Programmation Concurrente

Principes et concepts
2024-2025

Chapitre2: Threads Posix

Plan

- Rappel
- Thread Posix
- Les méthodes importantes
- Les exemples

Librairie Pthread

- Librairie normalisée POSIX
- Existe sur presque tous les systèmes actuels : Linux, Solaris, BSD, OSX, Windows, etc.
- Fichier d'en-tête:
- pthread.h
- Déclaration d'un thread

```
pthread_t thread;
```

Création d'un thread

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

- `thread` est un pointeur sur une variable de type **pthread_t**; il s'agit de l'identifiant du thread créé; c'est donc un argument de sortie
- `attr` permet de définir les attributs du thread (NULL = attributs par défaut)
- `start_routine` correspond à la fonction qui est exécutée par le thread créé. La fonction exécutée par le thread créé doit avoir le prototype suivant:
- void *fonction(void *data)
- `arg` est un argument passé à la fonction (NULL = pas d'argument)
- Renvoie 0 en cas de succès

Création d'un thread (Exemple_01)

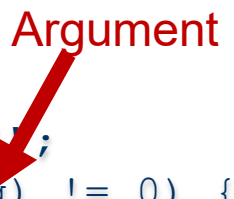
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *func(void *arg) {
  printf("Hello from our first thread!\n");
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t thread;
  if (pthread_create(&thread, NULL, func, NULL) != 0) {
    perror("thread creation error");
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Passage d'argument (Exemple_02)

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

void *func(void *arg) {
        char *msg = (char *) arg;
        printf("Message = %s\n", msg);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t t;
        char *msg = " Threads sont magnéfiques!";
        if (pthread_create(&t, NULL, func, msg) != 0) {
                perror("thread creation error");
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

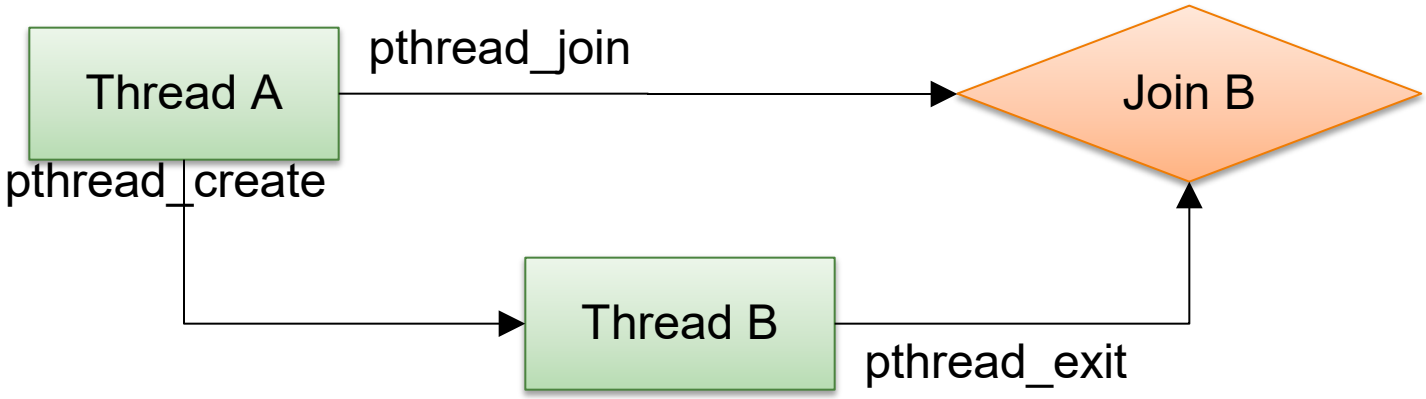
Passage d'argument (Exemple_02)

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

void *func(void *arg) {
        char *msg = (char *) arg;
        printf("Message = %s\n", msg);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t t;
        char *msg = "Threads sont magnéfiques!!";
        if (pthread_create(&t, NULL, func, msg) != 0) {
                perror("thread creation error");
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```



Jointure

- La jointure est une forme de communication entre threads
- Joindre un thread signifie bloquer jusqu'à la terminaison de celui-ci
- Le thread terminé peut retourner une valeur au thread ayant effectué la jointure



