Rapport du projet de Programmation concurrente

Rendu du 14 novembre 2016

Introduction au projet

La programmation concurrente est un paradigme de programmation informatique dans lequel plusieurs processus, threads ou tâches sont exécutés pendant des périodes de temps se chevauchant simultanément au lieu de séquentiellement.

Ce paradigme est utilisé pour modéliser des systèmes dans lequel les processus accèdent aux ressources de manière simultanée. Il est utilisé pour écrire des modèles qui interagissent avec le monde réel, qui lui est aussi concurrent. Il est donc très utiles mais pour autant il est très complexe à mettre en place. L'un des problèmes principaux de la programmation concurrente est l'indéterminisme de l'ordre d'exécution. Cela mène à la situation de compétition (Race condition) ou d'autres problèmes comme les interblocages (Deadlocks) ou encore un problème de famine (Starvation)

Afin d'aborder ces problèmes et de mieux les comprendre nous devons accomplir un projet. La première phase nécessite la mise en œuvre générale avec des threads POSIX sans synchronisation. La deuxième mettra en œuvre la synchronisation des variables à l'aide des conditions POSIX et la dernière implémentera la synchronisation des variables à l'aide des sémaphores POSIX.

Pour utiliser ces outils, nous devons modéliser le déplacement de personnes sur un terrain avec une taille et des obstacles prédéfinis. Les personnes doivent se déplacer vers un même point, le point de sortie, mais ne peuvent être en même temps sur le même endroit. Le programme permet à l'utilisateur de choisir le nombre de personnes présentes sur le terrain. L'utilisateur à également le choix sur le nombre de threads :

- Un seul qui fera avancer les personnes successivement
- Quatre : le terrain est découpé en quatre partie et chaque thread gère l'avancement des personnes sur sa partie
- Un thread par personne.

Dans la deuxième phase du projet nous devons mettre en place une synchronisation par sémaphores des ressources partagées.

Algorithme de déplacement d’une personne

Le déplacement des personnes sur le terrain est une action centrale au sein de ce projet. Nous devons par conséquent, essayer de comprendre les comportements des personnes afin qu'ils rejoignent au plus vite le deuxième muret.

Une personne peut se déplacer dans les huit directions. A savoir vers le haut, vers le bas, sur les côtés et sur les 4 diagonales. Il faut trouver un algorithme permettant de simuler le déplacement de personnes.

Pour que toutes les personnes puissent se diriger vers la « sortie » (le point azimut) , nous avons du mettre en place un algorithme simple. Nous devons faire avancer les personnes en fonction de leurs voisins et en fonction de leur hauteur (sur l'axe vertical) par rapport à l'azimut.

Notre algorithme de déplacement d'une personne sur le terrain :

```
ALGORITHME avancerUnePersonne

      personne :p
      terrain :t

 DEBUT

      SI (p est arrivé à destination) ALORS
            enlever p dans la liste de personnes de t
      SINON
            SI (p est dessus du milieu vertical du premier muret) ALORS
                  déplacer p dans la direction Sud-ouest
            SINON
                  SI (p est au-dessous du milieu vertical du second muret) ALORS
                        déplacer p dans la direction Nord-ouest
                  SINON
                        déplacer p dans la direction Ouest
                  FIN_SI
            FIN_SI
      FIN_SI

 FIN
```

Pour comprendre le déplacement des personnes présents sur le terrain, nous devons détailler les fonctions qui en découlent :

```
ALGORITHME déplacer dans la direction Sud-ouest

      personne :p
      terrain :t

 DEBUT

      SI (case au Sud-ouest de p est libre) ALORS
            se déplacer vers la case sud-ouest
```

```
      SINON
            se déplacer vers la case Sud
      FIN_SI

 FIN

ALGORITHME déplacer dans la direction Nord-ouest

      personne :p
      terrain :t

 DEBUT

      SI (case au Nord-ouest de p est libre) ALORS
            se déplacer vers la case Nord-ouest
      SINON
            se déplacer vers la case Nord
      FIN_SI

 FIN

ALGORITHME déplacer dans la direction Ouest

      personne :p
      terrain :t

 DEBUT

      SI (case Ouest de p est libre) ALORS
            se déplacer vers la case Ouest
      SINON
            Ne pas se déplacer
      FIN_SI

 FIN



ALGORITHME déplacer dans la direction Sud

      personne :p
      terrain :t

 DEBUT

      SI (case Sud de p est libre) ALORS
            se déplacer vers la case Sud
      SINON
            Ne pas se déplacer
      FIN_SI

 FIN



ALGORITHME déplacer dans la direction Nord
```

```
      personne :p
      terrain :t

DEBUT

      SI (case Nord de p est libre) ALORS
            se déplacer vers la case Nord
      SINON
            Ne pas se déplacer
      FIN_SI

FIN
```

