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

Durée indicative : entre 20 et 30 heures dont uniquement 6 heures seront faites lors des séances de TD. A faire par groupe de deux.

Vision d'ensemble du projet : travail 3 étapes cumulatives (i.e. chaque rapport doit compléter le précédent) :

Etape 1 – mise en œuvre générale avec des threads Posix sans synchronisation
- Plusieurs scénarios de créations de threads seront mis en œuvre

Etape 2 – synchronisation à l'aide des variables conditions Posix
- Plusieurs modèles de synchronisation seront mis en œuvre

Etape 3 – synchronisation à l'aide des sémaphores Posix
- Plusieurs modèles de synchronisation seront mis en œuvre

Objectifs : mettre en pratique divers éléments vus en cours ou en travaux dirigés dans le cadre du langage C/Posix. *La qualité du rapport et des explications données sont les critères principaux d'évaluation*, puis vient le code et pour finir, le respect des consignes.

Ce projet compte pour 1/3 des points dans le cadre du cours de programmation concurrence.

Spécification de la mise en œuvre

Le terrain sur lequel se déplace la foule fait 512 pixels x 128 pixels et sera représenté par une matrice de taille identique ayant 512 colonnes (numérotées de 0 à 511) et de 128 lignes (numérotée de 0 à 128). Sur le terrain sera placé 2 murs d'épaisseur 16 pixels. Le premier mur aura en son centre une ouverture de largeur 8 et le second mur aura une ouverture 16. La figure 1 représente le terrain.

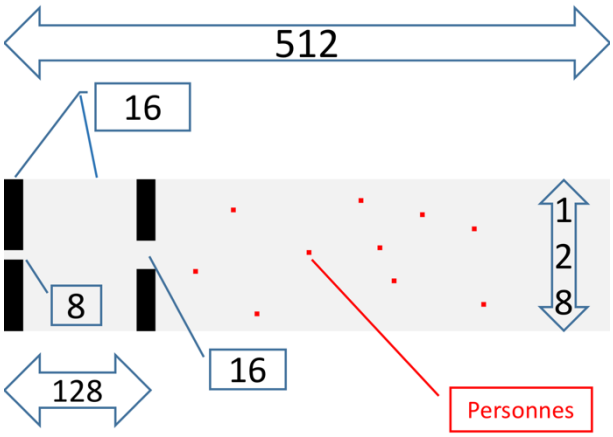

Fig. 1 – forme du terrain

Une personne fait 4 x 4 pixel et se déplace d'un seul pixel selon les directions Nord, Sud, Est et Ouest ainsi que les 4 diagonale Nord-Sud, Nord-Ouest, Sud-Est et Sud-Ouest. La figure 2 illustre les déplacements possibles pour une personne.

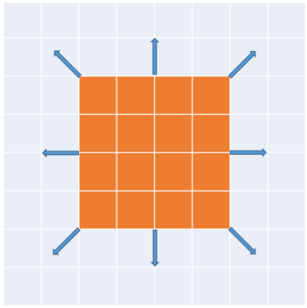

Fig. 2 - déplacement d'une personne

Après une phase d'initialisation qui met en place le terrain et qui distribue aléatoirement 2^p personnes, le simulateur fait avancer chacune des personnes vers la sortie matérialisée par le 'trou S'. A chaque déplacement la personne doit se rapprocher de la sortie. Pour cela, un point azimuth sera fixé et le déplacement choisi sera le déplacement possible qui minimise la distance avec le point (cf. figure 3). Bien évidemment une personne ne peut pas occuper une place occupée.

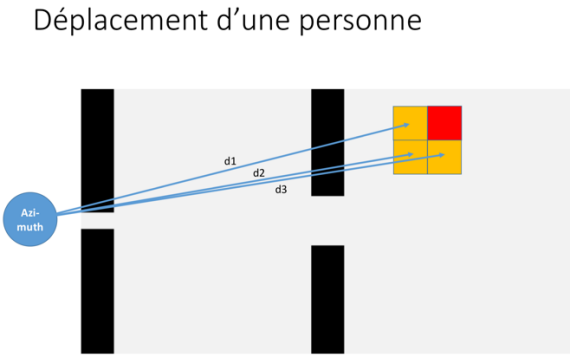

Fig. 3 – point azimuth

Afin de comparer les différentes mises en œuvres, il est souhaitable qu'à chaque exécution, les personnes partent de la même place.

