C et communications UNIX - Projet 2025-2026 S1

Lion Rayonnant

14 janvier 2026

Table des matières

Projet de gestion des spectacles

Enoncé

On considère une application client-serveur, comprenant un programme serveur et un programme client. Il n'y aura qu'une seule instance du programme serveur, en revanche il pourra y avoir k instances du programme client (un programme pour chaque client désirant se connecter au serveur).

Cette application permet de réserver des places pour un ensemble de spectacles.

Des clients émettent deux types de requêtes à destination du serveur :

Requête de consultation : permettant de consulter le nombre de places restantes pour un spectacle donné.

Requête de réservation : permettant de réserver n places pour un spectacle donné.

Le serveur de gestion de places de spectacles est composé de deux parties :

Partie Consultation : cette partie prend en compte les requêtes de consultation. Pour chacune d'elle, cette fonctionnalité renvoie le nombre de places disponibles pour le spectacle spécifié dans la requête.

Partie Réservation : cette partie prend en compte les requêtes de réservation. Pour chacune d'elle, cette fonctionnalité effectue la réservation si cela est possible (suffisamment de places disponibles pour la date et le spectacle indiqué). Dans le cas où la réservation a pu être faite, un acquittement de réservation est renvoyé au client et sinon un message d'erreur.

Les informations concernant les spectacles sont stockées dans une table en mémoire centrale du côté du serveur, qui est accessible par les deux parties du serveur (parties Consultation et Réservation). Une entrée de la table concerne un spectacle et donne les informations suivantes : Intitulé du spectacle, nombre de places restantes. Chaque spectacle est joué une seule fois.

Question 1

On vous demande de mettre en place une première version de cette application client-serveur en utilisant les tubes comme outil de communication. Vous considérez un seul client et un serveur.

1. Mettez en place la structure de votre application : processus, nombre de tubes nécessaires, maintien du tableau des réservations en mémoire, structure des données échangées via les tubes. Expliquez vos choix. Pour simplifier, vous pouvez faire afficher chez le client un

menu avec les spectacles associés chacun à un numéro. Le client transmet alors le numéro du spectacle et non son intitulé.
2. Codez votre solution en prenant soin de documenter votre code.
3. Réfléchissez aux difficultés que soulève le fait d'introduire plusieurs clients.

Réponse

Pour cette question il m'a fallu réfléchir à la structure même du projet. Quelle architecture pour le serveur ? Quelle interface pour le client ? J'ai commencé par le serveur car je trouvais plus simple de construire le client en conséquences plutôt que l'inverse. Je vais donc commencer par détailler le code du serveur.

Serveur

Tout d'abord on importe les dépendances nécessaires au bon fonctionnement du programme.

```c
// Import des dépendances
#include <stdio.h> // Pour les E/S (printf ici notamment)
#include <fcntl.h> // Pour utiliser O_RDONLY, O_WRONLY ...
#include <unistd.h> // Pour effectuer des read() et write(). API POSIX
#include <stdlib.h> // Pour exit()
#include <sys/types.h> // Pour les types, notamment pid_t
#include <sys/stat.h> // Pour la création des tubes avec mkfifo
#include <stdbool.h> // Pour utiliser les booléens (dans la réponse au client)
#include <sys/wait.h> // Pour que le père puisse attendre ses fils
#include <sys/shm.h> // Pour la gestion de la mémoire partagée
#include <semaphore.h> // Pour utiliser les sémaphores (sem.h pour SystemV)
#include <string.h> // Pour memcpy
```

A partir de ce moment là, on vois que j'ai utilisé `sys/shm.h` pour la gestion de la mémoire partagée. Cela n'est pas dans le cours mais cela fait partie des choses que j'ai dû étudier via Internet et un livre pour compléter mon code. Sans cela, impossible de partager la mémoire entre le père et les fils que nous allons voir plus tard.

Ensuite on définis quelques constantes.

```c
// Clefs statiques pour la mémoire partagée et les sémaphores
#define SHM_KEY 1234
#define SEM_KEY 5678

// Nombre maximal de spectacles
#define MAX_SPECTACLE 10

// Fichier de stockage des données de spectacles
#define DATA_FILE "spectacles.dat"
```

On pourra améliorer plus tard la définition des clefs en passant de clefs statiques à des clefs dynamiques grâce à `ftok()`.

J'ai également décidé de mettre une limite sur le nombre de spectacles, cela permets de s'assurer que le client ne demande pas un spectacle inexistant.

Enfin, je définis le fichier où stocker les données comme étant `"spectacles.dat"` qui doit être présent dans le répertoire courant du binaire.

```c
// Définition de la structure pour les spectacles
```

```c
typedef struct {
    int id;
    char nom[50];
    char date[11];
    int places;
} Spectacle;
```

Ici on définis une structure dédiée aux spectacles. On y retrouve les champs demandés, à savoir : le nom, la date et le nombre de places. Nous avons également ajouté un id pour faciliter la gestion. La structure se nomme donc `"Spectacle"`.

```c
// Création de la table des spectacles
Spectacle table_spectacles[MAX_SPECTACLE] = {
    {1, "Génération Céline - 4 voix pour une légende", "2027-01-15", 70},
    {2, "Claudio Capéo en concert", "2027-01-23", 80},
    {3, "The Music of Square Enix - Memories & Melodies", "2027-02-06", 90},
    {4, "Bonhomme (Jarry) - Seul en scène", "2027-02-26", 60},
    {5, "Marine (Delplace)", "2027-03-07", 55},
    {6, "Flora Fishbach en concert", "2027-03-12", 75},
    {7, "THÉA en concert", "2027-03-19", 50},
    {8, "Concert symphonique Saison Paris", "2027-04-05", 65},
    {9, "Spectacle théâtre à la Comédie Française", "2027-04-20", 40},
    {10, "Festival musique émergente", "2027-05-10", 30}
};
```

Il faut créer une table `table_spectacles` qui servira plus tard, dans le cas où le fichier de données `"spectacles.dat"` n'existe pas. Cela permets aussi de changer les spectacles en supprimant le fichier actuel et en ajoutant les valeurs souhaitées dans cette table.

```c
sem_t *mutex;
```

Le sémaphore (MUTEX) est déclaré en global afin que tous les processus créés par le père puissent y avoir accès.

```c
int shmid = -1;
Spectacle *shared_table = NULL;
```

