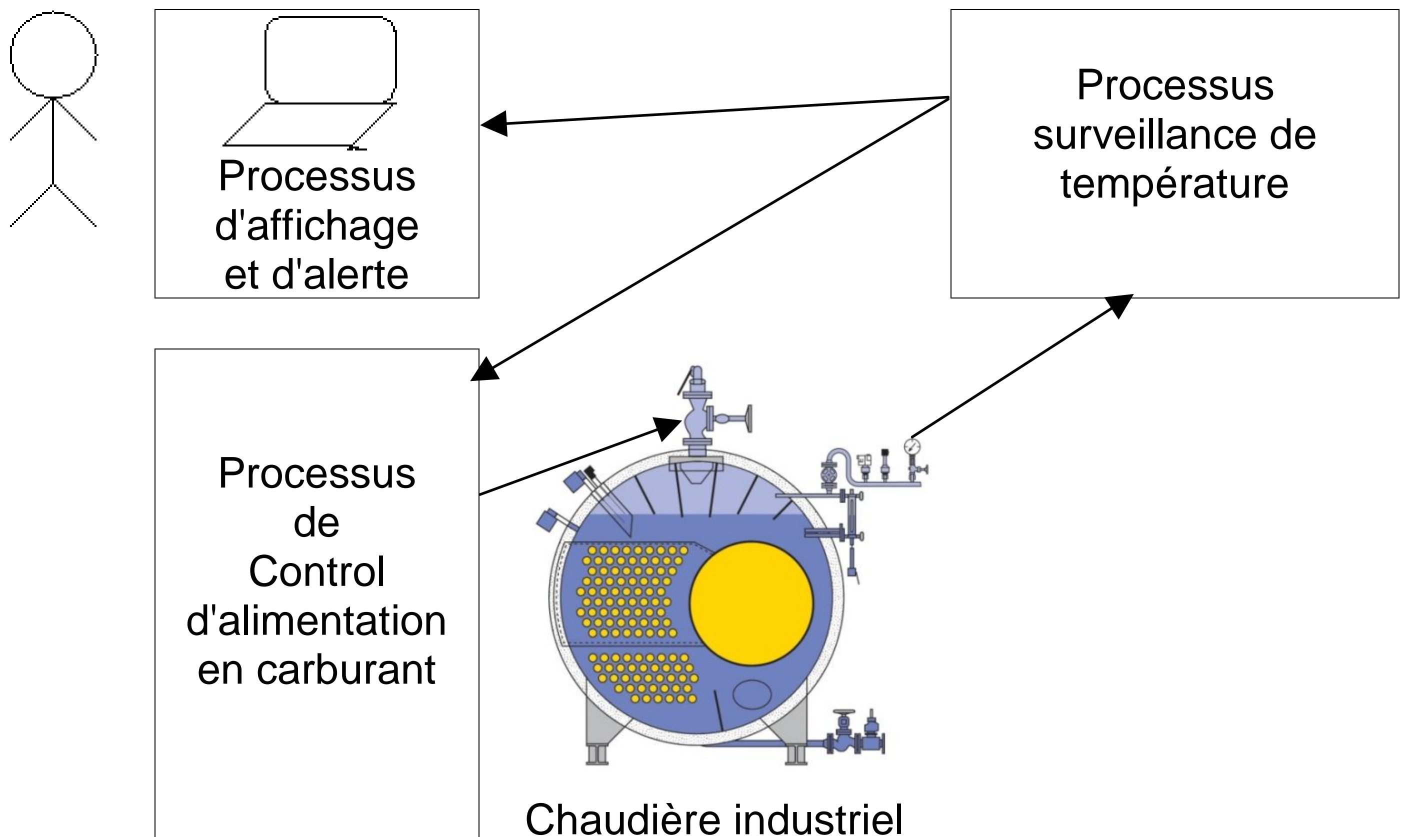
Chapitre 2
Communication interprocessus

2.1/ Introduction.

L'utilisation de processus dans le développement d'une application permet la délégation des tâches. Néanmoins, ces tâches peuvent ne pas être indépendantes et il est dès lors nécessaire que les processus sachent communiquer entre eux. pour se synchroniser ou pour communiquer, de l'information. Il existe de nombreuses façons de communiquer.

Pourquoi interagir avec les processus ?

Exemple : (contrôle d'une chaudière industrielle) :



Nous avons déjà vu un possibilité de communication entre fils par `exit(i)` et père par `WEXITSTATUS(status)`, voir ch1 p 32 et TP3.

