Exercice1 :

Faites un programme permettant de calculer le produit matriciel de deux matrice 3x3.
Trois threads (workers) se répartissent le travail, chaque thread calcule une ligne de la matrice résultat.
Vous allez utiliser les verrous pour afficher le résultat sous la forme suivante :

Ligne1 printf("| 1 | 2 | 3 |\n"); Thread1
Ligne2 printf("| 4 | 5 | 6 |\n");Thread2
Ligne3 printf("| 7 | 8 | 9 |\n");Thread3

Exercice2 :

Nous allons implémenter une variante du jeu du nombre mystère où l'ordinateur choisit un nombre entier aléatoire et ne répond aux propositions du joueur que par "Trop petit !", "Trop grand !" ou "Gagné !". Dans cette variante, le nombre mystère évolue au fil du temps :

Toutes les *t* secondes, l'ordinateur modifie le nombre mystère en lui ajoutant ou en lui retirant une valeur *x*.
Le joueur est informé de ces modifications par un message du type "Le prix a été augmenté de *13*."
Les valeurs de *t* et *x* sont choisies aléatoirement à chaque modification :
t est un nombre entre *5* et *10* secondes.
x est un nombre entre *0* et *50*, ajouté ou retiré au hasard.
Le nombre mystère doit toujours rester entre *0* et *200* après modification.

Le joueur dispose de *40* secondes pour trouver le nombre avant la fin de la partie.
Implémentation technique

Le jeu sera réalisé à l'aide de trois threads :

Thread principal : gère l'interaction avec le joueur (jeu du nombre mystère classique).
Thread de modification : modifie périodiquement le nombre mystère en fonction des règles ci-dessus.
Thread de gestion du temps : surveille le temps écoulé et met fin au jeu après *40* secondes.

- Le nombre mystère est choisi entre 0 et 200.
- Le jeu se déroule en console (ou une interface définie par l'enseignant).
- L'objectif est d'implémenter ce jeu en respectant les principes de gestion des threads et de synchronisation.

Exercice3 :

On aimerait implémenter notre propre version de barrière de synchronisation pour N threads à l'aide de `pthread_mutex_t` . Notre implémentation de barrière devra avoir un comportement similaire aux barrières `POSIX` vu en cours, excepté le `reset` de la barrière une fois tous les threads passés. Chaque thread doit signaler son arrivée à la barrière à l'aide d'une fonction, par exemple `barrier_wait()`.
On désire aussi minimiser l'attente active dans notre implémentation, ceci, sans faire appel aux fonctions `sleep` ou `usleep`.
Réaliser un programme de test permettant de montrer que votre implémentation fonctionne correctement.

Exercice4 :

Cet exercice propose de comparer l'utilisation de spinlocks par rapport à l'utilisation de mutex dans le cadre d'une section critique très courte.

Implémenter un programme de test permettant de comparer les vitesses d'exécution respectives entre une section critique protégée par un mutex versus une section critique protégée par un spinlock. A vous donc d'imaginer un scénario, puis de mesurer les timings obtenus.

- Quel scénario avez-vous implémenté ?
- Que constatez-vous ?