Threads Java versus Threads POSIX

Cette partie permet de montrer les différences de conception entre les threads Java et les Pthreads (threads POSIX).

Comme nous l'avons vu l'année dernière en SI3, un thread Java désigne un « fil d'exécution » dans un programme, et comme nous avons vu que le langage Java est multi-thread, nous pouvons faire vivre plusieurs fils d'exécution de façon indépendante. C'est uniquement depuis la version du JDK 5 que nous disposons du package *java.util.conccurent* qui permet la gestion de la concurrence.

Voici un tableau comparatif des outils utilisés pour la thread et la sémaphore entre le java et POSIX :

	Java	POSIX
Créer une thread	1. créer une classe qui hérite de la classe *java.lang.Thread et* redéfinir la méthode *run()*. Pour lancer ce thread de cette manière il faut créer l'objet et invoquer la méthode *Objet.start()* invoquant elle même la méthode run() que nous avons redéfinit. 2. Nous pouvons également créer une classe qui implémente l'interface *java.lang.Runnable* où il faudra toujours redéfinir la méthode *run()* . Pour lancer la classe : MyObject code = new MyObject() /*du code*/	#include <pthread.h> int pthread_create(pthread_t * thread, pthread_attr_t * attr,void *(*start_routine) (void *), void *arg);

	Thread tache = new Thread(code); où la classe MyObject implémente *Runnable*.	
Attendre la fin d'un thread	*thread.join()*	int pthread_join(pthread_t th, void **thread_return);
Destruction d'une thread	La destruction d'un thread Java intervient lorsque la méthode *run()* se termine.	void pthread_exit(void *ret);

Suspendre une thread	Thread.interrupt()
	
	[Cette méthode peut lever une exception si le processus est bloqué par une opération de synchronisation (Thread.join ou Object.wait…).]
		pthread_mutex_lock(&stopMutex);
		
		/*code */
		
		pthread_mutex_unlock(&stopMutex);

Initialiser une sémaphore	Semaphore sample = new Semaphore(1, true); //if true, fair semaphore	int sem_init(sem_t *semaphore, int pshared, unsigned int valeur)

Prendre /laisser la sémaphore	sample.acquire();
	//equivalent of down
	
	sample.release();
	// equivalent to up
		int sem_wait(sem_t *semaphore);
		
		Opération down sur un sémaphore.
		
		int sem_post(sem_t*semaphore);
		
		Opération up sur un sémaphore.

Détruire une sémaphore	Détruit par le Garbage Collector	Int sem_destroy(sem_t *semaphore);

Algorithme des threads du programme

L’enjeu principale du programme est de bien gérer le déplacement des personnes sur le terrain. Le problème majeur à gérer est le fait que deux personnes ne peuvent se trouver au même endroit au même instant. La bonne manipulation des threads est primordiale. En effet, une thread peut par exemple déplacer une personne sur une case à priori vide, et une autre thread n’étant pas encore notifié de l’occupation de cette case déplacera à son tour une personne sur cette même case. Ce qui créera un conflit. D’autres problématiques comme la condition d’arrêt d’une thread doivent être traités.

Algorithme avec 4 threads

Pour rappel, le terrain est divisé en quatre partie de même superficie et le programme principal lance une thread qui gérera les déplacements pour chaque partie du terrain.

Pour faciliter la compréhension des explications, les parties du terrain se trouvant vers le haut commenceront par « Nord » (inversement « Sud »), et les parties se trouvant à gauche se termineront par « Ouest » (inversement « Est »).