Idem ici j'initialise la mémoire partagée en global pour que tous les processus (père et fils) puissent y accéder. Je définis shmid avec la valeur par défaut `-1`. Je créé un pointeur vers `shared_table` à partir de la structure `Spectacle`. La valeur par défaut est `NULL`.

```c
/* Fonction de sauvegarde des spectacles dans le fichier */
void save_spectacles() {
    int fd = open(DATA_FILE, O_WRONLY | O_TRUNC | O_CREAT, 0666);
    // O_TRUNC évite d'écrire à la suite, cela réécris le fichier ;
    // O_CREAT créé le fichier si inexistant
    write(fd, shared_table, sizeof(table_spectacles));
    close(fd);
}
```

Ici on peu voir la fonction pour sauvegarder les modifications concernant les spectacles. Cette fonction va ouvrir le fichier défini à la constante `DATA_FILE` puis y écrire le contenu de la table partagée entre les processus. Les paramètres utilisés sont précisés en commentaires.

```c
int main() {
    // Entiers pour la consultation
    int request_view, response_view;
```

```
    // Entiers pour la réservation (requête)
    int request_reservation, request_reservation_nb;
    // Booléen pour la réponse à la réservation
    bool response_reservation;
```

On entre désormais dans la fonction principale. On commence par déclarer des variables. Elles serviront respectivement à :

1. `request_view, response_view` : entiers pour recevoir l'id du spectacle demandé et renvoyer le nombre de places (Consultation).
2. `request_reservation, request_reservation nb` : entiers pour recevoir le numéro du spectacle demandé et le nombre de places (Réservation)
3. `response_reservation` : Booléen pour la réponse de réservation (Confirmation)

```
// Supprime le sémaphore s'il existe déjà (pour éviter les erreurs)
   sem_unlink("/semaphore");
   // Ouverture du sémaphore (nommé)
   mutex = sem_open("/semaphore", O_CREAT, 0666, 1);
```

Ici il est question de la création du sémaphore. En premier lieu on fait un `sem_unlink` au cas où il serait déjà ouvert. Ensuite on ouvre le sémaphore nommé `"/semaphore"` et on enregistre la valeur de retour dans la variable `mutex`.

```
// Création d'un segment de mémoire partagée
shmid = shmget(SHM_KEY, sizeof(table_spectacles), IPC_CREAT | 0666);
// Attachement à la mémoire partagée via un cast
shared_table = (Spectacle *)shmat(shmid, NULL, 0);
```

Pour la partie mémoire partagée, on commence par créer un segment de mémoire partagée. Ensuite on y associe `shared_table`. On utilise un *cast* pour pouvoir accéder aux données comme un tableau de structure `Spectacle`.

```
// Ouverture des données dans le fichier de data
    int fd = open("spectacles.dat", O_RDONLY); //lecture seule

    // Si le fichier n'existe pas ou est vide, on copie les données d'exemple
    if (fd == -1) {
        memcpy(shared_table, table_spectacles, sizeof(table_spectacles));
        // Puis on sauvegarde dans le fichier
        save_spectacles();
    } else {
        // Sinon, on charge depuis le fichier vers la mémoire partagée
        read(fd, shared_table, sizeof(table_spectacles));
        close(fd);
    }
```

Cette partie du code permets de s'assurer que le fichier cible est présent et que la mémoire partagée en est remplie. Dans le cas `fd == -1`, on copie les données fournies en dur plus haut dans la mémoire partagée puis on utilise `save_spectacles()` qui va écrire dans le fichier. Sinon on charge le fichier vers la mémoire partagée.

```
// Création des tubes
mkfifo("tub1_view", 0666);
mkfifo("tub2_view", 0666);
mkfifo("tub1_reservation", 0666);
mkfifo("tub2_reservation", 0666);
mkfifo("tub3_reservation", 0666);
```

Pour la création des tubes, nous avons 5 tubes. Les deux premiers servent à échanger des informations pour la partie consultation. Il faut un tube pour que le client puisse communiquer le numéro du spectacle à consulter et un autre tube pour que le serveur puisse donner le nombre de places disponibles.

```
printf("[INFO] Serveur UP\n");
```

Ce message permets d'indiquer que la partie père du serveur est prête.

```
// Consultation

    if (fork() == 0) { // processus fils -> consultation

        int tub1 = open("tub1_view", O_RDWR);
        int tub2 = open("tub2_view", O_RDWR);
```

Le serveur comprends le père et deux fils. Chaque fils gère une fonctionnalité. Le premier est chargé de la partie consultation et le seconde de la partie réservation. Nous voyons ci-dessous le `fork()` permettant de créer le fils dédié à la consultation. Une fois créé, il ouvre les deux tubes qui lui sont dédié est lecture/écriture. Nous avons vu dans le cours qu'il ne fallait pas utiliser RDWR pour éviter les conflits de lecture écriture. J'ai fait ce choix pour éviter que les tubes ne se ferment une fois utilisés.

```
while(1) {
        read(tub1, &request_view, sizeof(int));
```

Ici il s'agit de la boucle principale du fils de consultation.

Il se mets en lecture sur le tube dédié à l'envoi de requêtes pour le client.

```
sem_wait(mutex);
```

Il attends de pouvoir prendre le sémaphore (ici `mutex`) avant d'effectuer des opérations avec la mémoire partagée.

```
if (request_view >= 0 && request_view < MAX_SPECTACLE) {
            response_view = shared_table[request_view].places;
            printf("[CONSULTATION] Spectacle %d : %d places\n",
        request_view, response_view);
       } else {
            printf("[CONSULTATION] Numéro invalide : %d\n", request_view);
        }
```

Une fois le mutex libéré, le processus peu, si le numéro est valide, renvoyer au client le nombre de places disponibles.

```
sem_post(mutex);
```

Une fois l'action sur la mémoire partagée effectuée, le sémaphore est relâché.

```
        write(tub2, &response_view, sizeof(int));
        }
    close(tub1); close(tub2);
    exit(0); // exit du fils
    }
```

Enfin, la réponse est écrite dans le tube où le client récupérera la réponse.

