IV - Connexion par adresse de noms

Connexion par adresse de noms

Pour l'humain, retenir une adresse composée de noms plutôt qu'une suite de nombres est plus simple.

Si on souhaite permettre à l'utilisateur d'entrer une adresse de noms,

- on ne peut plus utiliser les fonctions `inet_pton` et `inet_ntoa`,
- il faut donc pouvoir traduire ces adresses de nom en adresse IP :
  - On a vu que cela signifie qu'il faut interroger un dictionnaire (en général service DNS) pour obtenir la traduction d'une adresse de noms en adresse IP.
  - On peut vouloir ne retenir qu'un sous-ensemble d'adresses répondant à des critères particuliers (IPv4, IPv6, avec socket TCP ou UDP)
  - Mais comment fait-on en C ?

Connexion par adresse de noms

struct addrinfo

On commence par remplir une variable `hints` de type `struct addrinfo` avec des indications comme le type d'adresses ou de socket voulu.

```
struct addrinfo {
  int ai_flags;              /* options */
  int ai_family;             /* famille de protocols pour la socket */
  int ai_socktype;           /* type de socket */
  int ai_protocol;           /* protocole pour la socket */
  socklen_t ai_addrlen;      /* taille de l'adresse */
  struct sockaddr *ai_addr; /* addresse: IP, port... */
  char *ai_canonname;        /* nom canonique de l'adresse */
  struct addrinfo *ai_next; /* pointeur sur le suivant dans la liste */
};
```

Connexion par adresse de noms

struct addrinfo

Valeurs d'initialisation de `hints` pour obtenir des adresses IPv4 pour socket TCP :

- `ai_flags` : égal à `0` pour le moment
- `ai_family` : égal à `AF_INET`
- `ai_socktype` : égal à `SOCK_STREAM`
- `ai_protocol` : égal à `0` dans le cas de TCP
- `ai_canonname` : égal à `0` pour le moment

Tous les autres éléments de `struct addrinfo` passés via `hints` doivent être mis à `0` (ou au pointeur `NULL`).

Connexion par adresse de noms

Initialisation de `hints`

On dit que l'on veut récupérer des adresses IPv4 pour socket TCP

```
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
```

Et maintenant, comment interroger le dictionnaire pour traduire les noms d'adresses ?

Connexion par adresse de noms

getaddrinfo

On doit utiliser la fonction

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `node` : chaîne de caractères contenant l'adresse de noms
- `service` : chaîne de caractères contenant le numéro de port
- `hints` : contient des indications sur le type d'adresses, de socket...
- `res` : liste chaînée de pointeurs de `struct addrinfo` qui recevra les différentes adresses de socket possibles

Connexion par adresse de noms

getaddrinfo

`getaddrinfo` va remplir la liste chaînée `res`.

- Le champs `ai_addr` de chaque élément de `res` pointe sur une adresse de socket remplie.
- Ce champs est de longueur `ai_addrlen`.
- Le champs `ai_next` du dernier élément de la liste est égal à `NULL`.

```
struct addrinfo *r;
if ((getaddrinfo(hostname, port, &hints, &r)) != 0 || r == NULL)
  return -1;
```

- `hostname` et `port` sont des chaînes de caractères contenant respectivement l'adresse de noms et le numéro de port de l'entité dont on veut récupérer les adresses.
- `r` contient maintenant la liste chaînée des « adresses ».

Connexion par adresse de noms

client IPv4

Et maintenant, comment un client peut-il se connecter au serveur connaissant l'adresse de noms de ce dernier ?

Pour se connecter le client teste chaque élément `p` de la liste d'adresses `r` jusqu'à obtenir une connexion. Pour cela :

1. il crée une socket,
2. tente de se connecter avec la socket à l'adresse et au port de `p`,
3. si la connexion réussie, il passe à la suite...
4. sinon, il ferme la socket et recommence le test sur l'élément suivant tant que celui-ci n'est pas `NULL`.

