UE Programmation Unix

TP5 Gestion des Threads

Les Threads

Pour le système, il s'agit de flux (fil) d'instructions indépendants, appartenant à un même processus, et qui peuvent être ordonnancés pour le partage du processeur.

Pour le développeur, ils s'agit de fonctions, au sein d'un même processus, qui peuvent s'exécuter simultanément (concurrence).

Un processus "classique" peut donc être considéré comme n'ayant qu'un seul et unique thread.

Les programmes seront compilés avec l'option `-pthread`

```
gcc nom.c -pthread -o nom
```

La fonction main()

Arguments de la ligne de commandes

- Langage C offre des mécanismes qui permettent d'intégrer parfaitement un programme C dans l'environnement hôte
  - environnement orienté ligne de commande (Unix, Linux)
- Programme C peut recevoir de la part de l'interpréteur de commandes qui a lancé son exécution, une liste d'arguments

⇔ ligne de commande qui a servi à lancer l'exécution du programme

- Liste composée
  - du nom du fichier binaire contenant le code exécutable du programme
  - des paramètres de la commande

Entête à inclure

```
#include <stdlib.h>  // <cstdlib> en C++
```

Fonction *atoi*

```
int atoi( const char * theString );
```

Cette fonction permet de transformer une chaîne de caractères, représentant une valeur entière, en une valeur numérique de type int. Le terme d'atoi est un acronyme signifiant : ASCII to integer.

⚠ **ATTENTION :** la fonction atoi retourne la valeur 0 si la chaîne de caractères ne contient pas une représentation de valeur numérique. Du coup, il n'est pas possible de distinguer la chaîne "0" d'une chaîne ne contenant pas un nombre entier. Si vous avez cette difficulté, veuillez préférer l'utilisation de la fonction strtol qui permet bien de distinguer les deux cas.

La fonction main ()

Un processus débute par l'exécution de la fonction main() du programme correspondant

Definition

```
int main (int argc, char *argv[]);
ou
int main (int argc, char **argv);
```

- argc: nombre d'arguments de la ligne de commande y compris le nom du programme
- argv[]: tableau de pointeurs vers les arguments (paramètres) passés au programme lors de son lancement

A NOTER

- argv[0] pointe vers le nom du programme
- argv[argc]vaut NULL

- argc (argument count)
  - nombre de mots qui compose la ligne de commande (y compris le nom de la commande qui a servi à lancer l'exécution du programme)
- argv (argument vector)
  - tableau de chaînes de caractères contenant chacune un mot de la ligne de commande
  - argv[0] est le nom du programme exécutable

Exemple de cours

```
#include <stdio.h>

int main (int argc, char *argv[]){
int i;
for (i=0;i<argc;i++){
    printf ("argv[%d]: %s\n",i, argv[i]);
}
return (0);
}// main

>./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
```



```
[ij04115@saphyr:~/unix_tp1]:ven. sept. 11$ nano affich_param.c
```

saphyr.ens.math-info.univ-paris5.fr - PuTTY

```
 GNU nano 2.5.3            Fichier : affich param.c
#include <stdio.h>

int main(int argc, char *argv[]){
        int i;
        for (i = 0 ; i < argc ; i++){
                printf("argv[%d]: %s\n", i, argv[i]);
        }
        return (0);
}//main
```

Spécifier le nom de l'exécutable avec l'option `-o`

```
ProgC > gcc -o toto premierProg.c
ProgC > ls
toto    premierProg.c
ProgC > ./toto
```

```
[ij04115@saphyr:~/unix_tp1]:sam. sept. 12$ gcc -o affich_param affich_param.c
[ij04115@saphyr:~/unix_tp1]:sam. sept. 12$ ls
affich_param  affich_param.c
[ij04115@saphyr:~/unix_tp1]:sam. sept. 12$
```

```
[ij04115@saphyr:~/unix_tp1]:sam. sept. 12$ ./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
[ij04115@saphyr:~/unix_tp1]:sam. sept. 12$
```

Le fichier standard des erreurs stderr

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

Entête à inclure

```
#include <stdio.h>
```

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

```
FILE * stdin;
FILE * stdout;
FILE * stderr;
```

La structure FILE permet de stocker les informations relatives à la gestion d'un flux de données. Néanmoins, il est très rare que vous ayez besoin d'accéder directement à ses attributs.

Effectivement, il existe un grand nombre de fonctions qui acceptent un paramètre basé sur cette structure pour déterminer ou contrôler divers aspects.

- **stdin** (Standard input) : ce flot correspond au flux standard d'entrée de l'application. Par défaut, ce flux est associé au clavier : vous pouvez donc acquérir facilement des données en provenance du clavier. Quelques fonctions utilisent implicitement ce flux (**scanf**, par exemple).
- **stdout** (Standard output): c'est le flux standard de sortie de votre application.
  Par défaut, ce flux est associé à la console d'où l'application à été lancée. Quelques fonctions utilisent implicitement ce flux (**printf**, par exemple).
- **stderr** (Standard error) : **ce dernier flux est associé à la sortie standard d'erreur de votre application.** Tout comme stdout, ce flux est normalement redirigé sur la console de l'application.

fprintf it is the same as printf,
except now you are also specifying the place to print to :

```
printf("%s", "Hello world\n"); // "Hello world" on stdout (using printf)
fprintf(stdout, "%s", "Hello world\n"); // "Hello world" on stdout (using fprintf)
fprintf(stderr, "%s", "Stack overflow!\n");//Error message on stderr (using fprintf)
```

Messages d'erreurs affiches avec perror

C Library - <stdio.h>

C library function - perror()

Description

The C library function **void perror(const char *str)** prints a descriptive error message to stderr. First the string **str** is printed, followed by a colon then a space.

Declaration

Following is the declaration for perror() function.

```
void perror(const char *str)
```

Parameters

- **str** – This is the C string containing a custom message to be printed before the error message itself.