La partie `close` et `exit` n'est en réalité jamais atteinte par le fils puisqu'il est dans une boucle infinie `while(1)`. Il faudrait ajouter une gestion des signaux pour terminer proprement le fils.

```
// Réservation

    if (fork() == 0) { // processus fils -> réservation

        int tub3 = open("tub1_reservation", O_RDWR); // RDWR pour ne pa>
        int tub4 = open("tub2_reservation", O_RDWR);
        int tub5 = open("tub3_reservation", O_RDWR);
```

Dans cette section nous pouvons observer la seconde fonctionnalité du serveur, à savoir la gestion des réservations. Un second processus fils est créé, de la même façon que la processus dédié à la consultation.

```
while(1) { // Boucle infinie de traitement
        read(tub3, &request_reservation, sizeof(int)); // Attente d'une>
        read(tub4, &request_reservation_nb, sizeof(int));
        sem_wait(mutex); // Attente du sémaphore de type mutex
```

Tout comme le processus de consultation, une boucle infinie est utilisée pour traiter les requêtes. Le processus se mets en attente sur les tubes `tub1 reservation` et `tub2 reservation`, dédiés respectivements à la réception du numéro du spectacle et du nombre de places à réserver. Enfin, le processus attends que le mutex se libère avant de traiter la requête reçue.

```
// Vérification des entrées
        if (request_reservation ⩾ 0 && request_reservation
    < MAX_SPECTACLE && request_reservation_nb > 0 &&
    shared_table[request_reservation].places ⩾ request_reservation_nb) {
```

Avant d'effectuer quelconque action, le processus vérifie les entrées qui lui sont fournies. Il vérifie que le numéro de spectacle fourni existe, que le nombre de places demandées n'est pas égal à 0 et qu'il reste sufisamment de places disponibles.

```
shared_table[request_reservation].places -= request_reservation_nb;
            save_spectacles();
            response_reservation = true;
            printf("[RESERVATION] %d places réservées pour spectacle
        %d.\nPlaces restantes : %d\n", request_reservation_nb,
        request_reservation, shared_table[request_reservation].places);
```

Une fois l'information reçue, le mutex obtenu et les entrées vérifiées, le processus procède au traitement. Il commence par décrémenter le nombre de places demandées du nombre de places disponibles (en mémoire partagée). Ensuite il enregistre cette nouvelle valeur dans le fichier de stockage à l'aide de la fonction `save_spectacles()` et de la mémoire partagée. Enfin il assigne la valeur `True` à la variable qui sera retournée au client avant d'afficher une ligne de log.

```
} else {
            response_reservation = false;
            printf("[RESERVATION] Impossible pour le spectacle %d\n",
        request_reservation);
        }
```

Si les conditions sus-mentionnées ne sont pas remplies, la variable de response sera remplie d'un `False` et un log d'erreur sera affiché.

```
        sem_post(mutex);

        write(tub5, &response_reservation, sizeof(bool));
        }
    close(tub3); close(tub4); close(tub5);
    exit(0); // exit du fils
    }
```

Le processus relâche ensuite le mutex avant d'écrire sa réponse sur le tube. Comme pour la consultation, la partie fermeture des tubes et du fils n'est ici jamais atteinte.

```
    wait(NULL);
    wait(NULL);

    shmdt(shared_table);
    shmctl(shmid, IPC_RMID, NULL);
    sem_close(mutex);
    sem_unlink("/semaphore");
    return 0;
}
```

Cette dernière portion du serveur corresponds à l'attente des deux fils par le père avant la destruction de la mémoire partagée et du sémaphore.

Client

Dans cette seconde partie de la réponse à la Question 1, nous allons explorer le code du client.

Ce code doit être compilé puis exécuté distinctement du serveur. Il y a deux programmes distincts, fonctionnants sur une même machine et ayant le même répertoire courant.

```
#include <stdio.h> // Entrées sorties
#include <fcntl.h> // Options de open()
#include <unistd.h> // Donne accès à l'API POSIX
#include <stdbool.h> // Pour utiliser les booléens

#define MAX_SPECTACLE 10 // Identique au serveur

#define DATA_FILE "spectacles.dat" // Fichier de données, comme le serveur
```

On commence de nouveau avec les librairies puis on défini des constantes pour le numéro de spectacle le plus haut et le fichier de données.

```
typedef struct {
    int id;
    char nom[50];
    char date[11];
    int places;
} Spectacle;
```

On défini la structure pour lire les spectacles, de la même façon que le serveur. La différence majeure est qu'ici il n'y a pas besoin de mémoire partagée puisqu'il n'y a pas de processus concurrents.

```
/* Table des spectacles pour consultation */
Spectacle table_spectacles[MAX_SPECTACLE] = {};
```

Un tableau `table_spectacles` est créé via la structure `Spectacle` et en ayant `MAX_SPECTACLE` comme nombre d'entrées maximal.

```
void load_spectacles() {

    int fd = open(DATA_FILE, O_RDONLY); //lecture seule

    if (fd == -1) {
        printf("Aucun␣spectacle␣n'est␣disponible␣pour␣le␣moment.\n");

    } else {
        read(fd, table_spectacles, sizeof(table_spectacles)); //écriture dans la tab>
        close(fd);
    }

}
```

Ici on retrouve une fonction pour charger les spectacles dans le client, depuis le fichier de données. On utilise `open()` via le descripteur `fd` puis on écrit le contenu dans `table_spectacles` via la fonction `read()`.

```
int main() {
    int tub1, tub2, tub3, tub4, tub5;
    int choice;
    int index, nb_places;
    bool success;
```

La fonction principale du programme commence par la déclaration des descripteurs dédiés au tubes ainsi que des différentes variables proches de celles du serveur.

```
load_spectacles();
```

On utilise la fonction déclarée plus haut pour charger la liste des spectacles dans la MC du programme.

```
while (1) {
        printf("\n═══␣MENU␣═══\n");
        printf("1.␣Consultation\n");
        printf("2.␣Réservation\n");
        printf("0.␣Quitter\n");
        printf("Votre␣choix␣:␣");
        scanf("%d", &choice);
```

Ici le programme entre dans une boucle infinie avant d'afficher les différentes options à l'utilisateur à travers un menu interactif. Après affichage des options, il se met en attente d'une entrée clavier via `scanf()`.

```
if (choice == 0) break;
```

Si le choix est égal à 0, on sort de la boucle infinie et le programme se termine.

```
switch (choice) {
    case 1: /* CONSULTATION */

                printf("\n═══␣SPECTACLES␣DISPONIBLES␣═══\n"); // Affichage des
  ↪ spectacles
                for (int i = 0; i < MAX_SPECTACLE; i++) {
```

```
                    if (table_spectacles[i].id ≠ 0) {
                        printf("[%d] %s | Date : %s\n", i, table_spectacles[i].nom, >
                    }
                }

                printf("\nEntrez le numéro du spectacle à consulter : ");
                scanf("%d", &index); // Lecture de l'entrée utilisateur

                // Ouverture des tubes
                tub1 = open("tub1_view", O_WRONLY); // Tube pour l'envoi de la requête
                tub2 = open("tub2_view", O_RDONLY); // Tube pour la réception de la
 ↪ réponse

                write(tub1, &index, sizeof(int)); // Ecriture du numéro de spectacle
 ↪ dans le tube
                read(tub2, &nb_places, sizeof(int)); // Lecture de la réponse

                if (nb_places ⩾ 0) {
                    printf("Selon le serveur : %d places sont disponibles\n", nb_pla>
                } else {
                    printf("Erreur : spectacle inexistant\n");
                }

                close(tub1);
                close(tub2);
                break;
```

