Processus

Fabrication du fichier

gcc -Wall -o executable.exe fichier.c

Exécution du fichier

./executable.exe

```
int main(int argc, char *argv[]) {
```

La fonction main reçoit les arguments mis en paramètres à l'exécution.

Par exemple :
Avec *./tp1.exe 127.0.1.1 8080*
argc contient le nombre de paramètres (3) et
argv contient un tableau de pointeurs des paramètres saisis :

- « tp1.exe »
- 127.0.1.1
- 8080

Connexion à un serveur (en tant que client)

1. Création socket
2. Connexion au serveur
3. Echanges
    a. Réception
    b. Envoi
4. Fermeture socket

Fabrication d'un serveur

1. Création socket
2. Lier le socket à un port
3. Mise en écoute du socket
4. **Traitement des requêtes**
5. Fermeture socket

Traitement des requêtes (en tant que serveur)

1. Acceptation connexion entrante
2. Réception message (fermeture de sa socket en cas d'échec)
3. Traitement
4. (Envoi de la réponse)
5. Fermeture socket client

Mémoire partagée

1. Un processus créé la mémoire partagée avec une ftok()
2. Les autres l'utilisent avec son id

3. Le dernier l'utilisant la detruit

Fonctions utiles

Fonction	Explications
`perror("message d'erreur");`	Affiche une erreur importante. Passe devant le message par défaut
`fprintf(stderr, "message");`	Affiche le message en précisant son type de sortie (fichier, erreur…) stderr : erreur
`printf("Message %s", params);`	Affiche un message standard stdout
`char *fgets(char *str, int size, FILE *stream);`	Récupère un message saisi str : tableau de caractères size : nombre de caractères lus stream : Pointeur vers le fichier depuis lequel la ligne sera lue (stdin)
`int atoi(const char *str);`	String → int
`int socket(int domain, int type, int protocol);`	Pour créer un socket. Domain : AF_INET pour ipv4 Type : SOCK_STREAM pour TCP Protocol : 0 pour laisser le système choisir en général
`uint16_t htons(uint16_t hostshort);`	Traduit un port entier au format de l'ordinnateur
`int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`	Lie une @ip et un port à un socket local sockfd : socket addr : adresse retourne : -1 en cas d'erreur
`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`	Lie un socket client à un socket serveur sockfd : descripteur du socket addr : adresse a laquelle se connecter retourne : -1 en cas d'erreur
`close(server_socket);`	Pour fermer la socket
`listen(server_socket, 5)`	La socket écoute au maximum 5 connexions en attente.
`inet_pton(AF_INET, server_ip, &server_addr.sin_addr);`	Convertit une @ip string en @ip binaire
`send(client_socket, message, strlen(message), 0);`	Envoie des données à travers un socket
`recv(client_socket, response, sizeof(response), 0);`	Reçoit une réponse d'un socket. On doit indiquer la taille max de données qu'on veut recevoir. `'\0'` si rien n'est reçu.
`accept(server_socket, (struct sockaddr *)&client_addr, &client_len);`	Accepte une connexion entrante
`inet_ntop(AF_INET, &client_addr.sin_addr, client_ip,`	Convertit une @ip binaire en format réseau stockable en sockaddr_in. Permet d'afficher de manière lisible l'@

`sizeof(client_ip));`	
`ntohs(client_addr.sin_port) ;`	Convertit un port du format réseau au format hôte
`continue;`	Saute la boucle du while. Permet de passer a l'utilisateur suivant

`snprintf((char*)msg_formate,`
`sizeof(msg_formate),`
`"Message a formater :%n",`
`message);`
	Formate un message dans un tampon :
	Le message a formater utilisant les « % » ne dépassera pas la taille indiquée.

`key_t ftok(const char *pathname,`
`int proj_id);`
`ftok("/chemin/vers/monfichier.txt", 'A');`
	« File to Key »
	Génère une clé permettant d'accéder et de partager des ressources partagées.
	pathname : chemin vers le fichier contenant le processus ayant créé le segment.
	proj_id : entier pour identifier la clé générée par un fichier.