Mises-en œuvre : vous allez programmer un simulateur ultra-simpliste du déplacement d'une foule. Toutes les versions seront intégrées dans le même code et des options du programme permettront de choisir la version à exécuter :

-p [0123456789] : nombre de personnes présentent sur le terrain

- p varie de 0 à 9 et le nombre de personnes vaut 2^p (i.e. varie de 1 à 512)

-t [012] : scénario de créations des threads

- -t0 : un seul thread qui fait avancer successivement chacune des personnes de 1 pixel jusqu'à ce que toutes les personnes soient sorties du terrain ;
- -t1 : le terrain est partagé en 4 partie égale. Une thread est associée à chaque partie du terrain et chaque thread doit faire avancer successivement chacune des personnes présentes sur la portion de terrain que la thread gère ;
- -t2 : une thread est associée à chacune des personnes créées et chaque thread doit faire avancer la personne qu'elle gère.

-m : mesure du temps d'exécution

- mesure la consommation du CPU et le temps de réponse du programme
- lorsque des mesures sont effectuées : la phase d'initialisation du programme n'est pas prise en compte et l'affichage n'est pas actif
- pour effectuer les mesures, l'application est lancée 5 fois et la mesure est la moyenne des 3 valeurs intermédiaires

Pour simplifier la portabilité des programmes entre votre machine et la mienne, il doit être possible de compiler votre programme sans inclure la partie graphique qui doit visualiser le déplacement de la foule sur le terrain.

Etape 1 – mise en œuvre générale avec des threads Posix sans synchronisation

Dans cette première étape qui a pour unique objectif de mettre en place le code C/C++ nécessaire aux étapes suivante, aucun mécanisme de synchronisation sera mis à jour ce qui signifie que des exécutions peuvent être incorrecte.

Par contre, dans les scénarios ou la thread principale associée au main crée des threads fille (pthread_create[1]) elle doit ensuite attendre la terminaison des threads fille (pthread_join[2]) pour terminer l'exécution de l'application.

Vous effectuerez des mesures (option –m) de l'exécution pour les scénarios suivants : -t0, -t1, -t2 pour –p2 (4 personnes), -p4 (16 personnes), -p8 (256 personnes), **soit 9 mesures.**

Vous devrez rendre

- Dans une archive **tar gzip** (fichier tar compressé avec gzip) de nom `projet1-numero-groupe.tgz` se décompressant dans un répertoire de nom `projet1-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 `-t1 -p4 -m`
    - un fichier PDF de nom `numero-groupe.pdf` contenant votre rapport

Le rapport doit être rédiger comme un rapport et donc comporter outre les éléments attendus une introduction et une conclusion. Dans cette première étape, 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) ;
- analyser la correction de chacun des scénarios proposés.
- analyser de manière comparative les divers scénarios corrects proposés, cette analyse doit nécessairement utiliser les mesures effectuées.

Pour cette première phase la qualité du code ne sera pas évaluée mais il devra néanmoins se compiler et s'exécuter avec les options prévues.

[1] http://man7.org/linux/man-pages/man3/c.3.htm

[2] http://man7.org/linux/man-pages/man3/pthread_join.3.html

Quelques compléments de C

clock_t **clock**(void);

The **clock**() function determines the amount of processor time used since the invocation of the calling process, measured in CLOCKS_PER_SECs of a second.

La commande clock vous permet de calculer le temps CPU consommé.

time_t **time**(time_t *tloc);

The **time**() function returns the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time, without including leap seconds.

La commande time vous permet de calculer le temps utilisateur (ou temps de réponse).

Int **getrusage**(int who, struct rusage *r_usage);

getrusage() returns information describing the resources utilized by the current process, or all its terminated child processes. The who parameter is either RUSAGE_SELF or RUSAGE_CHILDREN. The buffer to which r_usage points will be filled in with the following structure:

```
struct rusage {
        struct timeval ru_utime; /* user time used */
        struct timeval ru_stime; /* system time used */
        long ru_maxrss;         /* max resident set size */
        long ru_ixrss;          /* integral shared text memory size */
        long ru_idrss;          /* integral unshared data size */
        long ru_isrss;          /* integral unshared stack size */
        long ru_minflt;         /* page reclaims */
        long ru_majflt;         /* page faults */
        long ru_nswap;          /* swaps */
        long ru_inblock;        /* block input operations */
        long ru_oublock;        /* block output operations */
        long ru_msgsnd;         /* messages sent */
        long ru_msgrcv;         /* messages received */
        long ru_nsignals;       /* signals received */
        long ru_nvcsw;          /* voluntary context switches */
        long ru_nivcsw;         /* involuntary context switches */
  };
```

Les deux premiers paramètres vous permettent de calculer le temps CPU consommé et le troisièmes l'empreinte maximale de votre programme.

Int **getopt**(int argc, char * const argv[], const char *optstring);

The **getopt**() function incrementally parses a command line argument list argv and returns the next known option character. An option character is known if it has been specified in the string of accepted option characters, optstring.

La commande getopt vous permet d'analyser les paramètres utiliser lors du 'lancement' du processus.