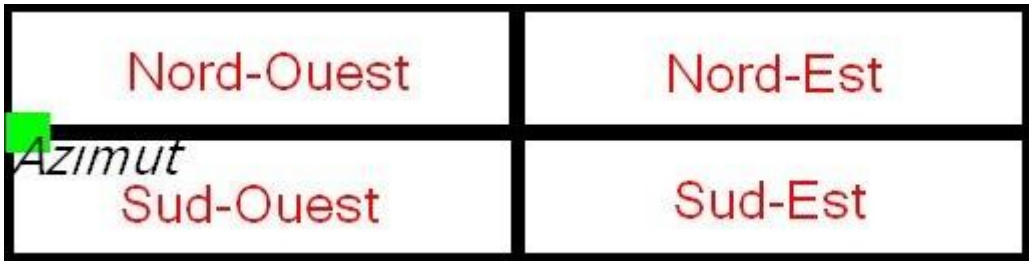


Illustration 1: Illustration des zones du terrain avec l'azimut

Le terrain est divisé en 4 parties. L'idée est d'avoir une thread principale (le main) qui crée 4 thread et chaque thread gérera les déplacements des personnes présentes sur sa partie.

(i) Sans sémaphores POSIX

Pour le lancement des threads :

debut lancement

Contexte ← terrain

pour chaque zone **dans** terrain

créer Thread(Contexte)

attendre_fin Thread

fin pour chaque

fin lancement

Algorithme des thread :

debut dans thread

tant que « il y a des personnes sur le terrain »

pour chaque personne **dans** terrain

si personne est dans la zone du thread

faire avancer la personne

fin si

fin pour chaque

fin tant que

fin dans thread

On remarque que le programme va vérifier pour chaque personne du terrain si elle est bien présente dans la zone que gère la thread pour la faire avancer ou non.

Une autre manière de faire serait d'avoir une liste de personne présente dans chaque zone. Ces listes seraient initialisées à l'initialisation du terrain.

Si le déplacement d'un personne l'a fait sortir de la zone, le programme la supprimerait de la liste de la zone et l'insérer dans la liste de sa nouvelle zone. Cette implémentation permettrait d'éviter des parcours inutiles sur toutes les personnes du terrain car on ne parcourrait que la liste des personnes présentes dans notre zone pour les faire avancer.

(ii) Avec sémaphores POSIX

Pour le lancement des threads:

debut lancement

pour chaque zone **dans** terrain

créer mutex

inserer mutex dans ListeMutex

créer sémaphore privée

fin pour chaque

pour chaque zone **dans** terrain

Contexte ← terrain, ListeMutex, sémaphore privée

créer Thread(Contexte)

fin pour chaque

pour chaque zone **dans** terrain

appeler down sur sémaphore privée

detruire semaphore privé et mutex

fin pour chaque

fin lancement

La création d'un sémaphore privée pour chaque zone est utilée pour attendre la terminaison de chaque thread.

Algorithme des thread:

debut dans thread

tant que « il y a des personnes sur le terrain »

pour chaque personne **dans** terrain

si personne est dans la zone du thread

si personne n'est proche d'aucune zone voisine où le deplacement est possible*

appeler down sur mutex de la zone du thread

faire avancer personne

appeler up sur mutex de la zone du thread

fin si

sinon

pour chaque zone **dans** zone voisine où le deplacement est possible*

appeler down sur mutex de la zone du thread

appeler down sur mutex de la zone voisine

faire avancer personne

appeler up sur mutex de la zone voisine

appeler up sur mutex de la zone du thread

fin pour chaque

fin sinon

fin tant que

appeler up sur la semaphore privée **

detruire la sémaphore privée

fin dans thread

* Conformément à notre algoritme de déplacement d'une personne, quand je suis dans une zone je ne peux me déplacer dans toutes les voisines. Par exemple une personne présente dans la zone Nord-Ouest ne pour effectuer un déplacement l'emmenant en zone Sud-Est.