Return Value

This function does not return any value.

```c
#include <stdio.h>

int main () {
   FILE *fp;

   /* first rename if there is any file */
   rename("file.txt", "newfile.txt");

   /* now let's try to open same file */
   fp = fopen("file.txt", "r");
   if( fp == NULL ) {
      perror("Error: ");
      return(-1);
   }
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will produce the following result because we are trying to open a file which does not exist –

```
Error: : No such file or directory
```

Tous les programmes devront être développés avec passage de leurs éventuels paramètres à la fonction

main (int argc, **char** ∗ argv [])

- **Les valeurs de retour des appels aux primitives devront être testées et les messages d'erreurs affiches avec** `perror`.
- **Les messages d'erreurs à destination de l'utilisateur se feront sur le fichier standard des erreurs** `stderr`.

1 Création de threads pthread_create ( ) , pthread_exit( )

Création

```
#include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
                    void * (*start_routine)(void *), void *arg);
```

- `thread` : identifiant du nouveau thread retourné par la fonction
- `attr` : paramètres de fonctionnement du thread (`NULL` possible):
  Détaché ou joignable, héritage de la politique de scheduling, politique de scheduling (FIFO (first-in first-out), RR (round-robin) ou OTHER (déterminé par le système), paramètres de scheduling, taille de la pile, adresse de la pile, etc
- `start_routine` : fonction exécutée par le thread
- `arg` : paramètres éventuels (NULL possible) de `start_routine`. Passés par référence par un pointeur avec un cast de type void.

Retourne :

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Terminaison

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

- `value_ptr` : Pointeur vers une variable contenant la valeur retournée par le thread qui se termine.
  Cette valeur pourra être consultée par tout thread effectuant un `pthread_join` avec le thread qui se termine.
  - Ne doit pas pointer vers une variable locale du thread qui se termine

Aucun retour de cette fonction

Fonctions exit ou _exit

- Un appel à l'une des fonctions `exit` a pour effet de terminer le processus auquel le thread appelant appartient
- La réception d'un signal par un thread peut avoir pour effet de terminer le processus auquel le thread appartient

a **Ecrivez un programme qui crée n threads.**

Chaque thread exécute une boucle suffisamment longue pour permettre l'observation et dans laquelle il écrit à chaque itération :

`Bonjour, je suis le thread tid`

Le thread 1 affiche son message au début de la ligne,
le thread 2 à une tabulation du début de la ligne,
le thread 3 à deux tabulations du début de la ligne, etc

```c
// Création et Terminaison de threads

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

#define MAX_THREADS 5
void* Bonjour(void* thread_id);

int main(int argc, char* argv[]) {
        pthread_t threads[MAX_THREADS]; // L'identifiant d'un thread est du typepthread_t
        int r;
        long t;

        for(t = 0 ; t < MAX_THREADS ; t++) {
                printf("Main : creation du thread %ld \n", t);

                /* int pthread_create (pthread_t *thread, pthread_attr_t *attr,void * (*start_routine)(void *), void *arg);
                 *      thread: identifiant du nouveau thread retourné par la fonction.
                 *      attr: paramètres de fonctionnement du thread (NULL possible).
                 *      void * (*start_routine)(void *) : fonction exécutée par le thread
                 *      arg: paramètres éventuels (NULL possible) de start_routine.
                 *      Passés par référence par un pointeur avec un cast de type void
                 */
                r = pthread_create(&threads[t], NULL, Bonjour, (void *)t);
                if(r) { // 0 en cas de succès ; une valeur autre que 0 en cas d'erreur
                        fprintf(stderr, "ERREUR : pthread_create() a retourné %d \n", r);
                        exit(-1); // Un appel à l'une des fonctionsexita pour effet de terminer le processus auquel le
threadappelant appartient.
                } // if
        } // for

        // void pthread_exit(void *value_ptr);
        pthread_exit(NULL); // terminaison du thread principal (main)
} // main

// Code du thread
void* Bonjour(void* thread_id) {
        long tid;
        int i;

        tid = (long) thread_id;

        for( i = 0 ; i < tid ; i++)
                printf("\t");

        printf("Bonjour ! Je suis le thread %ld \n", tid);

        /* void pthread_exit(void *value_ptr); Aucun retour de cette fonction
         * value_ptr: Pointeur vers une variable contenant la valeur retournée par le thread qui setermine.
         * Cette valeur pourra être consultée par tout thread effectuant unpthread_joinavec le threadqui se termine.
         * Ne doit pas pointer vers une variable locale du thread qui se termine
         */
        pthread_exit(NULL); // terminaison du thread
} // Bonjour()
```

```
[ij04115@saphyr:~/unix_tp5]:dim. janv. 03$ nano question1.c
[ij04115@saphyr:~/unix_tp5]:dim. janv. 03$ gcc question1.c -pthread -o question1
[ij04115@saphyr:~/unix_tp5]:dim. janv. 03$ ls
question1  question1.c
[ij04115@saphyr:~/unix_tp5]:dim. janv. 03$ ./question1
Main : creation du thread 0
Main : creation du thread 1
Main : creation du thread 2
Bonjour ! Je suis le thread 0
Main : creation du thread 3
                Bonjour ! Je suis le thread 2
        Main : creation du thread 4
                        Bonjour ! Je suis le thread 1
Bonjour ! Je suis le thread 3
                                Bonjour ! Je suis le thread 4
```

2 Synchronisation des threads

a Ecrire un programme qui crée :

- **n threads (n <= 26)**

  qui écrivent indéfiniment un caractère dans toutes les entrées d'un même tableau. Le thread écrivain 1 écrit le caractère A, le thread écrivain 2 écrit le caractère B, etc.

  Chaque thread écrivain est créé "détaché". Il reçoit en paramètre son numéro d'ordre de création

- **Un thread lecteur**

  qui affiche le contenu complet du tableau et qui totalise le nombre de fois où un caractère apparaît dans le tableau.

  Ce thread lecteur du tableau est créé "joignable" et sera attendu par le thread main. Il reçoit en paramètre le nombre de fois où il doit afficher le contenu du tableau.

  Lorsque le thread lecteur est terminé, le thread main affiche le nombre d'apparitions de chaque caractère constatées par le thread lecteur.

Appel : `./nom 3 5`

Résultat pour un tableau de 75 caractères:

```
DEBUT MAIN
CBBCAAAACCAACBBBBAAACBCCCCCBBBBBCCAABBBBBACCCCAAAAABBBCCCCCBBAAAACCCCCCCBCC
CCBAACCAAAAACCBBBBBBBACBBBBCAAACCBACCCCCACCCCAAAABBBBAAAAACCAAAAAAAABABBBCC
CAAAAAAAACBBBBAAAAAACCCCAAAACCCABBBBBAAAABBBBACCCCCABCCCAAAABBAAAAAAAAACBBB
CBCAAAAABBBACCCCCBBBCCCAAAAABBCCCCCABBBBBCCAABBBBAACCCCAACCCCCCAAAACAAAAAAA
CCCAAAACCCCCCAAACCCCCCCCCCCCCCCCCCCCAACCCCAACCCCAAACCCCCCCCAAAAACAAAAAAAACC
A: 147
B: 81
C: 147
FIN MAIN
```

```c
#include <pthread.h> // pthread_attr_init(), pthread_attr_setdetachstate(), thread_create(), pthread_attr_destroy(), pthread_join(), pthread_exit()
#include <stdio.h> // fprintf()
#include <stdlib.h> // exit(), EXIT_FAILURE, atoi()
#define MAX_THREADS 26 // max tableau des compteurs
#define MAX_TAB 75 // max tableau de caractères

char caractere_tab[MAX_TAB]; // tableau de caractères
int frequence_tab[MAX_THREADS + 1]; // tableau de compteurs ; frequence_tab[0] inutilisé.

void* ThreadEcrivain(void* thread_number);
void* ThreadLecteur(void* thread_iter);

int main(int argc, char *argv[]) {
	int function_result;
	long total = 0;
	pthread_t tid; // L'identifiant d'un thread est du type pthread_t

	/* Par exemple : ./pgm 3 6
	 * ca veut dire que je vais créer 3 threads écrivain
	 * et 1 thread lecteur qui va lire 6 fois le contenu du tableau de caractères.
	 */
	if(argc != 3) {
		fprintf(stderr, "Utilisation : %s nb_thread nb_iter \n", argv[0]);
		exit(EXIT_FAILURE); // Un appel à l'une des fonctionsexita pour effet de terminer le processus auquel le threadappelant appartient
	} // if

	int nb_thread = atoi( argv[1] ); // function to convert ASCII to int
	int nb_iter = atoi( argv[2] ); // function to convert ASCII to int
	printf("DEBUT MAIN\n");

	int i;
	for( i = 1 ; i < MAX_THREADS + 1 ; i++) // frequence_tab[0] inutilisé.
		frequence_tab[i] = 0; // Initialisez tous les éléments du tableau à zéro (tableau des compteurs).

	/* Joignabilité : Permet la synchronisation sur la terminaison des threads
	 * Un thread créé à l'état joignable peut être attendu par n'importe quel autre thread
	 * Joignabilité : mise en œuvre (Cours p. 23).
	 */
	pthread_attr_t attr; // Déclaration d'une variable de type pthread_attr_t
	pthread_attr_init( &attr ); // // Initialisation de cette variable avec pthread_attr_init()

	/* int pthread_attr_setdetachstate(pthread_attr_t *attr,int detachstate);
	 * attr : pointeur vers la variable d'état
	 * detachstate : PTHREAD_CREATE_JOINABLE ou PTHREA_CREATE_DETACHED
	 */
	pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); // Créer les threads qui ne seront jamais joints à l'état détaché
	// Création des threads qui écrivent dans le tableau de caractères
	for( i = 1 ; i <= nb_thread && i < MAX_THREADS ; i++) {
		/* int pthread_create (pthread_t *thread, pthread_attr_t *attr,void * (*start_routine)(void *), void *arg);
		 * thread : identifiant du nouveau thread retourné par la fonction
		 * attr : paramètres de fonctionnement du thread (NULL possible):
		 * start_routine : fonction exécutée par le thread
		 * arg : paramètres éventuels (NULL possible) destart_routine. Passés par référence par unpointeur avec un cast de type void.
		 */
		function_result = pthread_create(&tid, &attr, ThreadEcrivain, (void *) i);
		if( function_result != 0) {
			fprintf(stderr, "ERREUR : pthread_create() a retourné %d \n", function_result);
			exit(EXIT_FAILURE);
		} // if
	} // for
```

```
        pthread_attr_init( &attr ); // Initialisation d'une variable de type pthread_attr_t avec pthread_attr_init()
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); // Créer explicitement les threads à l'état joignable

        // Création du thread ThreadLecteur qui affiche et compte les caractères du tableau.
        function_result = pthread_create(&tid, &attr, ThreadLecteur, (void *) nb_iter);
        if( function_result != 0 ) {
                fprintf(stderr, "ERREUR : pthread_create() a retourné %d \n", function_result);
                exit(EXIT_FAILURE);
        } // if
        pthread_attr_destroy(&attr); // Libération des ressources avec pthread_attr_destroy() une fois la création du thread
terminée

        /* Attente de la fin du thread qui affiche et compte les caractères du tableau
         * int pthread_join(pthread_t th, void **thread_return);
         * th: thread dont la fin est attendue.
         * thread_return: valeur renvoyée par le thread qui s'est terminé
         */
        function_result = pthread_join(tid, NULL);
        if ( function_result != 0 ) { // 0 en cas de succès ou une autre valeur que 0 en cas d'erreur
                fprintf(stderr, "ERREUR : pthread_join() a retourné %d \n", function_result);
                exit(EXIT_FAILURE);
        } // if