Connexion par adresse de noms

client IPv4

```
struct addrinfo *p;
p = r;

while( p != NULL ){
  if((sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0){
    if(connect(sock, p->ai_addr, addrlen) == 0)
      break;

    close(sock);
  }

  p = p->ai_next;
}

// le client est maintenant connecté (sauf si p == NULL),
// la conversation avec le serveur peut commencer...
```

Connexion par adresse de noms

client IPv4

Lorsqu'on a plus besoin de la liste chaînée d'adresses, il faut libérer la mémoire allouée par getaddrinfo en faisant appel à la fonction freeaddrinfo :

```
void freeaddrinfo(struct addrinfo *res);
```

ce qui donne ici

```
if(r)
  freeaddrinfo(r);
```

Connexion par adresse de noms

Et pour un client IPv6 ?

```
struct addrinfo hints, *r, *p;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;     // socket IPv6
hints.ai_socktype = SOCK_STREAM;

if ((getaddrinfo(hostname, port, &hints, &r)) != 0 || r == NULL)
  return -1;

*addrlen = sizeof(struct sockaddr_in6);  //adresse IPv6
p = r;
while( p != NULL ){
  if((*sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0){
    if(connect(*sock, p->ai_addr, *addrlen) == 0)
      break;
    close(*sock);
  }
  p = p->ai_next;
}

freeaddrinfo(r);
```

IV - Concurrence

Communications en parallèle

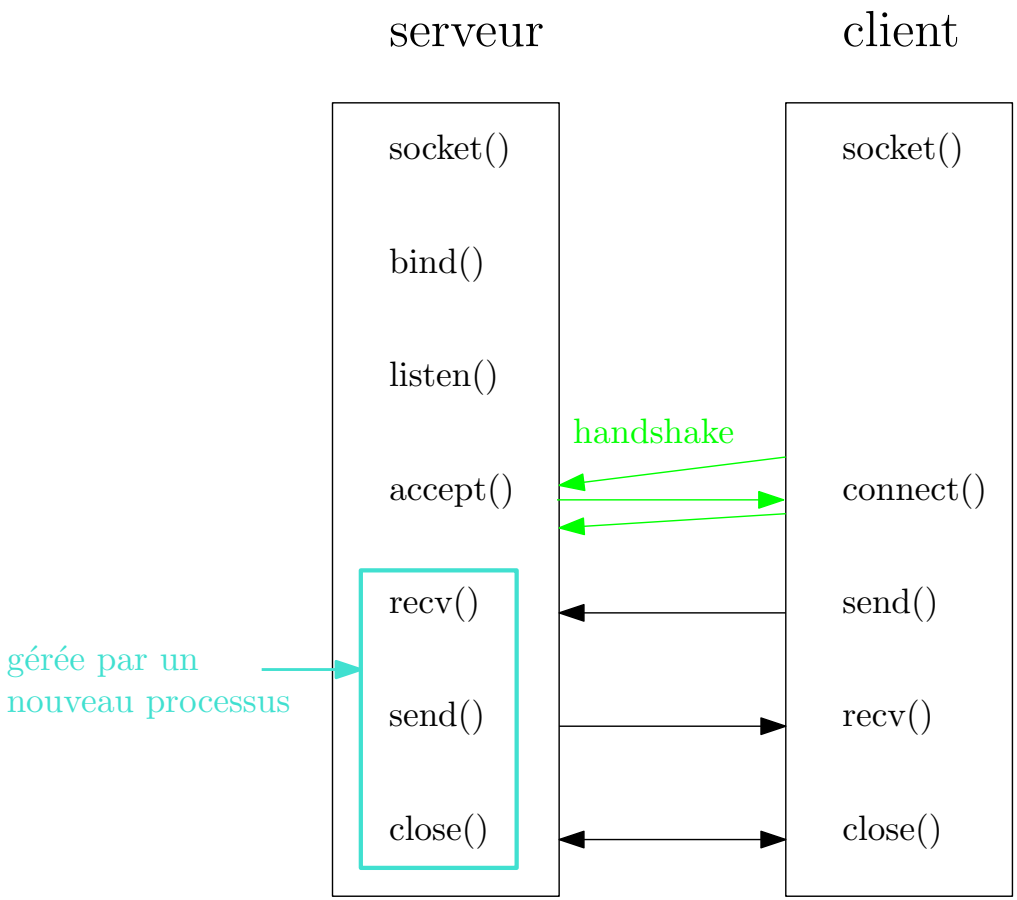
Un serveur qui ne peut s'occuper que d'un seul client à la fois peut vite être saturé, c'est-à-dire rejeter des demandes de connexion provenant de clients. Et les clients en attente de connexion peuvent s'impatienter...

Pour palier à ce problème, il y a plusieurs solutions :

- créer un nouveau processus (`fork`) à chaque nouvelle connexion d'un client,
- créer un nouveau processus léger (`thread`) à chaque nouvelle connexion d'un client,
- utiliser une socket non bloquante, c'est-à-dire, que les opérations d'acceptation d'une connexion, de reception et d'envoi sont non bloquantes.

