UNIVERSITÉ CAEN NORMANDIE

Les Thread

21.11.2018

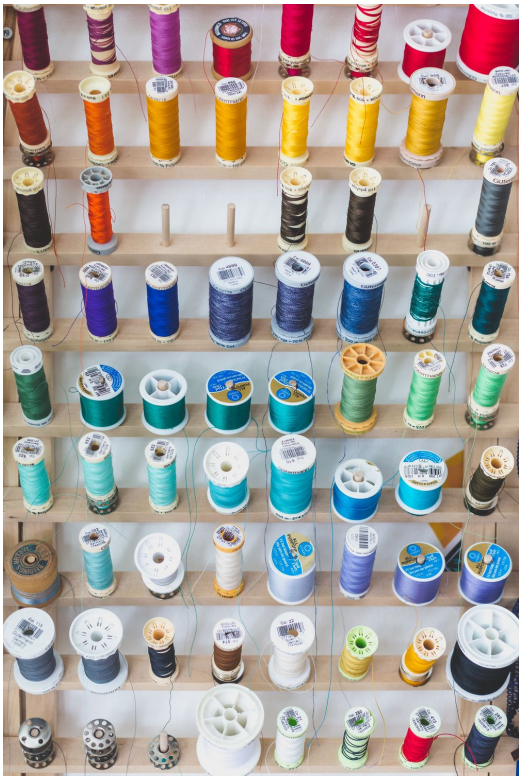
Le programme c'est la recette de la fabrication de la bière

le processus c'est fabriquer ce bon nectar et la boire entre ami(e)s, hum...

le thread c'est le gâteau qui ne fait pas grossir et on se partage les parts ! ☺

Auteur : Pascal Fougeray

Le fils avec tous ses fils a une sacrée bobine ?



Rappel sur les pointeurs ! ! !

*	Pointeur
(*)	Parenthèses nécessaires pour éviter une ambigüité
(*)()	Parenthèses suivant l'expression, indiquant qu'il s'agit d'une fonction, un **P** de fonction
(*)(void *)	Ces parenthèses contiennent en tout et pour tout un seul paramètre : un **P** *void* *
void *(*)(void *)	L'expression entière a un type : **void *.** C'est donc le type de la fonction proprement dite, celui de la valeur retournée
(void *(*)(void *))	Parenthèses autour de l'**expression entière** qui elle-même définit un type. Donc un transtypage.

Exemple :

```
// ici x est un pointeur sur un void
void *activite(void *x){
  // i = valeur (1ière étoile) d'un entier (cast) pointé par x
```

```
  int i =*((int *) x);
}

int *p; // p un pointeur qui pointe sur un entier
int a;  // a un entier
p=&a;   // p récupère l'adresse de a
```

1 Introduction

La programmation multithreading, de plus en plus utilisée surtout depuis la venue des microprocesseurs multi-coeurs.

Allez un peu de technologie avant d'entrer dans le vifs du sujet.

Si on prend un processeur ne serait-ce que pour ordinateur de bureau tel le Intelő Core™ i9-7960X X-series Processor, ce dernier contient 16 ciurs et 32 ***threads*** peuvent être exécutés en parallèles.

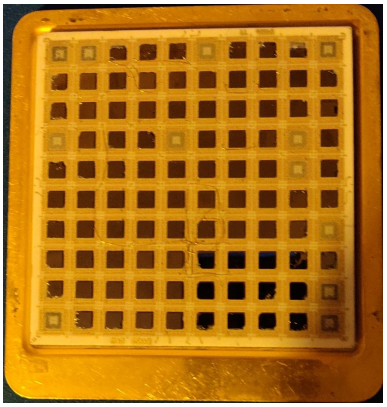
Le AMD Ryzen™ Threadripper™ 2970WX par encore sortie mais ça va venir lui possède 24 ciurs et peut exécutés 48 threads en parallèles

Quand à la famille des XEON, le Intelő Xeonő Processor E7-8894 v4 contient aussi 24 ciurs et peut exécutés 48 threads en parallèles

En 1995, création d'un standard d'interface pour l'utilisation des processus légers, il a pour nom ***pthread*** pour **POSIX thread**.

Les noyaux des SE sont tous aptes à gérer cela.

En 1985... celui que je vous ai montré !



2 Un thread c'est quoi, c'est pour quoi ?

Non ce n'est pas un fil... quoi que ?

La définition la plus légère... : **Processus léger.**

On parle aussi de **fil d'exécution** ou **d'activité**

Selon que l'on se place en système ou développeur, on ne voit pas un thread de la même façon.

— 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.

2.1 Les bénéfices pour le système sont

— Les threads utilisent les ressources du processus auquel ils appartiennent.
— Seules les ressources nécessaires à leur fonctionnement et à leur ordonnancement sont dupliquées
— La gestion des processus par le système consomme une quantité non négligeable de ressources (***overhead***)
— Création plus rapide que pour un processus (100 000 en moins de 2s !)
— Ne sollicite pas le gestionnaire de mémoire (MMU)
— Commutation rapide entre les threads
— Utilisation de plusieurs processeurs

2.2 Les bénéfices pour le développeur,

Ils s'agit de **fonctions** ou **méthodes**, au sein d'un même processus, qui peuvent s'exécuter simultanément, elles sont en **concurrence.**

Les **bénéfices** pour le **développeur** d'un point de vue applicatifs (applications)

- Simplification de l'expression et de l'implémentation du **parallélisme intrinsèque** d'une application (n fonctions en //)
  Facilite le partitionnement des programmes en tâches parallèles
  - Les interfaces graphiques sont toujours ***multi threadées***, pendant que l'application travaille, l'utilisateur peut interagir avec son interface.
    - Les services réseaux sont toujours ***multi threadés***, n clients pour m serveurs avec n >> m ! ! !
    - ...
  - Simplification de la gestion des évènements asynchrones (signaux)
    Un *thread* peut être associé à chaque évènement dont la gestion est alors codée de manière synchrone et donc plus simple.
  - Permet de tirer partie des architectures multi processeurs
- Les **Inconvénients** ou disons les précautions à prendre
  - Le développement d'applications, le codage, les tests et le débogage utilisant le parallélisme entre ses composants est bien plus délicat.
    - Combinatoire importante des états
    - L'utilisation d'une même bibliothèque (non réentrance) par plusieurs threads engendre souvent des corruptions de données
  - Rigueur et méthode de développement sont nécessaires pour obtenir les gains souhaités.
  - La Portabilité
    - Des limites varient d'un système à l'autre.
      - Le nombre maximum de threads par processus
      - La taille maximale de la pile d'un thread

Remarque : **Un processus de base n'a qu'un seul thread**

2.3 Avantages des threads / processus

- **Réactivité** (le processus peut continuer à s'exécuter même si certaines de ses parties sont bloquées),
- **Partage** de ressources (facilite la coopération, améliore la performance),
- **Économie** d'espace mémoire et de temps.
  Il faut moins de temps pour :
  - **créer**, **terminer** un thread
  - "**switcher**" entre 2 threads d'un même processus.

2.4 Threads Noyau / Threads Utilisateur

Il y a 2 types d'implémentation des threads

1. Les threads utilisateur qui ne sont pas connus du noyau !
   - L'état est maintenu en espace utilisateur.
   - Aucune ressource du noyau n'est allouée au thread.
   - Des opérations peuvent être réalisées indépendamment du système.
   - **Le noyau ne voit qu'un seul thread**
     Ce qui engendre que tout appel système bloquant une thread aura pour effet de bloquer son processus et par conséquent tous les autres threads du même processus.
2. Les threads Noyau, ils sont connus du noyau ! ce qui parait logique...
   - Les threads sont des entités du système (threads natives).
   - Le système possède un descripteur pour chaque thread.
   - Permettent l'utilisation des différents processeurs dans le cas des machines multiprocesseurs.

Lorsqu'un processus est créé, un seul flot d'exécution (thread) est associé au processus. Ce thread peut en créer d'autres.

3 Différences P & T

Sous Linux si on tape les commandes suivantes :
cat /proc/sys/kernel/pid-max renvoie **32768**
cat /proc/sys/kernel/threads-max renvoie **15859**
Que veulent dire ces 2 valeurs ?
On pourrait mal les interpréter, en effet on pourrait croire que l'on peut lancer jusqu'à 32768 processus et que seulement 15859 *threads*. Cela est faux ! ! !
On peut lancer des milliers de *threads* pour 1 seul processus !
Voyons ce que renvoient les 2 commandes suivantes : ***ps -elf | wc -l et ps -elfT | wc -l***

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# ps -elf | wc -l
115
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# ps -elfT | wc -l
231
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread#
```

J'en déduis que sur ma VM, j'ai
— 115 processus qui tournent et
— 231 threads,
le fait que ce soit le double n'est qu'un pur hasard !
En effet j'ai installé plusieurs services tels un serveur Web, un serveur de BDD et *docker*
La commande ***ps -elfT | grep apache2 | wc -l*** me renvoie 56 ...
j'ai donc 56 threads ***apache2*** qui attendent des clients
Pour **mysqld** c'est 28 et pour docker c'est 18.

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
ps aux | grep mysqld
mysql     594  0.0  0.0 686288   572 ?        Ssl  02:02   0:27 /usr/sbin/mysqld
root     3337  0.0  0.0  13080   948 pts/3    R+   15:38   0:00 grep mysqld
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
ps -p 594 -T
  PID  SPID TTY          TIME CMD
  594   594 ?        00:00:00 mysqld
  594   686 ?        00:00:00 mysqld
  594   694 ?        00:00:00 mysqld
  594   732 ?        00:00:01 mysqld
  594   733 ?        00:00:01 mysqld
  594   734 ?        00:00:01 mysqld
  594   735 ?        00:00:01 mysqld
  594   736 ?        00:00:01 mysqld
  594   737 ?        00:00:01 mysqld
  594   738 ?        00:00:01 mysqld
  594   739 ?        00:00:01 mysqld
  594   740 ?        00:00:01 mysqld
  594   741 ?        00:00:01 mysqld
  594   753 ?        00:00:00 mysqld
  594   803 ?        00:00:01 mysqld
  594   804 ?        00:00:02 mysqld
  594   805 ?        00:00:02 mysqld
  594   806 ?        00:00:00 mysqld
  594   807 ?        00:00:00 mysqld
  594   808 ?        00:00:00 mysqld
  594   818 ?        00:00:02 mysqld
  594   819 ?        00:00:01 mysqld
  594   883 ?        00:00:00 mysqld
  594   884 ?        00:00:01 mysqld
  594   902 ?        00:00:00 mysqld
  594   903 ?        00:00:00 mysqld
  594  1998 ?        00:00:00 mysqld
```

Même PID & SPID change !

