centralelille

Département
de
Mathématiques et Informatique

Système et réseau : module n° 2
Séminaire n° 4
Threads

Samir El Khattabi *et* Christian Vercauter

Version éditée le 2 avril 2020

Année 2019-2020

Table des matières

Introduction

Ce document est un recueil d'exercices complété de rappels de cours, qui a été réalisé à partir de polycopiés de cours et d'exercices créés par Étienne Craye, Samir El Khattabi et Christian Vercauter pour plusieurs enseignements effectués à EC-Lille et à IG2I Lens.

Il concerne le concept de thread, aussi appelé activité ou processus léger (lwp, pour *light-weight process*) comme unité d'exécution au cœur d'un système d'exploitation multitâche.

Les informations présentées ici, sont conformes aux standards `POSIX` de l'IEEE établis à partir de la fin des années 1980, complétés et périodiquement révisés depuis ; la dernière révision date de 2017 (IEEE Std 1003.1™-2017)

Les exemples pris et les exercices proposés ont été testés sur un système `Linux` installé sur un ordinateur d'architecture `x86-64` bits.

Ils font appel à :

- des compétences SHELL acquises dans le module 1 de cet électif ;
- des connaissances de base sur la programmation en langage C, qui seront enrichies lors de cette séance, par l'usage important de pointeurs de toutes sortes ;
- des compétences sur les outils de développement associés.

Le compilateur `Gcc`

L'éditeur de textes `Emacs`

L'éditeur de textes `vim`

Dans la version avec réponses aux questions, de ce document, les solutions présentées ont été obtenues à partir de plusieurs ordinateurs dans différentes configurations :

`Ubuntu 18.04 LTS` en mode virtuel avec `VirtualBox`

`Ubuntu 18.04 LTS` en mode natif

`Debian 8.10` en mode natif

Chapitre I

Présentation générale

1. Définition et caractéristiques

Un thread est une unité d'exécution au sein d'un processus : le conteneur est le processus et tous ses threads partagent le même espace d'adressage.

Chaque thread a sa propre identité, son propre état, un niveau de priorité, une politique d'ordonnancement et dispose de sa propre pile où sont créées ses propres variables locales.

Sous LINUX, l'ordonnancement est basé sur la notion de thread et non pas sur celle de processus ; un processus est le thread principal capable de créer plusieurs threads enfants.

Le standard IEEE Std 1003.1c-1995 (POSIX.1c, Threads extensions) spécifie une interface de programmation permettant la gestion de threads : création et la destruction, ordonnancement, synchronisation, gestion des signaux et des données partagées. Un thread est aussi appelé *processus léger*, car la plupart de ces opérations sont moins coûteuses en ressources (mémoire, CPU, …) que les opérations équivalentes sur processus.

2. Threads vs processus

De façon à comparer l'efficacité des processus et de threads, nous allons effectuer 50 000 exécutions :

- d'un processus qui ne fait rien d'utile ;
- d'un thread qui ne fait rien d'utile

Le processus comme le thread ne fait qu'initialiser une variable entière locale.

L'archive compressée (`Fork_vs_thread.tar.gz` est disponible sur le site Moodle de cet enseignement. Elle contient le code source des deux programmes ainsi qu'un `makefile` permettant de générer les exécutables et d'effectuer le test comparatif.

Question 1.

Après avoir décompressé cette archive et s'être placé dans le répertoire obtenu, exécuter la commande `make`.

Les tests effectués font appel à la commande `time` dont les résultats sont repris et affichés un tableau.

1. Que fait cette commande `time` ?

   Que représentent les résultats qu'elle produit ?

2. Comparer les résultats du test sur les processus et ceux sur les threads.

---

3. Threads Posix

La bibliothèque des threads Posix (*pthread*) utilisée ici est conforme à la révision de la norme POSIX 1003.1-2001.

Les fonctions de cette librairie offrent des services similaires à ceux fournis par diverses autres bibliothèques (Solaris LWP, Windows) ou langages (Modula-3, Java).

Une présentation générale de cette bibliothèque est accessible à l'aide de la commande `man 7 pthreads`

L'utilisation dans un programme `C` nécessite l'inclusion du fichier `pthread.h` contenant les définitions de types, de constantes et des prototypes de fonction.