        // Affichage des statistiques d'apparition de chaque caractère
        printf("\n");
        for( i = 1 ; i <= nb_thread ; i++) {
                printf("%c : %d \n", (char) (64 + i), frequence_tab[i]);
                total += frequence_tab[i];
        }
        printf("TOTAL : %ld\n", total);
        printf("FIN MAIN \n");
} // main()
```

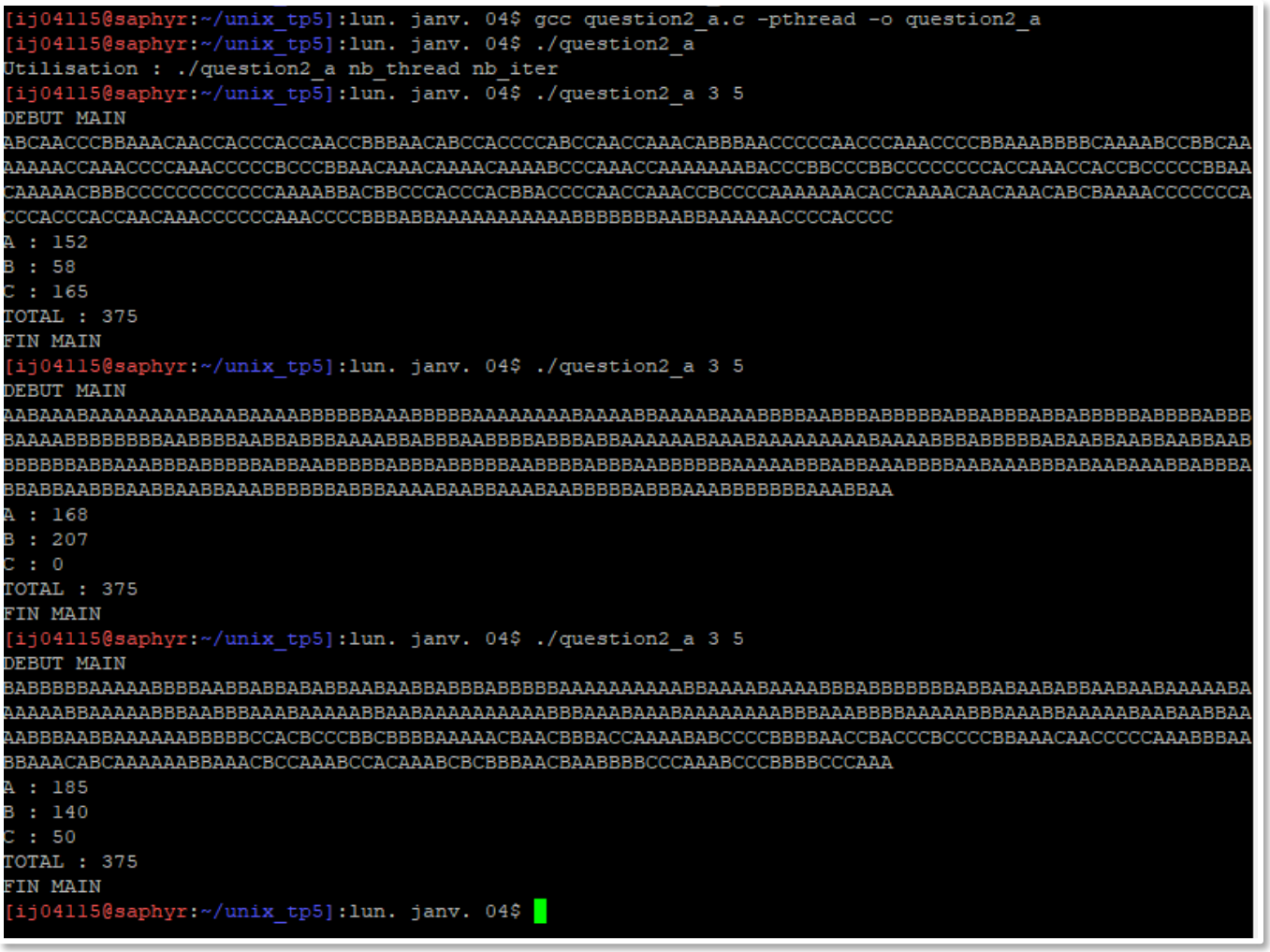
```
/* ThreadEcrivain : Threads qui ecrit dans le tableau de caractères : caractere_tab[]
 * Chaque ThreadEcrivain est "responsable" de l'écriture d'un certain caractère dans
 * le tableau de caractères et il le fait dans une boucle infinie.
 * Autrement dit, le ThreadEcrivain continue d'écrire «son caractère» aussi longtemps qu'il le peut.
 *
 * Chaque ThreadEcrivain écrit un caractère particulier en fonction du numéro de série de ce
ThreadEcrivain.
 * Par exemple: le premier ThreadEcrivain écrira le caractère A,
 * le second ThreadEcrivain écrira le caractère B, et ainsi de suite.
 * Par conséquent, la première action de la fonction est de trouver le code ASCII du caractère
 * concerné et de le sauvegarder dans la variable my_char
 */
void* ThreadEcrivain(void* thread_number) {
        long t_number = (long) thread_number;
        char my_char = 'A' + t_number - 1; // Calcul du code ASCII (cf cours L1 sur les équivalences entre
les entier et les codes ASCII).

        while(1) {
                int i;
                for( i = 0 ; i < MAX_TAB ; i++) // Ecriture de mychar dans tout le tableau
                        caractere_tab[i] = my_char;
        } // while
} // ThreadEcrivain()
```

```
/* ThreadLecteur : Il va lire un certain nombre de fois (thread_iter) le tableau dans lequel les
écrivains écrivent les caractères (caractere_tab[])
 * ThreadLecteur fait des statistiques. Donc, en fonction du caractère qui se trouve dans la case i_car,
 * il va incrémenter un compteur dans le tableau de compteurs. Dans le tableau il y a un compteur par
lettre de l'alphabet.
 * Par exemple : si j'utilise 3 threads donc ca veut dire que j'ai utilisé 3 compteurs pour compter les
occurrences de A, de B et de C.
 * ThreadLecteur affiche le caractère qui vient de comptabiliser
 */
void* ThreadLecteur(void* thread_iter) {
	long nb_iter = (long) thread_iter;
	int the_char;

	int i, i_car;
	for( i = 1 ; i <= nb_iter ; i++) {
		for( i_car = 0 ; i_car < MAX_TAB ; i_car++) {
			/* a */ if(caractere_tab[i_car] > 'A' - 1) { // Aucun ThreadEcrivain n'a peut etre
encore écrit