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
ps -p 594 -T | wc -l
28
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
```

On peut en déduire que les threads sont très utilisés pour rendre des services ?
La réponse est oui ! ! !

3.1 Les identifiants !

Pour les Processus nous avions le pid que le processus courant peut récupérer à l'aide de l'appel système ***getpid***() et aussi l'identifaint de son père à l'aide de l'appel système ***getppid***().

Pour les threads c'est un peu plus ambiguë !

Pour l'btention de l'identité de la thread courante on doit utiliser l'appel système

— ***pthread_t pthread_self (void);*** qui renvoie l'identificateur du thread courant.

On peut faire une comparaison entre 2 identificateurs de threads à l'aide de l'appel système ***pthread_equal();***

— ***pthread_t pthread_ equal(pthread_t t1, pthread_t t2);***

Test d'égalité : renvoie une valeur non nulle si t1 et t2 identifient la même thread

```
1  //bon-identifiant.c
   #include <stdio.h>
   #include <pthread.h>
   #include <unistd.h>
   #include <syscall.h>
6
   pthread_t liste[3];

   void* activite(void *i){
          printf("\t- je suis le %d-eme thread de pere %d: pthread_self() [%u] et de pid [%d]\n",
11               *((int *)i)+1, getppid(), (int)pthread_self(), (int)syscall(SYS_gettid));
          sleep(30); // pour avoir le temps de faire le pstree -p PID
          return NULL;
   }
   int main(){
16        printf("=== JE SUIS LE PROCESSUS PRINCIPAL D'ID %d ===\n", getpid());
          int i;
          for(i=0; i<3; i++) {
                 pthread_create(&liste[i], NULL, activite, (void *) &i);
                 sleep(1);
21        }
          printf("J'attends que mon dernier thread se termine...\n");
          printf("Mais avant lancez pstree -p %d dans une autre console vous avez 30s \n", getpid());
          for (i=0; i<3; i++) {
                 int rtn = pthread_join(liste[i], NULL);
26               if (!rtn) {
                        printf("Fin du thread de liste[%d] [%u]\n", i, (int)liste[i]);
                 }
          }
          printf("Tous les threads sont termines ;-)\n");
31        return 0;
   }
```

Résultat !

3.2 Différences entre P et T

Les 2 permettent d'effectuer des tâches en parallèles, cependant il existe de nombreuses différences entre les 2 !

— **La vitesse de création**

Je commence par celle-ci juste avec un tout petit programme pour en créer beaucoup de chaque et voir la comparaison de temps.

Soient les 2 codes suivants qui ne sont pas des exemples de programmation !

```
// fichier plein-fork.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>

int main(int argc, char *argv[]){
        int i = 0;
        int j = atoi(argv[1]);
        int pid;
                for (i=0; i<j;i++){
                pid=fork();
                if (pid==0){ // fils
                        pause(); // ne fait rien et attend
                        exit(0);
                }
        }
        // décommenter pour afficher
        // system ("ps jf");
        // system ("ps | grep plein-fork | wc -l");
        // le pere tue tous ses fils et lui-meme
        system("killall plein-fork");
        return  0;
}

// fichier plein-thread.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <sys/types.h>
void* jedors(void *arg){
        pause(); // // ne fait rien et attend
        pthread_exit(NULL); // Fermeture du thread
}
int main(int argc, char *argv[]){
        int i = 0;
        int j = atoi(argv[1]);
        int erreur;
        pthread_t thread[j];
        while(i < j) {
                erreur = pthread_create(&(thread[i]), NULL, &jedors, NULL);
                if (erreur != 0)
                        printf("\nThread ne peut être créé : [%s]", strerror(erreur));
                i++;
        }
        // décommenter pour afficher
        // system("ps -T | grep plein-thread | wc -l");
```

```
25          // system("ps -mT");

            while(i < j) {
            // Thread P attend autres threads
            // pthread_join(thread[i], NULL);
30          }
            return 0;
    }
```

On crée 10000 processus et 10000 *threads* !

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# time plein-fork 10000
Complété

real    0m0,829s
user    0m0,012s
sys     0m0,328s
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# time plein-thread 10000

real    0m0,269s
user    0m0,004s
sys     0m0,228s
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread#
```

On peut constater une différence significative même si ce n'est pas très précis !

— **L'empreinte mémoire**

Cette importante différence de temps vue précédemment vient des éléments créés lors de l'instanciation d'un ***thread*** ou d'un processus.

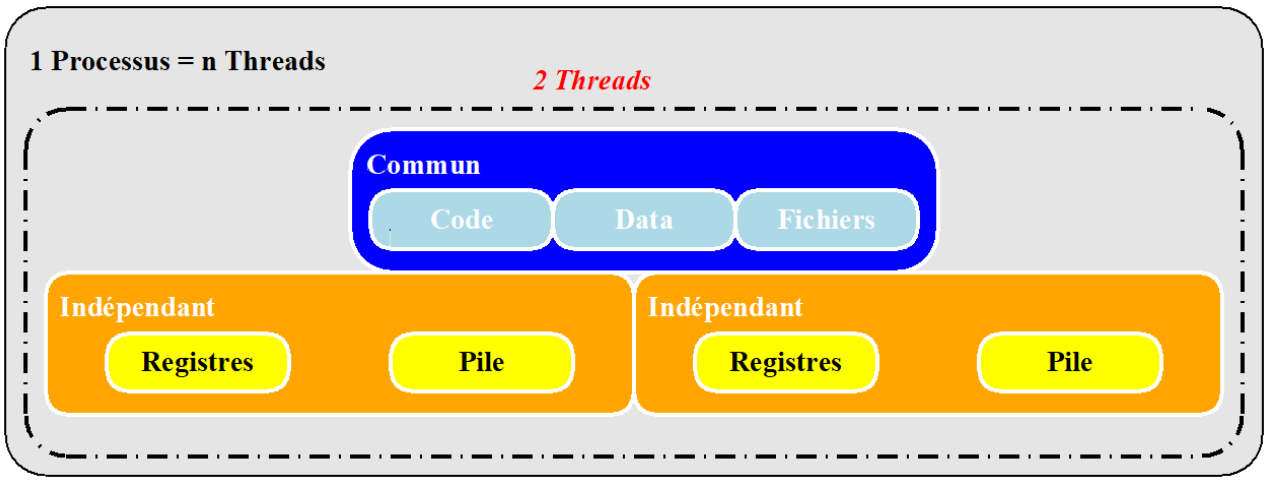
Quand on crée un nouveau processus, ce dernier possède son **code**, ses **données**, ses **fichiers**, ses **registres** et sa **pile**, en l'occurrence le fils (et non le fil ! ! !) est alors **dissociés** de son créateur le père.

Le nouveau processus commence sa vie à l'endroit où il a été créé.

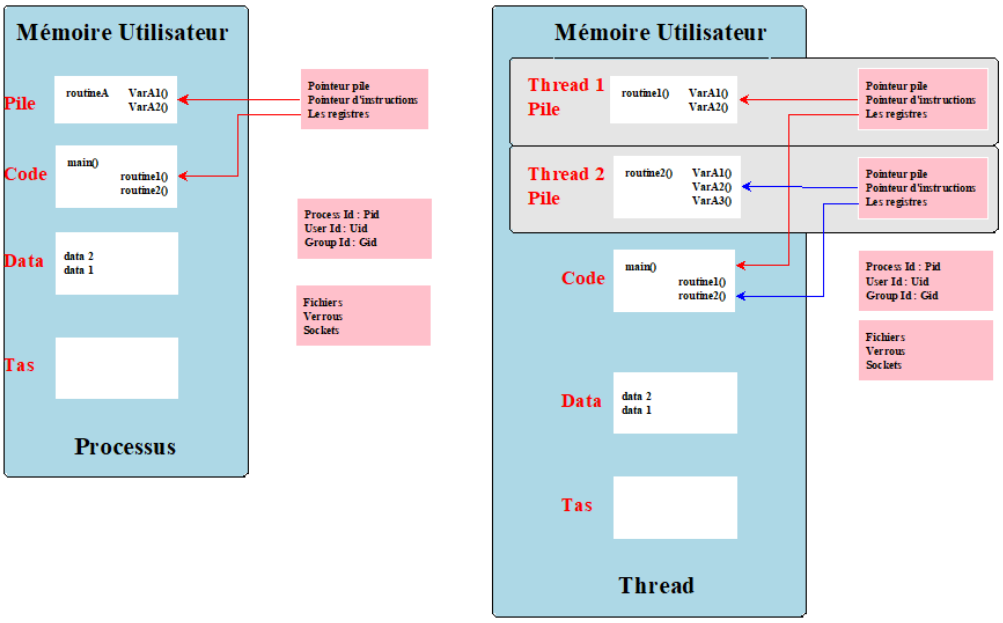
Un ***thread*** lui **partage** le code, les données et les fichiers avec son créateur.

Il possède seulement ses propres **registres** et sa propre **pile d'exécution.**

Un ***thread*** a un travail à effectuer **on lui indique juste la fonction qu'il devra exécuter à sa création**



2



3.3 Différences de comportement entre *fork*() et *pthread_create*()

— Lors de la création d'un nouveau **processus** à l'aide de l'appel système ***fork***(), le processus fils hérite du contexte du père tel qu'il est au moment de cet appel système *fork*().
Puis, chaque processus possède un contexte séparé, donc un espace d'adressage et de données séparés.
Les 2 processus peuvent communiquer avec les moyens de communications qui leur sont donnés, par exemple des **tubes** ou des **fils de messages** (pas vus en cours !).

— Lors de la création d'un nouveau **thread** à l'aide de l'appel système ***pthread_create()***, l'espace d'adressage et de données sont partagées, il n'y a pas de duplication. !
Mais... il faut faire attention à la synchronisation ! ! !
Le second fil (thread fils) démarre a la fonction appelée, le premier fil (thread père) continue après l'appel système thread_create().
L'exécution à lieu en pseudo-parallèle

Explication par 2 codes

```
// HelloPID.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

// Une variable globale appartient à toutes les fonctions !
int i;

int main() {
        // Variables
        pid_t pid;
        int status, waitsignal;
        // Initialisation
        i=0;
        // Création du fils
        pid =(int)fork();
        if (pid==0) { // fils !
            i+=10;
                printf("Bonjour du Processus fils et i vaut :%d\n", i);
```

```
23                  i+=20;
                    printf("Bonjour du Processus fils et i vaut :%d\n", i);
                    exit(EXIT_SUCCESS);
            }
            else{ // Père !
28              i+=1000;
                    printf("Bonjour du Processus père et i vaut :%d\n", i);
                    i+=2000;
                    printf("Bonjour du Processus père et i vaut :%d\n", i);
                    waitsignal = wait(&status); // Père attend fils !
33                  printf("\n\t\t Fils terminé ouf ! \n");
            }
            return 0;
    }

    // HelloTHREAD.c

    #include <stdio.h>
 4  #include <stdlib.h>
    #include <unistd.h>
    #include <sys/types.h>
    #include <pthread.h>

 9  // Une variable globale appartient à toutes les fonctions !
    int i;

    void *addition(){ // Fait par le thread Fils qu'on ralentit
            //sleep(1); // Pour être certain que le fils exécute après le père
14          i+=10;
            printf("Bonjour du thread fils et i vaut :%d\n", i);
            i+=20;
            printf("Bonjour du thread fils et i vaut :%d\n", i);
    }
19
    int main() {
            // Variables
            pthread_t thread_id;

24          // Initialisation
            i=0;
            // Création du thread fils
            pthread_create(&thread_id, NULL, &addition, NULL);

29          // Fait par le thread Père
            sleep(1); // Pour être certain que le père exécute après le fils
            i+=1000;
            printf("Bonjour du thread père et i vaut :%d\n", i);
            i+=2000;
34          printf("Bonjour du thread père et i vaut :%d\n", i);
            pthread_join(thread_id, NULL); // thread Père attend thread fils !
            printf("\n\t\t Thread fils terminé ouf ! \n");
            return 0;
    }
```

Explications sur les résultats obtenus !