3.1 Génération d'un exécutable *multithreadé*

La génération d'un programme exécutable utilisant des threads Posix nécessite la spécification de la librairie des primitives.

Il s'agit de :

- la librairie dynamique `libpthread.so` ;
- ou de la librairie statique `libpthread.a`

Ces deux bibliothèques de fonctions sont présentes dans le répertoire `/usr/lib/x86_64-linux-gnu/` sur un système LINUX sur base `x86-64`.

On utilise l'option `-lpthread`, pour générer un exécutable *multithreadé* comme dans l'exemple suivant :

```
gcc thread1.c -Wall -o thread1 -lpthread
```

3.2 Code d'erreur

La majorité des routines de la bibliothèque pthread retourne un résultat entier : la valeur 0 signale que l'opération demandée, a été correctement réalisée ; toute autre valeur, en général une valeur > 0 indique une erreur, comme par exemples :

`EAGAIN :` ressources insuffisantes ;

`EINVAL :` paramètre non valide ;

`EBUSY :` demande de destruction d'un objet utilisé ou test de verrouillage ;

`EPERM :` permission insuffisante …

La variable `errno` utilisée par les appels système UNIX n'est pas affectée par l'exécution des opérations sur threads Posix.

a) Macro `CHECK_T()`

Il est vivement conseillé de contrôler le résultat des fonctions sur les threads Posix.

La macro présentée ci-dessous, sera régulièrement utilisée dans les exemples et exercices de ce document.

```
#define CHECK_T(status, msg)                          \
  if (0 != (status))   {                              \
    fprintf(stderr, "pthread erreur : %s\n", msg);    \
    exit (EXIT_FAILURE);                              \
  }
```

3.3 Attributs

Chaque thread possède un ensemble d'attributs regroupé dans le type `pthread_attr_t`

Cet ensemble est notamment utilisé lors de la création d'un thread. En pratique, on utilisera dans la majorité des cas, les valeurs par défaut de ces attributs.

Il est également possible de changer dynamiquement leur valeur.

Voici quelques-uns de ces attributs et des valeurs pouvant leur être affectées

```
Detach state        = PTHREAD_CREATE_JOINABLE
Scope               = PTHREAD_SCOPE_SYSTEM
Inherit scheduler   = PTHREAD_INHERIT_SCHED
Scheduling policy   = SCHED_OTHER
Scheduling priority = 0
Guard size          = 4096 bytes
Stack address       = 0x40196000
Stack size          = 0x201000 bytes

Detach state        = PTHREAD_CREATE_DETACHED
Scope               = PTHREAD_SCOPE_SYSTEM
Inherit scheduler   = PTHREAD_EXPLICIT_SCHED
Scheduling policy   = SCHED_OTHER
Scheduling priority = 0
Guard size          = 0 bytes
Stack address       = 0x40197000
Stack size          = 0x3000000 bytes
```

Chapitre II

Opérations de base

1. Création

```
int pthread_create (pthread_t *thr, const pthread_attr_t *attr,
          void * (*start_routine)(void *), void *arg);
```

Cette fonction crée une nouvelle activité pour exécuter la fonction `start_routine`, appelée avec l'argument `arg`.

Un seul paramètre pour cette fonction, est-ce contraignant ?

Pas vraiment, l'argument transmis est une adresse, et elle peut être celle d'un tableau de plusieurs composantes ou celle d'une structure regroupant plusieurs champs de nature distincte.

L'argument `attr` pointe sur une structure `pthread_attr_t` dont le contenu est utilisé pendant la création pour déterminer les attributs du nouveau thread. Cette structure est initialisée avec `pthread_attr_init` et les fonctions similaires. Si `attr` vaut `NULL`, alors le thread est créé avec les attributs par défaut.

Un appel réussi à `pthread_create()` stocke l'identifiant du nouveau thread à l'adresse présente dans l'argument `thr`. Cet identifiant est utilisé pour se référer à ce thread dans les appels ultérieurs aux autres fonctions de pthreads.

2. Terminaison

```
void pthread_exit (void *status);
```

Cette fonction termine l'activité appelante en fournissant un code de retour, ou plus exactement l'adresse d'une variable de nature quelconque, contenant le code de terminaison.