Concurrence

Aujourd'hui, on s'intéresse aux deux premières solutions : créer un nouveau processus à chaque connexion d'un client.



Processus

Un processus est un programme en cours d'exécution.

Il est définit par

- un ensemble d'instructions à exécuter
- un environnement formé de
  - un espace d'adressage
  - ressources pour gérer les entrées/sorties de données

Plusieurs processus s'exécutent sur une même machine de façon quasi-simultanée.
C'est le système d'exploitation qui est chargé d'allouer les ressources mémoire, le temps processeur, les entrées/sorties.
D'un point de vue utilisateur, cela donne l'illusion du parallèlisme.

Rappels du cours de « Systèmes d'exploitation » :

- on crée un nouveau processus en faisant appel à la fonction `pid_t fork(void)`
- la fonction crée un nouveau processus et retourne
  - `0` pour le fils,
  - l'identifiant du nouveau processus (PID) pour le père
- à la création d'un processus fils, l'espace d'adressage du père est copié
- ensuite, les variables ne sont pas partagées entre les processus père et fils

Le serveur, après l'appel à

```
int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
```

crée un processus fils avec `fork`.

- Le processus fils doit alors :
  - fermer son descripteur `sock`
  - communiquer avec le client via son descripteur `sockclient`
  - fermer `sockclient` à la fin de la communication
  - terminer son exécution par `exit`

- Le processus père doit :
  - fermer son descripteur `sockclient`
    Les espaces d'adressage étant copiés à la création du fils, si le père ferme son descripteur `sockclient`, cela ne ferme pas celui du fils
  - retourner sur `accept` pour attendre une nouvelle connexion
  - récupérer les processus zombies avec appel non bloquant à `waitpid`

Rappel : `pid_t waitpid(pid_t pid, int *wstatus, int options);`

- `pid = -1` si le père attend n'importe quel fils
- `wstatus = NULL` si on ne veut pas récupérer d'information sur la terminaison du processus fils
- `options = WNOHANG` afin que `waitpid` ne soit pas bloquant

Concurrence

Processus avec fork()

```
while(1){
  int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
  if(sockclient == -1); //gérer l'erreur...

  switch(fork()){
  case -1 : break; //gérer l'erreur...
  case 0 :         //fils
    close(sock);
    int ret = communication(sockclient);
    exit(ret);
  default :        //père
    close(sockclient);
    affiche_connexion(adrclient);
    while(waitpid(-1, NULL, WNOHANG) > 0); //récupération des zombies
  }
}
```

Le problème avec `fork()` est que si les processus souhaitent partager des variables, ils doivent communiquer via une autre entité comme un tube.

Par exemple, dans ce programme, la variable `x` n'est pas partagée :

```
int main(){
  int x = 0;
  switch(fork()){
  case -1 : break; //gérer l'erreur...
  case 0 :         //fils
    x=25;
    printf("Valeur de x pour le fils %d\n",x);
    break;
  default :        //père
    sleep(2);
    printf("Valeur de x pour le pere %d\n",x);
    waitpid(-1, NULL, 0);
  }
  return 0;
}
```

L'exécution donne

```
Cours4$ ./pb_fork
Valeur de x pour le fils 25
Valeur de x pour le pere 0
```

→ les threads ou processus légers permettent le partage de variables.

Thread

Un thread est un fil d'exécution dans un programme, le programme étant lui même exécuté par un processus.

- Un processus peut avoir plusieurs threads $\rightarrow$ processus **multi-threadé**
- Chaque fil d'exécution est distinct des autres et est défini par
  - un **point courant d'exécution** (pointeur d'intstruction ou PC (Program Counter))
  - une **pile d'exécution** (stack)

Le processus principal et les threads qu'il a lancés, partagent :

- le **tas** (heap) → variables allouées avec `malloc`
- la **mémoire statique** (constantes, variables globales)
- le **code**

Un thread est donc un **processus léger** car le changement de contexte d'exécution est moins « lourd » que dans le cas d'un processus créé par `fork`. Le système à moins d'informations à charger lors du passage d'un fil d'exécution à l'autre.

En C, la bibliothèque POSIX `pthread` permet d'utiliser des threads.

