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

Etape 2 – mise en œuvre avec des threads Posix et une synchronisation par sémaphore

Cette seconde étape a pour unique objectif de proposer un simulateur correct (i.e. une personne ne peut se déplacer que sur une case réellement vide) et d'utiliser les sémaphores Posix pour synchroniser les différentes. Pour ceux qui ont déjà réalisé l'interface graphique, il faut aussi garantir que celle-ci affiche bien la situation réelle.

Dans cette seconde étape, vous devez aussi remplacer la synchronisation nécessaire entre la thread associée au main et la terminaison des threads filles créées par pthread_create pour utiliser les sémaphores (i.e. remplacer l'utilisation de la fonction pthread_join par l'utilisation de sémaphores Posix).

Une option –e<nombre> sera ajoutée :

- -e1 : exécute la première étape du projet,
- -e2 : exécute la seconde étape du projet,
- -e3 : exécutera la troisième et dernière étape du projet.

Comme dans l'étape 1 vous effectuerez des mesures (option –m) de l'exécution pour les scénarios suivants : -e1 et –e2 pour-t0, -t1, -t2 pour –p2 (4 personnes), -p4 (16 personnes), -p8 (256 personnes), **soit 18 mesures (9 sur étape 1 et 9 sur étape 2).**

Vous devrez rendre

- Dans une archive **tar gzip** (fichier tar compressé avec gzip) de nom `projet2-numero-groupe.tgz` se décompressant dans un répertoire de nom `projet2-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 `–e2 -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 deux 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.
- analyser de manière comparative les divers scénarios corrects proposés, cette analyse doit nécessairement utiliser les mesures effectuées.

Pour cette seconde 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, les primitives pour utiliser des sémaphores sont :

int sem_init(sem_t *semaphore, int pshared, unsigned int valeur)
    Création d'un sémaphore et préparation d'une valeur initiale.
int sem_wait(sem_t * semaphore);
    Opération down sur un sémaphore.
int sem_post(sem_t * semaphore);
    Opération up sur un sémaphore.
int sem_destroy(sem_t * semaphore);
    Destruction d'un sémaphore.