** Cet appel permet de signaler au programme appelant que la thread est terminée

Algorithme avec une thread associée à chaque personne

(i) Sans sémaphores POSIX

Pour le lancement des threads:

début lancement

pour chaque personne **dans** terrain

Contexte ← terrain , personne

créer Thread(Contexte)

ajouter thread à la liste de thread

fin pour chaque

pour chaque thread **dans** la liste de thread

attendre thread

fin pour chaque

fin lancement

Algorithme des threads:

debut dans thread

tant que « la personne n'est pas arrivée »

faire avancer la personne

fin tant que

fin dans thread

(ii) Avec sémaphores POSIX

Pour le lancement des threads:

début lancement

variable globale mutex

pour chaque personne **dans** terrain

Creer sémaphore privée

Contexte ← terrain , personne,mutex,sémaphore privée

créer Thread(Contexte)

ajouter sémaphore privée à la liste des sémaphores privées

fin pour chaque

pour chaque sémaphore privée **dans** la liste des sémaphores privées

appeler down sur la sémaphore privée

detruire la sémaphore privée

fin pour chaque

fin lancement

Algorithme des threads:

debut dans thread

tant que « la personne n'est pas arrivée »

appeler down sur mutex

faire avancer la personne

appeler up sur mutex

fin tant que

appeler up sur la sémaphore privée

fin dans thread

Avec cette implémentation nous bloquons le tout le terrain à chaque fois que l'on veut faire avancer une personne. Il n'y a donc plus de parallélisme. Mais ce ceci assure que le programme est correcte dans le sens où aucune personne ne se deplacera sur une case déjà occupé.
Cependant, nous avons pensé pour la suite associer une sémaphore par case, pour avoir du parallélisme :

debut déplacement

appel de down sur ma case

appel de downsur la case ou je vais me déplacer

la personne avance

appel de up sur la case où je suis

appel de up sur mon ancienne case

fin déplacement

Remplacer pthread_join

Pour la deuxième étape il fallait remplacer pthread_join, qui a 2 fonctionnalités :

- La première est d'attendre la fin du thread qui lui est donnée en paramètre
- La seconde est désallouer l'espace mémoire allouer dans la thread

Pour remplacer cette fonction nous avons utiliser une sémaphore initialisée à 0 (sémaphore privée) dans la thread principale. Une fois la thread lancée on fait un down sur cette sémaphore privée. Comme l'appel est bloquant la thread principale n'ira pas jusqu'à la fin de son bloc d'instruction sans avoir obtenu cette sémaphore. L'appel de up de la sémaphore privée est donc fait à la fin de la thread fille associée.

Analyse des scénarios

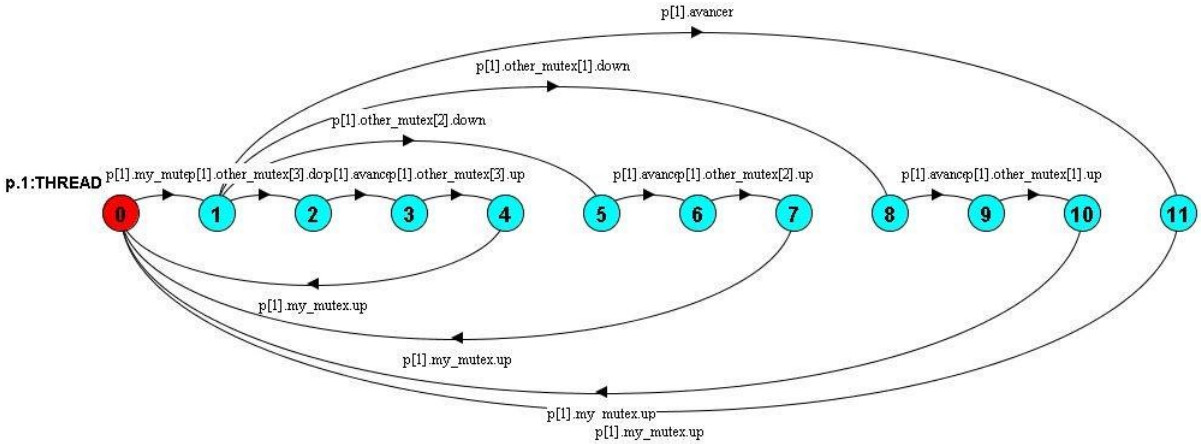
Partie FSP

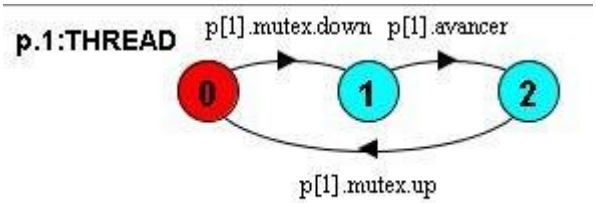
Nous pouvons établir des schéma FSP de nos algorithmes en t1 et t2