- il faut compiler un programme incluant `pthread.h` avec l'option `-pthread` : `gcc -Wall -pthread serveur.c -o serv`

Pour créer un thread, on a la fonction

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

- `thread` contient les données du thread créé
- `attr` contient les attributs donnés au thread à sa création (taille de la pile...). Mettre à `NULL` pour choisir les attributs par défaut
- la fonction `start_routine()` contient le code que va exécuter le thread. Elle prend en paramètre un `void *` et retourne un `void *`
- `arg` est un pointeur vers les arguments de la fonction `start_routine()`

Concurrence

création d'un thread

```
int *sockclient = malloc(sizeof(int));
*sockclient = accept(sock, (struct sockaddr *) &caller, &size);
if (*sockclient < 0); //gérer l'erreur...

pthread_t thread;
if (pthread_create(&thread, NULL, serve, sockclient) == -1){
  perror("pthread_create");
  continue;
}
```

Au retour de l'appel à `pthread_create`, le thread est créé et son exécution commence.

Le code exécuté par le thread est défini dans la fonction de prototype `void *serve(void *)`.

Cette fonction prend en paramètre un pointeur sur la socket `sockclient` de communication avec le client qui vient de se connecter.

Concurrence

création d'un thread

```
void *serve(void *arg) {
  int sock = *((int *) arg);       //on récupère le descripteur de socket
  char buf[SIZE_BUF+1];
  memset(buf, 0, sizeof(buf));

  int recu = recv(sock, buf, SIZE_BUF, 0);
  if (recu < 0){
    close(sock);
    return NULL;
  }
  if(recu == 0){
    close(sock);
    return NULL;
  }
  printf("recu : %s\n", buf);
  char c = 'o';
  int ecrit = send(sock, &c, 1, 0);
  if(ecrit <= 0)
    perror("erreur ecriture");

  close(sock);
  return NULL;
}
```

Attention, il faut passer en argument de la fonction `serve` une variable allouée sur le tas sinon cela peut créer des problèmes

Si on souhaite passer plusieurs arguments à la fonction `serve`

- si ces arguments sont de même type, on peut passer en argument un tableau dynamique les contenant,
- sinon, on peut passer en argument un pointeur sur une structure les contenant.

On peut également utiliser des variables globales qui sont partagées.

Si le programme principal termine avant des threads qu'il a lancés, ces derniers sont détruits
$\rightarrow$ il faut donc que le processus principal attende la fin d'exécution de ses threads. Il peut alors libérer la mémoire allouée sur le tas.

On utilise pour cela la fonction

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

- la fonction est bloquante
- `thread` : thread attendu
- `retval` : si non `NULL`, permet de récupérer la valeur de retour du thread

Concurrence

retour d'un thread

```
int compt = 0;
int *tsock[5];
pthread_t tpthread[5];
while(compt < 5){
  //*** on enregistre les pointeurs sur les descripteurs de socket client ***
  tsock[compt] = malloc(sizeof(int));
  *(tsock[compt]) = accept(sock, NULL, NULL);

  //*** on enregistre les threads ***
  if (*(tsock[compt]) >= 0) {
    if (pthread_create(&(tpthread[compt]), NULL, serve, tsock[compt]) == -1) {
      perror("pthread_create");
      continue;
    }
    compt++;
  }
}

//*** le processus principal attend les 5 threads et libère ***
//*** chaque pointeur de descripteur ***
for(int i=0; i<5; i++){
  pthread_join(tpthread[i], NULL);
  close(*tsock[i]);  // fermeture socket client
  free(tsock[i]);
}
```

Pour terminer un thread, on peut utiliser la fonction

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

- `retval` est la valeur retournée par le thread.
- ne libère pas les ressources partagées (descripteurs, verrous...)
  $\Rightarrow$ ne pas oublier de fermer la socket client
- équivalent à l'utilisation de `return` avec valeur de retour

Attention : un appel à `exit()` fait terminer le processus ! ! !

Concurrence

retour d'un thread

```
void *serve(void *arg) {
  int sock = *((int *) arg);
  char buf[SIZE_BUF+1];
  memset(buf, 0, sizeof(buf));

  int recu = recv(sock, buf, SIZE_BUF, 0);
  if (recu <= 0) return NULL;

  int *ret = malloc(sizeof(int));

  if(buf[0] >= 'a' && buf[0] < 'p'){
    *ret = 1;
    pthread_exit(ret);
  }
  else{
    *ret = 2;
    return ret;
  }
}
```

Le processus principal attend la terminaison du thread

```
int *val;
pthread_join(thread1, (void **) &val);

if(val)
  printf("valeur de retour : %d\n", *val);
```