— Pour le programme ***helloPID***, comme la variable **globale** i est initialisée à 0 et que l'appel système *fork*() a lieu après, il y **a duplication** de cette variable globale et l'addition (incrémentation **i+=10**

ou **i+=1000** etc...) dans la fonction ***main***, les résultats dans les processus Père et fils sont cohérents et surtout **indépendants** !

— Pour le programme ***helloTHREAD***, comme la variable **globale** i est initialisée à 0 et que l'appel système *pthread_create*() a lieu après, il n'y **a pas duplication** de cette variable globale et la fonction ***addition***() attachée au thread fils est exécutée avant ou après (on ne peut pas le prévoir !) l'addition (incrémentation **i+=1000** etc...) dans la fonction ***main***, les résultats dans les Threads Père et fils sont **incohérents** et surtout **dépendants** !

Selon que l'on commente ou pas les 2 lignes sleep(1) du programme ***helloTHREAD***, le programme renvoie des résultats différents comme le montre la figure suivante !

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/diff_fork_pthread_create $
./helloPID
Bonjour du Processus père et i vaut :1000
Bonjour du Processus fils et i vaut :10
Bonjour du Processus fils et i vaut :30
Bonjour du Processus père et i vaut :3000

                Fils terminé ouf !
```

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/diff_fork_pthread_create
./helloThread
Bonjour du thread fils et i vaut :10
Bonjour du thread fils et i vaut :30
Bonjour du thread père et i vaut :1030
Bonjour du thread père et i vaut :3030

                Thread fils terminé ouf !

root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/diff_fork_pthread_create
./helloThread
Bonjour du thread père et i vaut :1000
Bonjour du thread père et i vaut :3010
Bonjour du thread fils et i vaut :1010
Bonjour du thread fils et i vaut :3030

                Thread fils terminé ouf !
```

4 l'API C

La programmation par *thread* repose sur des fonctionnalités (fonctions ou méthodes) sous-jacentes que l'on appelle aussi activités.

— Soit mises à disposition par l'OS

— Soit développées par un programme plus simple qu'un OS.

En C et sous *gnu*/linux en particulier, mais pas que, la bibliothèque de plus bas niveau est la bibliothèque ***pthread*** pour Posix Thread.

Posix définissant un standard d'appels systèmes (plus d'informations ici :
https://fr.wikipedia.org/wiki/Appel_syst%C3%A8me),
par exemple ***fork***() est un appel système.
Un système se conforme à un standard existant.

La bibliothèque ***pthread*** est de loin la plus répandue, elle est disponible sur tous les systèmes Unix et aussi sur Windows
(https://sourceforge.net/projects/pthreads4w/) !

La bibliothèque ***pthread*** permet de gérer les *thread* ainsi que les mécanismes de verrouillage nécessaires pour accéder aux ressources partagées !

Les fonctionnalités offertes par cette bibliothèque sont catégorisées de la façon suivante :

— Gestion des **threads**
— Gestion des **attributs**
— Gestion des **exclusions mutuelles**
— Gestion des **variables conditionnelles**
— Gestion des **verrous de lecture/écriture**
— Gestion des **contextes pour chaque threads**
— Gestion du n**ettoyage des ressources**

Nous ne verrons pas tout cela, du moins pas en 2h de CM et 1h30 de TP !

Pour compiler un programme avec le compilateur ***gcc*** vous devrez ajouter ***-pthread*** ! ! !
Sinon vous obtiendrez au moment de l'édition des liens un message d'erreur de ce type
/tmp/ccdBAFHI.oă : Dans la fonction ńămainăżă :
thread-identifiant.c :(.text+0x6d)ă : référence indéfinie vers ńăpthread_createăż

thread-identifiant.c :(.text+0x8a)ă : référence indéfinie vers ńăpthread_createăż
thread-identifiant.c :(.text+0xd7)ă : référence indéfinie vers ńăpthread_joinăż
thread-identifiant.c :(.text+0xe8)ă : référence indéfinie vers ńăpthread_joinăż
collect2 : error : ld returned 1 exit status

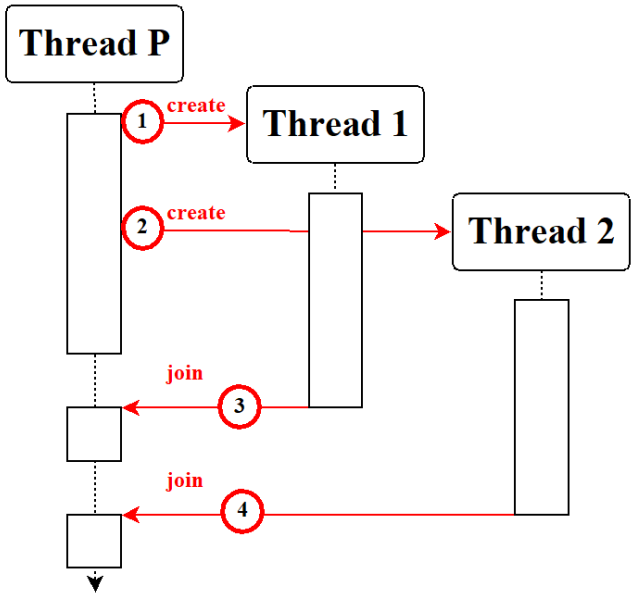
ld étant l'éditeur des liens !

5 Anatomie d'un simple programme "threadé"

Tout programme ***multithread*** simple consiste en un ***thread* principal**, le créateur et les fonctions associées que les **threads fils** exécuteront.

Les modèles de threads déterminent la manière dont les threads sont créés et gérés.

Ils peuvent être créés en une fois ou sous certaines conditions.




```
#include <pthread.h>

void *tache1(void *X){ //définit la tâche exécutée par Thread1
    //...
    printf("je suis thread 2 \n");
    pthread_exit(NULL);
}

void *tache2(void *X){ //définit la tâche exécutée par Thread2
    //...
    printf("je suis thread 2 \n");
    pthread_exit(NULL);
}

int main(int argc, char *argv[]){
    pthread_t Thread1,Thread2; // declaration des TID threads

    pthread_create(&Thread1,NULL,tache1,NULL); // create threads
    pthread_create(&Thread2,NULL,tache2,NULL);
    // Autre travail pour le Thread P
    pthread_join(Thread1,NULL); // Thread P attend Thread 1
    pthread_join(Thread2,NULL); // Thread P attend Thread 2
    return(0);
}
```

6 Création et exécution des threads

6.1 Création

Chaque thread d'un processus est caractérisé par un identifiant de thread !
Il faut utiliser le type **pthread_t**.
Lors de sa création, chaque thread exécute une fonction (activité) de thread.
Il s'agit d'une fonction ordinaire contenant le code que doit exécuter le thread.
Lorsque la fonction se termine, le thread se termine également.
Sous GNU/Linux, les fonctions de thread ne prennent
— qu'un seul paramètre de type ***void**** et
— ont un type de retour ***void****.
Ce paramètre est l'argument de thread : GNU/Linux passe sa valeur au thread sans y toucher.
Le programme peut utiliser ce paramètre pour passer des données à un nouveau thread.
Tout comme, il peut utiliser la valeur de retour pour faire en sorte que le thread renvoie des données à son Thread main lorsqu'il se termine.
La fonction ***pthread_create()*** crée un nouveau thread.
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
Voici les 4 paramètres dont elle a besoin :

1. Un pointeur vers une variable ***pthread_t***, dans laquelle l'identifiant du nouveau thread sera stocké ;
2. Un pointeur vers un objet d'attribut de thread.
   Cet objet contrôle les détails de l'interaction du thread avec le reste du programme.
   Si on passe **NULL** comme argument de thread, le thread est créé avec les attributs par défaut.
3. Un pointeur vers la fonction (**activité**) de thread.
   Il s'agit d'un pointeur de fonction ordinaire de type : *void** (*)(*void**) ;
   Exemple (le minimum ! ! !) A comprendre par ***c'est sur*** ! ! !

```
void *activite(void *pasutilisé) {
}
int main(){
  pthread_t thread_id;
  pthread_create (&thread_id, NULL, activite, NULL );
}
```

4. Une valeur d'argument de thread de type ***void**.
   Quoi que vous passiez, l'argument est simplement transmis à la fonction de thread lorsque celui-ci commence à s'exécuter.

Exemple : programme affichant Bonjour (le *Thread* Esclave ou fils) et Bonsoir (le *Thread* Maitre ou père) ...

```
// fichier Bonjour-Bonsoir.c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h> // pour sleep

void * affiche_bonjour(void * inutilise) {
        while (1){// Affiche bonjour et ne finit jamais.
                printf("Bonjour\n");
                sleep(1)        // pour ralentir
        }
        return 0;
}

int main () {
        pthread_t thread_id; // Identifiant du thread
        // On Crée un nouveau thread exécutant la fonction affiche_bonjour
        pthread_create (&thread_id, NULL, &affiche_bonjour, NULL );
```

```
        // Affiche Bonsoir  et ne finit jamais.
19      while (1){// Affiche bonjour et ne finit jamais.
                printf("Bonsoir\n");
                sleep(1)        // pour ralentir
        }
        return 0;
24  }
```

```
 ~/L3/TD_TP/thread/C $
gcc -lpthread Bonjour-Bonsoir.c -o Bonjour-Bonsoir
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
./Bonjour-Bonsoir
Bonsoir          PID: 2700
Bonjour          PID: 2700
Bonjour          PID: 2700
Bonsoir          PID: 2700
Bonjour          PID: 2700
Bonsoir          PID: 2700
^C
```