Jointure

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

- Attend que le thread passé en paramètre se termine
- Le thread appelant est bloqué jusqu'à la terminaison du thread spécifié
- `value_ptr` contient la valeur de retour du thread terminé
- Renvoie 0 en cas de succès

Jointure/exemple_03

```
void *func(void *arg) {
    char *msg = (char *) arg;
    printf("Message = %s\n", msg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t;
    char *msg = "Threads sont magnéfiques!";
    if (pthread_create(&t, NULL, func, msg) != 0) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }
    if (pthread_join(t, NULL) != 0) {
        perror("pthread_join");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Jointure/exemple_03

```
void *func(void *arg) {
    char *msg = (char *) arg;
    printf("Message = %s\n", msg);
    int val = 33;
    return (void*)val;
}

int main(int argc, char *argv[]) {
    pthread_t t;
    char *msg = " Threads sont magnéfiques!";
    if (pthread_create(&t, NULL, func, msg) != 0) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }
    int *exit_code;
    if (pthread_join(t, (void **)&exit_code) != 0) {
        perror("pthread_join");
        return EXIT_FAILURE;
    }
    printf("Thread exit code = %d\n", exit_code);
    return EXIT_SUCCESS;
}
```

Jointure/exemple_04

```
void *func1(void *arg) {
        static int exit_code = 0;
        printf("Thread 1\n");
        return &exit_code;
}

void *func2(void *arg) {
        static int exit_code = 4;
        printf("Thread 2\n");
        pthread_exit((void *)&exit_code);
}

int main(int argc, char *argv[]) {
        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, func1, NULL);
        pthread_create(&thread2, NULL, func2, NULL);
        int *status1, *status2;
        pthread_join(thread1, (void **)&status1);
        pthread_join(thread2, (void **)&status2);
        printf("Status1: %d\n", *status1);
        printf("Status2: %d\n", *status2);
        return EXIT_SUCCESS;
}
```

Jointure/Exemple_05

```
void *func(void *arg) {
    char *msg = (char *) arg;
    printf("Message = %s\n", msg);
    int *code = (int *) malloc(sizeof(int));
    *code = 47;
    return code;
}

int main(int argc, char *argv[]) {
    pthread_t t;
    char *msg = "Threads are awesome!";
    if (pthread_create(&t, NULL, func, msg) != 0) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }
    int *exit_code;
    if (pthread_join(t, (void **)&exit_code) != 0) {
        perror("pthread_join");
        return EXIT_FAILURE;
    }
    printf("Thread exit code = %d\n", *exit_code);
    return EXIT_SUCCESS;
}
```

Détachement d'un thread

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

- Un thread détaché est un thread ne pouvant plus être joignable; son exécution devient donc indépendante des autres threads
- Il est possible de détacher un thread en utilisant le fonction `pthread_detach`
- Le thread passé en paramètre devient détaché
- Renvoie 0 en cas de succès

Détachement d'un thread

- Tout thread créé est joignable par défaut (propre à l'implémentation Linux)
  - Il est possible de spécifier si un thread est joignable ou pas à la création en:
    - déclarant une variable d'attribut de thread de type `pthread_attr_t` avec la fonction `pthread_attr_init`
    - mettant l'attribut détaché à `PTHREAD_CREATE_DETACHED` ou `PTHREAD_CREATE_JOINABLE` avec la fonction `pthread_attr_setdetachstate`
    - passant cet attribut à la fonction `pthread_create`
- Une fois terminé, l'attribut doit être détruit avec la fonction `pthread_attr_destroy`
- Rien ne garantit qu'un thread crée avec pthread_create sera joignable par défaut ! Par soucis de portabilité, il est donc recommandé de spécifier cet attribut au moment de la création du thread.

Terminaison d'un thread

- La terminaison d'un thread peut être exécutée depuis :
  - Le thread lui-même :
    - Instruction return
    - Fonction pthread_exit
    - Un autre thread (annulation) :
    - Fonction pthread_cancel
- ATTENTION !
  - Mal terminer un thread peut laisser le système dans un état incohérent !
  - Plus spécifiquement depuis un autre thread.

Auto terminaison