Si le choix est égal à 1, le client bascule en mode consultation. Il affiche alors les spectacles disponibles en se basant sur sa MC qui est elle même basée sur le fichier de données. Il passe ensuite en attente d'une entrée utilisateur. L'utilisateur indique le numéro du spectacle qu'il souhaite consulter et le programme transmet l'information au serveur via un `write()` sur le tube dédié. Il bascule ensuite en attente de réponse du serveur via un `read()` sur le tube dédié. Si le nombre de places reçues n'est pas égal à 0, la valeur est affichée à l'écran à destination de l'utilisateur. Le cas échéant, une erreur est affichée. Enfin, les tubes sont fermés et le client revient au menu principal.

```
case 2: /* RESERVATION */
                printf("\n=== RÉSERVATION ===\n");
                printf("Numéro du spectacle à réserver : ");
                scanf("%d", &index);
                printf("Nombre de places à réserver : ");
                scanf("%d", &nb_places);

                tub3 = open("tub1_reservation", O_WRONLY);
                tub4 = open("tub2_reservation", O_WRONLY);
                tub5 = open("tub3_reservation", O_RDONLY);

                write(tub3, &index, sizeof(int));
                write(tub4, &nb_places, sizeof(int));
                read(tub5, &success, sizeof(bool));

                if (success) {
                    printf("Réservation OK. A bientôt dans nos salle !\n");
                } else {
                    printf("Réservation NOK. Reessayez avec un nombre de places plus>
                }
```

```
                close(tub3);
                close(tub4);
                close(tub5);
                break;
```

Si le choix de l'utilisateur est égal à 2, le client bascule en mode réservation. Il n'affiche pas la liste des spectacles et se mets directement en attente d'une entrée utilisateur. Cette dernière corresponds au numéro du spectacle à réserver. Le programme attends ensuite une seconde entrée qui corresponds au nombre de places à réserver.

Puis il ouvre les tubes dédiés à travers des descripteurs.

Il écrit ensuite les données de l'utilisateur sur les tubes, à destination du serveur. Pour cela il utilise la fonction `write()`. Il bascule en attente d'une réponse du serveur via `read()`.

Une fois la réponse reçue, le processus indique à l'utilisateur si la réservation est validée ou non. Si la réponse corresponds à True la réservation est validée, sinon la réservation est invalide.

Enfin, il ferme les tubes et revient au menu principal.

```
 default:
                printf("Choix invalide\n");
```

Ces deux lignes permettent de gérer une entrée invalide dans le menu en affichant `"Choix invalide"`.

```
        }
    }

    return 0;
}
```

Enfin, la fonction principale retourne 0. De nouveau, cette partie n'est jamais atteinte en raison de la boucle infinie sans condition de sortie.

Conclusion

Au cours de cette première question nous avons défini une base nécessaire à la suite des questions. Certains points sont anticipés, comme les sémaphores et la mémoire partagée mais cela me semblait cohérent des les introduire directement, dans la mesure où cela facilite l'extension future des programmes. Cela évitera de faire de trop gros changement d'architecture par la suite. D'autres points comme la persistance des données ne sont pas demandés mais me semblaient utiles pour livrer un serveur concret et utilisable. J'ai donc dû explorer les options disponibles en m'éloignant un peu du contenu du cours.

Question 2

On vous demande de mettre en place une seconde version de cette application client-serveur en utilisant à présent des files de messages.

A présent, vous avez plusieurs clients qui s'exécutent en parallèle. Le tableau des réservations et sa manipulation ne sont pas changés.

Le processus serveur traite les requêtes client reçues dans la MSQ en FIFO.

Mettez en place la structure de votre application : processus, file de messages, structure des messages. Expliquez vos choix. Codez votre solution en prenant soin de documenter votre code.

Réponse

Cette question nécessite quelques ajustement sur la façon dont sont échangées les données entre le serveur et les clients mais pas sur le traitement interne au serveur. Cette version semble permettre une plus grande flexibilité dans le traitement des requêtes des clients. Cela permets également de mieux séparer et compartimenter les requêtes. En effet, comme vu en cours, nous allons utiliser les MSQ ainsi que le PID du client pour renvoyer la réponse au bon client.

Il va falloir déterminer une structure cohérente pour le traitement des requêtes. Le client devra toujours fournir son PID ainsi que les valeurs associées à sa demande. On peu imaginer mettre en place une structure ressemblant à ceci :

Consultation :

```
|Code de requête|PID client|
```

Réservation :

```
|Code de requête|PID client|n° du spectacle|nombre de places|
```

Le mieux me semble être de créer deux structures distinctes pour les deux requêtes. Une pour la demande de consultation avec seulement le PID du client et une seconde avec le numéro du spectacle et le nb de places à réserver. Bien que cela rajoute des lignes de code, je trouve cela plus propre et approprié.

Admettons que la requête avec le code “1” soit une consultation et que celle avec le code “2” soit une réservation. Le processus fils de consultation ne traitera alors que les requêtes avec le code “1” et celui de réservation celles avec le code “2”.

Dans ma réponse je ne vais pas re détailler tout le code mais simplement les changements apportés. L’ensemble du code est cependant accessible dans le répertoire “q2”.

Serveur

```c
#include <sys/types.h> // Permets d'utiliser les types de données (pid_t ... )
#include <sys/ipc.h> // Permets la gestion des IPCs (ftok ... )
#include <sys/msg.h> // Permets d'utiliser les MSQ
```

J’ai commencé par ajouter les dépendances nécessaires. On retrouve ici `types.h`, `ipc.h` et `msg.h` pour respectivement : les types de données, les IPCs et les MSQs.

```c
// Types de requêtes
#define TYPE_LIST 1 // Consultation
#define TYPE_RESERV 2 // Réservation
```

J’ai modifié les types de requêtes en nommant “consultation” en “list”. En effet, le client demandera désormais le liste complète des spectacles avec le nombre de places. Avant, le client lisait la liste en “dur” depuis un fichier partagé avec le serveur. Pour rendre ma copie plus réaliste et préparer la question bonus, j’ai décidé de créer une relation client-serveur plus proche de la réalité. Seul le serveur détient la vérité.

```c
// Obtention de la clef via ftok
key_t key = ftok("question_2_server", 42);
```

```
// Rejoindre la MSQ et la créer si inexistante
int msgid = msgget(key, 0666 | IPC_CREAT);
```

Ici on retrouve la création de la clef (via `ftok()`) et la création de la MSQ via cette même clef. J'ai décidé de tout de suite utiliser `ftok()`. Cela n'est pas encore le cas pour les autres clefs (shm, semaphore) mais je l'implémenterais plus tard. Je préffère pour l'instant me concentrer sur le sujet du devoir et le peaufinner à la fin.