premier FSP pour -t1 :

```
const Max = 1
range Int = 0..Max
const NBPERSONNE=8

THREAD = (my_mutex.down->AVANCER),
AVANCER = (avancer ->my_mutex.up-> THREAD
          |other_mutex[i:1..3].down->avancer->other_mutex[i].up->my_mutex.up->THREAD).
SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up->SEMA[v+1]
              |when(v>0) down->SEMA[v-1]),
SEMA[Max+1] = ERROR.
||SEMADEMO = (p[1..NBPERSONNE]:THREAD || {p[1..NBPERSONNE]}::mutex:SEMAPHORE(1)).
```

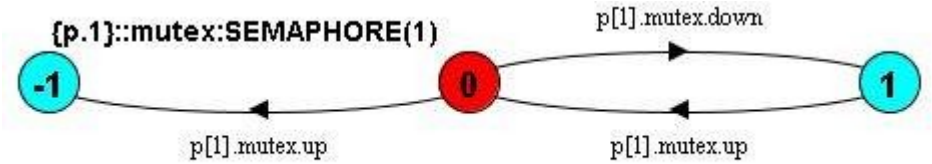


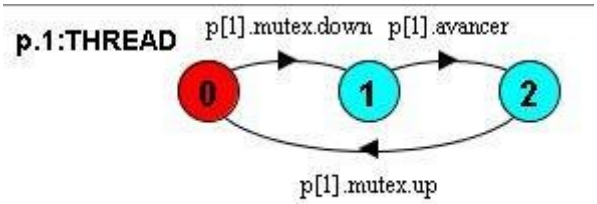


Et pour t2, voici le code et les schémas :

```
const Max = 1
range Int = 0..Max
const NBPERSONNE=8

THREAD      = (mutex.down->avancer->mutex.up->THREAD).
SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up->SEMA[v+1]
              |when(v>0) down->SEMA[v-1]),
SEMA[Max+1] = ERROR.
||RUN = (p[1..NBPERSONNE]:THREAD ||{p[1..1]}::mutex:SEMAPHORE(1)).
```





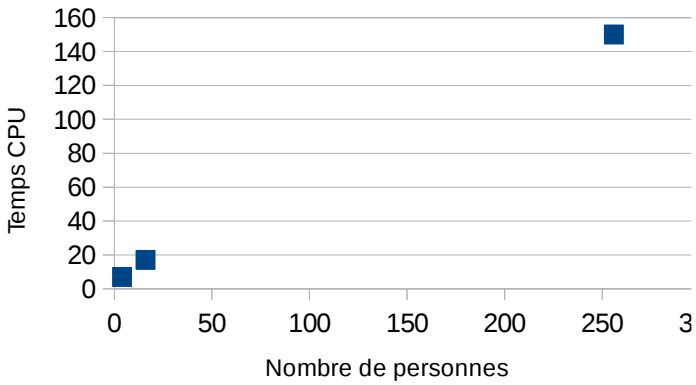
Mesures du programme

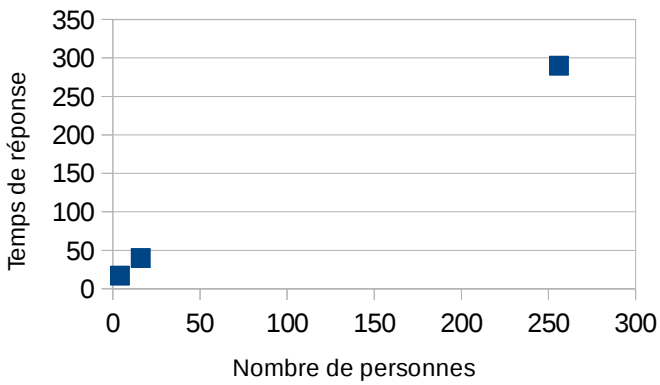
Il faut noter que l 'exécution complète du programme se bloquait souvent avec l'option -t1 et -e1. Le problème a été résolu. Cependant, depuis les options -t0 et -t2 ne fonctionnent plus.

