Université Paul Sabatier
31062 Toulouse Cedex 09

**Licence 3 Informatique**

# Programmation Système
# Travaux pratiques n°1

***Vous devez rendre le travail effectué en séance à la fin de la séance (dans le bon dépôt Moodle). Vous pouvez modifier votre dépôt jusqu'à la date limite imposée.***

## Exercice 1 – Comparaison processus Unix / threads Posix

On veut écrire une application qui permet de créer N activités **parallèles** qui affichent un message personnalisé un certain nombre de fois avant de se terminer. Le nombre d'affichages à réaliser avant de se terminer devra être communiqué à la création, ainsi que le rang de création de l'activité de manière à ce que le message affiché soit personnalisé.

Le nombre N d'activités à créer ainsi que le nombre d'affichages à réaliser constituent les **paramètres** de l'application.

- Version 1 : les activités parallèles sont des processus.

**Exemple d'exécution avec des processus :**

*%./exo1a 5 1*

*[5 processus affichant 1 fois leur message « rang de création / identité » ]*

Je suis le processus de rang 4, mon identificateur est 2700
Je suis le processus de rang 2, mon identificateur est 2698
Je suis le processus de rang 3, mon identificateur est 2699
Je suis le processus de rang 0, mon identificateur est 2696
Je suis le processus de rang 1, mon identificateur est 2697

- Version 2 : les activités parallèles sont des threads.

**Exemple d'exécution avec des threads :**

*%./exo1b 5 1*

*[5 threads affichant 1 fois leur message « rang de création / identité » ]*

Je suis le thread de rang 4, mon identificateur est 3042315072
Je suis le thread de rang 3, mon identificateur est 3050707776
Je suis le thread de rang 2, mon identificateur est 3059100480
Je suis le thread de rang 1, mon identificateur est 3067493184
Je suis le thread de rang 0, mon identificateur est 3075885888

## Exercice 2 – Threads Posix : pratique du mécanisme exit/join

Compléter la version 2 de l'exercice précédent pour que chaque activité soit exécutée par un thread compagnon qui, lors de sa terminaison, **transmettra comme compte-rendu d'exécution** au thread initial/principal une valeur choisie au hasard entre 1 et 10 (par exemple).

À chaque réception d'un compte-rendu d'exécution, le thread principal affiche cette information.

Une fois que tous les threads compagnons sont terminés, le thread principal affiche la somme des valeurs récupérées.

Le nombre N de threads compagnons à créer constitue un **paramètre** de l'application.

Remarque : Voir manuel en ligne des fonctions *srand* et *rand*. La graine peut être le numéro du thread obtenu par pthread_self(), l'opérateur modulo permet de borner la valeur générée par rand.

**Exemple d'exécution :**

*%./exo2 4*

Je suis le thread de rang 3, mon identificateur est 3050527552, je mourrai en retournant 9
Je suis le thread de rang 2, mon identificateur est 3058920256, je mourrai en retournant 8
Je suis le thread de rang 1, mon identificateur est 3067312960, je mourrai en retournant 9
Je suis le thread de rang 0, mon identificateur est 3075705664, je mourrai en retournant 1
Valeur retournée par le thread 3075705664 = 1
Valeur retournée par le thread 3067312960 = 9
Valeur retournée par le thread 3058920256 = 8
Valeur retournée par le thread 3050527552 = 9
Je suis le thread principal, mon identificateur est 3075708672, la somme des valeurs reçues est 27

## Exercice 3 – Threads Posix : partage d'une variable

Transformer l'application précédente pour que chaque thread créé par le thread principal incrémente une **variable partagée** par tous les threads. Cette variable partagée représente la somme des valeurs qui sera affichée par le thread principal à la fin de leur exécution.

**Exemple d'exécution :**

*%./exo3 3*

Je suis le thread de rang 2, mon identificateur est 3059493696, j'ai ajoute 8
Je suis le thread de rang 1, mon identificateur est 3067886400, j'ai ajoute 7
Je suis le thread de rang 0, mon identificateur est 3076279104, j'ai ajoute 8
Valeur retournee par le thread 3076279104 = 8
Valeur retournee par le thread 3067886400 = 7
Valeur retournee par le thread 3059493696 = 8
Je suis le thread principal, mon identificateur est 3076282112, la somme des valeurs vaut 23

## Exercice 4 – Threads Posix : partage de plusieurs variables

On se propose d'écrire une application dans laquelle cohabitent deux types d'activités parallèles qui partagent des informations via un tableau à nbCases cases. Chaque case du tableau contient deux informations – une valeur entière et un type associé (parmi deux possibles, que l'on peut coder 0 ou 1, par exemple) – et sera représentée par une structure nommée Info.