Dans ce chapitre, nous allons étudier les différents moyens qu'ont a leur disposition les programmeurs en langage C pour faire communiquer des processus :

- communication par signaux
- communication par tubes
- communication par sockets
- communication par variables et fichiers commun

Il y a des lagunages envolés permettent la communication entre processus comme par exemple : MPI, qui ne feront pas parti de ce cours.

2.2/ Communication par sockets

Qu’est-ce qu’une socket ?

On peut représenter de façon imaginée le mécanisme des sockets sous la forme suivante : une ligne de communication existe au départ ; deux processus veulent communiquer ; ils demandent chacun au système l’attribution d’un appareil (la socket) connecté à la ligne (appel système **socket**) ; puis chacun affecte un nom externe (une adresse) à son appareil (appel system **bind**) pour pouvoir être identifié par son interlocuteur ; chacun peut alors déposer sur son appareil un message qui sera envoyé par le système à son destinataire (apppel **send**) ; il peut également recueillir sur son appareil un message qui lui est adressé (appel system **recv**)

Modes et types de communication

Pour qu'un échange de messages entre processus soit possible, il faut que la destination des messages soit connue. On distingue deux **modes** de communication qui seront référencées respectivement par :

- SOCK_DGRAM (**le mode non connecté**)
- SOCK_STREAM (**le mode connecté**).

Pour chaque mode correspond deux **types** :

1. en mode **connecté** à travers Internet : TCP
2. en mode **connecté** à travers Unix.

3. en mode **non connecté** à travers Internet : UDP
4. en mode **non connecté** à travers Unix.

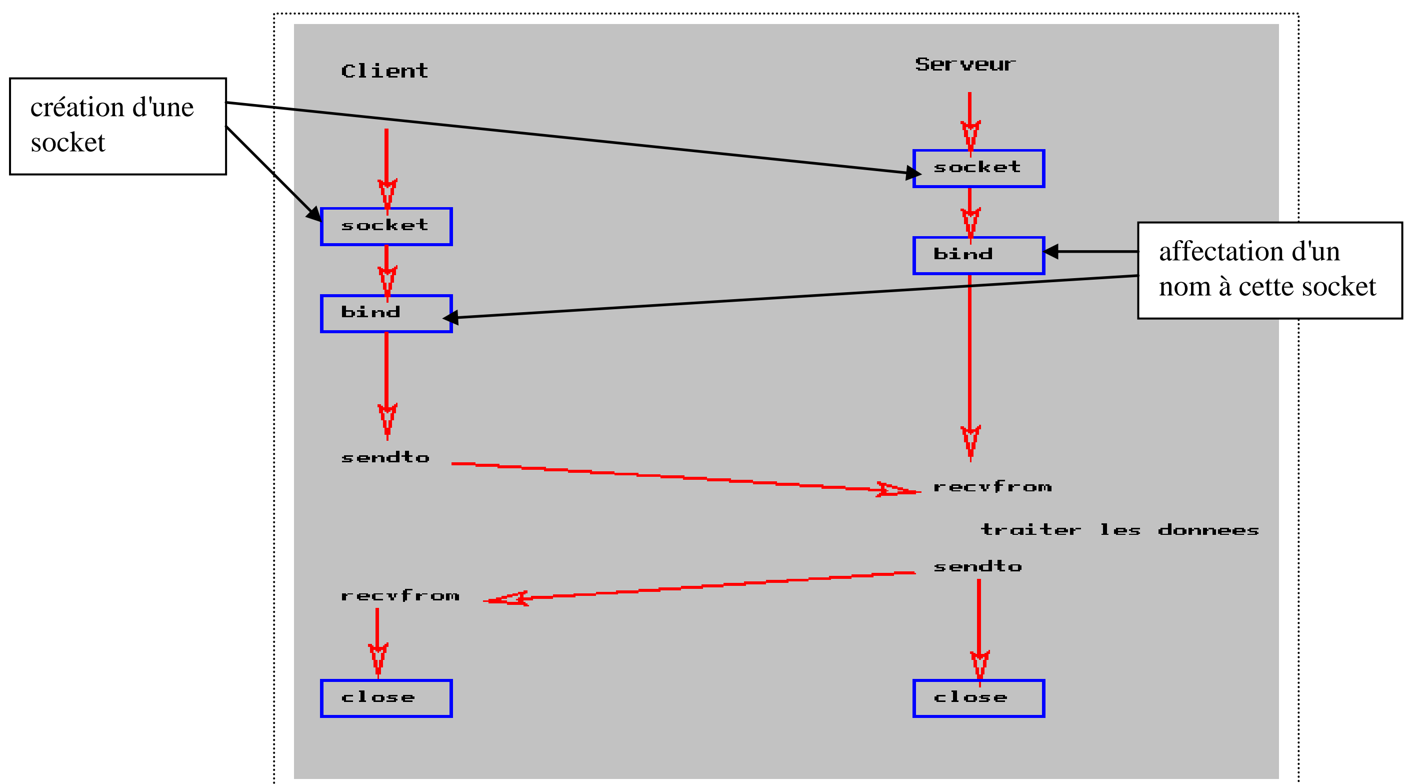
C'est les modes à travers UNIX qui seront etudiés dans ce chapitres