Un thread se termine également dès que l'on exécute l'instruction **`return`** de sa fonction d'implémentation. Ce **`return`** est généralement suivi de l'adresse de la variable contenant le code de terminaison.

3. Attente de la terminaison

```
int pthread_join (pthread_t thread, void **status);
```

L'espace d'adressage — l'espace mémoire — qu'utilise un thread est celui du processus conteneur. Si ce dernier se termine alors tout l'espace mémoire qu'il occupait, est libéré, mettant ainsi fin à l'exécution

de tous les threads enfants qu'il a créés.

Si une telle opération est souvent présente dans le thread principal, il est également permis de l'utiliser dans n'importe quel autre thread pour synchroniser la poursuite de son traitement sur la terminaison d'un autre.

La fonction ci-dessus attend la terminaison de l'activité indiquée et permet de récupérer son code de retour. Elle a aussi pour effet de libérer les ressources utilisées par cette activité.

Remarque. L'opération `pthread_join()` ne doit pas être réalisée sur un thread détaché (voir 5).

La valeur `NULL` utilisée pour l'argument `status` indique qu'il n'y a pas de code de terminaison à récupérer.

S'il ne vaut pas `NULL`, alors `status` représente l'adresse d'une variable de type pointeur de n'importe quoi. À l'issue de l'exécution de cette fonction, cette variable contient l'adresse des cases de mémoire, où le code de terminaison du thread a été rangé.

Lors d'un appel de fonction, pour récupérer les modifications apportées la valeur d'un paramètre de type pointeur, c'est son adresse qu'il faut lui fournir et c'est ainsi qu'apparaissent des arguments de type pointeur de pointeur.

Exemple d'utilisation

Le thread principal crée un pointeur d'entier long nommé `status` et l'initialise à `NULL` et imaginons que cette variable soit rangée en mémoire à l'adresse `0x123456`.

```
long * status = NULL;
```

alors

- `status` → NULL
- `*status` → indéfini
- `&status` → 0x123456

Le thread enfant a rangé dans une variable [1] entière longue le résultat qu'il veut transmettre, lorsqu'il se termine, au thread principal

Soit `resultat`, cette variable rangée en mémoire à l'adresse `0x1abcdef`. Le thread enfant d'identité `tid` a rangé la valeur `25` dans cette variable, puis se termine en exécutant, `pthread_exit((void *)&resultat);` ou plus simplement `return (void *)&resultat;`

À cet instant, nous avons :

- `resultat` → `25`
- `&resultat` → `0x1abcdef`

Le thread principal exécute l'opération `pthread_join (tid, (void **)&status)` qui a pour effet de ranger à l'adresse de la variable `status` la valeur transmise en paramètre du `pthread_exit()`, ou la valeur retournée par un `return` en fin d'exécution du thread enfant.

1. nous verrons plus loin comment doit être créée cette variable

Nous avons ainsi :

- `&status` → `0x123456` — cela n'a pas changé —
- `status` → `0x1abcdef`
- `*status` → `25`, qui représente bien le code de terminaison du thread enfant.

Remarque importante

La variable `resultat` ne peut être une variable locale ordinaire de la fonction qui implémente le thread : l'espace mémoire qu'elle occupe, doit persister en fin d'exécution de cette fonction, or la durée de vie d'une variable locale se limite à la durée d'exécution de la fonction contenant sa déclaration.

La solution consistant à définir cette variable `resultat` avec l'attribut `static` peut parfois convenir : en effet, une variable locale statique est une variable rémanente, dont la durée de vie est celle du programme et non celle de la fonction contenant sa déclaration.

Pourquoi la solution d'une variable statique ne convient pas toujours ?

Parce que plusieurs threads peuvent partager la même fonction et dans ce cas, ils partageront aussi cette même variable.

Dans cette situation, examinons le scénario suivant :

1. le premier thread se termine en retournant l'adresse de `resultat` dont la valeur vaut à cet instant, 25 par exemple ;
2. le thread principal n'a pas encore récupéré cette adresse, quand un second thread incrémente la valeur de cette variable statique `resultat` ;
3. le thread principal, qui enfin effectue l'opération `pthread_join` sur le premier thread, récupère l'adresse puis accède au contenu, et lira 26 au lieu de la valeur 25 que souhaitait retourner le premier thread.