Le premier type d'activité écrit des informations dans une case du tableau partagé. Le second type lit une information dans le tableau pour l'afficher.

L'accès à une case pour une écriture ou pour une lecture est séquentiel i.e. on accède à la case qui suit celle précédemment modifiée lors d'une écriture ou à celle qui suit celle précédemment consultée lors d'une lecture. Enfin, les accès sont gérés de manière circulaire (l'opération qui suit un accès à la dernière case se fera sur la première case du tableau).

Les **données partagées** entre les activités parallèles sont les suivantes :

- Le tableau, partagé par toutes les activités.
- L'indice de la prochaine case où écrire une information, partagé par les activités qui veulent écrire. Cet indice est incrémenté (% nbCases) à chaque écriture.
- L'indice de la prochaine case où lire, partagé par les activités qui veulent consulter. Cet indice est incrémenté (% nbCases) à chaque lecture.

Écrire une application qui permette de créer nbEcr threads voulant modifier le tableau et nbLec threads voulant y lire une information. Chaque thread, selon sa nature, exécutera une boucle de nbEcr écritures (d'une information de type 0 ou 1) ou de nbLec lectures.

L'application sera **paramétrée** par : nbEcr, nbLec, nbEcr, nbLec, nbCases.

Exécutez l'application pour **différentes configurations** (notamment celle données dans l'exemple ci-dessous) et examinez les résultats obtenus.

En supposant que ce tableau serve à communiquer des informations entre les deux types d'activités, les résultats obtenus vous paraissent-ils **cohérents** ? **Pourquoi** ?

***Remarque*** : Si les affichages sont trop rapides, il est possible de temporiser l'exécution d'un thread pendant quelques microsecondes ou nanosecondes à l'aide des primitive *int usleep (useconds_t usec)* ou *int nanosleep(const struct timespec *req, struct timespec *rem).* Voir le manuel en ligne pour leur utilisation (*man 3 usleep* ou *man 2 nanosleep*).

On peut utiliser une valeur générée aléatoirement (*rand*) pour varier les délais d'attente d'un thread à un autre. La fonction *srand()* permet d'initialiser le générateur aléatoire avec une graine donnée.

**Attention** : **Les résultats obtenus par une application temporisée doivent être les mêmes que si on n'avait pas eu recours à une telle temporisation !**

**Exemple d'exécution :**

*%exo4 1 2 2 2 1*

*[1 thread écrit, 2 threads lisent, celui qui écrit le fait 2 fois, les threads qui lisent le font 2 fois, dans un tableau d'1 case]*

Creation thread (Ecr) de rang 0 -> 0/1
Creation thread (Lec) de rang 0 -> 1/2
Creation thread (Lec) de rang 1 -> 2/2
Thd Lec 2 : Je veux lire 0 - - > le 3e thread (2e lecteur) veut lire une 1re fois
Thd Lec 2 : Info lue = [ -1/-1 ] - - > [ valeur / type d'info ] -1 signifie absence d'info
Buffer : [ -1/-1 ]
Thd Lec 1 : Je veux lire 0 - - > le 2e thread (1er lecteur) veut lire une 1re fois
Thd Lec 1 : Info lue = [ -1/-1 ]
Buffer : [ -1/-1 ]
Thd Lec 1 : Je veux lire 1 - - > le 2e thread veut lire une 2e fois
Thd Lec 1 : Info lue = [ -1/-1 ]
Buffer : [ -1/-1 ]
Thd Lec 2 : Je veux lire 1
Thd Lec 2 : Info lue = [ -1/-1 ]
Buffer : [ -1/-1 ]
Thd Ecr 1 : Je veux ecrire 0 - - > le 1er thread (écrivain) veut écrire une 1re fois
Thd ecr 1 : Info deposee = [1/0 ]
Buffer : [ 1/0 ]
Thd Ecr 1 : Je veux ecrire 1
Thd ecr 1 : Info deposee = [1/0 ]
Buffer : [ 1/0 ]
Fin de l'execution du main