Les modes à travers Internet font partie du cours Reseaux.

2.2.I. Le mode non connecté

A. Schéma général:

La destination est spécifiée chaque fois qu'un message est envoyé. Dans ce mode, la même socket peut être utilisée par un processus pour envoyer successivement un message sur plusieurs sockets différentes. Le nom de l'adresse réceptrice est ajouté à chaque datagramme. Ce type (SOCK_DGRAM) est présumé n'être pas fiable, ne pas préserver l'ordre des messages et éventuellement dupliquer des messages. Le schéma de communication est le suivant :



La création d'une socket s'effectue au moyen de l'appel système socket déclaré dans le fichier sys/socket.h par

```
int socket(int domaine, int type, int protocole);
```

Le retour d'un appel à socket est un descripteur de socket utilisé dans les opérations ultérieures, -1 en cas d'échec.

- Le **`domaine`** détermine les règles selon lesquelles un nom de socket (son "adresse") doit être formé. On distique deux domaines : UNIX, INTERNET. Il leur correspond les deux valeurs que peut prendre l'argument domaine: **AF_UNIX, AF_INET**

- Le **`type`** peut être
  - SOCK_DGRAM pour une communication en mode non connecté et par data grammes,
  - SOCK_STREAM pour une communication en mode connecté et par flot.
- Le paramètre **`protocole`** prend le plus souvent la valeur 0

Exemple : `desc = socket(AF_UNIX, SOCK_DGRAM, 0)`

Rappelons que ici, nous ne considérons que le domaine UNIX et donc domaine a pour valeur AF_UNIX. Dans le domaine UNIX, un nom de socket n'est autre qu'une référence de fichier. Plus précisément, il est formé selon le type `struct sockadd_un` défini dans le fichier `sys/un.h`

```
struct sockaddr_un {unsigned short sun_family;     /* AF _UNIX */
char sun_path[108]; };          /* reference de "fichier" */
```

Exemple :

```
struct sockaddr_un agence = {AF_UNIX, "Opera"};
```

Nommage d'une socket : `bind`

Pour que une socket soit accessible aux autres processus, il faut lui donner un nom externe. L'affectation d'un nom à une socket pour laquelle on dispose déjà d'un descripteur s'effectue au moyen de l'appel système `bind` déclaré dans le fichier `sys/socket.h`

par

```
int bind(int descripteur, struct sockaddr *nom, int longueur);
```

- **`nom`** est un pointeur sur une structure `sockaddr` où est mémorisé le nom à affecter à la socket désignée par son descripteur. On notera que le type structure `sockaddr` n'est qu'un type générique: lors d'un appel effectif à `bind`, il est remplacé par le type structure spécifique correspondant au domaine particulier de la socket. Ainsi dans le domaine UNIX, nom sera un pointeur sur une structure de type `sockaddr_un`.
- Le troisième argument est la longueur du nom.
- Le retour d'un appel à socket est 0 en cas de succès, -1 en cas d'échec.

Exemple d'utilisation :

```
bind(desc, (struct sockaddr *) &agence, sizeof(agence))
```

Dans le domaine UNIX, le nom affecté à une socket apparaît comme un élément de l'arborescence des fichiers (une socket se reconnaît par la lettre s au début de l'affichage de la commande ls -a). Lorsque le propriétaire de la socket a fini de l'utiliser, il est bon qu'il supprime son lien par appel système `unlink` ou par la commande `rm` (voire exercice 1 ci-dessous) ; à défaut, un autre programme ne pourra en réutiliser le nom. De même si un programme qui nomme une socket doit être employé plusieurs fois, il doit supprimer le lien de la socket avant de la nommer.