Une meilleure solution

Une meilleure solution consiste à déclarer la variable `resultat` comme une variable locale, mais au lieu d'en faire un entier long — *pour notre exemple* — :

1. on la déclare comme un pointeur d'entier long ;
2. on alloue dynamiquement l'espace mémoire pour un entier long et on range l'adresse de cette zone mémoire dans `resultat`
3. on accède et modifie la valeur rangée dans cet espace mémoire, en utilisant la notation `*resultat` ;
4. on retourne l'adresse de cet espace mémoire par l'instruction `return (void *) resultat;`

Ce sera à la charge du thread principal qui a récupéré l'adresse et a accédé à la valeur, de libérer l'espace mémoire dynamiquement créé à l'aide de la fonction `free()`.

4. Identification

```
pthread_t pthread_self (void);
```

Cette fonction renvoie l'identificateur de l'activité appelante.

```
int pthread_equal (pthread_t thread_1, pthread_t thread_2);
```

Elle permet de vérifier si les deux identifiants fournis en paramètres, désignent la même activité.

5. Libération des ressources

```
int pthread_detach (pthread_t thread);
```

Cette fonction détache l'activité `thread`.

Normalement, les ressources allouées pour l'exécution d'une activité (espace mémoire réservé pour la pile d'exécution du thread, notamment …) ne sont libérées que lorsque l'activité s'est terminée et qu'un appel à `pthread_join()` pour cette activité a été effectué.

Pour éviter de devoir se synchroniser sur la terminaison d'une activité dont on compte ignorer le code retour, on peut détacher cette activité, et dans ce cas, les ressources sont automatiquement libérées dès la terminaison de l'activité.

Il est interdit d'attendre la terminaison (`join`) d'une activité détachée.

Chapitre III

Variations sur un exemple simple

1. Première version

1.1 Objectif

Le code ci-dessous permet au thread principal implémenté par la fonction `main()`, de créer deux threads enfants qui partagent le même code, celui de la fonction `monThread()`.

Le code de cette fonction se contente d'afficher un message au début de son exécution, puis s'endort quelques secondes en appelant la fonction `sleep()` et affiche un second message avant de se terminer. La durée de l'endormissement est tirée au sort dans l'intervalle $[1, \text{DUREE\_MAX}]$

Le thread parent attend ensuite la terminaison de ses threads enfants avant de se terminer lui même.

Chaque thread enfant dispose de sa propre pile et donc de ses propres variables locales. Il ne peut accéder aux variables locales d'un autre thread, y compris le thread principal, mais peut accéder aux variables globales du processus conteneur.

1.2 Code source

L'utilisation des types, constantes et fonctions relatifs aux threads POSIX nécessite d'inclure le fichier `pthread.h` en tête de programme.

Question 2.

Quelle commande permet de rechercher puis d'afficher le contenu de ce fichier ?

Le code de ce programme, également disponible sur la page Moodle de cet enseignement, est le suivant :

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

#include <pthread.h>

#define DUREE_MAX 5

#define CHECK_T(status, msg)                          \
  if (0 != (status))   {                              \
    fprintf(stderr, "pthread erreur : %s\n", msg);    \
    exit (EXIT_FAILURE);                              \
  }

void  *monThread (void * arg);
void  bye(void);

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

```c
    pthread_t tid1, tid2;  /* Identifiants de thread    */

    atexit(bye);
    printf("Le processus %d va créer deux threads\n",
           getpid());

    CHECK_T(pthread_create (&tid1, NULL, monThread, NULL),
            "pthread_create(1)");

    CHECK_T(pthread_create (&tid2, NULL, monThread, NULL),
            "pthread_create(2)");

    printf("\tatttente de la terminaison des threads\n");

    CHECK_T(pthread_join (tid1, NULL),"pthread_join(1)");
    CHECK_T(pthread_join (tid2, NULL),"pthread_join(2)");
    return EXIT_SUCCESS;
}

void  *monThread (void * arg)
{
    printf("Thread fils : \tDébut de traitement\n");
    sleep (rand() % DUREE_MAX + 1);
    printf("Thread fils : \tFin de traitement\n");
    return NULL;
}

void bye (void)
{
    printf("Fin du processus %d\n", getpid());
}
```

Listing III.1 – Le programme `thread1.c`

Dans cet exemple, la macro `CHECK_T()` est utilisée pour contrôler la validité des opérations sur les threads.