			/* b */		frequence_tab[ caractere_tab[i_car] - 'A' + 1 ]++;
			/* c */		printf("%c", caractere_tab[i_car]);
				} // if
		} // for
	} // for

	/* void pthread_exit(void *value_ptr); Aucun retour de cette fonction
	 * value_ptr: Pointeur vers une variable contenant la valeur retournée par le thread qui setermine.
	 * Cette valeur pourra être consultée par tout thread effectuant unpthread_joinavec le threadqui se
termine.
	 * Ne doit pas pointer vers une variable locale du thread qui se termine
	 */
	pthread_exit( (void *) NULL );
} // ThreadLecteur()
```

b **Procédez à plusieurs exécutions de :**

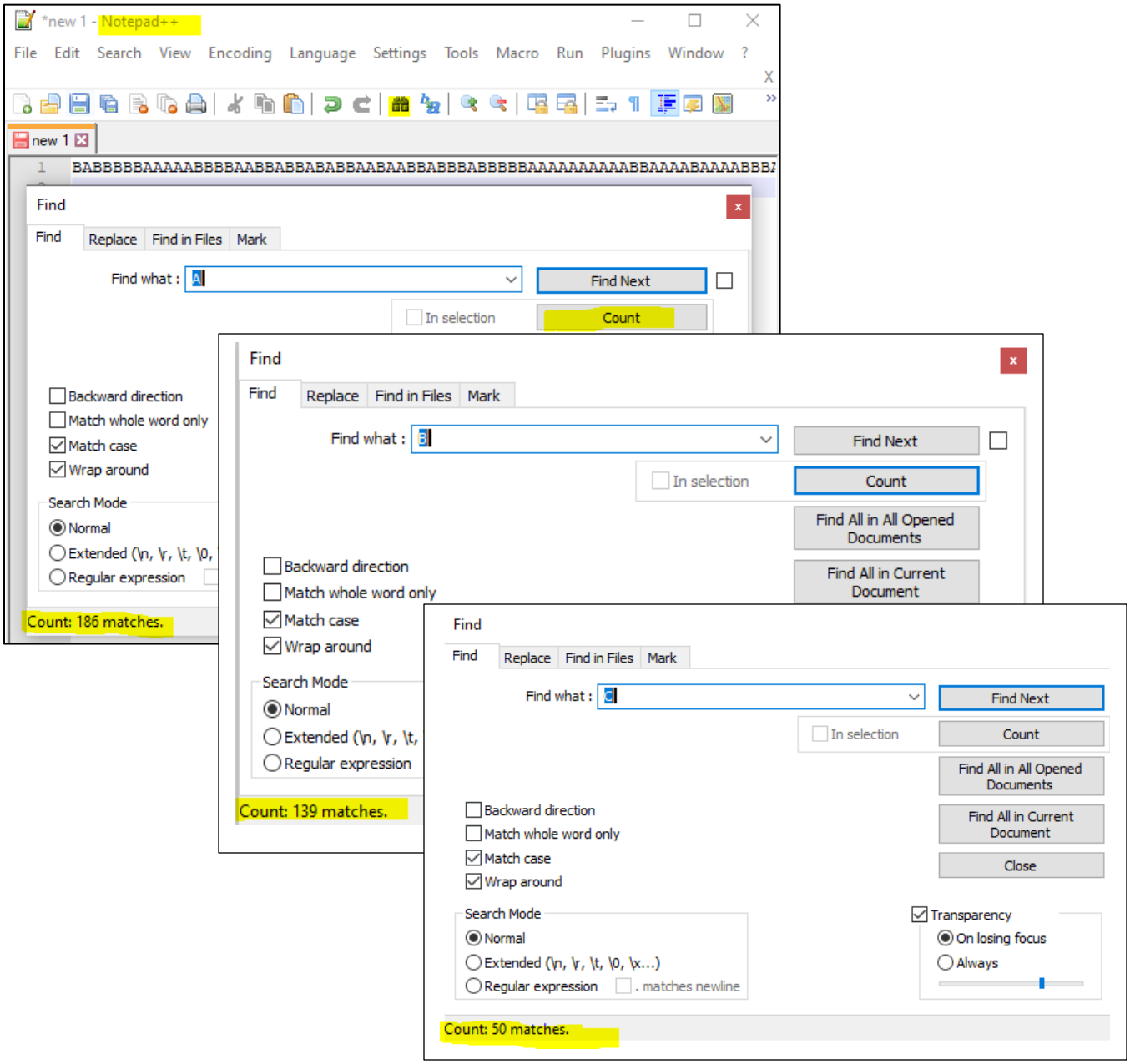
`./nom 3 5`

Expliquez ce que vous constatez



Les statistiques qui sont affichées ne correspondent pas toujours au tableau.

Par exemple : le résultat de la dernière exécution



On voit que le total est correct. Par contre le total individuel de chaque caractère n'est pas toujours correct.

Ou est ce qu'il peut y avoir un problème dans ce programme ? A quel moment ce genre de situation peut se produire ?

C'est entre la lecture et l'écriture.

Pendant qu'on fait la lecture ici :

```
/* ThreadLecteur : Il va lire un certain nombre de fois (thread_iter) le tableau dans lequel les
écrivains écrivent les caractères (caractere_tab[])
 * ThreadLecteur fait des statistiques. Donc, en fonction du caractère qui se trouve dans la case i_car,
 * il va incrémenter un compteur dans le tableau de compteurs. Dans le tableau il y a un compteur par
lettre de l'alphabet.
 * Par exemple : si j'utilise 3 threads donc ca veut dire que j'ai utilisé 3 compteurs pour compter les
occurrences de A, de B et de C.
 * ThreadLecteur affiche le caractère qui vient de comptabiliser
 */
void* ThreadLecteur(void* thread_iter) {
	long nb_iter = (long) thread_iter;
	int the_char;

	int i, i_car;
	for( i = 1 ; i <= nb_iter ; i++) {
		for( i_car = 0 ; i_car < MAX_TAB ; i_car++) {
			/* a */ if(caractere_tab[i_car] > 'A' - 1) { // Aucun ThreadEcrivain n'a peut etre
encore écrit