TP4. Première partie

Exercice 1

I. Récupérer à partir de `corton --> /tmp/info2#/tp4` le programme `creesock.c`, l'analyser, le compiler et l'exécuter. On remarque que ce programme va créer une socket, puis lui donne le nom ("l'adresse") Opera.

II. Vérifier (`ls -a`) l'existence du ficher "Opera", quel il est le type de ce fichier ?

III. Exécuter à nouveau ce programme, puis analyser le message d'erreur, proposer une solution. Indication : Utiliser la fonction `unlink`

B. Les primitives de communication (en mode non connecté) :

Les primitives d'envoie et de réception en mode non connecté

```
int sendto(int desc, char *msg, int lg, int option,
                        struct sockaddr_Un *dest, int lgdest);
```

- Effet : dépose le message `msg` et de longueur `Ig` sur la socket locale de descripteur `desc` en vue de son émission vers la socket de nom externe `*dest`, nom dont la longueur est `Igdest`.
- La seule valeur possible pour option en mode non connecté est o.
- Le retour de `sendto` est le nombre de caractères envoyés, -1 en cas d'échec.

Exemple d'utilisation de `sendto`:

```
sendto(desc,msg,sizeof(msg),0,(struct sockaddr *)
&agence,sizeof(agence))
int recvfrom(int desc, char *msg, int lg, int option,
                      struct  sockadd_in *exp, int *lgexp);
```

- Effet : place dans le buffer d'adresse `msg` et de longueur `lg` un message (Le. un datagramme) reçu sur la socket locale de descripteur

desc en provenance d'une socket dont le nom externe est placé dans le buffer d'adresse exp ; à l'appel la taille du buffer est chargée à l'adresse Igexp; au retour la taille réelle du nom de la socket expéditrice est chargée à cette même adresse.

- Le retour de recvfrom est le nombre de caractères reçus, -1 en cas d'échec.
- Si le paramètre exp a pour valeur NULL, l'adresse de l'expéditeur n'est pas récupérée.
- recvfrom est **un appel bloquant**.

Exemple d'utilisation de recvfrom:

```
lgclient = sizeof(client);
recvfrom(desc, msg, LGMESS, 0, (struct sockaddr *) &client,
&lglient);
```

TP4. suit

(ne pas oublier d'envoyer un **CR** à nakech@free.fr, sujet tp4ex1&2)

Exercice 2

I. On vous fournit sur corton --> /tmp/info2#/tp4 (voir également l'annexe) deux squelettes client_unix_dgram.c, serveur_unix_dgram.c de code **client** et **serveur** que vous devez recopiez et compléter à partir des éléments du cours décrits plus haut.

II. Lancer le serveur en background (serveur_unix_dgram &) puis le client.

Remarque : Pour tuer le serveur, soit revenir en foreground (fg) et faire <Ctrl-C>, soit tuer le processus par kill après l'avoir repéré son pid par "ps ax".

III. Modifier les deux programmes précédents pour que le serveur envoie une confirmation (accusé de reception) au client.

Annexe

```
/*    programme creesock.c  */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>
struct sockaddr_un adresse = {AF_UNIX, "Opera"};
main ()
{int desc;               /* pour le descripteur */
 int lng;      /* longueur de l'adresse */
 if ((desc = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){ perror (" socket");
exit (1); }
 adresse.sun_family = AF_UNIX;
 lng = sizeof(adresse);
 if(bind(desc, (struct sockaddr *)  &adresse, lng) == -1){
perror("bind"); exit (1); }
 close(desc);}
```