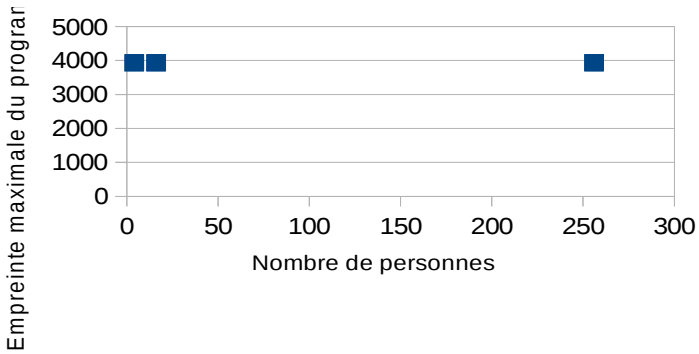
Pour l'option -t2, en -e1 le problème vient du fait que les personnes n'avancent plus donc que les threads ne se terminent jamais. Cela peut venir du fait que notre l'algorithme pour faire avancer les personnes sur le terrain fait que chaque personne est bloquée ; ou alors que le programme n'enregistre pas bien qu'une personne est arrivée.

En -e2, c'est une fuite de mémoire, avec l'aide de valgrind nous pouvons voir que cela vient de blocs mémoire non-libérés et initialisés par pthread_create. Comme pthread_join permet d'attendre la terminaison d'un thread et de supprimer les blocs mémoires allouer à l'interieur, nous pensons qu'ils s'agit de mémoire allouer dans le thread non libérée.

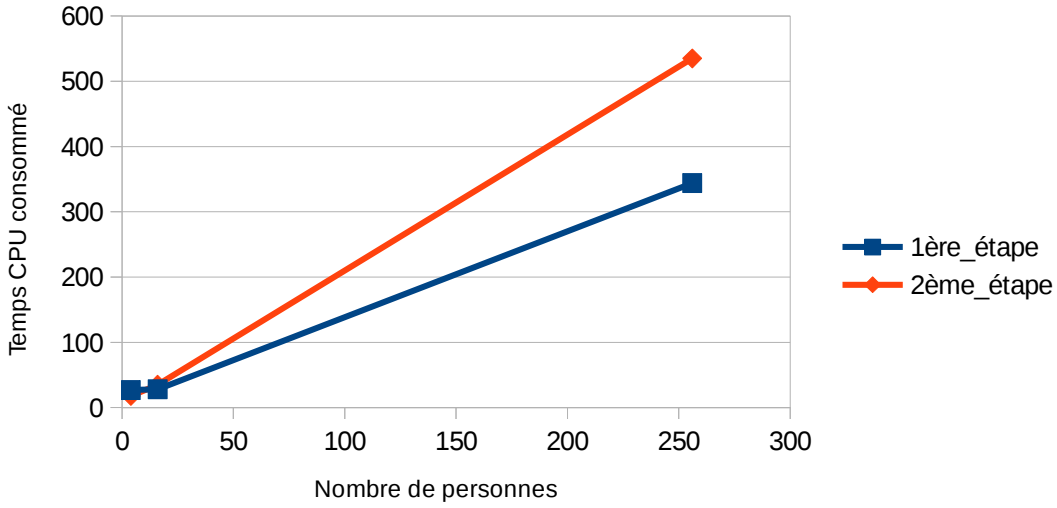
Mesures avec un thread :