			/* b */		frequence_tab[ caractere_tab[i_car] - 'A' + 1 ]++;
			/* c */		printf("%c", caractere_tab[i_car]);
				} // if
		} // for
```

Il y a un risque que la tableau de caractères a change entre temps.

Ça montre les conséquences de la concurrence, les conséquences de l'accès concurrents à une structure de données.

Ici lorsque le lecteur détecte un caractère, il va incrémenter le compteur qui correspond à ce caractère dans le tableau de fréquences, mais on peut imaginer par exemple qu'il a vu un B dans la case i du tableau et donc qu'il va incrémenté le compteur de fréquences correspondant au B. MAIS, il peut très bien se passée que bien qu'il ait vu un B, il peut perdre le processeur, (par exemple entre les lignes /*a*/ et /*b*/ et a l'indice i du tableau ou il avait vu un B, il y a un écrivain qui a écrit autre chose. Donc on a compté un B qui au final ne sera pas affichée parce qu'un écrivain a déjà modifié la valeur de cette case pendant que le lecteur regardez ce qu'ils avaient dans la case.

Solution 1 : Stocker le caractère qu'on a lu.

Ça ne voudra pas dire que le caractère i n'aura pas changé, mais ca ne faussera pas la cohérence entre ce qui est affiché par le programme et ce qui est compter au niveau des statistiques.

```
// ------------------------------------------------
    void *ThreadLecteur(void *thread_iter) {
// ------------------------------------------------
        long nb_iter;
        int i, i_car;
        int the_char;

        nb_iter = (long) thread_iter;

        for (i=1; i<=nb_iter; i++){

            for (i_car=0; i_car<MAX_TAB; i_car++) {

                the_char=caractere_tab[i_car];
                if (the_char>'A'-1) {// Aucun ThreadEcrivain n'a
                                     // peut être encore écrit
                    frequence_tab[the_char-'A'+1]++;
                    printf("%c", the_char);

            } // for

            printf ("\n");
        }// for

        pthread_exit((void *)NULL);

    } // ThreadLecteur
```

Solution 2 : *Question c*

C Modifiez le programme précédent afin que

1. **Le thread lecteur**
   - Affiche chaque caractère lu du tableau et le remplace par un espace
   - Ne s'exécute que lorsque le tableau ne contient que des caractères différents de l'espace
2. **Chaque thread écrivain**
   - Soit le seul à écrire son caractère dans le tableau au moment où il écrit ce caractère.
   - N'écrive son caractère que dans les cases du tableau contenant un espace
   - Ne s'exécute que lorsque le tableau a été rempli d'espaces par le thread lecteur.

On nous dit que chaque écrivain doit être le seul à accéder au tableau au moment où il écrit un caractère.

Donc : il va falloir protéger. Si on veut que chaque thread écrivain soit le seul à accéder à la fois au tableau, contrairement a ce qu'on a vu tout à l'heure quand on fait pas du tout de synchronisation, on va utiliser un sémaphore d'exclusion mutuelle pour l'accès au tableau.

Ensuite on nous dit que les écrivains doivent attendre, avant d'écrire, que le tableau était vidé par le lecteur, et le lecteur doit attendre avant de lire que le tableau a été remplies par les écrivains. Donc là on voit qu'il y a une condition qui intéressent à la fois les écrivains et les lecteurs : c'est l'état du tableau. Donc cette condition c'est l'état du tableau.

Pour les lectures il faut que le tableau soit plain et pour les écrivains ils ne peuvent pas écrire tant que le tableau n'a pas été vidé par le lecteur. Donc il faudra une variable condition qui en fait l'état du tableau et on a vu que chaque variable condition devaient être assortie d'un sémaphore pour y accéder.

Du moment ou on parle de synchronisation il y a toujours un sémaphore..

Donc il y a un sémaphore ici pour **l'accès exclusif** au tableau mais comme on utilise aussi une variable condition qui va donc représenter l'état du tableau, on a aussi besoin de sémaphore qui va être associé à cette variable condition (un sémaphore qui a été verrouillé au préalable ; cf cours).

Mise en œuvre

Donc les écrivains, il d'abord vouloir entrer en section critique pour accéder aux tableau, ils vont faire ce qu'ils ont à faire pour accéder sur le tableau. Une fois qu'ils sont rentrés dans leurs section critique, faut qu'ils écrivent le caractère qu'il devrait écrire à condition que la case sur laquelle il tombe soit vide, ça aussi c'est une des contraintes du cahier des charges.

Une fois qu'ils ont fait ça, il va falloir qu'ils entrent en section critique pour savoir l'état du tableau : est-ce que le tableau est plein ou pas, et quand il sera plain, il va se mettre en attente de recevoir un signal de la part du lecteur qui dit que le tableau a été vidée.

Maintenant si on prend la logique du lecteur, il faut d'abord que le tableau soit rempli avant qu'il puisse être lu, donc ça veut dire que c'est d'abord les écrivains qui vont devoir travailler, donc le lecteur va attendre que le tableau soit plain (autrement dit : il attend tant qu'il est vide). Une fois que cette condition sera satisfaite, il va chercher le tableau.

Quand on parle d'espace = une case vide

(Un espace qui symbolise que la case est vide).

