INTRODUCTION AUX SYSTEMES D'EXPLOITATION

TD4
Threads POSIX

S O M M A I R E

1.Rappel sur les threads

1.1. Présentation

Sous Unix, le programmeur dispose d'une bibliothèque de gestion des activités : la bibliothèque normalisée Posix. Cette bibliothèque se nomme `libpthread.a`. Il faut donc compiler vos programmes avec l'option **`-lpthread`**. Certaines fonctions concernant l'ordonnanceur (comme `sched_yield`) sont disponibles dans une autre bibliothèque : `librt.a`. Dans ce cas, il faut indiquer en plus **`-lrt`**.

1.2. Partie 1 : précautions

Les threads sont des activités concurrentes accédant à un espace d'adressage commun, il est donc nécessaire de les synchroniser avec des sémaphores de type `pthread_mutex_t` (voir plus loin).
Toujours à cause de l'espace d'adressage commun, un autre problème se pose. Il faut veiller à écrire des fonctions **réentrantes**, c'est à dire des fonctions sans donnée statique.

Rappel : une donnée statique dans une fonction existe en 1 seul exemplaire durant toute la durée de l'exécution du programme.

Exemple de fonction non réentrante

Problème : Contenu de buffer indéterminé si invocations simultanées

```
const char * writeHexa(int i)
{
      static char buffer[0x10];
      sprintf(buffer,"0x%.8x",i);
      return(buffer);
}
```

Version réentrante de cette fonction. Chaque invocation utilise des données différentes (transmises)

```
char * writeHexa_r(int i,char * buffer)
{
      sprintf(buffer,"0x%.8x",i);
      return(buffer);
}
```

Choix de primitives systèmes réentrantes

- Fonction ayant l'extention `_r` (Posix.1c)
- Ex : `asctime()` et `asctime_r()`

1.3. Partie 2 : Création, terminaison, attente

1.3.1. Création

La fonction **`pthread_create`** crée et initialise une nouvelle activité.

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
                   void * (*routine)(void *), void *arg);
```

En entrée, la fonction reçoit :

- l'adresse d'un `pthread_t`, `pthread_t` qui contiendra l'identifiant du thread créé,
- l'adresse d'un `pthread_attr_t`, `pthread attr_t` qui contient des attributs de création du thread ; pour un usage « courant », on peut indiquer NULL,
- l'adresse de la fonction contenant le programme exécute dans l'activité,
- un argument `arg` indiqué sous la forme d'un pointeur sur une zone mémoire ; ce pointeur sera transmis à la fonction exécutée par le thread.
- Cette fonction `pthread_create` retourne un code éventuel d'erreur de création et 0 en cas de réussite.

1.3.2. Terminaison

La fonction `pthread_exit` provoque l'arrêt de l'activité qui l'exécute.

```
#include <pthread.h>

void pthread_exit(void * value_ptr);
```

En entrée, cette fonction reçoit une adresse qui sera transmise (on peut indiquer NULL) à une autre activité attendant la fin de l'activité courante (voir `pthread_join`).
Remarque : la fin normale (avec return) de la fonction exécutée par l'activité est équivalente à un `pthread_exit`.

1.3.3. Attente

La fonction `pthread_join`, fonction de synchronisation, permet de bloquer une activité en attente de la fin d'une autre activité.

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **status);
```

En entrée, la fonction reçoit :

- l'identifiant du thread à attendre,
- l'adresse d'un pointeur de pointeur permettant de récupérer l'adresse indiquée lors du `pthread_exit` (ou du return) afin d'accèder à la valeur de retour.

1.4. Partie 3 : synchronisation

Les threads étant des activités concurrentes, il faut les synchroniser vis-à-vis de l'accès à des variables partagées. Il faut utiliser des sémaphores binaires => `pthread_mutex_t`.
Il peut également être intéressant d'avoir un mécanisme permettant à un thread d'avertir d'autres threads, qu'il a effectué un changement de la valeur d'une variable partagée => `pthread_cond_t`.