`ftok()` va donc créer une clef unique à partir du fichier `question_2_server` - lui même donc - et du chiffre 42. `msgget()` ensuite utiliser cette clef pour rejoindre ou créer une MSQ. L'id retourné sera stocké dans msgid.

```
if (fork() == 0) { // CONSULTATION

        // Demande de la liste
        typedef struct {
            long mtype;
            pid_t pid;
        } Request_list;

        // Réponse de la liste
        typedef struct {
            long mtype;
            pid_t pid;
            Spectacle spectacles[MAX_SPECTACLE];
        } Response_list;

        Request_list request_list;
        Response_list response_list;
```

J'ai modifié le premier fils, celui dédié à la consultation. Désormais il utilisera les structures visibles ci dessus. Ces dernière ne contiennent pas de numéro de spectacle particulier à consulter. Le client demandera toute la liste à jour et le serveur la lui renverra dans sa réponse via le champ spectacles.

```
// Boucle infinie de traitement

        while(1) {

            // Attente d'un message sur la MSQ
            msgrcv(msgid, &request_list, sizeof(Request_list) - sizeof(long), TYPE_LIST
 ↪ , 0);

            // Préparation de la réponse avec le PID fourni par le client.
            response_list.mtype = request_list.pid;
            // On précise notre PID
            response_list.pid = getpid();

            // Demande du mutex
            sem_wait(mutex);

            // Copie du contenu de la mémoire partagée dans la réponse
            memcpy(response_list.spectacles, shared_table, sizeof(table_spectacles));

            // Relâchement du mutex
            sem_post(mutex);
```

```
            // Envoi de la réponse
            msgsnd(msgid, &response_list, sizeof(Response_list) - sizeof(long), 0);
    }
```

La boucle de traitement s'est également vu quelque peu modifiée. On attends toujours un message du client avant de demander le mutex mais cette fois-ci via la MSQ et non les tubes. `msgrcv()` attends un nouveau message sur la MSQ avec l'id `msgid`. Il stockera le contenu dans l'espace mémoire associé à `request_list`. Ce contenu a une taille correspondant à la taille de la structure de requête moins le mtype. Enfin, le serveur n'écoute que les messages dont le `mtype` est égal à `TYPE_LIST`, soit 1.

Il prépare ensuite sa réponse en mettant le PID fourni par le client dans le mtype. Il précise également son PID (pour de potentielles implémentations futures mais non essentielles pour le moment).

Il demande le mutex via `sem_wait()`.

Il copie le contenu de sa mémoire partagée (à chaud) dans la réponse. C'est le contenu le plus récent. Pour cela on utilise `memcpy()`.

Il relâche le mutex via `sem_post()`.

Enfin, il envoie la réponse via msgsnd. La réponse contiendra la liste des spectacles à jour, son PID et le PID du client en mtype.

```
if (fork() == 0) { // RESERVATION

        // Requête pour la réservation
        typedef struct {
            long mtype;
            pid_t pid;
            int spectacle;
            int nb_places;
        } Request_reservation;

        // Réponse pour la réservation
        typedef struct {
            long mtype;
            pid_t pid;
            bool ack;
        }  Response_reservation;

        Request_reservation request_reservation;
        Response_reservation response_reservation;
```

A l'instar du processus de consultation, le processus de réservation se voit également doté de deux nouvelles structures. Une pour les requêtes et une pour les réponses. On retrouvera cette même logique d'échange des PIDs entre le client et le serveur mais quelques autres champs diffèrent. En effet, ici il s'agit plus d'une requête de traitement qu'une requête informative.

Dans la requête on aura le numéro du spectale à réserver ainsi que le nombre de places. Dans la réponse un simple "ack" - comprendre ACKnowledgment - sous la forme d'un booléen et le PID du serveur.

```
while(1) { // Boucle infinie de traitement

        // Attente d'une requête sur la MSQ
```

```
        msgrcv(msgid, &request_reservation, sizeof(Request_reservation) - sizeof(long),
  ↪  TYPE_RESERV, 0);

        // Demande du mutex

        sem_wait(mutex);
```

Comme pour l'autre fils, on a une boucle infinie qui attends un message sur la MSQ. Les différences sont qu'ici, nous écrirons le contenu dans `request_reservation` et que nous attendons un `mtype` égal à `TYPE_RESERV` (2). La taille varie aussi, conformément à la requête de réservation. Une fois qu'un message est arrivé, le fils demande le mutex.

```
// Vérification des entrées
        if (request_reservation.spectacle ⩾ 0 && request_reservation.spectacle < MAX_
  ↪ SPECTACLE &&
                request_reservation.nb_places > 0
        && shared_table[request_reservation.spectacle].places ⩾ request_reservation.nb
  ↪ _places
                 && shared_table[request_reservation.spectacle].id ≠ 0) {
```

Avant de procéder à n'importe quelle opération, le fils de réservation effectue des vérifications des entrées qui lui sont fournies.

Il vérifie les conditions suivantes :

- Le numéro du spectacle est valide (`0 ⩾ et < MAX_SPECTACLE`).
- Le nombre de places demandées et supérieur à 0.
- Le nombre de places disponibles est supérieur ou égal au nombre de places demandé.
- Le spectacle existe

```
// Décrémentation du nombre de places
            shared_table[request_reservation.spectacle].places -= request_reservation.
  ↪ nb_places;

            // Enregistrement des modifications sur le fichier persistant
            save_spectacles();
```

Une fois que ces conditions sont valides, le fils décrémente le nombre de places demandées et enregistre en dur sur le fichier de données.

```
 // ACK sur True pour confirmer au client
            response_reservation.ack = true;

            // Affichage d'un message de log
            printf("[RESERVATION] %d places réservées pour spectacle %d.\nPlaces 
  ↪ restantes : %d\n",
                   request_reservation.nb_places, request_reservation.spectacle,
           shared_table[request_reservation.spectacle].places);
```

La valeur du ACK qui sera émis en réponse est définie sur True avant d'afficher un message de log sur le console.

```
} else {

            // ACK sur False pour infirmer au client
            response_reservation.ack = false;
```

```
            // Affichage d'une erreur
            printf("[RESERVATION] Impossible pour le spectacle %d\n",request_
  ↪ reservation.spectacle);
        }

        // Relâchement du mutex
        sem_post(mutex);

        // Défnition du mtype sur le PID du client
        response_reservation.mtype = request_reservation.pid;

        // On mets le PID du serveur dans la réponse
        response_reservation.pid = getpid();

        // Envoi de la réponse sur la MSQ
        msgsnd(msgid,&response_reservation, sizeof(Response_reservation) - sizeof(long)
  ↪ , 0);
```

Si les conditions ne sont pas remplies, le ACK est défini sur False et un message d'erreur s'affiche sur la console.