```c
// pthread_attr_init(), pthread_attr_setdetachstate(), thread_create(), pthread_attr_destroy(), pthread_join(), pthread_exit(),
// pthread_mutex_init(), pthread_mutex_lock() pthread_mutex_unlock(), pthread_cond_signal(), pthread_cond_wait(),
pthread_cond_broadcast()
#include <pthread.h>
#include <stdio.h> // fprintf(), perror()
#include <stdlib.h> // exit(), EXIT_FAILURE, atoi()

#define MAX_THREADS 26 // max tableau des compteurs
#define MAX_TAB 75 // max tableau de caractères
#define VIDE 0
#define PLEIN 1

char caractere_tab[MAX_TAB]; // tableau de caractères
int frequence_tab[MAX_THREADS + 1]; // tableau de compteurs ; frequence_tab[0] inutilisé.

pthread_mutex_t mutex_acces_tableau;
pthread_mutex_t mutex_etat_tableau; // mutex associé à condition_etat_tableau.
pthread_cond_t condition_etat_tableau;
int etat_tableau = VIDE;
int nb_caractere_tableau = 0;

void* ThreadEcrivain(void* thread_number);
void* ThreadLecteur(void* thread_iter);

int main(int argc, char *argv[]) {
        int function_result;
        long total = 0;
        pthread_t tid; // L'identifiant d'un thread est du type pthread_t
        int i;

        /* Par exemple : ./pgm 3 6
         * ca veut dire que je vais créer 3 threads écrivain
         * et 1 thread lecteur qui va lire 6 fois le contenu du tableau de caractères.
         */
        if(argc != 3) {
                fprintf(stderr, "Utilisation : %s nb_thread nb_iter \n", argv[0]);
                exit(EXIT_FAILURE); // Un appel à l'une des fonctionsexita pour effet de terminer le processus auquel le
threadappelant appartient
        } // if

        int nb_thread = atoi( argv[1] ); // function to convert ASCII to int
        int nb_iter = atoi( argv[2] ); // function to convert ASCII to int
        printf("DEBUT MAIN\n");

        //Initialisation du tableau de caractères
        for( i = 0 ; i < MAX_TAB ; i++)
                caractere_tab[i] = ' ';

        // Initialisation du tableau de comptage des caractères.
        for( i = 1 ; i < MAX_THREADS + 1 ; i++) // frequence_tab[0] inutilisé.
                frequence_tab[i] = 0; // Initialisez tous les éléments du tableau à zéro (tableau des compteurs).

        // Initialisation du sémaphore  mutex_acces_tableau
        function_result = pthread_mutex_init(&mutex_acces_tableau, NULL);
        if(function_result != 0) {
                perror("ERREUR pthread_mutex_init() : ");
                exit(EXIT_FAILURE);
        }

        // Initialisation du sémaphore mutex_etat_tableau
        function_result = pthread_mutex_init(&mutex_etat_tableau, NULL);
        if(function_result != 0) {
                perror("ERREUR pthread_mutex_init()");
                exit(EXIT_FAILURE);
        }
```

```c
        /* Joignabilité : Permet la synchronisation sur la terminaison des threads
         * Un thread créé à l'état joignable peut être attendu par n'importe quel autre thread
         * Joignabilité : mise en œuvre (Cours p. 23).
         */
        pthread_attr_t attr; // Déclaration d'une variable de type pthread_attr_t
        pthread_attr_init( &attr ); // // Initialisation de cette variable avec pthread_attr_init()

        /* int pthread_attr_setdetachstate(pthread_attr_t *attr,int detachstate);
         * attr : pointeur vers la variable d'état
         * detachstate : PTHREAD_CREATE_JOINABLE ou PTHREA_CREATE_DETACHED
         */
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); // Créer les threads qui ne seront jamais joints à l'état
détaché
        // Création des threads qui écrivent dans le tableau de caractères
        for( i = 1 ; i <= nb_thread && i < MAX_THREADS ; i++) {
                /* int pthread_create (pthread_t *thread, pthread_attr_t *attr,void * (*start_routine)(void *), void *arg);
                 * thread : identifiant du nouveau thread retourné par la fonction
                 * attr : paramètres de fonctionnement du thread (NULL possible):
                 * start_routine : fonction exécutée par le thread
                 * arg : paramètres éventuels (NULL possible) destart_routine. Passés par référence par unpointeur avec un cast de
type void.
                 */
                function_result = pthread_create(&tid, &attr, ThreadEcrivain, (void *) i);
                if( function_result != 0) {
                        fprintf(stderr, "ERREUR : pthread_create() a retourné %d \n", function_result);
                        exit(EXIT_FAILURE);
                } // if
        } // for

        pthread_attr_init( &attr ); // Initialisation d'une variable de type pthread_attr_t avec pthread_attr_init()
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); // Créer explicitement les threads à l'état joignable

        // Création du thread ThreadLecteur qui affiche et compte les caractères du tableau.
        function_result = pthread_create(&tid, &attr, ThreadLecteur, (void *) nb_iter);
        if( function_result != 0 ) {
                fprintf(stderr, "ERREUR : pthread_create() a retourné %d \n", function_result);
                exit(EXIT_FAILURE);
        } // if
        pthread_attr_destroy(&attr); // Libération des ressources avec pthread_attr_destroy() une fois la création du thread
terminée

        /* Attente de la fin du thread qui affiche et compte les caractères du tableau
         * int pthread_join(pthread_t th, void **thread_return);
         * th: thread dont la fin est attendue.
         * thread_return: valeur renvoyée par le thread qui s'est terminé
         */
        function_result = pthread_join(tid, NULL);
        if ( function_result != 0 ) { // 0 en cas de succès ou une autre valeur que 0 en cas d'erreur
                fprintf(stderr, "ERREUR : pthread_join() a retourné %d \n", function_result);
                exit(EXIT_FAILURE);
        } // if

        // Affichage des statistiques d'apparition de chaque caractère
        printf("\n");
        for( i = 1 ; i <= nb_thread ; i++) {
                printf("%c : %d \n", (char) (64 + i), frequence_tab[i]);
                total += frequence_tab[i];
        }
        printf("TOTAL : %ld\n", total);
        printf("FIN MAIN \n");
} // main()
```

```c
/* ThreadEcrivain : Threads qui ecrit dans le tableau de caractères : caractere_tab[]
 * Chaque ThreadEcrivain est "responsable" de l'écriture d'un certain caractère dans
 * le tableau de caractères et il le fait dans une boucle infinie.
 * Autrement dit, le ThreadEcrivain continue d'écrire «son caractère» aussi longtemps qu'il le peut.
 *
 * Chaque ThreadEcrivain écrit un caractère particulier en fonction du numéro de série de ce ThreadEcrivain.
 * Par exemple: le premier ThreadEcrivain écrira le caractère A,
 * le second ThreadEcrivain écrira le caractère B, et ainsi de suite.
 * Par conséquent, la première action de la fonction est de trouver le code ASCII du caractère
 * concerné et de le sauvegarder dans la variable my_char
 */
void* ThreadEcrivain(void* thread_number) {
        int function_result;
        long t_number = (long) thread_number;
        char my_char = 'A' + t_number - 1; // Calcul du code ASCII (cf cours L1 sur les équivalences entre les entier et les codes ASCII).