Signalons qu'une variable condition de type `pthread_cond_t` est une variable partagée par les différents threads. Il faut donc un `pthread_mutex_t` associé pour en protéger l'accès.

1.4.1. Les sémaphores d'exclusion mutuelle

Création d'un verrou (`man 3 pthread_mutex_init`)

- Type : `pthread_mutex_t`
- Initialisation statique :
  - `pthread_mutex_t myMutex=PTHREAD_MUTEX_INITIALIZER;`
- Initialisation dynamique :
  -
```
int pthread_mutex_init(pthread_mutex_t * mtx,
                       pthread_mutexattr_t * attr);
```

- Généralement `attr` est nul (initialisation par défaut)

Destruction d'un verrou (man 3 pthread_mutex_destroy)
- `int pthread_mutex_destroy(pthread_mutex_t * mtx);`
- Le verrou n'est plus utilisable
- Retour : 0 si ok, non nul si erreur
- Erreur si le verrou est monopolisé → erreur `EBUSY`

Opération de verrouillage (man 3 pthread_mutex_lock)
- `int pthread_mutex_lock(pthread_mutex_t * mtx);`
- Si le verrou est libre :
  - il est monopolisé par le thread courant
  - le thread courant poursuit son traitement
- Si le verrou n'est pas libre
  - le thread courant est bloqué jusqu'à la libération du verrou
  - Retour : 0 si ok, non nul si erreur

Opération de déverrouillage (`man 3 pthread_mutex_unlock`)
- `int pthread_mutex_unlock(pthread_mutex_t * mtx);`
- Libère le verrou
- Si des threads sont bloqués en attente sur ce verrou
  - l'un d'eux est débloqué et monopolise le verrou
- Retour : 0 si ok, non nul si erreur

Tentative de verrouillage (`man 3 pthread_mutex_trylock`)
- `int pthread_mutex_trylock(pthread_mutex_t * mtx);`
- Si le verrou est libre
  - il est monopolisé par cet appel qui retourne 0
- Si le verrou n'est pas libre
  - cet appel retourne immédiatement un résultat non nul
  - il ne faut pas utiliser les données protégées par le verrou systématiquement après cet appel

1.4.2. Les variables conditions : `pthread_cond_t`

Création d'une condition (`man 3 pthread_cond_init`)
- Type : `pthread_cond_t` (doit être associé à un mutex)
- Initialisation statique
  - `pthread_cond_t myCond=PTHREAD_COND_INITIALIZER;`
- Initialisation dynamique
  - `int pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t * attr);`
- Généralement attr est nul (initialisation par défaut).

Destruction d'une condition (`man 3 pthread_cond_destroy`)
- `int pthread_cond_destroy(pthread_cond_t * cond);`
- La condition n'est plus utilisable
- Retour : 0 si ok, non nul si erreur
- Erreur si la condition est utilisée, erreur `EBUSY`

Attente d'une condition (`man 3 pthread_cond_wait`)
- `int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mtx);`
- Bloque le thread courant
  - débloqué quand une modification est signalée sur `cond`
  - évite l'attente active
- `cond` n'a aucune valeur logique (sert à la synchronisation)

- `mtx` doit être monopolisé avant et libéré après
- Interruptible par les signaux → relance

Démarche usuelle

```
pthread_mutex_lock(&mtx);
while(!conditionIsSatisfied())
      pthread_cond_wait(&cond,&mtx);
pthread_mutex_unlock(&mtx);
```

Attente temporisée d'une condition (`man 3 pthread_cond_timedwait`)

- `int pthread_cond_timedwait(pthread_cond_t * cond, pthread_mutex_t * mtx, const struct timespec * date);`
- Même principe que `pthread_cond_wait()`
- Renvoie `ETIMEDOUT` si `date` est dépassée
- `date` est une limite, pas un délai !
  - Prendre la date courante (`time(), gettimeofday()`)
  - Ajouter un délai (structure `timespec`)
  - Basé sur le temps universel