Le mutex est relâché.

Le serveur défini le PID fourni par le client comme mtype pour la réponse.

Il précise également son PID à titre informatif.

La réponse est envoyée via `msgsnd()`. Le serveur utilise l'id `msgid` et envoie le contenu du response reservation. La taille de la réponse corresponds à la taille de la structure de réponse moins la taille du mtype (un long).

```
  // Exit du fils
    exit(0);

    }

    wait(NULL);
    wait(NULL);

    // Destruction de la file de messages
    msgctl(msgid, IPC_RMID, NULL);

    // Suppression de la table partagée
    shmdt(shared_table);

    // Suppression de l'espace de mémoire partagée
    shmctl(shmid, IPC_RMID, NULL);

    // Fermeture du mutex
    sem_close(mutex);

    // Suppression du sémaphore
    sem_unlink("/semaphore");

    return 0;
}
```

La fin du programme est similaire à ce qui a été vu dans la question 1. La destruction de la MSQ avec `msgctl()` a été ajouté. On détruit la MSQ via son id et `IPC_RMID`.

Voilà pour la partie serveur. Seul le mode de communication a été modifié, pas le traitement interne.

Passons désormais à la partie client.

Client

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

Tout comme le serveur, de nouvelles librairies sont incluses au programme.

```c
// Types de requêtes
#define TYPE_LIST 1 // Consultation
#define TYPE_RESERV 2 // Réservation
```

Modification des types des requêtes également.

```c
// Demande de consultation
typedef struct {
    long mtype;
    pid_t pid;
} Request_list;

// Réponse de consultation
typedef struct {
    long mtype;
    pid_t pid;
    Spectacle spectacles[MAX_SPECTACLE];
} Response_list;

// Demande de réservation
typedef struct {
    long mtype;
    pid_t pid;
    int spectacle;
    int nb_places;
} Request_reservation;

// Réponse de réservation
typedef struct {
    long mtype;
    pid_t pid;
    bool ack;
} Response_reservation;

Request_reservation request_reservation;
Response_reservation response_reservation;

Request_list request_list;
Response_list response_list;
```

Les structures nécessaires à la communication avec le serveur sont déclarées à l'identique. La différence est qu'ici il n'y a qu'un seul processus donc tout est déclaré au même endroit. Pour le serveur, étant donné que j'utilise deux fils distincts, j'ai préféré séparer la déclaration des requêtes dans chacun d'eux. Ainsi, chaque fils n'a accès qu'à la structure de données dont il a besoin.

```c
int main() {
    int choice;
    int index, nb_places;

    // ftok pour générer la clef
    key_t key = ftok("question_2_server", 42);

    // On rejoins la MSQ
    int msgid = msgget(key, 0666 | IPC_CREAT);
```

On observe ici que les variables nécessaires pour les descripteurs de fichiers (avec les tubes) ont disparues. Idem pour le booléen qui servait de ACK, il est désormais intégré à la structure. On génère également la clef via ftok. Cette génération de clefs suppose que le client et le serveur sont présents dans le même répertoire.

Le client rejoint ensuite la MSQ via `msgget()` et la clef retournée par ftok.

```c
switch (choice) {
            case 1: // CONSULTATION

                // Type de la requête : LIST (1)
                request_list.mtype = TYPE_LIST;
                // On précise notre PID
                request_list.pid = getpid();
                // Envoi du message sur la file
                msgsnd(msgid,&request_list, sizeof(Request_list) - sizeof(long), 0);
                // Attente d'une réponse avec notre PID dans le mtype
                msgrcv(msgid, &response_list, sizeof(Response_list) - sizeof(long),
  ↪ request_list.pid, 0);

                // Affichage de la réponse
                printf("\nRéponse␣du␣serveur␣:\n");

                for (int i = 0; i < MAX_SPECTACLE; i++) {
                    if (response_list.spectacles[i].id != 0) {
                        printf("[%d]␣%s␣|␣%s␣|␣%d␣places\n",
                            i,
                            response_list.spectacles[i].nom,
                            response_list.spectacles[i].date,
                            response_list.spectacles[i].places);
                    }
                }
                break;
```

La partie consultation a beaucoup évolué. En effet, comme mentionné plus haut cette dernière ne lis plus un fichier un “dur” avant de demander le nombre de places disponibles au serveur. Cette configuration était peu réaliste et ne permettait pas au client d’avoir des informations récentes. Si un spectacle était modifié / remplacé, impossible de le savoir sans relancer le client. Il s’agit désormais d’une véritable relation client-serveur où le client accède aux ressources via le serveur uniquement.

Le client commence par préparer sa requête en définissant le contenu du mtype à TYPE LIST (1). Il précise ensuite son PID dans le champ dédié.

Le message est envoyé sur la MSQ via `msgsnd()`.

Le point intéressant se situe dans l’attente de la réponse du serveur. En effet, le client va attendre

une réponse sur la MSQ dont le mtype corresponds à son PID. Nous avons vu précédement que le serveur prends soin de mettre le PID du client dans le mtype de sa réponse. Cette astuce permets de renvoyer la réponse au bon client dans un cas de gestion multi-clients. Si plusieurs clients écoutennt sur la même MSQ une réponse du serveur, seul celui avec le PID correspondant prendra en compte la réponse. Cela empêche des confusions dans l'échange des données.