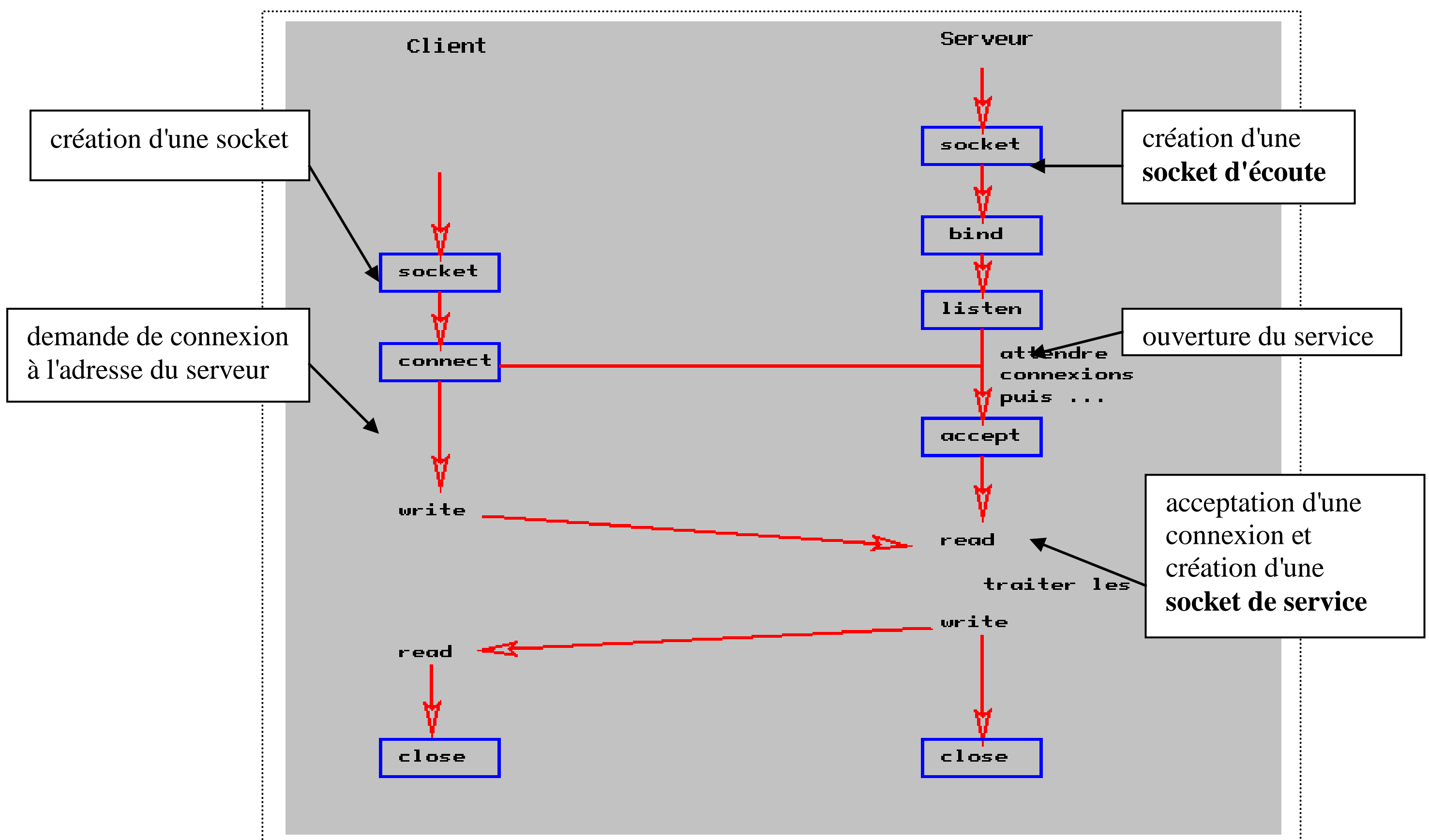
```
/*    client_unix_dgram.c : */
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>
#include <unistd.h>      /* declaration de unlink */
struct sockaddr_un agence = {AF_UNIX,"Opera"};
char msg[] = "Reservez moi main 16 places au Volcan";
int main(){
 int desc;     /* Descripteur de socket */
 int lg;  /* Longueur de l'adresse */
/* Creation d'une socket */
 if ((desc = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
    { perror (" socket"); exit (1);}
 lg = sizeof(agence);
 /* on envoie le message a l'agence Opera */
 if (sendto(desc,msg,sizeof(msg),0, (struct sockaddr *) ..., lg) == -1)
     {perror("sendto"); exit (1);}
}
```

```
/*   serveur_unix_dgram.c  */
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>
#include <unistd.h>
/* declaration de unlink */
struct sockaddr_un agence = {AF_UNIX,"Opera"};
#define LGMESS 80             /* longueur d'un message attendu */
char msg[LGMESS];             /* pour placer le message recu */
int main() {
 int desc;                    /* pour le descripteur */
 int lg;                 /* longueur de l'adresse */
 int lgrecu;
                         /* creation d'une socket */
 if ((desc = socket(..., 0)) == -1){
     perror (" socket"); exit (1);
     }
/* on nomme la socket Opera */
 unlink(...);
 lg = sizeof(agence);
 if (bind(desc,...    ,lg) == -1) {
     perror ("bind"); exit (1);
     }
 lgrecu = -1;                 /* on attend un message */
 while(lgrecu == -1) {             /* boucle d'attente */
     lgrecu = recvfrom(desc,msg,LGMESS,0,... , ...);
     }
 printf("Message recu du client %s \n%s\n", client.sun_path, msg);
close(desc);
 exit(0);}
```