2 Threads
1 PID !

6.2 Identifier un thread

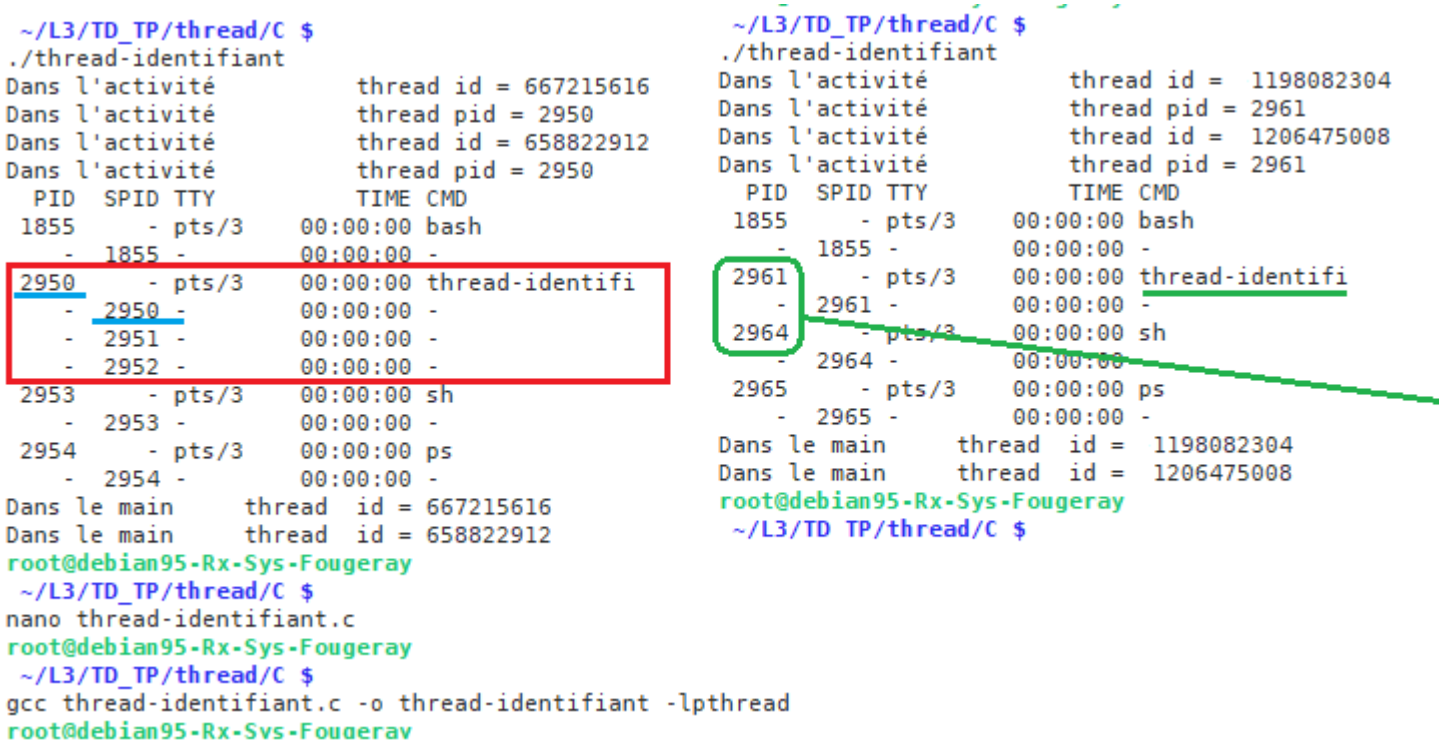
Si on ne s'occupe que du PID, on ne le peut pas (normalement...) avec la commande ps ... !
Voyons cela avec le programme suivant

```
1   #include <sys/types.h>
    #include <unistd.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <pthread.h>
6
    void* appel(void* args) {
            printf("Dans l'activité \t thread id = %d\n", pthread_self());
            printf("Dans l'activité \t thread pid = %d \n", getpid());
            sleep(1); // pour que le thread P ait le temps de faire ps -mT
11          return NULL;
    }

    int main() {
            pthread_t thread1=33, thread2=1664; // declare thread  identifiant
16          pthread_create(&thread1, NULL, appel, NULL);
            pthread_create(&thread2, NULL, appel, NULL);
            // sleep(2); // si on décommente les valeurs TID  changent pas,
            // mais les PID sont perdus dans ps -mT
            system("ps -mT");
21          printf("Dans le main \t thread  id = %d\n", thread1);
            printf("Dans le main \t thread  id = %d\n", thread2);
            pthread_join(thread1, NULL);
            pthread_join(thread2, NULL);
            return 0;
26  }
```

Le Thread principal lance 2 activités et chacune affiche son identifiant !

— Si l'appel système de la fonction main ***system("ps -mT");*** est lancé **avant** la fin des 2 activités alors on peut voir que chaque thread a un PID...

— Si l'appel système de la fonction main ***system("ps -mT");*** est lancé **après** la fin des 2 activités alors on ne peut pas voir que chaque thread a un PID...



6.3 Retour et fin

Dans des circonstances normales, un thread peut se terminer de 2 façons.

1. La valeur de retour de la fonction de thread est considérée comme la valeur de retour du thread. (voir ***join*** un peu plus loin ! ! !)
2. Un thread peut également se terminer explicitement en appelant l'appel système ***pthread_exit***().

   Cette fonction peut être appelée depuis la fonction de thread ou depuis une autre fonction appelée directement ou indirectement par la fonction de thread.

   L'argument de ***pthread_exit***() est la valeur de retour du thread.

6.4 Transmettre des données à un thread

L'argument de thread est une méthode pratique pour passer des données à un thread.

Comme son type est ***void****, cependant, on ne peut pas passer beaucoup de données directement en l'utilisant.

Au lieu de cela, utiliser l'argument de thread pour passer un **pointeur** vers une **structure** ou un **tableau** de données.

Une technique souvent utilisée est de définir une structure de données pour chaque argument de thread, qui contient les paramètres attendus par la fonction de thread.

Il est possible d'utiliser le 4ème paramètre de ***pthread_create()*** pour passer **l'adresse** d'un objet quelconque à un thread (variable simple, tableau, structure) au moment de la création de la tâche.

Il est bien évident que la durée de vie de cette variable doit être supérieure ou égale à la durée de vie du thread qui l'utilise.

S'il y a plusieurs paramètres, on définit une **structure**, on créée une instance de thread par tache avec les paramètres désirés et on passe **l'adresse** de la structure au moment de la création du thread.

Ensuite la thread récupère l'adresse en paramètre. Il suffit alors d'initialiser **un pointeur du bon type avec l'adresse**, et on a alors accès aux champs.

Si vous ne comprenez pas le code ci-dessous, je vous invite à lire le cours sur les structures ! ! !

```
//passage-parametre.c
#include <stdio.h>
#include <pthread.h>
struct data{
    char const *id;
```

```
    int n;
};
void *tache (void *p_data){
    struct data *p = p_data; // un cast
    int i;
    for (i = 0; i < p->n; i++){
        printf ("Hello world %s (%d)\n", p->id, i);
        }
    return 0;
}
int main (void){
    pthread_t ta;
    pthread_t tb;
    struct data data_a = { "A", 5 };
    struct data data_b = { "B", 7 };
    pthread_create (&ta, NULL, tache, &data_a);
    pthread_create (&tb, NULL, tache, &data_b);
    pthread_join (ta, NULL); // en commentaire pour Synchroniser des threads
    pthread_join (tb, NULL);
    return 0;
}
```

Le résultat est :

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# gccp passage-parametre.c -o passage-parametre
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# passage-parametre
Hello world B (0)
Hello world B (1)
Hello world B (2)
Hello world B (3)
Hello world B (4)
Hello world B (5)
Hello world B (6)
Hello world A (0)
Hello world A (1)
Hello world A (2)
Hello world A (3)
Hello world A (4)
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread#
```

Explication du code, notamment la manipulation des pointeurs et des structures ☺

```
 4  struct data{
 5      char const *id;
 6      int n;
 7  };
 8  void *tache (void *p_data){
 9      struct data *p = p_data;
10      int i;
11      for (i = 0; i < p->n; i++){
12          printf ("Hello world %s (%d)\n", p->id, i);
13          }
14      return 0;
15  }
16  int main (void){
17      pthread_t ta;
18      pthread_t tb;
19      struct data data_a = { "A", 5 };
20      struct data data_b = { "B", 7 };
21      pthread_create (&ta, NULL, tache, &data_a);
22      pthread_create (&tb, NULL, tache, &data_b);
23      pthread_join (ta, NULL);
24      pthread_join (tb, NULL);
25      return 0;
```

6 3 4 5 5 1 2

1. Lignes 19 et 20, on déclare 2 structures de données : ***data_a*** et ***data_b***

2. Lignes 21 et 22, on passe en argument les 2 adresses de ces 2 structures ***&data_a*** et ***&data_b***
3. Le pointeur ***p_data*** de la ligne 8 récupère l'une de ces 2 adresses (selon la ligne 21 ou 22 ! ! !), il pointe sur un type ***void***, il faudrait le faire pointer sur un type ***struct data***
4. On déclare un second pointeur **P** qui lui pointe sur une donnée de type ***struct data*** et prend la même valeur que ***p_data***
5. ***p->n*** représente donc le contenu de la variable ***n*** de
   (a) soit la structure ***data_a***
   ou
   (b) soit la structure ***data_b***,
   idem pour p->id
6. Rappel de la structure de type ***data***.

Vive les pointeurs ?

6.5 Valeurs de retour des threads

Si le second argument passé à l'appel système ***pthread_join*** n'est pas NULL, la valeur de retour du thread sera stockée à l'emplacement pointé par cet argument.

La valeur de retour du thread, comme l'argument de thread, est de type void*.

Si on désire renvoyer un simple **int** ou un autre pe*tit nombre, on peut le faire facilement en convertissant la valeur en void* puis en le reconvertissant vers le type adéquat après l'appel système de* ***pthread_join().*** voir l'exemple suivant !

6.5.1 Un exemple simple avec des entiers (*int*)

```
//exemple_join.c
/* Un exemple sur pthread_join().
   Voir que l'on obtient la valeur de retour de chaque
   thread dans les appels pthread_join () respectifs. */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void * activite(void *x){ // ici x est un pointeur sur un void
  int i =*((int *) x); // i = valeur (1ière étoile) d'un entier (cast) pointé par x
  printf("valeur de i : %d\n",i);
  for (; i < 5; ++i) {
    printf("thread %d (loop #%d)\n", *((int *) x), i);
    sleep(1); // pour ralentir
  }
  return ((void *)x);
}
int main(void){
  void *p1, *p2; // P1 et P2 des pointeurs sur des types void !
  pthread_t t1, t2;
  int n1 = 1, n2 = 2;
  pthread_create(&t1, NULL, activite, &n1); // on passe l'adresse !
  pthread_create(&t2, NULL, activite, &n2);
  printf("Création des threads\n");
  printf("Threads principal attend ses threads fils...\n");
  pthread_join(t1, &p1); // 1 valeur de retour
  pthread_join(t2, &p2); // 1 seconde valeur de retour
  // P1 et P2 deviennent des pointeurs sur des entiers (cast)
  // l'étoile devant permet de récupérer la valeur de ce qui pointé par le pointeur.
  printf("thread %d returned %d\n", n1, *((int *)p1));
  printf("thread %d returned %d\n", n2, *((int *)p2));
  return (0);
}
```

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C/mff.devnull.cz $
./exemple_join
Création des threads
Threads principal attend ses threads fils...
valeur de i : 1
thread 1 (loop #1)
valeur de i : 2
thread 2 (loop #2)
thread 2 (loop #3)
thread 1 (loop #2)
thread 2 (loop #4)
thread 1 (loop #3)
thread 1 (loop #4)
thread 1 returned 1
thread 2 returned 2
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C/mff.devnull.cz $
```

Thread 1 commence à 1 car n=1

Thread 2 commence à 2 car n=2

Résultat :

