INFO-F-201 – OS
TP9 : Programmation réseau multi-client et programmation multi-threads

Yannick Molinghen, Alexis Reynouard

25 novembre 2021

ULB

1 Programmation réseau multi-clients

Il est souvent nécessaire pour un serveur d'accepter plusieurs connexions à la fois, comme le ferait un serveur web. Il existe plusieurs manières de gérer des clients simultanées :

1. Gérer chaque client dans un processus
2. Gérer les clients dans un thread
3. Utiliser l'appel système `select`

Si les deux premières options peuvent être de bonnes options dans certains cas, elles n'en sont pas moins lourdes à gérer pour l'OS. Nous allons dans un premier temps explorer la troisième option qui permet de gérer plusieurs connexions simultanément sans utiliser de multi-processing ni de multi-threading.

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

Listing 1 – Appel système `select` et ses macros

L'appel système `select` se base sur le principe de `fd_set`, pour *file descriptor set* ou ensemble de *file descriptors*. L'idée est de mettre dans des `fd_set` tous les *file descriptors* que l'on veut gérer, et de les donner comme entrée à `select` qui sera blocante jusqu'à ce qu'au moins un des *file descriptors* dans les `fd_set` soit prêt.

On distingue trois `fd_set`

— Ceux en lecture qui débloqueront `select` dès que l'un des sockets a des données à lire
— Cenx en écriture qui débloqueront `select` dès qu'il est possible d'écrire dans l'un d'eux
— Ceux susceptibles de générer des erreurs qui débloqueront le `select` dès qu'une erreur s'y produit.

Ainsi, un serveur peut mettre dans son `fd_set` tous les sockets de ses clients, et attendre que des données soient disponibles sur n'importe lequel des sockets (donc une requête) pour ensuite lui répondre.

Les macros présentes dans le Listing 1 sont des utilitaires pour gérer les structures `fd_set`, comme montré dans le Listing 2.

```c
int main(int argc, char *argv[]) {
  int opt = 1;
  int master_socket = socket(AF_INET, SOCK_STREAM, 0);
  setsockopt(master_socket, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt));

  struct sockaddr_in address;
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = INADDR_ANY;
  address.sin_port = htons(8080);
  bind(master_socket, (struct sockaddr *)&address, sizeof(address));
  listen(master_socket, 3);

  socklen_t addrlen = sizeof(address);
  fd_set readfds;
  FD_ZERO(&readfds);
  FD_SET(master_socket, &readfds);
  int max_fd = master_socket;
  // Le seul file descriptor ajouté est celui du socket "maître"
  select(max_fd + 1, &readfds, NULL, NULL, NULL);

  if (!FD_ISSET(master_socket, &readfds)) {
    fprintf(stderr, "Impossible d'arriver ici (en théorie)\n");
    exit(1);
  } else {
    // Vu que le master socket qui a des donnees, c'est une nouvelle connexion.
    int nouveau_sock = accept(master_socket, (struct sockaddr *)&address, &addrlen);
    // ...
    // Il faut maintenant faire quelque chose d'utile avec ce nouveau socket,
    // comme l'ajouter au fd_set pour détecter lorsqu'il y a
    // de nouvelles données à lire.
  }
  return 0;
}
```

Listing 2 – Exemple de serveur utilisant `select`

Exercice 1. *Reprenez la solution de l'exercice 3 de la semaine passée et modifiez-le pour que votre serveur accepte plusieurs connexions simultanées avec `select`. Le client reste identique.*

2 Programmation multi-threads

Le concept de thread est proche de celui de processus car tous deux sont un moyen de paralléliser un programme. Toutefois, là où chaque processus possède sa propre mémoire virtuelle (heap et stack), les threads d'un même processus se partagent le heap. Par contre, vu que les threads ont des exécutions différentes, ils ne peuvent de toute évidence pas partager le même stack.
Nous allons dans ce TP utiliser les threads conformes à la norme POSIX, ou `pthread`, que l'on peut créer à l'aide de la fonction `pthread_create`. La fonction `pthread_exit` permet, elle de mettre fin à un thread.

```
#include <pthread.h>

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

Le premier paramètre, `thread`, contiendra l'identifiant unique du thread créé, il sert principalement comme paramètre pour les autres fonctions liées aux threads que nous verrons ci-après.
Le second paramètre, `attr`, est une structure de données utilisée lors de la création du thread pour déterminer les attributs de ce dernier. Dans le cadre de ce cours, ce paramètre prendra la valeur `NULL`.
Le dernier paramètre, (`start_routine`), est un pointeur vers la fonction à exécuter pour le thread. En effet, un thread a pour vocation à effectuer une unique fonction conforme à cette signature, come illustré ci-dessous.

```
// Une fonction compatible avec pthread_create
void* nom_de_la_fonction (void*);
```

Le dernier paramètre contient le paramètre qui sera passé en paramètre de la fonction `start_routine`.

Une fois l'appel à `pthread_create` effectué, l'exécution reprend à l'instruction suivante dans le thread principal, et dans la fonction dans le nouveau thread.
Tout comme il est utile d'attendre la fin d'un processus enfant, il est souvent utile, voire nécéssaire, de synchroniser les threads entre eux et d'attendre la fin d'un thread en particulier. Cela se fait à l'aide de la fonction `pthread_join`.

```
int pthread_join(pthread_t thread, void **retval)
```

La fonction prend en paramètre l'identifiant du thread à attendre, et un pointeur sur un espace mémoire qui contiendra la valeur de retour du thread attendu.
Le thread se termine lorsque l'un des cas suivants se produit :

1. le thread fait un appel à l'appel système suivant :

   ```
   void pthread_exit(void *retval)
   ```

   Cette fonction permet de spécifier une valeur de retour. Comme énoncé précédemment, la valeur de retour peut être récupérée par le thread principal avec un appel à `pthread_join`,
2. le thread effectue un retour classique de la fonction pour laquelle il a été créé. Ce type de terminaison est équivalente à un appel à `pthread_exit`,
3. le thread principal effectue un appel à la fonction suivante :

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

Le paramètre `thread` est l'identifiant unique du thread à interrompre.

4. Le thread effectue un appel à `exit` provoquant la terminaison du thread appelant, du thread principal et de tous les autres threads lancés par le thread principal.

Exercice 2. *Écrivez un programme C créant un thread. Le thread principal écrira cent fois 1 et le second thread écrira cent fois 2.*

Exercice 3. *Modifiez votre solution de l'exercice 1 pour utiliser la programmation multi-thread au lieu de `select`. Chaque connexion avec un client doit être gérée dans son propre thread.*

Exercice 4. *Ecrivez un programme qui calcule la moyenne d'un tableau de nombres aléatoirement générés de manière distribuée à l'aide de threads.*