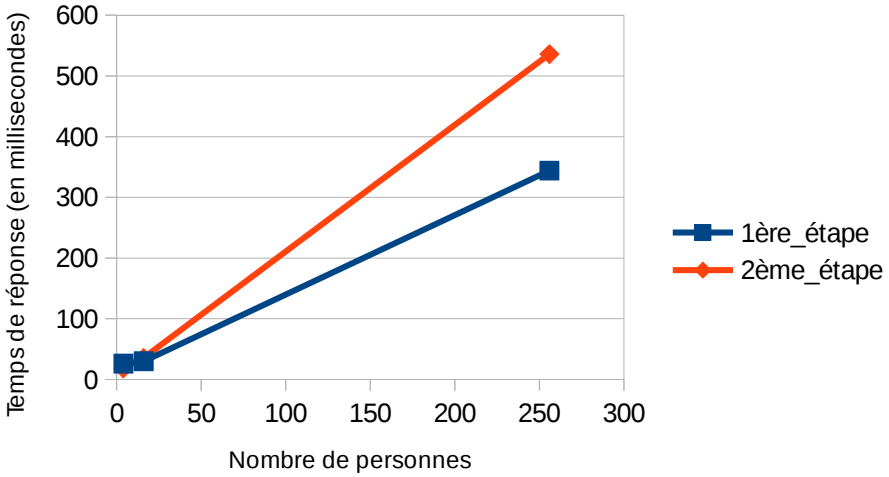


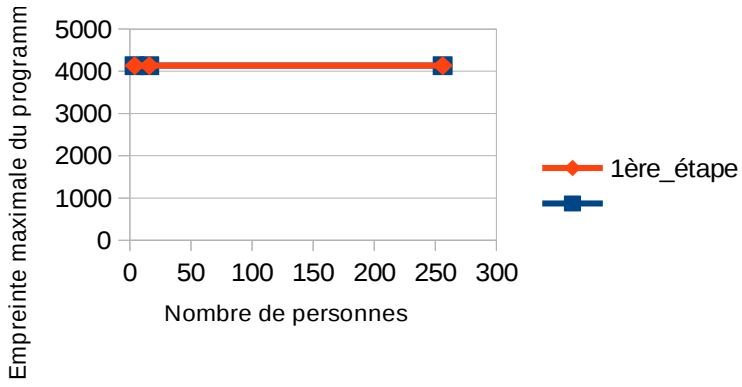




Mesures avec un thread qui gère chaque quart du terrain :







Mesures avec un thread pour chaque personne :

Nombres de personnes	Temps CPU consommé *(en millisecondes)*		Temps de réponse *(en millisecondes)*	Empreinte maximale du programme
4	7		7	5770
16	33		32,25	8028
256				

-e2			
4	22,66	22,33	6133
16	24	24,2	6490
256	?	?	?

Le manque de données nous permet difficilement de tirer des conclusions. Cependant pour l'option -t0 on observe que l'empreinte maximale du programme ne varie pas en fonction du nombre de personne. Ainsi que le temps CPU consommé et le temps utilisateur suivent approximativement la même évolution en fonction du nombre de personnes.
Inversement, avec une seul thread par personne l'empreinte du programme varie en fonction du nombre de personne et le temps CPU augmente bien plus vite que le temps utilisateur en fonction du nombre de personnes.

Le temps CPU est l'addition du temps utilisateur et du temps système. Nous pouvons constater que pour l'option -t0 le temps système est très court, ce qui suppose que le temps pour exécuter des appels systèmes par le noyau est quasiment nul.

Nous constatons également que l'option -t2 demande en revanche plus d'appels systèmes pour créer les threads.

Analyse des mises en œuvre des scénarios

Le simulateur dans l'étape 1 ne présente aucune synchronisation ce qui présente un risque dans l'utilisation de donnée partagée (le terrain). Notre programme ne respecte pas les normes que nous avons étudiées en cours. L'utilisation de sémaphore dans la seconde étape permet de pallier au problème de synchronisation. Outre les problèmes d'interblocages qu'ils peuvent provoquer, les sémaphores ne protègent pas les programmeurs de l'erreur courante qui consiste à bloquer par un processus un sémaphore qui est déjà bloqué par ce même processus, et d'oublier de libérer un sémaphore qui a été bloqué

Conclusion

Durant la première phase du projet, la partie la plus délicate représente la manipulation des threads avec une seule ressource partagée : la liste de personnes présentes sur le terrain. Nous avons bien identifié la difficulté des threads et de la conception de nos classes afin de répondre aux exigences de cette première phase.

Nous pouvons déterminer de manière évidente que les options augmentent en difficulté.

Nous pouvons assurer que le code n'est pas dans une version optimale et de multiples corrections seront faites afin de délivrer un programme soigné et efficace à la fin du projet.

Cependant, il nous manque certains outils pour synchroniser nos données partagées afin de ne pas avoir des incohérences. Nous devons synchroniser nos données pour pouvoir gérer de façon efficace l’utilisation de threads sur le terrain.

Les programmes FSP nous permettent de d'affirmer que l'algo est correcte mais la mise en pratique des algorithmes pour -t1 et -t2 nous empêche de l'affirmer.