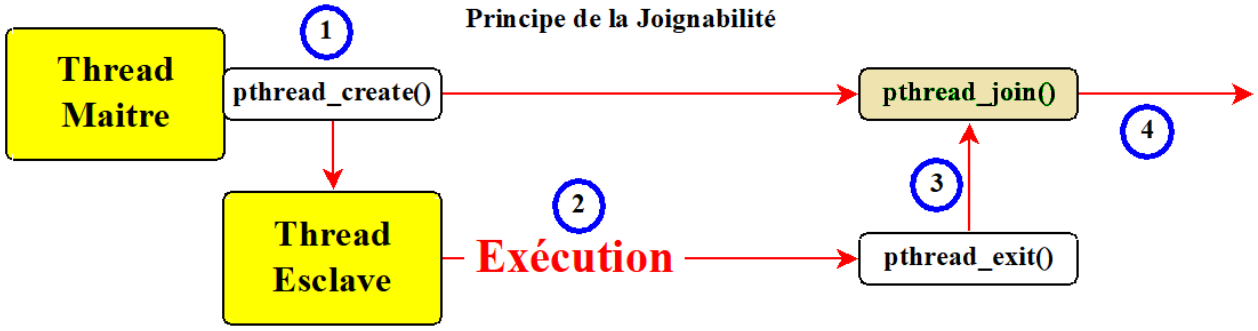
6.5.2 Un exemple plus “compliqué” avec une structre (*struc*)

```
//passage-parametre2.c
#include <stdio.h>
#include <pthread.h>
struct data{
   char const *id;
   int n;
};
void *tache (void *p_data){
   struct data *p = p_data; // un cast
   int i;
   for (i = 0; i < p->n; i++){
      printf ("Hello world %s (%d)\n", p->id, i);
      }
    printf("Valeur de i est 3*3 que je retourne %d \n",i*i);
        i=i*i;
    return (void*) i;
}
int main (void){
   int valeur_retourA;
   pthread_t ta;
   struct data data_a = { "A", 3 }; // Fait 3 fois
   pthread_create (&ta, NULL, tache, &data_a);
   pthread_join (ta, (void*) &valeur_retourA);
   printf("La valeur de retour est %d \n", valeur_retourA);
   return 0;
}
```

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
./passage-parametre2
Hello world A (0)
Hello world A (1)
Hello world A (2)
Valeur de i est 3*3 que je retourne 9
La valeur de retour est 9
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/C $
```

6.6 Synchroniser des threads

LA JOIGNABILITE ! ! ! <– ça existe ce mot ? ? ?



Questions !

1. Que se passe-t-il si le thread principal se termine avant les threads qu'il a créé ?
   Et bien tout le monde est terminé et la gestion de la mémoire... bof bof...
2. Que se passe-t-il si un thread d'activité fait un appel système ***exit(1)*** ?
   Le **FIN** sera-t-il affiché ? Réponse : **NON** !

```
//thread-fils-quitte.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
void *activite(void *arg) {
  sleep(1);
  exit(1);
  //pthread_exit(NULL);
}
int main() {
  pthread_t tid;
  int rep;
  tid=pthread_create(&tid, NULL, &activite, NULL);
  printf("thread creee\n");
  pthread_join(tid, NULL);
  printf("FIN\n");
  return 0;
}
```

3. Que se passe-t-il si un thread d'activité fait un appel système ***exit(1)*** le retour... ?
   Le **FIN** sera-t-il affiché ? Réponse : **OUI** !

```
//thread-fils-quitte-join.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
void *activite(void *arg) {
  exit(1);
}
int main() {
  pthread_t tid;
  tid=pthread_create(&tid, NULL, &activite, NULL);
  printf("thread creee\n");
  pthread_join(tid, NULL);
  printf("FIN\n");
  return 0;
}
```

```c
//thread-fils-quitte.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
void *activite(void *arg) {
  sleep(1);
  exit(1);
}
int main() {
  pthread_t tid;
  tid=pthread_create(&tid, NULL, &activite, NULL);
  printf("thread creee\n");
  sleep(5);
  printf("FIN\n");
  return 0;
}
```

```c
//thread-fils-quitte-join.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
void *activite(void *arg) {
  exit(1);
}
int main() {
  pthread_t tid;
  tid=pthread_create(&tid, NULL, &activite, NULL);
  printf("thread creee\n");
  pthread_join(tid, NULL);
  printf("FIN\n");
  return 0;
}
```

```
~/L3/TD_TP/thread/C/Synchro $
./thread-fils-quitte
thread creee
root@debian95-Rx-Sys-Fougeray
~/L3/TD_TP/thread/C/Synchro $
./thread-fils-quitte-join
thread creee
FIN
root@debian95-Rx-Sys-Fougeray
~/L3/TD_TP/thread/C/Synchro $
```

Conclusion :

Il faut synchroniser **le thread principal avec les thread fils** et vice-versa un peu comme le ***wait***() et les ***fork***()...

Une solution possible est de forcer main à attendre la fin des autres threads.

Ce dont nous avons besoin est une fonction similaire à ***wait*** pour les processus qui attendent la fin d'un thread au lieu de celle d'un processus.

Cette fonction est ***pthread_join()***, qui prend 2 arguments :

1. un pointeur vers une variable void* qui recevra la valeur de retour du thread s'étant terminé.
   Si la valeur de retour du thread n'est pas utile, on passe NULL pour ce second argument.

Cette fonction renvoie un entier, celui-ci vaudra 0 si le thread se termine correctement, dans le cas contraire, la valeur sera le code correspondant à l'erreur rencontrée.

1. Le premier argument de la fonction est le **TID** (l'identifiant du thread à attendre) du thread pour lequel nous désirons attendre la fin de son exécution ou disons de son activité.
2. Le second argument est (accrochez vous ! ! !) **un pointeur de pointeur**.
   Ce pointeur de pointeur servira à récupérer l'adresse de la valeur renvoyée par le ***pthread_exit()*** du thread.
   Nous verrons 2 applications de pthread_join, la première sera une amélioration du programme du cours précédent pour que celui-ci attende la fin du thread avant de se couper.
   La seconde application permettra de récupérer grace à pthread_join() la valeur renvoyée par le pthread_exit().

Reprenons l'exemple ***//passage-parametre.c*** du paragraphe **Transmettre des données à un thread**.

Mettons en commentaire les 2 lignes ***pthread_join (ta, NULL);*** et ***pthread_join (tb, NULL);***

On relance le programme, le résultat donne : **RIEN NE S'AFFICHE ! ! !**

Normal le thread principal est terminé avant les autres et un thread ne peut exister sans le thread principal.

Mais plus étrange ou disons plus normal si on y réfléchi bien !

Créons d'abord le thread B avant le A et mettons en commentaire le join du thread B comme le montre le source suivant

```c
#include <stdio.h>
#include <pthread.h>
struct data{
    char const *id;
    int n;
};
void *tache (void *p_data){
    struct data *p = p_data; // un cast
    int i;
    for (i = 0; i < p->n; i++){
       printf ("Hello world %s (%d)\n", p->id, i);
       }
    return 0;
}
int main (void){
    pthread_t ta;
    pthread_t tb;
    struct data data_a = { "A", 5 };
```

```
19      struct data data_b = { "B", 7 };
        //pthread_create (&ta, NULL, tache, &data_a);
        pthread_create (&tb, NULL, tache, &data_b);
        pthread_create (&ta, NULL, tache, &data_a);
        pthread_join (ta, NULL);
24      //pthread_join (tb, NULL);
        return 0;
    }
```

Les résultats seront les suivants, ce n'est pas complètement aléatoire, c'est logique et explicable.

Résultat **Code**

```
Hello world A (0)
Hello world A (1)
Hello world A (2)
Hello world A (3)
Hello world A (4)
```

```
//pthread_create (&ta, NULL, tache, &data_a);
pthread_create (&tb, NULL, tache, &data_b);
pthread_create (&ta, NULL, tache, &data_a);
pthread_join (ta, NULL);
//pthread_join (tb, NULL);
return 0;
```

Cas 1

```
Hello world B (0)
Hello world B (1)
Hello world B (2)
Hello world B (3)
Hello world B (4)
Hello world B (5)
Hello world B (6)
Hello world A (0)
Hello world A (1)
Hello world A (2)
Hello world A (3)
Hello world A (4)
```

```
pthread_create (&ta, NULL, tache, &data_a);
pthread_create (&tb, NULL, tache, &data_b);
//pthread_create (&ta, NULL, tache, &data_a);
pthread_join (ta, NULL);
//pthread_join (tb, NULL);
return 0;
```

Cas 2

```
Hello world B (0)
Hello world B (1)
Hello world B (2)
Hello world B (3)
Hello world B (4)
Hello world B (5)
Hello world B (6)
```

```
pthread_create (&ta, NULL, tache, &data_a);
pthread_create (&tb, NULL, tache, &data_b);
//pthread_create (&ta, NULL, tache, &data_a);
//pthread_join (ta, NULL);
pthread_join (tb, NULL);
return 0;
```

Cas 3

7 Gestion des signaux en présence de threads

Nous avons vu dans le cours sur les signaux, que ces derniers étaient délivrés aux processus et non aux threads.

De plus un signal a besoin de connaitre le PID du processus et tous les threads issus d'un thread principal ou main possèdent le même PID (???)!!!

1. La gestion d'un signal est assurée pour l'ensemble de l'application en employant l'appel système ***sigaction();*** !
2. Chaque thread possède son masque de signaux et son ensemble de signaux pendants.
3. Le masque d'un thread est hérité à sa création du masque de la thread le créant mais les signaux pendants ne sont pas hérités.

   int pthread_sigmask (int mode, sigset_t *pEns, sigset_t *pEnsAnc);

   Permet de consulter ou modifier le masque de signaux du thread appelant.

7.1 Envoi d'un signal à un thread

On utilise l'appel système ***pthread_kill() ;*** qui a le même comportement que l'appel système kill(). L'émission du signal au sein du même processus

int pthread_kill (pthread_t tid, int signal) ;

Renvoie 0 en cas de succès ou **ESRCH** si le thread de TID tid n'existe pas !

7.2 Attente de signal

On utilise l'appel système ***sigwait()*** ;

int sigwait(const sigset_t *ens, int *sig) ;

Extrait un signal de la liste de signaux pendants appartenant à **ens**.

Le signal est récupéré dans **sig** et renvoyé comme valeur de retour de la fonction.

7.3 Signal traité par une thread spécifique

C'est **synchrone** !

Évènement lié à l'exécution du thread actif.

Le signal est délivré au thread fautif. Par exemple SIGSEGV, SIGPIPE !

Signal envoyé par une autre thread en utilisant l'appel système ***pthread_kill() ; ! ! !***

7.4 Signal traité par une thread quelconque

C'est asynchrone ! et reçu par le processus. !

Le signal sera pris en compte par un des threads du processus parmi ceux qui ne masquent pas le signal en question.

Soit le code suivant permettant de détourner les signaux, du moins tous sauf **SIGKILL** et **SIGSEGV** ! ! !

7.5 Remarque et Solution

Les fonctions POSIX qui permettent de manipuler les Pthreads ne sont pas nécessairement **réentrantes**.

Par conséquent elles ne doivent pas être appelées depuis un gestionnaire de signaux !

Il est nécessaire de créer un thread dédié à la réception des signaux, qui boucle indéfiniment en utilisant l'appel système ***sigwait().***

int sigwait (const sigset_t * masque, int * num_sig) ;

Attente de l'un des signaux contenus dans le champ masque . !

Si un signal arrive, la fonction se termine en sauvegardant le numéro du signal reçu dans ***num_sig.**

Point d'annulation

Possibilité d'utiliser les fonctions de la bibliothèque Pthreads.

Tous les autres threads doivent bloquer les signaux attendus.

7.6 Exemple 1