```
void pthread_exit(void *value);
```

- La fonction met fin au thread et retourne value au thread attendant grâce à une jointure (fonction `pthread_join`)
- Un thread peut aussi utiliser le mot clé return pour se terminer et retourner une valeur à son père
- Lorsque `pthread_exit` est appelé depuis le thread principal (main), ce dernier bloque et attend que tous ses threads enfants soient terminés avant de se terminer lui-même !
- La fonction **exit**(**int** status) permet de terminer le processus, ce qui a pour effet de terminer instantanément tous les threads du processus; ceci, quel que soit le thread ayant appelé **exit**

Annulation d'un thread

```
int pthread_cancel(pthread_t thread);
```

- `thread` est le thread à terminer (annuler)
- Cette fonction permet d'annuler (terminer) un thread depuis un autre
- Le seul moyen pour savoir si un thread a été annulé est de faire un pthread_join puis d'inspecter la valeur de sortie du thread; en cas d'annulation cette dernière est alors `PTHREAD_CANCELED`
- La terminaison s'effectue sur un point d'annulation
- Si la fonction du thread s'est terminée avant l'appel à `pthread_cancel`, alors la valeur `ESRCH` est retournée (thread non trouvé); la constante `ESRCH` est définie dans le header errno.h
- Renvoie 0 en cas de succès

Politique d'annulation

```
int pthread_setcancelstate(int state, int *oldstate);
```

- Permet à un thread de définir sa politique d'annulation
- `state` peut prendre les valeurs suivantes :
  - `PTHREAD_CANCEL_ENABLE` : Autorise l'annulation (défaut)
  - `PTHREAD_CANCEL_DISABLE` : Interdit l'annulation
- `oldstate` contient ensuite l'ancienne politique d'annulation
- Renvoie 0 en cas de succès

Politique d'annulation

- Si le thread autorise l'annulation, il peut alors définir son type d'annulation avec :

```
int pthread_setcanceltype(int type, int *oldtype);
```

- type peut prendre les valeurs suivantes :
  - PTHREAD_CANCEL_DEFERRED : Autorise l'annulation en des points précis (défaut)
  - PTHREAD_CANCEL_ASYNCHRONOUS : Autorise l'annulation n'importe quand
- oldtype contient ensuite l'ancien type d'annulation
- Renvoie 0 en cas de succès

Point d'annulation

- Un thread peut placer ses points d'annulation avec :

```
void pthread_testcancel(void);
```

- L'annulation est donc permise en ces points
- ATTENTION !
  - Certains appels systèmes sont des points d'annulation, en particulier les fonctions d'entrée/sortie, car potentiellement bloquantes; exemple: fopen, write, fprintf, etc.

Auto-identification

- Un thread peu obtenir son identifiant avec la fonction:

```
pthread_t pthread_self();
```

- Il s'agit donc de la même valeur d'identifiant que celle retournée en premier argument de la fonction pthread_create
- Chaque thread dispose d'un numéro d'identifiant unique
- Sur linux x64, l'identifiant est un entier long non signé (unsigned long int)
- Attention: il n'est pas garanti que l'identifiant d'un thread soit un entier ! Le type est propre à l'implémentation (voir slide suivante pour comparer des identifiants de threads)

Comparaison d'identification

- Comparer deux identifiants de threads doit se faire avec la fonction:

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

- Si les threads t1 et t2 sont égaux, alors la fonction renvoie une valeur différente de zéro

- Il est nécessaire d'utiliser cette fonction pour comparer les identifiants, car rien ne garantit que les types pthread_t soient des valeurs comparables avec l'opérateur d'égalité =

Rendre le processeur

- Il est possible de forcer le thread appelant à relâcher le processeur avec la fonction:

```
#include <sched.h>

   int sched_yield();
```

- Après l'appel à cette fonction, le thread est placé à la fin de la file d'attente des threads en attente du processeur
- Le thread suivant en attente du processeur est alors activé
- Renvoie zéro en cas de succès

Fonctions « async-cancel-safe »

- Une fonction est dite « async-cancel-safe » si elle peut être annulée de manière asynchrone (annulée concurremment)
- Toute fonction non « async-cancel-safe » annulée peut potentiellement engendrer un crash du programme !
- Parmi toutes les fonctions de la librairie POSIX Threads, seules les 3 fonctions suivantes sont garanties être async-cancel-safe:
- pthread_cancel
- pthread_setcancelstate()
- pthread_setcanceltype()

Conclusion

- Se référer au document complet :

  «Multithreaded Programming Guide»

- Libraire riche avec plusieurs implémentations

- Portabilité

- Même dans certain OS mobile