si oui, dites pourquoi.
    - Etape 3 : comment avez-vous utilisez les moniteurs pour synchroniser les différentes threads filles entre-elles. Etes-vous capable de modéliser votre programme en FSP ? Etes-vous capable de démontrer que votre programme ne présente pas de risque d'interblocage ? Si oui, dites pourquoi.
- analyser de manière comparative les divers scénarios proposés, cette analyse doit nécessairement utiliser les mesures effectuées.

Pour cette troisième étape, la qualité du code de synchronisation sera bien évidemment évaluée. La correction de la mise en œuvre est primordiale : un programme peu efficace est toujours préférable à un programme faux.

Rappel : en Posix, un moniteur Posix est l'association :

- d'un mutex (type pthread_mutex_t) qui sert à protéger la partie de code où l'on teste les conditions de progression ;
- d'au moins une variable condition (type pthread_cond_t) qui sert de point de signalisation.

- Attente sur la variable de synchronisation par la primitive :
    - pthread_cond_wait(&laVariableCondition,&leMutex)

- Reveil sur cette variable avec la primitive :
    - pthread_cond_signal(&laVariableCondition);