```
//http://mff.devnull.cz/pvu/src/pthreads/sigwait.c
// fichier signal-CTRL-C.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#define CYCLES  100
sigset_t sset;
void * thread(void *x) {
        int i;
        for (i = 0; i < CYCLES; ++i) {
                printf("thread %d (loop #%d)\n", *((int *) x), i);
                sleep(3);
```

```
        }
        return (NULL);
}
int main(void) {
        pthread_t t1, t2;
        int sig, n1 = 1, n2 = 2;

        sigfillset(&sset);
        pthread_sigmask(SIG_SETMASK, &sset, NULL);

        pthread_create(&t1, NULL, thread, &n1);
        pthread_create(&t2, NULL, thread, &n2);
        system("clear");
        printf("Vous␣pouvez␣m'envoyer␣^C.␣\t\tou␣'kill␣-9␣%d'␣\t\t" \
            "d'un␣autre␣terminal␣pour␣m'atteindre␣!!!␣\n\n\n", getpid());
        while (1) {
                sigwait(&sset, &sig);
                printf("--->␣Reçu␣le␣signal␣N°␣%d\n", sig);
        }
        return (0);
}
```

Le résultat est :

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# nano
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# signal-CTRL-C
thread 2 (loop #0)
thread 1 (loop #0)
Vous pouvez m'envoyer ^C.               ou 'kill -9 14828'              d'un autre terminal pour m'

---> Reçu le signal N° 17      SIGCHLD !!!
thread 2 (loop #1)
thread 1 (loop #1)
thread 2 (loop #2)
thread 1 (loop #2)
^C---> Reçu le signal N° 2     Ctrl-c
thread 2 (loop #3)
thread 1 (loop #3)
thread 2 (loop #4)
thread 1 (loop #4)
^C---> Reçu le signal N° 2
thread 2 (loop #5)
thread 1 (loop #5)
thread 2 (loop #6)
thread 1 (loop #6)
Processus arrêté
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# kill -L
```

7.7 SIGFPE ou autre signal

Imaginons le cas suivant, un thread qui déclenche un signal par exemple de type FPE que se passe-t-il ?

```
//signal-fpe.c

void* jedivisepar0(void *arg){
        int a,b,c;
        a=0; b=10;
        sleep(2); // dort 2s pour que l'autre thread puisse W un peu
        c=b/a;     // La division par 0
        pthread_exit(NULL); // Fermeture du thread
}
void* jaffiche(void *arg){
        while(1){
                printf("jaffiche␣\n");
```

```
                usleep(250000); // dort 0,25s
        }
        pthread_exit(NULL); // Fermeture du thread
}
int main(int argc, char *argv[]){
        int i;
        pthread_t thread_id1, thread_id2;
        pthread_create(&thread_id1, NULL, &jaffiche, NULL);
        pthread_create(&thread_id2, NULL, &jedivisepar0, NULL);

        //sleep(1);

        for(i=0; i<2; i++)
                printf("Thread P j'ai que 2 fois à écrire ouf \n");

        // Thread P attend autres threads
        pthread_join(thread_id1, NULL);
        pthread_join(thread_id2, NULL);
        return 0;
}
```

Le résultat est : ... catastrophique ...☹Les 3 threads vont être fermés !

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# gccp signal-fpe.c -o signal-fpe
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# signal-fpe
Thread P j'ai que 2 fois à écrire ouf
Thread P j'ai que 2 fois à écrire ouf
jaffiche
  PID  SPID TTY          TIME CMD
 1260  1260 pts/0    00:00:01 bash
14866 14866 pts/0    00:00:00 signal-fpe
14866 14867 pts/0    00:00:00 signal-fpe
14866 14868 pts/0    00:00:00 signal-fpe
14869 14869 pts/0    00:00:00 sh
14870 14870 pts/0    00:00:00 ps
jaffiche
jaffiche
jaffiche
jaffiche
jaffiche
jaffiche
jaffiche
Exception en point flottant
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread# ps -T
  PID  SPID TTY          TIME CMD
 1260  1260 pts/0    00:00:01 bash
14871 14871 pts/0    00:00:00 ps
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/thread#
```

PS -T montre 3 threads

3 threads terminés pour 1 thread qui a fait une division par 0

Alors que faire ?

8 Protéger les données partagées

Comme les données sont partagées entre les différents threads, et bien il faut les protéger ! ! !

8.1 Quelques définitions

— **réentrante** : Une fonction est réentrante, lorsqu'elle peut être interrompue, une autre invocation est faite dans l'intervalle, puis reprise, et les 2 invocations fournissent un résultat correct.
  — Pas d'utilisation de variable globale ou statique
  — Pas de retour de pointeur statique
  — Pas d'appel de fonction non-reentrantes
  Ex. : *malloc*, *free* ou les fonctions d'entrée/sortie de la ***libc*** ne le sont pas !

— **Exclusion mutuelle** : une méthode qui permet d'assurer que 2 processus ou threads distincts n'utilisent pas simultanément une ressource partagée
Exemple : 2 processus souhaitent placer des fichiers dans une file d'impression. Il est souhaitable que les 2 n'agissent pas simultanément sur les éléments de cette file
— **Section critique** : portion de code où un programme souhaite un accès exclusif à une ressource donnée.
D'un point de vue principe, On veut réunir les 4 conditions suivantes :

1. 2 processus ou thread ne doivent pas se retrouver simultanément en section critique
2. Pas de supposition sur le nombre ou la vitesse des processus ou thread
3. Aucun processus ou thread hors de sa section critique ne doit bloquer d'autre processus ou thread
4. Aucun processus ne doit attendre indéfiniment pour pouvoir entrer en section critique

— **Producteur - Consommateur** : Il s'agit de coordonner 2 ou plus processus/threads qui échangent des objets au travers d'un tampon de taille fixée
  — Problématique :
    — Interruption du processus ou thread consommateur alors qu'il vient de tester le vide de la file mais n'est pas encore passé en ***sleep***
    — Le producteur lance un ***wakeup*** qui n'est pas reçu
    — Le Consommateur s'endort ensuite, n'est plus jamais réveillé
    — Ne réveille jamais le producteur lorsque celui-ci s'endort à son tour
  — Solution :
    — Le problème est que les messages envoyés en avance ne sont jamais reçus...
    — d'où Les **mutex** ou exclusion mutuelle
— Les **mutex** : ou sémaphore binaire
  — des verrous utilisateurs définis à l'aide de primitives matérielles
  — Ils réalisent des **opérations atomiques.**
  — Ils reposent sur l'attente active
  — Ils permettent de résoudre la problématique producteur/consommateur
— Les **Sémaphores** (1965 : E.W. Dijkstra, Néerlandais)
  — Elles offrent une généralisation des **mutex**
  — Autorisant plusieurs libérations de la ressource
  — **P** : verrouille (***sleep***, ***wait***) ***P****roberen* (tester)
  — **V** : déverrouille (***wakeup***) ***V****erhogen* (incrémenter)
  Sous Linux : ***man sem_overview*** explique TB leur utilisation

8.2 Problématique

Tous les ***threads*** d'un programme partagent la même mémoire, ils peuvent ainsi accéder à toutes les variables du programme.

C'est très pratique mais aussi très dangereux, car tous les threads tournent en parallèle, cela signifie qu'une variable ou une fonction pourrait très bien être utilisée depuis plusieurs threads en même temps.

Et si l'opération en question n'est pas ***thread-safe***, le résultat est indéterminé et peut "planter" ou corrompre des données partagées.

Il existe plusieurs outils de programmation pour aider à protéger les données partagées et rendre le code ***thread-safe***, ce sont les "***primitives de synchronisation***".

Les plus communs sont les ***mutexs***, les **sémaphores**, les **conditions d'attente** et les ***spin locks***.

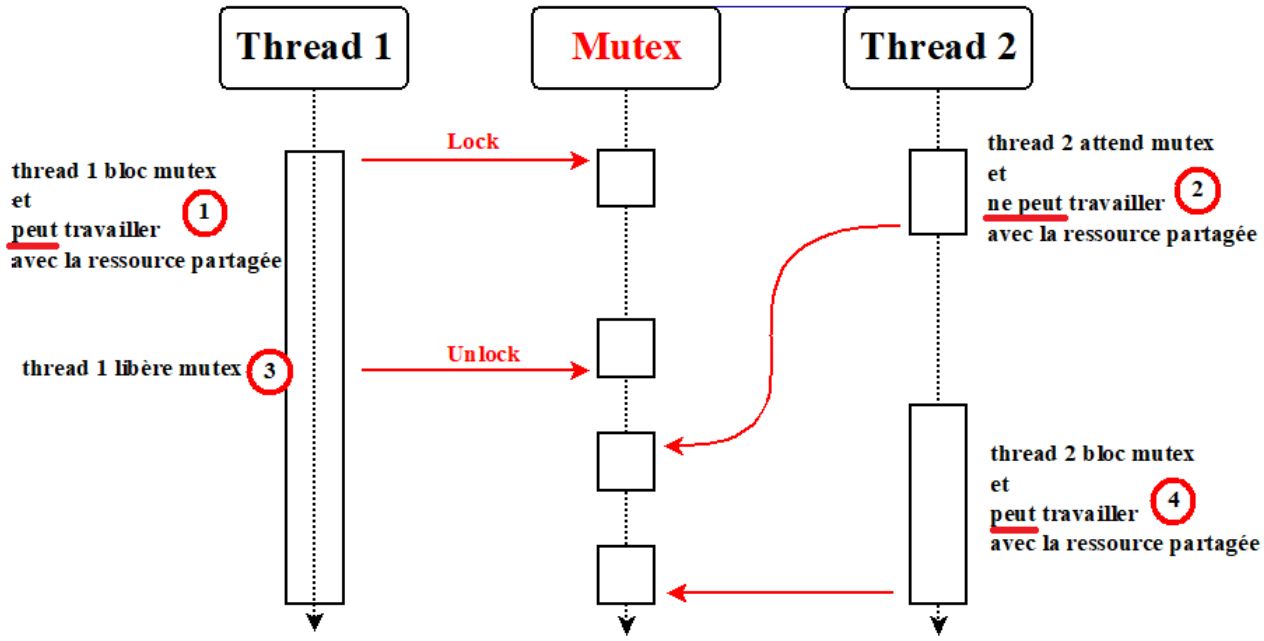
Ils sont tous des variations du même concept : **ils protègent un morceau de code en autorisant son accès uniquement à certains (1 à la fois) threads, tout en bloquant les autres.**

La primitive de synchronisation la plus basique donc la plus utilisée est le ***mutex*** pour "***EXclusion MUTuelle***" :

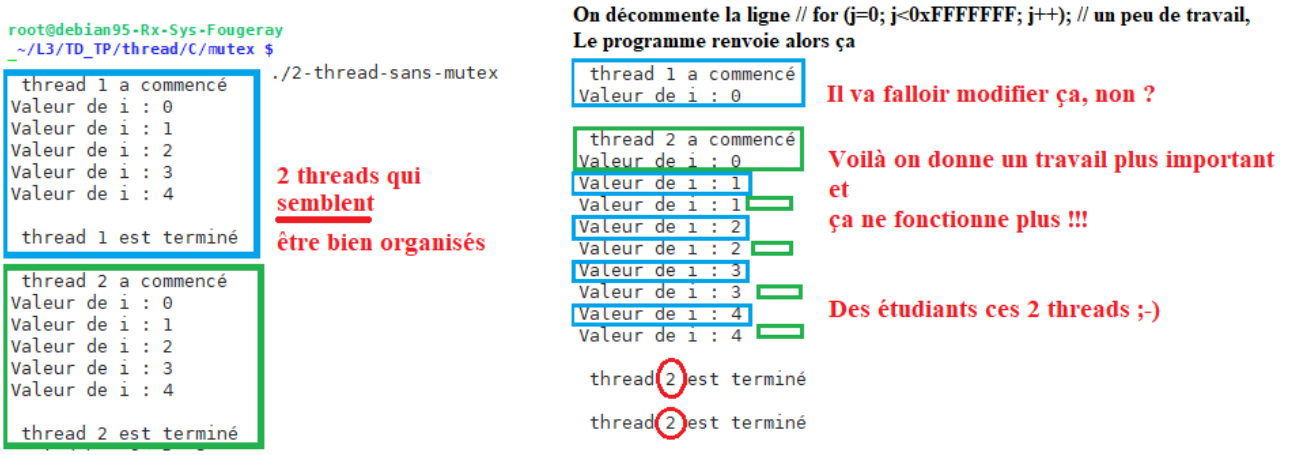
Le ***mutex*** autorise un seul ***thread*** à la fois à accéder aux "bouts" de code qu'il encadre.

8.3 Un exemple simple

Prenons l'exemple suivant où 2 ***threads*** créés ont la même tâche à réaliser, la fonction ***travail***, mais selon que ce travail soit "long" ou pas, le résultat n'est pas le même comme le montre les 2 figures suivantes