Signaler une condition (`man 3 pthread_cond_broadcast`)

- int `pthread_cond_broadcast(pthread_cond_t * cond);`
- Débloque tous les threads attendant la condition `cond`
- Le verrou associé à `cond` doit être monopolisé avant et libéré après

Démarche usuelle :

```
pthread_mutex_lock(&mtx);
/* make this condition become true */
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mtx);
```

- `int pthread_cond_signal(pthread_cond_t * cond);`
  - Ne débloque qu'un thread parmi ceux qui attendent `cond`
- Un peu plus efficace que `pthread_cond_broadcast`
- Bien moins général (incohérence si plusieurs threads en attente !)

2.Exercice N°1

Ci-dessous un premier programme permettant de se familiariser avec l'utilisation des threads POSIX. Ce programme est issu du (très bon) livre de C. Blaess « Développement système sous Linux ».

2.1. Analyse du programme

Analysez puis exécutez ce programme. N'oubliez pas qu'Il faut compiler vos programmes avec l'option « **`-lpthread`** ».
Vous pouvez modifier la durée des `sleep` (voire utiliser usleep() avec des valeurs différentes afin de voir l'impact de l'ordonnancement).

```
// ------------------------------------------------------------------
// exemple-pthread-create-1.c
// Fichier d'exemple du livre "Developpement Systeme sous Linux"
```

```
// (C) 2000-2019 - Christophe BLAESS <christophe@blaess.fr>
// https://www.blaess.fr/christophe/
// -----------------------------------------------------------------------

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_function(void *arg)
{
      while (1) {
            fprintf(stderr, "New thread\n");
            sleep(1);
      }
}


int main (void)
{
      pthread_t thr;

      if (pthread_create(&thr, NULL, thread_function, NULL) != 0) {
            fprintf(stderr, "Error during pthread_create()\n");
            exit(EXIT_FAILURE);
      }
      while (1) {
            fprintf(stderr, "Main thread\n");
            sleep(1);
      }
}
```

2.2. Passage d'argument

Nous souhaions reprendre le programme précédent et créer plusieurs threads (en utilisant une boucle), disons 5 threads. Le père enverra à chaque thread l'indice de la boucle en argument de l'appel à `pthread_create`.

Afficher l'argument récupéré par chaque thread. Que constatez vous ? comment réctifier ce comportement ?

2.3. Fin d'un thread

Voici quelques tests à réaliser afin de comprendre le fonctionnement de la terminaison d'un thread :

1. Terminez les threads avec un appel à exit(), que se passe-t-il ?
2. Utilisez `pthread_exit(NULL)` cette fois-ci, que se passe-t-il ?
3. Même question dans le programme principal, que se passe-t-il lorsqu'un `exit()` est réalisé et lorsqu'un appel à `pthread_exit(NULL)` est fait ?

Nous souhaitons à présent avoir la possiblité de renvoyer une valeur via `pthread_exit()` au père. Faites un calcul quelconque dans le fils et renvoyez la valeur au père (qui la récupérera via un `pthread_join()`) et assurez vous que vous récupérez la bonne valeur pour chaque thread.

2.4. Utilisation de Mutex

Dans cet exercice, nous reprenons le code précédent. Nous souhaitons que chaque thread puisse utiliser le flux `stdout` (sortie standard) de manière exclusive.

- Chaque thread exécutera une boucle avec un nombre d'itérations aléatoire (entre 5 et 10)
- Chaque thread demande l'accès au flux de sortie standard en utilisant un Mutex
- Chaque thread s'endort pendant une durée aléatoire entre 1 et 3 secondes dans la section critique (pour éviter une séquence dans l'ordre croissant pour l'exécution des threads).

- Chaque thread affiche l'instant ou il a pris le Mutex et l'instant ou il l'a relaché (l'identité du thread est à afficher aussi, le numéro passé en paramètre est suffisant). L'utilisation de la fonction time() est suffisante ici.

