# Le serveur multiprocessus

Fonctionnement : un serveur multiprocessus crée un processus fils à chaque nouvelle connexion, ce processus fils s'occupera de la communication avec le client.

Reprenons l'exemple de notre architecture client/serveur simple. Dans ce style d'architecture, un client à la fois pouvait se connecter, si un autre client voulait se connecter il lui fallait attendre la déconnexion du premier. Dans l'architecture multiprocessus, chaque processus fils s'occupe d'un client ce qui permet la connexion simultanée de plusieurs clients.

Adaptons notre serveur qui mélange les chaines de caractères :

```
int main(){

    signal(SIGCHLD, SIG_IGN);

    int nbLus;
    char message[BUFFER_SIZE];

    int secoute = socket(AF_INET, SOCK_STREAM, 0);
    if (secoute == -1){
        perror("Erreur lors de la création de la socket");
    }
    int sservice;

    struct sockaddr_in saddr = {0};
    struct sockaddr_in caddr = {0};
    unsigned int caddlren;

    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(PORT);
    saddr.sin_addr.s_addr = inet_addr(ADRESSE);

    if(bind(secoute, (struct sockaddr *) &saddr, sizeof(saddr)) == -1){
        perror("Erreur lors du bind");
        return -1;
    }

    if(listen(secoute, 5) == -1){
        perror("Erreur lors du listen");
        return -1;
    }
    while(1){
        caddlren = sizeof(caddr);
        printf("En attente de connexion\n");
        sservice = accept(secoute, (struct sockaddr *) &caddr, &caddlren);
        if(sservice == -1){
            perror("Erreur lors de la création de sservice");
        }
```

```
        if(fork() == 0){
            printf("Client connecté\n");
            close(secoute);
            while(1){

                nbLus = read(sservice, message, BUFFER_SIZE-1);
                if(nbLus == -1){
                    perror("Erreur lors du read");
                    return 1;
                }
                if(nbLus == 0){
                    break;
                }

                message[nbLus] = '\0';

                shakeMessage(message, nbLus-1);
                write(1, message, nbLus);
                write(sservice, message, nbLus);

            }
            shutdown(sservice, SHUT_RDWR);
            close(sservice);
            exit(0);
        }

        close(sservice);
    }

    shutdown(secoute, SHUT_RDWR);
    close(secoute);
}
```

Comment il fonctionne :

1. **Mise en place d'un gestionnaire de signaux**

   La première chose à faire est de mettre en place un gestionnaire de signaux pour éviter l'accumulation de processus zombies.

   Pour mettre en place un gestionnaire de signaux, on utilise la primitive `signal()` qui prend 2 paramètre :

   - Le numero ou le nom du signal.
   - La fonction gestionnaire

   Dans l'argument de la fonction gestionnaire, on peut également mettre les valeurs suivante :

   - SIG_IGN : ignore le signal
   - SIG_DFL : restaure le signal par défaut

   Voici les différents signaux :

| Nom | N° | Par défaut | Commentaire |
|---|---|---|---|
| SIGHUP | 1 | terminaison | déconnexion du terminal gérant |
| SIGINT | 2 | terminaison | interruption par l'utilisateur (touche ) |
| SIGKILL | 9 | terminaison | terminaison immédiate, ne peut pas être capturé ou ignoré |
| SIGSEGV | 11 | terminaison | accès à l'adresse mémoire invalide (segmentation fault) |
| SIGPIPE | 13 | terminaison | écriture dans un tube (pipe) sans lecteur |
| SIGALRM | 14 | terminaison | temporisation mise par alarm() écoulée |
| SIGTERM | 15 | terminaison | terminaison immédiate |
| SIGUSR1 | 10 | terminaison | signal utilisateur |
| SIGUSR2 | 12 | terminaison | signal utilisateur |
| SIGCHLD | 17 | ignoré | processus fils terminé ou arrêté |
| SIGCONT | 18 | continuation | continuer l'exécution si arrêté |
| SIGSTOP | 19 | arrêt | arrêt immédiat, ne peut pas être capturé ou ignoré |
| SIGTSTP | 20 | arrêt | arrêt par l'utilisateur (touche ) |

2. **On crée la socket d'écoute et les structures d'adresse**

   On crée une socket *secoute* sur laquelle les clients vont faire les demandes et une socket *sservice* sur laquelle on communiquera avec les clients La structure d'adresse *saddr* correspond aux paramètres réseaux du serveur via lesquels il sera accessible, et *caddr* va stocker les paramètres du client.

3. **On associe la structure d'adresse à la socket**

4. **On ouvre l'écoute**

5. **On accepte les connexions**

   On place le decripteur dans sservice, qui nous servira à communiquer.

6. **On crée un fils**

   A chaque fois qu'une connexion est acceptée, on va créer un processus fils, avec `fork()`.

   *Rappel :*

   - *si fork() == 0 on est dans le processus fils.*
   - *si fork() > 0 on est dans le processus père.*
   - *si fork() == -1 il y a une erreur.*

   On se place dans le processus fils puis

   - on ferme la socket d'écoute : on ne s'en servira pas dans le fils, la communication se fera via sservice.

- On entre dans la boucle de communication : le principe est le même que dans le serveur simple :

  On lit le message envoyé par le client (et on fait les tests de reception) puis on melange le message et on le renvoie.

- quand la connexion est terminée, on peut couper la communication et fermé la socket sservice, puis mettre fin au processus.

```
shutdown(sservice, SHUT_RDWR); //Coupe la communication en
lecture/ecriture
close(sservice); //ferme la socket de service
exit(0); //Termine le processus
```

7. **Dans le processus père**

   Une fois que le fils est lancé, on peut fermer la socket de service (elle est gérée par le fils). Et on réitère la boucle pour attendre une nouvelle connexion.

8. **A la coupure du serveur, on ferme la socket d'écoute**