La fonction `atexit()` permet d'installer un gestionnaire de terminaison de processus : c'est une fonction automatiquement appelée lors de l'exécution de l'appel système `exit()` ; il y a empilement de gestionnaires, lorsque plusieurs appels à `atexit()` sont effectués.

1.3 Génération de l'exécutable

Question 3.

a) Que signifient les messages d'erreurs produits lors qu'on compile le programme `thread1.c` comme suit :

```
gcc thread1.c -Wall -o thread1
```

b) Comment corriger ces problèmes ?

1.4 Résultat de l'exécution

L'exécution de ce programme par la commande `./thread1` produit ce genre d'affichage :

```
$ ./thread1
Le processus 28751 va créer deux threads
Thread fils :   Début de traitement
        atttente de la terminaison des threads
Thread fils :   Début de traitement
Thread fils :   Fin de traitement
```

```
Thread fils :   Fin de traitement
Fin du processus 28751
```

2. Seconde version

2.1 Identification des threads

Le programme précédent présente le défaut de ne pas identifier les threads qui s'exécutent, lors des affichages par exemple.
La fonction `pthread_create ()` range à l'adresse fournie comme premier argument une information d'identification du thread créé : c'est une information de type `pthread_t`, autrement dit de type *opaque*, uniquement utilisée par d'autres fonctions sur les threads (`pthread_join()` notamment).
Pour les curieux, examiner le résultat de la commande
`less $(locate bits/pthreadtypes.h) | grep "pthread_t;"`
N'en tirez pas de conclusion trop hâtive car cette définition est dépendante de l'implémentation des threads Posix.

2.2 Cahier des charges

Commencer par créer une copie du programme précédent, qui sera nommée `thread2.c`, puis apporter ensuite les modifications permettant de :

1. créer `NBMAX_THREADS` threads qui partagent le même code, c'-à-d. la même fonction.

   Cette création doit être réalisée à l'aide d'une structure de contrôle itérative `for`, par exemple ;

2. Chaque thread affiche son numéro d'ordre de création : 1 pour le premier, etc. Entre l'affichage de son début de traitement et celui de sa fin de traitement, chaque thread s'endort pendant une durée tirée au sort dans l'intervalle $[1, \text{DUREE\_MAX}]$

   L'affichage des message d'un thread doit être effectué avec un niveau d'indentation dépendant de l'ordre de sa création.

3. le thread principal attend la terminaison des threads enfants, selon leur ordre de création, et non pas selon leur ordre de terminaison

2.3 Exemple de résultat à obtenir

```
$ ./thread2
Le processus 29051 va créer 4 threads
  Début du thread n°1
    Début du thread n°2
      Début du thread n°3
        attente de la terminaison des threads
        Début du thread n°4
        Fin du thread n°4
    Fin du thread n°2
      Fin du thread n°3
  Fin du thread n°1
Fin du processus 29051
```

2.4 Indications

1. Ranger dans une variable globale définie comme une table de `NBMAX_THREADS`, les identifiants de chaque thread créé.

2. Chaque thread peut accéder à cette variable globale et peut obtenir son ordre de création en y recherchant la présence de son propre identifiant (cf. `pthread_self()`)

3. Ne pas utiliser l'opérateur ordinaire de comparaison (`==`) pour la recherche mentionnée ci-dessus, mais utiliser la fonction `pthread_equal()`

2.5 Exercice pratique

Question 4.
Écrire et tester le programme `thread2.c` répondant aux spécifications présentées plus haut, et permettant d'obtenir un résultat comparable à celui de la section 2.3.

3. Troisième version

3.1 Motivation

L'utilisation de variables globales partagées par l'ensemble des threads est souvent source de problèmes, notamment lorsque ceux-ci effectuent des accès en lecture et en modification (cf. chapitre suivant sur les mécanismes de communication/coopération/synchronisation)
Dans le cas de notre exemple :

- une modification intempestive réalisée par un thread sur la table des identifiants peut conduire à un programme qui boucle sans fin ;

- les threads enfants s'exécutent de façon indépendante et n'ont pas besoin de communiquer entre eux.