3.Exercice N°2 : la factorielle

1. Ecrire un programme `calculfactorielleLocal.c` qui crée deux threads, pour calculer indépendamment la factorielle de 2 nombres.
   Chaque activité exécute le code décrit dans la fonction `calculerFactorielle`. Cette fonction reçoit en paramètre l'adresse d'un entier pour lequel la factorielle doit être calculée (vous pouvez utiliser une version itérative du calcul de la factorielle).
   Le calcul est effectué dans une variable locale de l'activité.
   Le résultat est communiqué au thread principal via l'adresse d'une zone allouée dynamiquement (avec un `malloc`) dans `calculerFactorielle`. Cette zone doit bien sûr être libérée (avec un `free`) dans le thread principal une fois que le résultat est affiché par celui-ci.

2. Modifier le programme précédent pour que les calculs soient effectués dans **une variable globale** ; les activités se terminent alors sans argument de sortie. Ecrire pour cela un programme `calculfactorielleGlobal.c`. Est-ce que ça marche ? Comment peut-ont résoudre le problème ?

4. Exercice 3 : synchronisation

Nous souhaitons ici créer des threads, synchroniser leurs exécutions et attendre leurs fins.

4.1. Produit scalaire multi-threads

Nous souhaitons réaliser le produit scalaire de deux vecteurs à l'aide d'une multitude de threads. Si les vecteurs sont de dimension N il y aura N threads dédiés aux multiplications et un thread dédié à la somme.

Lorsque les deux vecteurs sont initialisés par l'activité principale, les threads de multiplication peuvent effectuer leurs calculs. Lorsque ces multiplications ont toutes été effectuées, le thread d'addition peut commencer son calcul. Quand cette somme est calculée l'activité principale est alors en mesure d'afficher le résultat.

Nous souhaitons de plus que notre programme soit capable de réaliser plusieurs produits scalaires les uns à la suite des autres (sur des vecteurs de même taille). Les threads de multiplication et d'addition seront donc créés une fois pour toutes au début du programme et tourneront en boucle pour chaque nouveau produit scalaire à calculer.

La synchronisation des différentes activités (threads + main()) aura lieu autour d'une unique structure `Product` contenant les données suivantes :