2.2.II. Le mode connecté

A. Schéma général:

Les deux processus s'accordent pour que tout message déposé sur la socket de l'un soit récupéré sur la socket de l'autre. Tout se passe comme si un **circuit virtuel était établi entre les deux sockets**. Pendant tout le temps où les deux sockets sont connectés, la destination des messages est donc parfaitement déterminée. Ce type (SOCK_STREAM) est présumé fiable, respecter l'ordre des messages et ne pas les dupliquer. Le schéma de communication est le suivant :



Pour établir une communication en mode connecté, chacun des deux processus crée une socket de type `SOCK_STREAM`.

```
desc = socket(AF_UNIX, SOCK_STREAM, 0);
```

La communication est précédée par l'établissement d'une connexion dont ces deux sockets constituent les extrémités. Dans cette phase préliminaire, le rôle des deux processus est dissymétrique: l'un d'entre eux (appelé client) demande à l'autre (appelé serveur) s'il accepte de communiquer. Dans la phase de communication proprement dite, le rôle des processus redevient symétrique. Lorsque la connexion s'est établie, le serveur et le client peuvent communiquer par les appels système `write` et `read`, le **client** sur sa **socket**, le **serveur** sur la **socket de service**.

Description des primitives en mode connecté :

```
int listen(int desc, int nb);
```

- Cet appel a deux effets :
  Il signale au système local que le serveur est prêt à recevoir des demandes de connexion sur la socket d'écoute de descripteur `desc` ;
- Il crée une file d'attente de taille `nb` pour mémoriser les demandes de connexion reçues.

Le retour de `listen` est 0 en cas de succès, -1 en cas d'échec.
Exemple d'utilisation : `listen(desc, 5);`

```
int accept(int desc, struct sockaddr *nom, int *lg);
```

- Si la file d'attente associée à la socket d'écoute `desc` n'est pas vide, les opérations suivantes sont effectuées :
  - création d'une socket de service ;
  - connexion de cette socket de service à la socket pendante la plus ancienne ;
  - affectation du nom de la socket pendante à l'adresse `nom` et de sa longueur à `*lg` ; à l'appel, `*lg` doit avoir pour valeur la taille du buffer fourni à l'adresse `nom` ; rappelons que `sockaddr` est un type structure générique et doit être remplacé dans un appel effectif par le type convenant au domaine (`sockadd_un` pour le domaine UNIX).
- Sinon, si la socket d'écoute est en mode bloquant (mode par défaut), le processus est mis en sommeil.
- Le retour de accept est le descripteur de la socket de service et -1 en cas d'échec.

Exemple d'utilisation :

```
lgclient = sizeof(client);
accept(desc, (struct sockaddr *) &client, &lgclient)) ;
```

```
int connect(int desc, struct sockaddr *nom, int lg);
```

Effet: tentative de connexion de la socket locale `desc` à la socket dont le nom est à l'adresse `nom` sur une longueur `lg`.

Le retour de `connect` est 0 en cas de succès, -1 en cas d'échec. Causes d'échec:
- Les paramètres ne sont pas corrects.
- Les sockets client et serveur ne sont pas compatibles (domaine, type, protocole) .
- Le serveur n'est pas prêt à recevoir des demandes de connexion.

Exemple : `connect (desc, (struct sockaddr *) &agence, sizeof(agence))`

TP4 Exercice 3

I On vous demande maintenant de modifier les programmes précédents (premières versions exercice II) du serveur et du client de telle sorte que la communication entre le nouveau serveur et le nouveau client se fasse en mode connecté, c'est-à-dire en établissant un canal de communication préliminaire entre le client et le serveur. On nommera le serveur `serveur_unix_stream.c` et le client `client_unix_stream.c`

II Idem que l'exercice 2.III, modifier les programmes précédents pour que le serveur envie une confirmation au client.

Exercice 4

Modifier le résultat de l'exercice 3.II pour réaliser un serveur qui reçoit (en mode connecté) un message du cient1 puis l'envoie à un autre client2.

Exercice 5

Un serveur et n clients dialoguent ensemble. Un client (le numéro 1) envoie un message `m` au serveur, le serveur diffuse ce message `m` aux autres clients. Modifier les résultats de l'exercice 4 pour réaliser cette discussion à plusieurs.