`int shmget(key_t key,`
`size_t size,`
`int shmflg);`
`int shmid = shmget(key, 1024, 0666 | IPC_CREAT);`
	Permet d'accéder ou de créer un segment de mémoire partagée.
	key : clé de la mémoire partagée générée avec ftok()
	size : taille du segment de mémoire partagée en octets.
	shmflg : ensemble de droits et d'options (IPC_CREAT, SHM_R, SHM_W)

`void *shmat(int shmid,`
`const void *shmaddr,`
`int shmflg);`
`void *shm_ptr = shmat(shmid, NULL, 0);`
	Permet d'attacher un segment de mémoire à au processus en cours
	Shmaddr : Adresse de base à laquelle attacher le segment (Généralement Null pour laisser le système choisir)
	Shmflg : ensemble de droits et d'options (SHM_R ou SHM_W)
	Retourne : adresse a laquelle la mémoire s'est attachée

`int shmdt(const void *shmaddr);`	Détache un segment de mémoire partagée d'une adresse

`int shmctl(int shmid,`
`int cmd,`
`struct shmid_ds *buf);`
`shmctl(shmid, IPC_RMID, NULL);`
	Effectuer des opérations de contrôle sur un segment de mémoire partagée telle que la suppression.
	Cmd : commande de contrôle :
	• IPC_STAT récupère les infos de la MP
	• IPC_SET copie les donnees de buf dans la MP
	• IPC_RMID supprime le segment
	Buf : Buffer qui stocke les données s'il y en a un (mettre null si on récupère rien).

Structures / types de données

```
struct sockaddr_in {// Structure surtout utilisee pour IPV4
    short sin_family;        // Famille d'adresse (AF_INET)
    unsigned short sin_port; // Numéro de port (htons(server_port))
    struct in_addr sin_addr; // Adresse IP : (INADDR_ANY)
};

struct sockaddr {// Structure plus generique
    unsigned short sa_family; // Famille d'adresse (AF_INET, AF_INET6, etc.)
    char sa_data[14];         // Adresse IP générique
};

socklen_t client_len = sizeof(client_addr);

char client_ip[INET_ADDRSTRLEN]; //INET_ADDRSTRLEN = longueur max d'une @ IPv4
```

Client

```c
if (argc != 3) {
    fprintf(stderr, "Utilisation : %s <adresse IP du serveur> <port>\n", argv[0]);
    exit(1);
}

const char *server_ip = argv[1];
int server_port = atoi(argv[2]);

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

struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(server_port);
if (inet_pton(AF_INET, server_ip, &server_addr.sin_addr) <= 0) {
    perror("Erreur lors de la conversion de l'adresse IP");
    close(client_socket);
    exit(1);
}

if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
    perror("Erreur lors de la connexion au serveur");
    close(client_socket);
    exit(1);
}

printf("Connecté au serveur %s:%d\n", server_ip, server_port);

// Envoi d'un message au serveur
char message[100];
printf("Entrez un message à envoyer au serveur : ");
fgets(message, sizeof(message), stdin);

if (send(client_socket, message, strlen(message), 0) == -1) {
    perror("Erreur lors de l'envoi du message au serveur");
    close(client_socket);
    exit(1);
}

// Réception de la réponse du serveur
char response[100];
int bytes_received = recv(client_socket, response, sizeof(response), 0);
if (bytes_received == -1) {
    perror("Erreur lors de la réception de la réponse du serveur");
```

```
    close(client_socket);
    exit(1);
}

response[bytes_received] = '\0';
printf("Réponse du serveur : %s\n", response);

// Fermez la socket client lorsque vous avez terminé.
close(client_socket);
```