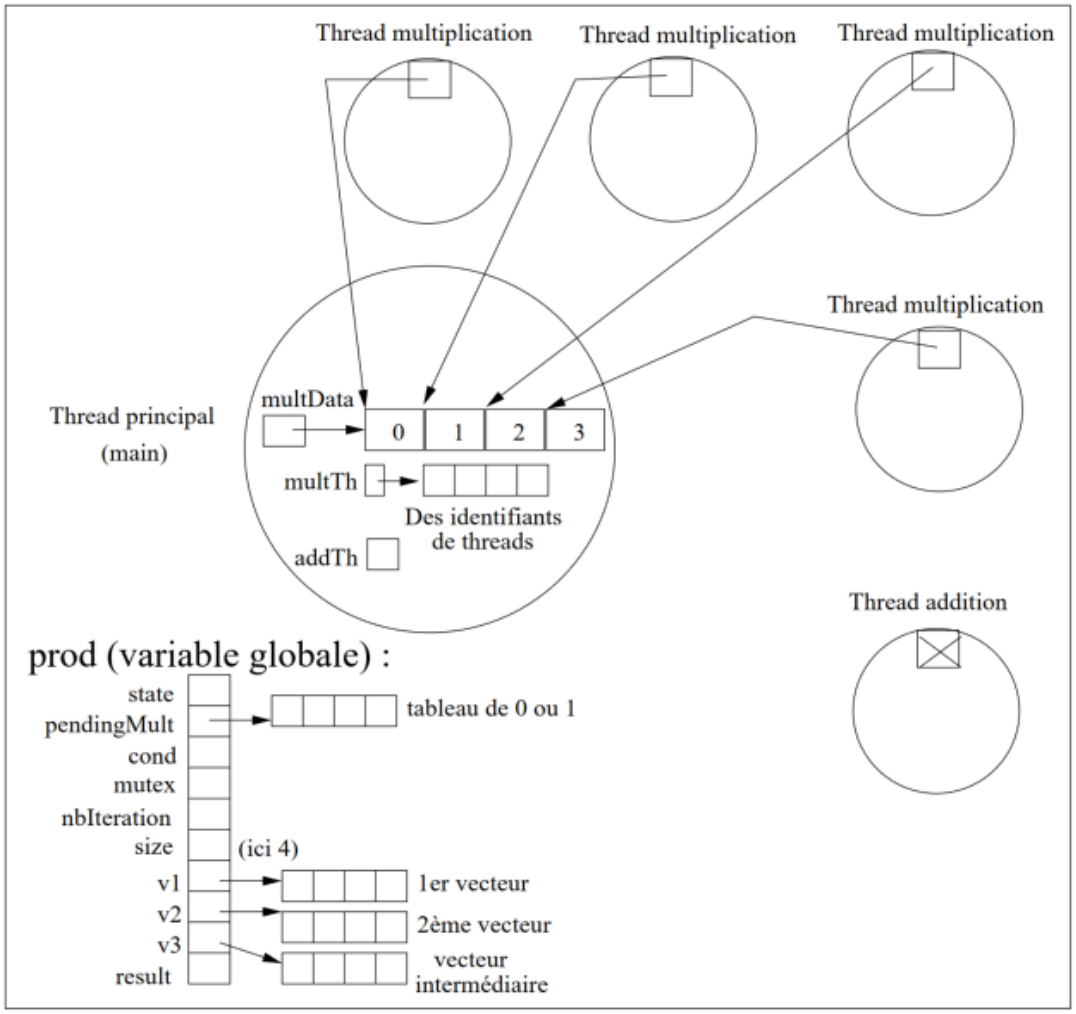
- `state` : l'étape au sein du produit scalaire courant
- `pendingMult` : les multiplications non terminées
- `cond` : la variable condition synchronisant les évolutions des champs précédents
- `mutex` : le verrou associé à la condition précédente
- `nbIteration` : le nombre d'iterations à effectuer
- `size` : la taille des vecteurs
- `v1` et `v2` : les vecteurs dont il faut calculer le produit scalaire
- `v3` : le vecteur contenant les résultats des multiplications
- `result` : le résultat du produit scalaire

L'état du produit scalaire courant est représenté par les constantes suivantes :

- `STATE_WAIT` : début du programme
- `STATE_MULT` : les multiplications peuvent commencer
- `STATE_ADD` : l'addition peut commencer
- `STATE_PRINT` : l'affichage du résultat peut avoir lieu

Une instance de la structure Product est communiquée aux différents threads via une variable globale. Pour le thread d'addition, cette information est suffisante, mais pour les threads de multiplication, il est nécessaire d'indiquer à quel index des vecteurs le thread concerné doit calculer.

Il faudra passer au thread concerné l'adresse d'un entier contenant l'index. Il doit exister un entier différent pour chaque thread de multiplication.



Les opérations de multiplication et d'addition ayant une durée très courte, nous pourrons éventuellement simuler de longs calculs en utilisant la fonction `wasteTime()`. Celle-ci réalise une attente active afin de ne pas perturber l'ordonnanceur par la mise en sommeil des threads.

Les algorithmes des traitements à effectuer sont donnés dans le code source à compléter. Il faudra veiller à bien attendre que toutes les multiplications soient terminées (fonction `nbPendingMult()`) avant de commencer l'opération d'addition. C'est le rôle du champ `pendingMult` qui doit être initialisé (fonction `initPendingMult()`) avant les multiplications et qui doit être modifié à chaque fois qu'une multiplication se termine. Les manipulations des champs `state` et `pendingMult` devront toujours avoir lieu sous le contrôle des champs `cond` et `mutex` sans quoi certaines modifications risquent de ne pas être détectées.

