Processus

# Compilation

*gcc -Wall -o executable.exe fichier.c*

# Exécution

*./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
   1. Réception
   2. 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 |
| uint16\_t ntohs(uint16\_t netshort);  ntohs(client\_addr.sin\_port) ; | « Network to Host »  Convertit un port du format réseau au format hôte. Big endian 🡪 little endian  Netshort : Entier court non signé au format réseau (big endian)  Retourne : |
| uint16\_t htons(uint16\_t hostshort); | Fonction inverse |
| 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  Retourne : descripteur de fichier identifiant le socket |
| 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,  sizeof(client\_ip)); | Convertit une @ip binaire en format réseau stockable en sockaddr\_in. Permet d’afficher de manière lisible l’@ |
| continue; | Saute la boucle du while. Permet de passer a l’utilisateur suivant |
| int snprintf(char \*str,               size\_t size,               const char \*format,               ...);  snprintf(buffer, tailleBuffer,           "Message a formater :%n", message); | Formate une chaine de caractères dans un tampon.  Str : Buffer de destination  Size : Nb de caractères max à formater  Format : Chaine à formater (en utilisant les %)  … : Variables à remplir dans la chaine |
| 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

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);

    }

    // 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

// 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);

    // Libération de la mémoire partagée

    shmctl(shmid, IPC\_RMID, NULL);

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

    return 0;