```
//

pthread_t tid[2];
int compteur;
// pthread_mutex_t lock; // Pour les MUTEX
void* travail(void *arg) {
        // pthread_mutex_lock(&lock);
        int i = 0, j;
        compteur += 1;
        printf("\n thread %d a commencé\n", compteur);
        for(i; i<5;i++){
            printf("Valeur de i : %d \n", i);
                // for (j=0; j<0xFFFFFFFF; j++); // un peu de travail
        }
        printf("\n thread %d est terminé\n", compteur);
        // pthread_mutex_unlock(&lock); // Pour les MUTEX
        return NULL;
}
int main(void) {
        int i = 0;
        int erreur;
        if (pthread_mutex_init(&lock, NULL) != 0) {
                printf("\n mutex init a raté\n");
                return 1;
        }
        while(i < 2) {
                erreur = pthread_create(&(tid[i]), NULL, &travail, NULL);
                if (erreur != 0)
                        printf("\nThread ne peut être créé : [%s]", strerror(erreur));
                i++;
        }
        pthread_join(tid[0], NULL); pthread_join(tid[1], NULL);
        // pthread_mutex_destroy(&lock); // Pour les MUTEX
        return 0;
}
```



On constate que **Thread 2 est terminé** 2 fois alors que cela n'est pas possible.

Et pourtant si l'ordinateur l'affiche c'est que c'est vrai ?

Réalité : c'est la variable partagée i qui va valoir 2 même pour le **Thread 1.**

Le thread 1 est obligatoirement lancé en premier, mais il n'a pas le temps de faire tout son travail.

L'ordonnanceur l'arrête avant et donne la main entre autres à **Thread 2** qui incrémente la variable **partagée** i qui passe donc à 2, si si 1+1=2 ☺

Une Solution parmi d'autres :

Faire en sorte que Thread 2 ne puisse pas commencer tant que Thread 1 n'a pas terminé, pour cela on utilise les Mutex.

Si on enlève les commentaires des 4 lignes, le programme fonctionne et **c'est le noyau qui gère la synchronisation ! ! !**

- // ***pthread_mutex_t lock ;***
- // ***pthread_mutex_lock(&lock) ;***
- // ***pthread_mutex_unlock(&lock) ;***
- // ***pthread_mutex_destroy(&lock) ;***

Cela semble merveilleux, mais ce n'est pas top d'un point de vue vitesse d'exécution.

En effet la seconde solution qui semble la meilleure est la plus lente, même 2 fois plus lente (10s vs 5,5s ici) pour preuve, un ***time*** sur les 2 processus renvoie

```
time ./2-thread-sans-mutex
 thread 1 a commencé
Valeur de i : 0

 thread 2 a commencé
Valeur de i : 0
Valeur de i : 1
Valeur de i : 1
Valeur de i : 2
Valeur de i : 2
Valeur de i : 3
Valeur de i : 3
Valeur de i : 4
Valeur de i : 4

 thread 2 est terminé

 thread 2 est terminé

real    0m5,586s
user    0m5,564s
sys     0m0,000s
```

```
time ./2-thread-avec-mutex
 thread 1 a commencé
Valeur de i : 0
Valeur de i : 1
Valeur de i : 2
Valeur de i : 3
Valeur de i : 4

 thread 1 est terminé

 thread 2 a commencé
Valeur de i : 0
Valeur de i : 1
Valeur de i : 2
Valeur de i : 3
Valeur de i : 4

 thread 2 est terminé

real    0m10,007s
user    0m0,004s
sys     0m0,000s
```

Mais bon le but dans cet exemple n'est pas la vitesse mais voir l'intérêt des **mutex**.

8.4 Définition des primitives mutex

Comme nous venons de le voir dans l'exemple précédent, pour utiliser le mécanisme des mutex, nous avons utilisés certaines primitives dont voici les définitions.

8.5 Les sémaphores

Les mutex c'est bien pour une exclusion mutuelle, c'est à dire soit l'un soit l'autre mais ne fonctionnent pas si plus de 2 threads.

Pour cela on peut utiliser les sémaphores.

Il existe 2 types de sémaphores :

1. **Sémaphores nommés** : Un sémaphore nommé est identifié par un nom donc une chaine terminée par un caractère NULL.

   2 processus peuvent utiliser un même sémaphore nommé en passant le même nom à ***sem_open().***

   La fonction ***sem_open*** crée un nouveau sémaphore nommé ou en ouvre un existant.

   Après l'ouverture de ce sémaphore, il peut être utilisé avec ***sem_post()*** et ***sem_wait()***. Lorsqu'un processus ou thread a fini d'utiliser le sémaphore, il peut utiliser ***sem_close()*** pour le fermer.

   Lorsque tous les processus ou thread ont terminé de l'utiliser, il peut être supprimé du système avec ***sem_unlink()***.

2. **Sémaphores anonymes (**sémaphores en mémoire) : Un sémaphore anonyme n'a pas de nom. Il est placé dans une région de la mémoire qui est partagée entre plusieurs threads (sémaphore partagé par des threads) ou processus (sémaphore partagé par des processus).

   Un sémaphore partagé par des threads est placé dans une région de la mémoire partagée entre les threads d'un processus, par exemple **une variable globale**.

   Un sémaphore partagé par des processus doit être placé dans une région de mémoire partagée (par exemple un segment de mémoire partagée créé avec ***shmget***(), ou un objet de mémoire partagée POSIX créé avec ***shm_open***()).

9 En JAVA

Tout est possible en JAVA... je vous invite à lire le cours d'Alexis Lechevry sur ecampus !

Mais voici Juste une application Client-Serveur Multithreading qui permet que la connexion !

1. Le serveur : Il écoute sur le port 2009

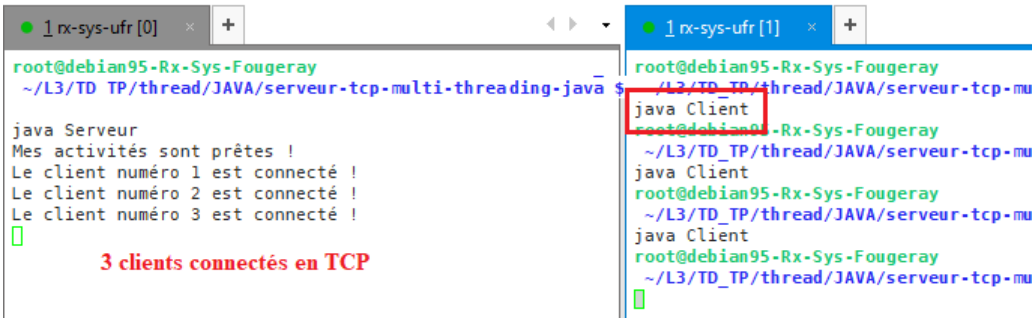
```
// Serveur.java dans thread/JAVA/serveur-tcp-multi-threading-java
import java.io.IOException;
import java.net.*;

public class Serveur {
        public static void main(String[] zero){
                ServerSocket socket;
                try {
                socket = new ServerSocket(2009);
                Thread t = new Thread(new Accepter_clients(socket));
                t.start(); // on lance le thread
                System.out.println("Mes activités sont prêtes !");
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
class Accepter_clients implements Runnable {
            private ServerSocket socketserver;
            private Socket socket;
            private int nbrclient = 1;
                public Accepter_clients(ServerSocket s){
                        socketserver = s;
```

```
                }
25              public void run() {
                try {
                        while(true){
                          socket = socketserver.accept(); // Un client se connecte on l'accepte
                          System.out.println("Le client numéro "+nbrclient+ " est connecté !");
30                        nbrclient++;
                          socket.close(); // On ferme la socket d'écoute du thread
                        }