Exemple d'exécution :

```
$ ./produit
usage: ./produit nbIteration vectorSize
$ ./produit 3 4
Begin mult(0)
Begin mult(1)
Begin mult(2)
Begin mult(3)
Begin add()
--> mult(0)
--> mult(2)
--> mult(1)
--> mult(3)
<-- mult(2) : 0.632*1.51=0.956
<-- mult(0) : 0.751*2.16=1.63
<-- mult(1) : -4.14*-3.5=14.5
<-- mult(3) : -1.76*3.52=-6.19
--> add
<-- add
ITERATION 0, RESULT=10.9
--> mult(1)
--> mult(0)
--> mult(2)
--> mult(3)
<-- mult(2) : -3.47*-3.84=13.3
<-- mult(1) : 0.678*-3.11=-2.11
<-- mult(3) : -1.59*1.72=-2.75
<-- mult(0) : -0.863*2.48=-2.14
--> add
<-- add
ITERATION 1, RESULT=6.32
--> mult(3)
--> mult(1)
--> mult(2)
--> mult(0)
<-- mult(2) : 3.76*0.553=2.08
Quit mult(2)
<-- mult(0) : -3.64*-3.89=14.1
Quit mult(0)
<-- mult(3) : -0.181*4.62=-0.838
Quit mult(3)
<-- mult(1) : 4.8*2.65=12.7
--> add
Quit mult(1)
<-- add
ITERATION 2, RESULT=28.1
Quit add()
$
```