Aussi, allons-nous déplacer le tableau des identifiants de threads pour qu'il ne soit accessible que par le thread principal (la fonction `main()`)
Il faut maintenant être capable de fournir un argument au thread créé.

3.2 Argument de thread

L'argument de la fonction qui implémente le corps d'un thread est de type `void *`, autrement dit un pointeur de *n'importe quoi*.

a) Une mauvaise idée

Il serait tentant d'utiliser l'adresse de la variable de boucle utilisée lors de la création des threads, mais ce serait une erreur car chaque thread récupérerait son numéro dans la même variable or cette variable change de valeur dans la boucle de création et généralement aussi dans la boucle d'attente de terminaison — il est classique d'utiliser la même variable dans ces deux boucles —
Voici un exemple de résultat auquel on peut aboutir lorsqu'on procède de cette manière :

```
Le processus 29256 va créer 4 threads
    Début du thread n°2
        Début du thread n°4
        attente de la terminaison des threads
  Début du thread n°1
  Début du thread n°1
  Fin du thread n°1
  Fin du thread n°1
  Fin du thread n°1
  Fin du thread n°1
Fin du processus 29256s
```

b) Solution

Ce n'est pas l'adresse de cette variable de boucle qu'il faut fournir mais sa valeur.

1. Cette valeur doit être de même taille qu'une adresse pour l'architecture matérielle et logicielle utilisée.

   Pour une version de LINUX de 64 bits, le code ci-dessous …

   ```c
   printf("Taille d'un entier ........ = %ld\n", sizeof (int));
   printf("Taille d'un pointeur ...... = %ld\n", sizeof (void *));
   printf("Taille d'un entier long ... = %ld\n", sizeof (long));
   ```

   … produit ce résultat :

   ```
   Taille d'un entier ........ = 4
   Taille d'un pointeur ...... = 8
   Taille d'un entier long ... = 8
   ```

   On définira la variable de contrôle de la boucle de création des threads comme un entier long.

2. Pour simplifier l'accès à la valeur transmise à la fonction qui implémente le thread, son en-tête sera défini comme suit :

   ```c
   void  *monThread (long  no)
   ```

3. Cette signature n'étant plus compatible avec ce qu'attend la fonction `pthread_create()`, on crée un type pointeur de fonction ayant un seul paramètre de type `void *` et qui retourne un résultat de même nature.

   ```c
   typedef void * (*pf_t)(void *);
   ```

   Cette déclaration semble compliquée mais en procédant par étape, on y arrive facilement :

   a) définir l'en-tête d'une fonction qui a un argument de type `void *` et qui retourne un `void *`

      ```c
      void * pf_t (void *);
      ```

   b) faire de `pf_t` un pointeur de fonction qui …

      ```c
      void * (*pf_t)(void *);
      ```

   c) faire de `pf_t` un nouveau type de pointeur de …, on ajoute simplement `typedef` devant la déclaration précédente

      ```c
      typedef void * (*pf_t)(void *);
      ```

4. On effectue les forçages de type nécessaires lors de l'appel de la fonction `pthread_create()`

```
pthread_create (&tid[i], NULL, (pf_t)monThread, (void *)i);
```

3.3 Exercice

Question 5.
Créer une copie du programme précédent, qui sera nommée `thread3.c` et lui apporter les modifications nécessaires pour prendre en compte les indications présentées dans ce paragraphe.

4. Quatrième version

4.1 Traitement du code de terminaison

La fonction `pthread_join()` permet de récupérer le code de terminaison d'un thread.
Un thread se termine en exécutant soit une instruction `return` soit une opération `pthread_exit()`. La valeur retournée dans le premier cas, ou l'argument utilisé dans le second cas, peut être :

- `NULL` lorsqu'on n'a pas de résultat particulier à transmettre ;
- l'adresse d'une variable quelconque contenant le résultat à fournir.

Cette adresse peut être récupérée lors d'une opération `pthread_join()` et rangée dans une variable capable de la recevoir.

Comment déclarer et utiliser cette variable

Il faut tenir compte des remarques suivantes :

1. Cette variable doit être déclarée comme l'a été la variable spécifiée dans l'instruction `return` ou le `pthread_exit()` du thread qui se termine.