                } catch (IOException e) {
35                              e.printStackTrace();
                        }
                }
    }
```

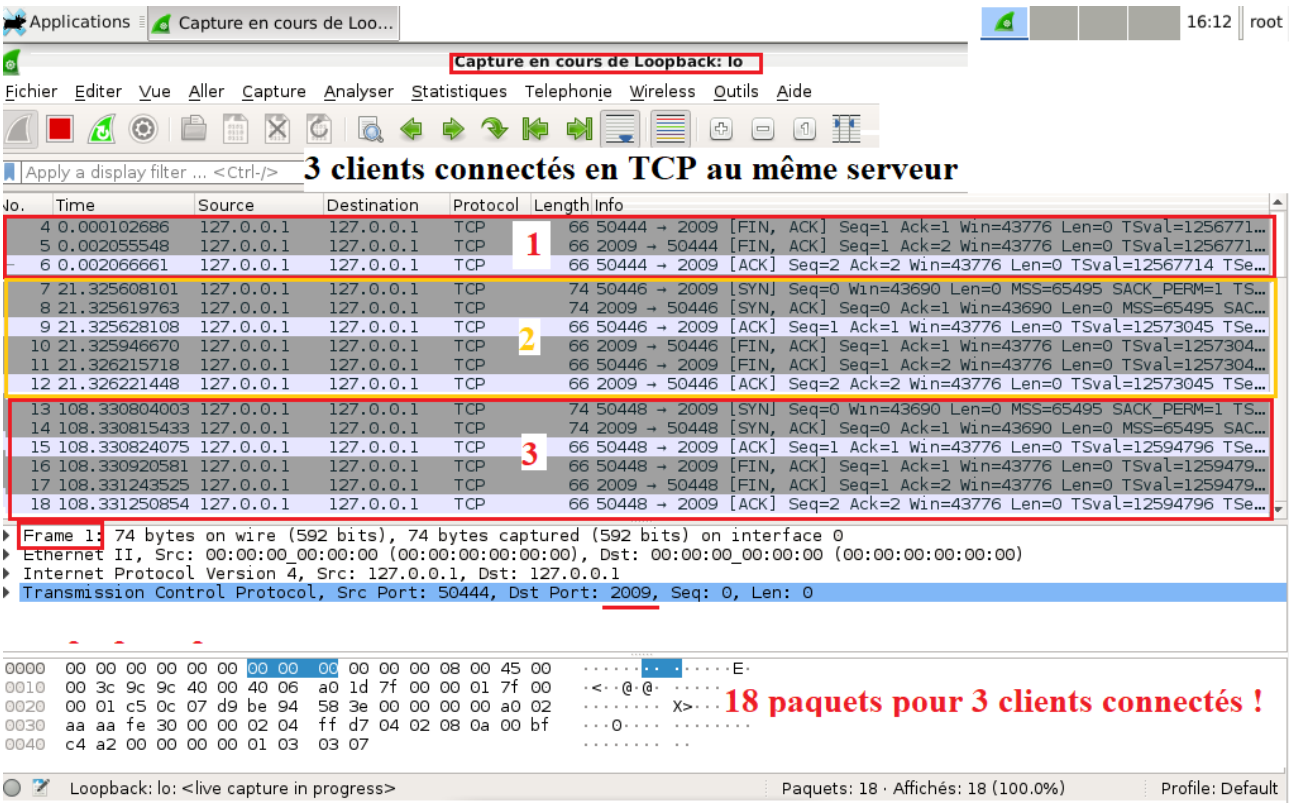
2. Le client

```
    // Client.java dans thread/JAVA/serveur-tcp-multi-threading-java
 2  import java.io.IOException;
    import java.net.*;

    public class Client {
            public static void main(String[] zero){
 7                  Socket socket;
                    try {
                    socket = new Socket("localhost",2009);
                    socket.close();
                    } catch (IOException e) {
12                          e.printStackTrace();
                    }
            }
    }
```

3. Lancement et résultat : rien de plus simple ☺



4. un peu de wireshark



10 En Python

Pas plus compliqué qu'en C ou en JAVA même au contraire je pense, les principes sont les mêmes !
Si vous désirez en connaitre davantage, je vous invite à lire
https://docs.python.org/fr/3/library/threading.html
Un exemple de code en 3 parties sans avoir à expliquer à nouveau les concepts !

```
# Mot-par-Mot.py
import random
import os
import sys
from threading import Thread
import time
class Afficheur(Thread):
    """Thread chargé simplement d'afficher une lettre dans la console."""
    def __init__(self, mot):
        Thread.__init__(self)
        self.mot = mot
    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 8:
            sys.stdout.write(self.mot)
            sys.stdout.flush()
            attente = 0.2
            attente += random.randint(1, 60) / 100
            time.sleep(attente)
            i += 1
# Création des threads
thread_1 = Afficheur(" Etudiant ")
thread_2 = Afficheur(" Enseignant ")
# Lancement des threads
thread_1.start()
```

```
thread_2.start()
# Attend que les threads se terminent
thread_1.join()
thread_2.join()
# Un retour à la ligne
print("\n")

# Lettre_par_Lettre.py
import os
import random
import sys
from threading import Thread
import time
class Afficheur(Thread):
    """Thread chargé simplement d'afficher un mot dans la console."""
    def __init__(self, mot):
        Thread.__init__(self)
        self.mot = mot
    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 5:
            for lettre in self.mot:
                sys.stdout.write(lettre)
                sys.stdout.flush()
                attente = 0.2
                attente += random.randint(1, 60) / 100
                time.sleep(attente)
            i += 1
# Création des threads
thread_1 = Afficheur(" Etudiant ")
thread_2 = Afficheur(" Enseignant ")
# Lancement des threads
thread_1.start()
thread_2.start()
# Attend que les threads se terminent
thread_1.join()
thread_2.join()
# Un retour à la ligne
print("\n")

# Lettre_par_Lettre-synchro.py
import os
import random
import sys
from threading import Thread, RLock
import time
verrou = RLock()
class Afficheur(Thread):
    """Thread chargé simplement d'afficher une lettre dans la console."""
    def __init__(self, mot):
        Thread.__init__(self)
        self.mot=mot
    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 8:
            with verrou:
```

```
                for lettre in self.mot:
                    sys.stdout.write(lettre)
                    sys.stdout.flush()
                    attente = 0.2
22                  attente += random.randint(1, 60) / 100
                    time.sleep(attente)
            i += 1
# Création des threads
thread_1 = Afficheur(" Etudiant ")
27 thread_2 = Afficheur(" Enseignant ")
# Lancement des threads
thread_1.start()
thread_2.start()
# Attend que les threads se terminent
32 thread_1.join()
thread_2.join()
```

Le résultat du lancement de ces 3 programmes :

```
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/Python $
python3 Mot-par-Mot.py
 Etudiant  Enseignant  Enseignant  Etudiant  Enseignant  Etudiant  Enseignant  Etudiant
 Enseignant  Etudiant  Etudiant  Enseignant  Enseignant  Etudiant  Etudiant  Enseignant

root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/Python $
python3 Lettre_par_Lettre.py
  EEntsuediaignnt a nEtt  uEdiansnetig n aEntutd  iEannste ig nEatntu d iaEnnste i Egtnu
adinatn t E nseignant

root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/Python $
python3 Lettre_par_Lettre-synchro.py
 Etudiant  Etudiant  Etudiant  Etudiant  Etudiant  Etudiant  Etudiant  Etudiant  Enseign
ant  Enseignant  Enseignant  Enseignant  Enseignant  Enseignant  Enseignant  Enseignant
root@debian95-Rx-Sys-Fougeray
 ~/L3/TD_TP/thread/Python $
```

Mot par Mot

Lettre par Lettre sans synchronisation

Lettre par Lettre avec synchronisation

11 Comparaison

Juste un peu de réflexion !

— Que veut-on ?
— Un programme rapide ?
— Que faire ?
— 1 seul processus ?
— Plusieurs threads ?
— Plusieurs Processus ?

Voyons le programme suivant écrit en python et récupéré ici

https://medium.com/practo-engineering/threading-vs-multiprocessing-in-python-7b57f224eadb

Attention, si vous désirez le lancer avec Python3, il faut remplacer **xrange** par **range ! ! !**

Ce que j'ai fait ! ! !

Si vous lancez cette nouvelle version (donc avec **range** et non **xrange**) vous allez multiplier les temps par "beaucoup" ! ! !

```
import random
2 from threading import Thread
from multiprocessing import Process
size = 10000000   # Number of random numbers to add to list
threads = 2 # Number of threads to create, 2 puis 3 puis 4
```

```
my_list = []
for i in range(0,threads):
    my_list.append([])
def func(count, mylist):
    for i in range(count):
        mylist.append(random.random())
def multithreaded():
    jobs = []
    for i in range(0, threads):
        thread = Thread(target=func,args=(size,my_list[i]))
        jobs.append(thread)
    # Start the threads
    for j in jobs:
        j.start()
    # Ensure all of the threads have finished
    for j in jobs:
        j.join()
def simple():
    for i in range(0, threads):
        func(size,my_list[i])
def multiprocessed():
    processes = []
    for i in range(0, threads):
        p = Process(target=func,args=(size,my_list[i]))
        processes.append(p)
    # Start the processes
    for p in processes:
        p.start()
    # Ensure all processes have finished execution
    for p in processes:
        p.join()
if __name__ == "__main__":
        multithreaded() //
        #simple()
        #multiprocessed()
```

Résultats on en conclut quoi ?

```
root@debian95-Rx-Sys-Fougeray
~/L3/TD_TP/thread/le_GIL_python/comparaison-thread-multiprocessing $
```

Avec 4 coeurs !!!

2 Nb Threads !	3	4
time python3 comparaison.py	time python3 comparaison.py	time python3 comparaison.py
real 0m10,087s	real 0m14,737s	real 0m19,764s
user 0m9,780s	user 0m14,396s	user 0m19,192s
sys 0m0,320s	sys 0m0,388s	sys 0m0,668s
root@debian95-Rx-Sys-Fougeray	root@debian95-Rx-Sys-Fougeray	root@debian95-Rx-Sys-Fougeray
time python3 comparaison.py	time python3 comparaison.py	time python3 comparaison.py
real 0m9,454s	real 0m14,702s	real 0m19,643s
user 0m9,200s	user 0m14,380s	user 0m19,100s
sys 0m0,244s	sys 0m0,312s	sys 0m0,540s
root@debian95-Rx-Sys-Fougeray	root@debian95-Rx-Sys-Fougeray	root@debian95-Rx-Sys-Fougeray
time python3 comparaison.py	time python3 comparaison.py	time python3 comparaison.py
real 0m4,264s	real 0m4,245s	real 0m4,324s
user 0m8,136s	user 0m12,068s	user 0m16,440s
sys 0m0,264s	sys 0m0,428s	sys 0m0,528s

multithreaded()

simple()

multiprocessed()

Conclusion fausse ! (celle que l'on vient de faire ! ! !) pas celle qui suit ! ! ! !

Voir le souci du **GIL** en Python

Le ***Global Interpreter Lock*** est un verrou qui évite à plusieurs ***threads*** de **modifier le même objet en même temps**.

Dans les langages bas niveau, on fait la disrinction entre un tableau ou une liste qui supporte les accès concurrentiels ou non. Si elle ne les supporte pas, les accès sont plus rapides mais suppose que le développeur s'occupe de gérer les problèmes de synchronisation si besoin (mutex !).

Le langage Python protège listes et dictionnaires par l'intermédiaire de ce verrou qui est unique pour toutes les listes afin de pouvoir gérer efficacement le garbage collector (voir module gc).

En conséquence, si le langage Python est ***multithread*** par design, dans les faits, il ne l'est presque pas car le **GIL** est sans cesse utilisé.

Une excellente page pour apprendre Python :
http ://www.xavierdupre.fr/app/teachpyx/helpsphinx/c_parallelisation/thread.html
d'où j'ai tiré cette conclusion après l'avoir constatée ! ! !

12 Conclusion

Qu'est-ce qui est le mieux, les Processus ou les Thread ?
Qu'est-ce qui est le mieux Langage C, JAVA ou Python ?