Serveur

```
if (argc != 2) {
    fprintf(stderr, "Utilisation : %s <port>\n", argv[0]);
    exit(1);
}

int server_port = atoi(argv[1]);

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

struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(server_port);
server_addr.sin_addr.s_addr = INADDR_ANY;

if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
    perror("Erreur lors de la liaison du socket au port");
    close(server_socket);
    exit(1);
}

if (listen(server_socket, 5) == -1) {
    perror("Erreur lors de la mise en écoute du socket");
    close(server_socket);
    exit(1);
}

printf("Serveur en écoute sur le port %d...\n", server_port);

while (1) {
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);

    int client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_len);
```

```
        if (client_socket == -1) {
            perror("Erreur lors de l'acceptation de la connexion entrante");
            continue;  // Passer à la prochaine connexion
        }

        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, sizeof(client_ip));
        printf("Nouvelle connexion acceptée de %s:%d\n", client_ip, ntohs(client_addr.sin_port));

        // Réception du message de l'utilisateur
        char message[100];
        int bytes_received = recv(client_socket, message, sizeof(message), 0);
        if (bytes_received == -1) {
            perror("Erreur lors de la réception du message de l'utilisateur");
            close(client_socket);
            continue;  // Passer à la prochaine connexion
        }

        message[bytes_received] = '\0';
        printf("Message reçu de l'utilisateur : %s\n", message);

        // Traitement du message (multiplication par 2)
        int number = atoi(message);
        int response = number * 2;
        char response_message[100];
        snprintf(response_message, sizeof(response_message), "Réponse : %d\n", response);

        // Envoi de la réponse à l'utilisateur
        if (send(client_socket, response_message, strlen(response_message), 0) == -1) {
            perror("Erreur lors de l'envoi de la réponse à l'utilisateur");
        }

        // Fermez la socket client lorsque vous avez terminé.
        close(client_socket);
    }
```

Serveur avec threads

```
void *handle_client(void *client_socket_ptr) {
    int client_socket = *((int *)client_socket_ptr);
    free(client_socket_ptr);

    char buffer[1024];
    int bytes_received;

    while ((bytes_received = recv(client_socket, buffer, sizeof(buffer), 0) > 0)) {
        buffer[bytes_received] = '\0';
        printf("Message from client %d: %s", client_socket, buffer);
```

```
        // Répondre au client (écho)
        send(client_socket, buffer, bytes_received, 0);
    }

    if (bytes_received == 0) {
        printf("Client %d disconnected.\n", client_socket);
    } else {
        perror("Error receiving data from client");
    }

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

Dans le while :

```
struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);

        int *client_socket_ptr = malloc(sizeof(int));
        *client_socket_ptr = accept(server_socket, (struct sockaddr *)&client_addr, &client_len);
        if (*client_socket_ptr == -1) {
            perror("Error accepting new client connection");
            free(client_socket_ptr);
            continue;
        }

        printf("New client connected from %s:%d\n", inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));

        pthread_t client_thread;
        if (pthread_create(&client_thread, NULL, handle_client, client_socket_ptr) != 0) {
            perror("Error creating client thread");
            close(*client_socket_ptr);
            free(client_socket_ptr);
        } else {
            pthread_detach(client_thread);
        }
```

Création mémoire partagée

```
    key_t key;
    int shmid;
    char *shmaddr;

    // Création de la clé IPC (même que le premier programme)
    key = ftok("exo-1.c", 'A');
    if (key == -1) {
        perror("ftok");
        exit(1);
    }
```

```c
// Accès à la mémoire partagée
shmid = shmget(key, SHM_SIZE, 0);
if (shmid == -1) {
    perror("shmget");
    exit(1);
}

// Attachement à la mémoire partagée en lecture seulement
shmaddr = shmat(shmid, NULL, SHM_RDONLY);
if (shmaddr == (char *)-1) {
    perror("shmat");
    exit(1);
}

// Affichage de la chaîne contenue dans la mémoire partagée
printf("Chaîne dans la mémoire partagée : %s\n", shmaddr);

// Détachement de la mémoire partagée
shmdt(shmaddr);

// Libération de la mémoire partagée
shmctl(shmid, IPC_RMID, NULL);

printf("Mémoire partagée libérée.\n");
```

Utilisation de la mémoire partagée

```c
// COMME AU DESSUS
// Accès à la mémoire partagée
shmid = shmget(key, SHM_SIZE, 0);
if (shmid == -1) {
    perror("shmget");
    exit(1);
}

// Attachement à la mémoire partagée en lecture seulement
shmaddr = shmat(shmid, NULL, SHM_RDONLY);
if (shmaddr == (char *)-1) {
    perror("shmat");
    exit(1);
}

// Affichage de la chaîne contenue dans la mémoire partagée
printf("Chaîne dans la mémoire partagée : %s\n", shmaddr);

// Détachement de la mémoire partagée
shmdt(shmaddr);
```

```c
// Libération de la mémoire partagée
shmctl(shmid, IPC_RMID, NULL);

printf("Mémoire partagée libérée.\n");

return 0;
```