TP 2 : threads et prog. parallèle

R. Raffin
IUT Aix-Marseille
dépt. Informatique, Arles
romain.raffin[AT]univ-amu.fr

2014

1 Threads

Les `threads` en Unix sont soumis à la norme Posix, cela implique une construction et une gestion identique quel que soit le système d'exploitation. La majeure différence par rapport à une création de processus (`fork`), vient du fait qu'un `thread` n'est qu'une duplication du compteur ordinal et des registres mais que la mémoire utilisée par le père devient commune à tous.

Prototype :

```
#include <pthread.h>
```

Cette inclusion est obligatoire pour programmer des `threads` (ainsi que l'option `-lpthread` dans `g++`), voir le manuel.

```
int pthread_create(pthread_t* dthread, pthread_attr_t*attr,
  void*(*start_routine)(void*), void* arg)
```

Cela crée un `thread`, dont le descripteur est retourné dans `dthread`, qui lance la `start_routine` avec les arguments (paramètres) `arg`. Les attributs servent à gérer la vie des ressources utilisées par le `thread` (`Joignable`, si on veut s'en resservir. `Detached` sinon, cf `man`).

```
int pthread_join(pthread_t th, void** thread_return)
```

Cela permet de fixer un point d'attente avec un thread `th`, jusqu'à ce que celui-ci se termine. Les valeurs de retour de ce processus sont fixées dans `thread_return`.

1.1 Utilisation

(voir le fichier « `premierThread.cpp` »)

```
void * fct_thread(void * _arg) {
for( unsigned int i = 0; i < 10; i++)
{
     std::cout << "t1 " << i << std::endl;
     sleep(1);
}
return (NULL);
}

int main() {

pthread_t p; //identifiant de thread (type spécifique, cf pthread.h)

pthread_create(&p, NULL, fct_thread, NULL); //lancement du thread, via la fonction fct_thread

std::cerr << "MAIN(), pendant le  thread\n";  //le programme principal peut continuer son exécution

pthread_join(p, NULL); //le programme principal bloque en attendant la fin du thread (return)

         std::cerr << "MAIN(), threads terminés\n";  //après le join() seul reste le programme principal

return(EXIT_SUCCESS); //fin du programme principal
}
```

1.2 À vous de jouer !

- prendre l'exemple de « `premierThread.cpp` », le compiler et le modifier pour pouvoir voir le (ou les processus) en action (`ps -ef` et affichage d'états).
- programmez 2 threads ayant une exécution concurrente, permettant de calculer les cosinus d'angles compris entre 0 et 360°, par pas de 10°. Utilisez un tableau global au programme ou des tableaux locaux à chaque thread, envoyés lors du `pthread_create`, en lecture/écriture. Remarque : compilation avec la librairie `math`

  ```
  g++ mesthreads.cpp -o mesthread -lpthread -lm
  ```
- programmez l'interaction de thread sur des structures composées d'un entier, un réel, un tableau de caractères, un pointeur de type `char *`). Déclarer des structures locales au `main()` qui seront envoyées en lecture/écriture dans chaque thread. Afficher les structures aux différentes étapes.

Besoins : (Linux,) librairie Posix `pthread`, g++ > 4.0

Le problème majeur des threads, qui est aussi un avantage, est le partage des données entre le programme principal et ses threads. On a donc un fonctionnement maître-esclaves avec un passage d'informations via la RAM, donc pas besoin de mettre en place signaux ou *shared-memory*. Pour utiliser les threads sans être obligé de gérer la concurrence de l'accès aux données, on effectue d'abord un découpage des données. Les threads sont plus faciles à mettre en place par le système (*lightweight process*), puisqu'on ne clone pas l'intégralité du processus.

Les minima pour utiliser les threads sont donc :

1. `#include <pthread.h>` en en-tête,

2. avoir un ou des identifiants de thread, `pthread_t th1` pour l'identifiant `th1`,
3. utiliser `pthread_create(&th1, NULL, (void*) fct_thread1, NULL)`, pour lancer une fonction où les paramètres sont (identifiant de thread, options, fonction - de prototype `void * fct_thread1(void *)`, les paramètres de la fonction (de type `void*`).
4. `pthread_join(th1, NULL)` pour attendre (par un rendez-vous) la fin de l'exécution du thread `th1`, avec le 2ème paramètre le retour éventuel, de type `void *`.

De la documentation supplémentaire est disponible dans le manuel de `pthread_create`, notamment les threads « attachés » ou « détachés ».

1.3 Exemple simple

Voilà un exemple de calcul de la moyenne arithmétique de N données scalaire d_i situées dans un tableau, par des threads. La formule est :

$$\text{Moyenne} = \frac{\sum_{i=0}^{N-1} d_i}{N}$$

Les données ne sont pas dépendantes, pour éviter des problèmes d'accès on découpe le tableau initial en sous-tableaux (selon le nombre de threads), on passe ensuite chaque sous-tableau à un thread qui calcule la somme, on récupère les résultats de chaque thread et il ne reste qu'une division à faire.