Programmation des threads POSIX : Exercice n°3

Points de matière concernés :

- Les mutex pour threads et les sections critiques
- Les variables de condition et la synchronisation par événements
- Les variables spécifiques aux threads

Etape 1 (mise en place de l'application → mise en évidence de la concurrence)

Soit la structure suivante :

```
typedef struct
{
      char  nom[20];
      int   nbSecondes;
} DONNEE;
```

et les variables globales :

```
DONNEE data[] = { "MATAGNE",15,
                  "WILVERS",10,
                  "WAGNER",17,
                  "QUETTIER",8,
                  "",0 };

DONNEE Param;
```

Le thread principal va lancer autant de threads secondaires qu'il y a de données non nulles dans le vecteur data, et cela dans une boucle qui contient

memcpy(&Param,&data[i],sizeof(DONNEE)) ;
pthread_create(…,&Param) ;

Après avoir lancé tous les threads secondaires, le thread principal attendra la fin de ceux-ci (utilisation de pthread_join pour le moment), avant de se terminer lui-même.

Chaque thread secondaire doit utiliser la même fonction dans laquelle il

- affiche une chaîne "Thread 10529.5636553 lancé" (PID = 10529 et TID = 5636553).
- affiche le nom passé en paramètre dans la structure.
- attend un nombre de secondes (nanosleep), fourni par la structure.
- affiche une chaîne "Thread 10529.5636553 se termine".
- se termine sans rien retourner.

Après compilation et exécution du programme, observez et mettez en évidence le problème de concurrence qui se produit actuellement.

Etape 2 (résolution du problème de concurrence – mise en place d'un premier mutex)

Déclarer et initialiser correctement le **mutexParam** qui permettra de protéger l'accès à la variable globale **Param**. A vous de trouver les « lock » et « unlock » à mettre en place afin de résoudre le problème de concurrence de l'étape 1.

Etape 3 (synchronisation par événements)

Une variable globale **compteur** va à présent représenter le nombre de threads secondaires en cours d'exécution dans l'application. Dès lors, cette variable sera incrémentée au démarrage de chaque thread secondaire et décrémentée au moment où il se termine.

A partir de maintenant, le thread principal n'attendra plus la fin des threads secondaires à l'aide d'un pthread_join mais bien à l'aide d'un couple « mutex-variable de condition » : **mutexCompteur** et **condCompteur**. Ce couple correctement utilisé devra lui permettre d'attendre la réalisation de l'événement suivant :

« Tant que compteur est positif, j'attends… »

avant de se terminer.

Etape 4 (variable spécifique) : BONUS

Le thread principal doit à présent mettre en place une clé **cle** (**pthread_key_t**) qui sera mise à disposition des threads secondaires (utilisation de **pthread_key_create**).

Dès leur démarrage, les threads secondaires doivent allouer dynamiquement une chaîne de caractères qui recevra le nom reçu en paramètre (« Matagne », « Wagner », …) et mettre dans leur zone spécifique associée à **cle** le pointeur obtenu (utilisation de **pthread_setspecific**).

Pendant l'exécution du processus, on lui enverra le signal SIGINT (soit par un <CTRL-C>, soit en utilisant la commande kill). Seuls ces threads secondaires pourront recevoir ce signal. Vous devez donc le masquer correctement dans les autres threads.

Dès réception du signal SIGINT, le thread en question entrera dans un handler dans lequel il devra

1. récupérer sa variable spécifique associée à **cle**,
2. afficher un message du genre « Thread 10529.25333 s'occupe de « WAGNER » »

A son retour du handler, le thread devra reprendre son attente où il l'avait laissée (utilisation du second paramètre de **nanosleep**).

Les chaines de caractères allouées dynamiquement devront être libérées (free) automatiquement par une fonction « destructeur » mise en place lors du **pthread_key_create**.