        while(1) {
                int i, nb;
                for( i = 0 , nb = 0 ; i < MAX_TAB ; i++)  { // Ecriture de mychar dans tout le tableau
                        // Début de section critique mutex_acces_tableau
                        function_result = pthread_mutex_lock(&mutex_acces_tableau);
                        if(function_result != 0) {
                                perror("ERREUR pthread_mutex_lock()");
                                exit(EXIT_FAILURE);
                        }

                        if(nb_caractere_tableau < MAX_TAB) {
                                if(caractere_tab[i] == ' ') { // Ecriture seulment si l'entrée i est
                                        caractere_tab[i] = my_char;
                                        nb_caractere_tableau++;
                                }
                        }

                        // Fin de section critique mutex_acces_tableau
                        function_result = pthread_mutex_unlock(&mutex_acces_tableau);
                        if( function_result != 0 ) {
                                perror("ERREUR pthread_mutex_unlock()");
                                exit(EXIT_FAILURE);
                        }
                } // for( i = 0 , nb = 0 ; i < MAX_TAB ; i++)

                if(nb_caractere_tableau == MAX_TAB) { // Le tableau est plein

                        // Debut de section critique mutex_etat_tableau
                        function_result = pthread_mutex_lock(&mutex_etat_tableau);
                        if( function_result != 0 ) {
                                perror("ERREUR pthread_mutex_lock()");
                                exit(EXIT_FAILURE);
                        }

                        etat_tableau = PLEIN;

                        // Le tableau est plein : envoi du signal
                        function_result = pthread_cond_signal(&condition_etat_tableau);
                        if( function_result != 0 ) {
                                perror("ERREUR pthread_cond_signal()");
                                exit(EXIT_FAILURE);
                        }

                        // Attente que le tableau soit vide
                        while(etat_tableau == PLEIN) {
                                function_result = pthread_cond_wait(&condition_etat_tableau, &mutex_etat_tableau);
                                if( function_result != 0 ) {
                                        perror("ERREUR pthread_cond_wait()");
                                        exit(EXIT_FAILURE);
                                } // if
                        } // while

                        // Fin de section critique mutex_etat_tableau
                        function_result = pthread_mutex_unlock(&mutex_etat_tableau);
                        if(function_result != 0) {
                                perror("ERREUR pthread_mutex_unlock()");
                                exit(EXIT_FAILURE);
                        }
                } // if
        } // while (1)
} // ThreadEcrivain()
```

```c
/* ThreadLecteur : Il va lire un certain nombre de fois (thread_iter) le tableau dans lequel les écrivains écrivent les caractères (caractere_tab[])
 * ThreadLecteur fait des statistiques. Donc, en fonction du caractère qui se trouve dans la case i_car,
 * il va incrémenter un compteur dans le tableau de compteurs. Dans le tableau il y a un compteur par lettre de l'alphabet.
 * Par exemple : si j'utilise 3 threads donc ca veut dire que j'ai utilisé 3 compteurs pour compter les occurrences de A, de B et de C.
 * ThreadLecteur affiche le caractère qui vient de comptabiliser
 */
void* ThreadLecteur(void* thread_iter) {
        int function_result;
        long nb_iter = (long) thread_iter;
        int the_char;

        int i, i_car;
        for( i = 1 ; i <= nb_iter ; i++) {
                // Début de section critique mutex_etat_tableau
                function_result = pthread_mutex_lock(&mutex_etat_tableau);
                if( function_result != 0 ) {
                        perror("ERREUR pthread_mutex_lock()");
                        exit(EXIT_FAILURE);
                }

                // Attente que le tableau soit rempli
                while (etat_tableau == VIDE) {
                        function_result = pthread_cond_wait(&condition_etat_tableau, &mutex_etat_tableau);
                        if(function_result != 0) {
                                perror("ERREUR pthread_mutex_lock()");
                                exit(EXIT_FAILURE);
                        }
                }

                for( i_car = 0 ; i_car < MAX_TAB ; i_car++) {
                        the_char = caractere_tab[i_car];
                         if( (the_char > 'A' - 1) && (the_char != ' ') ) {
                                        // Aucun ThreadEcrivain n'a peut etre encore écrit
                                        frequence_tab[ the_char - 'A' + 1 ]++;
                         } // if
                         printf("%c", the_char);
                         caractere_tab[i_car] = ' ';
                } // for
                nb_caractere_tableau = 0;

                // Fin de section critique mutex_acces_tableau
                function_result = pthread_mutex_unlock(&mutex_acces_tableau);
                if(function_result != 0) {
                        perror("ERREUR pthread_mutex_unlock()");
                        exit(EXIT_FAILURE);
                }

                etat_tableau = VIDE;

                // Le tableau est vide : envoi du signal à TOUS les threads écrivains
                function_result = pthread_cond_broadcast(&condition_etat_tableau);
                if(function_result != 0) {
                        perror("ERREUR pthread_cond_broadcast()");
                        exit(EXIT_FAILURE);
                }

                // Fin de section critique mutex_etat_tableau
                function_result = pthread_mutex_unlock(&mutex_etat_tableau);
                if(function_result != 0) {
                        perror("ERREUR pthread_mutex_unlock()");
                        exit(EXIT_FAILURE);
                }
        } // for( i = 1 ; i <= nb_iter ; i++)

        /* void pthread_exit(void *value_ptr); Aucun retour de cette fonction
         * value_ptr: Pointeur vers une variable contenant la valeur retournée par le thread qui setermine.
         * Cette valeur pourra être consultée par tout thread effectuant unpthread_joinavec le threadqui se termine.
         * Ne doit pas pointer vers une variable locale du thread qui se termine
         */
        pthread_exit( (void *) NULL );
} // ThreadLecteur()
```

a