Une fois la réponse reçue, le client l'affiche lignes à lignes avec une boucle for.

```
case 2: // RESERVATION
                // Demande à l'utilisateur le n° du spectacle
                printf("Numéro du spectacle à réserver : ");
                scanf("%d", &index);
                // Demande à l'utilisateur le nombre de places
                printf("Nombre de places à réserver : ");
                scanf("%d", &nb_places);
```

Nous arrivons sur la partie réservation. Pour les entrées utilisateur, rien n'a changé.

```
// Type de la requête : RESERV(2)
                request_reservation.mtype = TYPE_RESERV;
                // On précise notre PID
                request_reservation.pid = getpid();
                request_reservation.spectacle = index;
                request_reservation.nb_places = nb_places;
```

Le client prépare ensuite sa requête en remplissant les champs avec le type (2), son PID et les valeurs fournies par l'utilisateur.

```
// Envoi du message sur la file
                msgsnd(msgid, &request_reservation,
                    sizeof(Request_reservation) - sizeof(long), 0);

                // Attente d'une réponse avec notre PID dans le mtype
                msgrcv(msgid, &response_reservation,
                    sizeof(Response_reservation) - sizeof(long),
                    request_reservation.pid, 0);
```

D'une façon similaire à la partie consultation, la requête est émise sur la MSQ avant d'attendre une réponse où le mtype correponds au PID du client.

```
 // Vérification de la confirmation du serveur
                if (response_reservation.ack) {
                    printf("Réservation OK. À bientôt dans nos salles !\n");
                } else {
                    printf("Réservation NOK. Réessayez.\n");
                }
                break;
```

Enfin, le client vérifie le retour du serveur. Si la réponse est égale à `True`, il affiche un succès sinon un échec.

Conclusion

En conclusion cette question a permis de rendre plus opérationel la solution proposée. En effet, en utilisant les MSQ on s'inscrit dans une démarche plus proche de la réalité avec l'utilisation des IPCs. Je trouve également à titre personnel que cela apporte de la lisibilité et de la clarté au code. J'ai également essayé d'étayer mes commentaires pour rendre le code aussi intelligible

que possible. J'ai pris une liberté par rapport au sujet en remaniant un peu la fonctionnalité de consultation. En effet il est demandé de pouvoir consulter le nombre de places d'un spectacle à la fois. J'ai trouvé plus pertinent d'obtenir la liste complète et à jour. Cela peu susciter des questions de sollicitation serveur mais en réalité à l'échelle du projet, cela n'est pas un problème. J'ai dû effectuer des recherches sur Internet pour la partie affichage avec la boucle for et donc, m'éloigner un peu du cours. Je pense néanmoins que cela est pertinent dans une démarche de projet concret. Avec la définition des structures de requête et de réponse, on commence à voir se dessiner un protocole. Il pourrait être intéressant d'imaginer un protocole complet, basé sur les PIDs par exemple. Si on a plusieurs serveurs et plusieurs client on pourrait imaginer cela :

- Le client émet une requête sur la MSQ
- Un serveur la récupère et lui réponds en précisant son PID
- Le client et le serveur précisent désormais le PID de l'autre en tant que mtype

Ainsi on pourrait avoir un serveur pour un client.

J'ai également conscience que cette version n'intègre pas de gestion des erreurs concrète. Ce sera un ajout à faire pour les dernières versions au même titre que `ftok()` pour toutes les clefs.

Question 3

A présent, le processus serveur créé d'une part un thread qui traite les consultations, d'autre part un thread qui traite les réservations. Chaque thread prélève les messages qui le concerne dans la MSQ, les traite et renvoie la réponse.

Mettez en place la structure de votre application en considérant que serveur et client sont sur une même machine. Codez votre solution en prenant soin de documenter votre code.

Réponse

Cette question n'a pas induit de gros changements d'architecture étant donné que j'avais déjà divisé le serveur en deux. Aucune modification n'a été apportée au client. Il a fallu remplacer les fils par des threads. La différence sera donc qu'on va désormais utiliser des processus légers plutôt que des processus fils qui sont plus lourds. Les processus légers vont pouvoir partager la mémoire du père ce qui va donner lieu à des simplifications de mon code.

Serveur

```
#include <pthread.h> // Permets d'utiliser les threads
```

On commence par ajouter une dépendance pour pouvoir utiliser les threads.

```
// Clefs statiques pour les sémaphores
#define SEM_KEY 5678
```

Ici on peu voir que j'ai supprimé la clef destinée à la mémoire partagée (SHM). J'ai décidé de supprimer le système de mémoire partagée. En effet, étant donné que nous utilisons désormais des threads et non des processus fils, la mémoire est nativement partagée. Il n'est plus nécessaire d'allouer un espace dédié dans la mémoire. Cela simplifie le code et le rends plus lisible. Je suppose également que les performances sont meilleures car l'accès mémoire est direct.

```
// Déclaration du mutex
sem_t mutex;
```

L'utilisation des sémaphores a également été revue. En effet, je n'utilise plus de sémaphores nommés car de nouveau, les threads ont un espace de mémoire partagé. Je développerais ce point plus bas dans le document.

```
// Fonction de sauvegarde dans le fichier
void save_spectacles() {
    int fd = open(DATA_FILE, O_WRONLY | O_TRUNC | O_CREAT, 0666); // O_TRUNC évite d'
  ↪ écrire à la suite, cela réécris le fichier ; O_CREAT créé le fichier si inexistant
    write(fd, table_spectacles, sizeof(table_spectacles));
    close(fd);
}
```

Etant donné que nous n'utilisons plus de mémoire partagée, les données sont stockées dans table spectacle au lieu de shared table. Cela implique donc de modifier toutes les interactions associées dans le code. Ici il s'agit de la fonction d'enregistrement des données sur le fichier dédié. On écrit depuis table spectacles au lieu de shared table.

```
// Fonction pour le thread de consultation
void* thread_consultation(void* arg) {
    int msgid = *(int*)arg;

    // Demande de la liste
    typedef struct {
    long mtype;
    pid_t pid;
    } Request_list;

    // Réponse de la liste
    typedef struct {
    long mtype;
    pid_t pid;
    Spectacle spectacles[MAX_SPECTACLE];
    } Response_list;

    Request_list request_list;
    Response_list response_list;

    // Boucle infinie de traitement

    while(1) {

    // Attente d'un message sur la MSQ
    msgrcv(msgid, &request_list, sizeof(Request_list) - sizeof(long), TYPE_LIST, 0);

    printf("[CONSULTATION] - Demande de %i.\n",request_list.pid);
    // Préparation de la réponse avec le PID fourni par le client.
    response_list.mtype = request_list.pid;
    // On précise notre PID
    response_list.pid = getpid();

    // Demande du mutex
    sem_wait(&mutex);

    // Copie du contenu de la table dans la réponse
```

```
    memcpy(response_list.spectacles, table_spectacles, sizeof(table_spectacles));

    // Relâchement du mutex
    sem_post(&mutex);

    // Envoi de la réponse
    msgsnd(msgid, &response_list, sizeof(Response_list) - sizeof(long), 0);

    }

}
```