   Ainsi, pour un thread qui se termine par `return msg;` avec `msg` défini comme suit `char * msg`, la variable `status` du thread principal, destinée à recevoir le résultat, doit aussi être déclarée comme un `char *`

2. Pour que cette variable puisse recevoir ce résultat lors du `pthread_join()`, c'est donc son adresse qu'il faut utiliser … et ceci explique que le second argument de `pthread_join()` soit défini comme un `void **`

   Ainsi, pour reprendre l'exemple ci-dessus, l'opération qui attend la terminaison du thread dont l'identifiant est présent dans la variable `tid` sera effectué comme suit :

   ```
   pthread_join (tid, &status);
   ```

   Pour éviter la génération de message d'avertissement sur l'instruction ci-dessus, il suffit d'effectuer le forçage de type, présenté ici :

   ```
   pthread_join (tid, (void **)&status);
   ```

4.2 Application

Créer une nouvelle version du programme nommée `thread4.c` à partir de la version précédente.
On souhaite que chaque thread qui se termine retourne son numéro d'ordre de création, comme résultat.
On prépare dans la fonction `main()`, une variable capable de récupérer l'adresse où sera rangé le résultat. Ce résultat étant un entier long, cette variable est définie ainsi :

```
long * status;
```

Elle sera utilisée de la façon suivante dans la boucle d'attente de la terminaison de chaque thread créé :

```
for (i = 0; i < NBMAX_THREADS; i++) {
    CHECK_T(pthread_join (tid[i], (void **) & status),"pthread_join()");
    printf("--> fin du thread n°%ld avec le code %ld\n", i + 1, *status);
}
```

Il est évident que l'adresse reçue dans la variable `status` doit exister après la terminaison du thread : cela ne peut donc pas être l'adresse d'une variable locale !
C'est donc :

- l'adresse d'une variable globale. Ce n'est pas conseillé car une variable globale est accessible par tous les threads, or seul le thread principal s'en servira ;
- l'adresse d'une variable créée dynamiquement sur le tas (*heap*) par le thread enfant et restituée par le thread principal, dès que la valeur a été récupérée.

  On ajoutera par conséquent l'instruction `free(status);` après l'affichage de la valeur `*status` présenté ci-dessus.

Question 6.
Apporter les modifications au programme `thread4.c` de façon à pouvoir produire un résultat comparable à ceci :

```
$ ./thread4
Le processus 29832 va créer 4 threads
Début du thread n°1
  Début du thread n°2
        attente de la terminaison des threads
      Début du thread n°4
    Début du thread n°3
    Fin du thread n°3
  Fin du thread n°2
      Fin du thread n°4
Fin du thread n°1
--> fin du thread n°1 avec le code 1
--> fin du thread n°2 avec le code 2
--> fin du thread n°3 avec le code 3
--> fin du thread n°4 avec le code 4
Fin du processus 29832
```

Bibliographie

[1] W. Richard Stevens, Stephen A. Rago
Advanced Programming in the UNIX Environment, Addison-Wesley, Third Edition, 2013.

[2] Robert Love
Linux System Programming, O'Reilly, 2007.

[3] Christophe Blaess
Programmation système en C sous Linux, Eyrolles, 2e édition, 2005.

[4] Patrick Cegielski
Conception de systèmes d'exploitation : Le cas Linux, Eyrolles, 2e édition, 2004.

[5] Vincent Lozano
UNIX : Pour aller plus loin avec la ligne de commande
In Libro Veritas, Framabook, 2010
(Téléchargeable ici)

[6] Mendel Cooper, Linux Documentation Project
Advanced Bash-Scripting Guide

- Version anglaise en ligne ;
- Traduction en français en ligne.

[7] Robert Mecklenburg
Managing Projects with GNU Make, O'Reilly, 3rd edition, 2004
(en ligne sur le projet Open Books)

[8] Richard M. Stallman and the GCC Developer Community
Using the GNU Compiler Collection, GNU Press

- Version anglaise en ligne
- Version anglaise au format pdf

[9] O. Bonaventure, G. Detal, C. Paasch
Cours Systèmes Informatiques (2014) de l'Université catholique de Louvain

- Version en ligne
- Version au format pdf