4.2. Squelette du programme

Le squelette du programme est disponible dans l'archive proposée.

```
#include <pthread.h> /* produit.c */
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>


/*********** Data Type ***********/
typedef enum
{
      STATE_WAIT,
      STATE_MULT,
      STATE_ADD,
      STATE_PRINT
} State;

typedef struct
{
      State state;
      int * pendingMult;
      pthread_cond_t cond;
      pthread_mutex_t mutex;
      size_t nbIterations;
      size_t size;
      double * v1;
      double * v2;
      double * v3;
      double result;
} Product;


/*********** Data structure ***********/
Product prod;
```

```
/*********** Function ***********/
void initPendingMult(Product * prod)
{
	size_t i;
	for(i=0;i<prod->size;i++)
	{
		prod->pendingMult[i]=1;
	}
}


int nbPendingMult(Product * prod)
{
	size_t i;
	int nb=0;
	for(i=0;i<prod->size;i++)
	{
		nb+=prod->pendingMult[i];
	}
	return nb;
}


void wasteTime(unsigned long ms)
{
	unsigned long t,t0;
	struct timeval tv;
	gettimeofday(&tv,(struct timezone *)0);
	t0=tv.tv_sec*1000LU+tv.tv_usec/1000LU;
	do
	{
		gettimeofday(&tv,(struct timezone *)0);
		t=tv.tv_sec*1000LU+tv.tv_usec/1000LU;
	} while(t-t0<ms);
}


/*****************************************************************************/
void * mult(void * data)
{
	size_t index;
	size_t iter;

	/*=>Recuperation de l'index, c'est a dire index = ... */


	fprintf(stderr,"Begin mult(%d)\n",index);

	/* Tant que toutes les iterations n'ont pas eu lieu  */
	for(iter=0;iter<prod.nbIterations;iter++)
	{
		/*=>Attendre l'autorisation de multiplication POUR UNE NOUVELLE
		ITERATION...*/


		fprintf(stderr,"--> mult(%d)\n",index); /* La multiplication peut
		commencer */

		/*=>Effectuer la multiplication a l'index du thread courant... */
		wasteTime(200+(rand()%200)); /* Perte du temps avec wasteTime()
		*/

		/* Affichage du calcul sur l'index */
		fprintf(stderr,"<-- mult(%d) : %.3g*%.3g=%.3g\n",
		index,prod.v1[index],prod.v2[index],prod.v3[index]);
```

```
            /*=>Marquer la fin de la multiplication en cours... */


            /*=>Si c'est la derniere ... */


            {
            /*=>Autoriser le demarrage de l'addition... */

            }
      }
      fprintf(stderr,"Quit mult(%d)\n",index);
      return(data);
}


/*************************************************************************/
void * add(void * data)
{
      size_t iter;
      fprintf(stderr,"Begin add()\n");
      /* Tant que toutes les iterations */
      for(iter=0;iter<prod.nbIterations;iter++) /* n'ont pas eu lieu */
      {
            size_t index;
            /*=>Attendre l'autorisation d'addition... */


            fprintf(stderr,"--> add\n"); /* L'addition peut commencer */
            /* Effectuer l'addition... */

            prod.result=0.0;

            for(index=0;index<prod.size;index++)
            {
                  /*=>A faire... */
            }
            wasteTime(100+(rand()%100)); /* Perdre du temps avec wasteTime()
            */

            fprintf(stderr,"<-- add\n");
            /*=>Autoriser le demarrage de l'affichage... */


      }
      fprintf(stderr,"Quit add()\n");
      return(data);
}


/*************************************************************************/


int main(int argc,char ** argv)
{
      size_t i, iter;
      pthread_t *multTh;
      size_t *multData;
      pthread_t addTh;
      void *threadReturnValue;
      /* A cause de warnings lorsque le code n'est pas encore la...*/
      (void)addTh; (void)threadReturnValue;

      /* Lire le nombre d'iterations et la taille des vecteurs */
```

```
if((argc<=2)||
(sscanf(argv[1],"%u",&prod.nbIterations)!=1)||
(sscanf(argv[2],"%u",&prod.size)!=1)||
((int)prod.nbIterations<=0)||((int)prod.size<=0))
{
      fprintf(stderr,"usage: %s nbIterations vectorSize\n",argv[0]);
      return(EXIT_FAILURE);
}


/* Initialisations (Product, tableaux, generateur aleatoire,etc) */
prod.state=STATE_WAIT;
prod.pendingMult=(int *)malloc(prod.size*sizeof(int));


/*=>initialiser prod.mutex ... */


/*=>initialiser prod.cond ... */


/* Allocation dynamique des 3 vecteurs v1, v2, v3 */
prod.v1=(double *)malloc(prod.size*sizeof(double));
prod.v2=(double *)malloc(prod.size*sizeof(double));
prod.v3=(double *)malloc(prod.size*sizeof(double));


/* Allocation dynamique du tableau pour les threads multiplieurs */
multTh=(pthread_t *)malloc(prod.size*sizeof(pthread_t));


/* Allocation dynamique du tableau des MulData */
multData=(size_t *)malloc(prod.size*sizeof(size_t));


/* Initialisation du tableau des MulData */
for(i=0;i<prod.size;i++)
{
      multData[i]=i;
}

/*=>Creer les threads de multiplication... */



/*=>Creer le thread d'addition... */



srand(time((time_t *)0)); /* Init du generateur de nombres aleatoires
*/

/* Pour chacune des iterations a realiser, c'est a dire : tant que
toutes les iterations n'ont pas eu lieu */
for(iter=0;iter<prod.nbIterations;iter++)
{
      size_t j;

      for(j=0;j<prod.size;j++)
      {
            prod.v1[j]=10.0*(0.5-((double)rand())/((double)RAND_MAX));
            prod.v2[j]=10.0*(0.5-((double)rand())/((double)RAND_MAX));
      }
```

```
            /*=>Autoriser le demarrage des multiplications pour une nouvelle
            iteration..*/

            /*=>Attendre l'autorisation d'affichage...*/

            /*=>Afficher le resultat de l'iteration courante...*/
        }

        /*=>Attendre la fin des threads de multiplication...*/

        /*=>Attendre la fin du thread d'addition...*/

        /*=> detruire prod.cond ... */

        /*=> detruire prod.mutex ... */

        /* Detruire avec free ce qui a ete initialise avec malloc */
        free(prod.pendingMult);
        free(prod.v1);
        free(prod.v2);
        free(prod.v3);
        free(multTh);
        free(multData);
        return(EXIT_SUCCESS);
}
```