Ci-dessus on retrouve la fonction qui sera appelée par le thread de consultation. Il s'agit du fils de consultation vu dans la question précédente avec les modifications évoquées plus haut (pas de SHM et sémaphore non nommé).

```
// Fonction pour le thread de réservation
void* thread_reservation(void* arg) {
    int msgid = *(int*)arg;

    // Requête pour la réservation
        typedef struct {
            long mtype;
            pid_t pid;
            int spectacle;
            int nb_places;
        } Request_reservation;

        // Réponse pour la réservation
        typedef struct {
            long mtype;
            pid_t pid;
            bool ack;
        }  Response_reservation;

        Request_reservation request_reservation;
        Response_reservation response_reservation;

        while(1) { // Boucle infinie de traitement

        // Attente d'une requête sur la MSQ

        msgrcv(msgid, &request_reservation, sizeof(Request_reservation) - sizeof(long),
  ↪  TYPE_RESERV, 0);

        // Demande du mutex

        sem_wait(&mutex);

        // Vérification des entrées
        if (request_reservation.spectacle ⩾ 0 && request_reservation.spectacle < MAX_
  ↪ SPECTACLE &&
                request_reservation.nb_places > 0
        && table_spectacles[request_reservation.spectacle].places ⩾ request_
  ↪ reservation.nb_places
                 && table_spectacles[request_reservation.spectacle].id ≠ 0) {

            // Décrémentation du nombre de places
```

```
            table_spectacles[request_reservation.spectacle].places -= request_
↪ reservation.nb_places;

            // Enregistrement des modifications sur le fichier persistant
            save_spectacles();

            // ACK sur True pour confirmer au client
            response_reservation.ack = true;

            // Affichage d'un message de log
            printf("[RESERVATION] - %i a réservé %d place(s) pour le spectacle %d. 
↪ Places restantes : %d\n",
                    request_reservation.pid, request_reservation.nb_places, request_
↪ reservation.spectacle,
                table_spectacles[request_reservation.spectacle].places);
        } else {

            // ACK sur False pour infirmer au client
            response_reservation.ack = false;

            // Affichage d'une erreur
            printf("[RESERVATION] - Réservation impossible pour le spectacle %d et le 
↪ client %i.\n",request_reservation.spectacle, request_reservation.pid);
        }

        // Relâchement du mutex
        sem_post(&mutex);

        // Défnition du mtype sur le PID du client
        response_reservation.mtype = request_reservation.pid;

        // On mets le PID du serveur dans la réponse
        response_reservation.pid = getpid();

        // Envoi de la réponse sur la MSQ
        msgsnd(msgid,&response_reservation, sizeof(Response_reservation) - sizeof(long)
↪ , 0);
    }

}
```

Ci-dessus on retrouve la fonction qui sera appelée par le thread de consultation. Il s'agit du fils de consultation vu dans la question précédente avec les modifications évoquées plus haut (pas de SHM et sémaphore non nommé).

```
int main() {

    // Ouverture des données dans le fichier de data
    int fd = open("spectacles.dat", O_RDONLY); //lecture seule

    // Si le fichier n'existe pas ou est vide, on copie les données d'exemple sur le
↪ fichier de données
    if (fd == -1) {
        save_spectacles();
    } else {
        // Sinon, on charge depuis le fichier vers la mémoire partagée
        read(fd, table_spectacles, sizeof(table_spectacles));
        close(fd);
```

```
    }
```

Dans le main on commence par charger les données avant de lancer les threads.

```
    // Déclaration des threads
    pthread_t thread_consult, thread_reserv;

    // Création et lancement des threads
    pthread_create(\&thread_consult, NULL, thread_consultation, &msgid);
    pthread_create(\&thread_reserv, NULL, thread_reservation, &msgid);

    printf("[INFO] Serveur UP\n");

    // Attente de la fin des threads

    pthread_join(thread_consult, NULL);
    pthread_join(thread_reserv, NULL);
```

Cette partie constitue le coeur de la question. On commence par déclarer les threads avec le type associé. Ensuite on créer et lance les threads. On précise la fonction qu'on veut lancer (en lien avec celles définies plus haut). La valeur `&msgid` est passée en paramètre pour permettre aux threads de rejoindre la MSQ une fois lancés. Enfin, `join` permets d'attendre que les threads se terminent avant de continuer l'éxécution du main.

```
    // Destruction de la file de messages
    msgctl(msgid, IPC_RMID, NULL);

    // Fermeture du mutex
    sem_destroy(&mutex);

    return 0;
```

Enfin, cette partie du programme détruit la MSQ et le mutex. Cette partie n'est jamais atteinte car les threads tournent à l'infini.

Conclusion

Cette question a de nouveau permis d'apporter de la clarté au code. En utilisant les threads au lieu du `fork()`, on élimine plusieurs problématiques dont la mémoire partagée et les sémaphores nommés. Je pense que j'ai eu la bonne démarche en voulant séparer la traitement des deux requêtes mais je n'ai pas directement utilisé les bons outils. On a désormais un serveur plus léger et facile à maintenir / développer. J'ai profité de cette question pour rendre plus exhaustifs les logs dans la console, permettant un meilleur déboggage.

Pour constater l'impact direct sur les performances d'un tel choix technique, j'ai regardé les informations retournées par la commande `ps` et les informations contenues dans `/proc` :

Pour la solution proposée en question 2 :

```
user@03102025:~/cnam/nsy103/project$ ps aux |grep question_2_server
user        5500  0.0  0.0   2576  1632 pts/0    S+   22:59   0:00 ./bin/q2/question_2_
  ↪ server
user        5501  0.0  0.0   2576   820 pts/0    S+   22:59   0:00 ./bin/q2/question_2_
  ↪ server
user        5502  0.0  0.0   2576   820 pts/0    S+   22:59   0:00 ./bin/q2/question_2_
  ↪ server
```

```
user@03102025:~/cnam/nsy103/project$ cat /proc/5500/status
Name:   question_2_serv
Umask:  0002
State:  S (sleeping)
Tgid:   5500
Ngid:   0
Pid:    5500
PPid:   4503
TracerPid:  0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 100 1000
NStgid: 5500
NSpid:  5500
NSpgid: 5500
NSsid:  4503
Kthread:    0
VmPeak:     2576 kB
VmSize:     2576 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      1632 kB
VmRSS:      1632 kB
RssAnon:          108 kB
RssFile:         1520 kB
RssShmem:           4 kB
VmData:      232 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1596 kB
VmPTE:        44 kB
VmSwap:        0 kB
HugetlbPages:           0 kB
// Autres informations
```

Pour la solution proposée dans la question 2, avec les fils on a donc ceci :

- 4 processus au total
- Une utilisation CPU et mémoire de 0.0%
- Un VmRSS de 1632 KB

Regardons désormais la solution proposée en question 3 :

```
user@03102025:~/cnam/nsy103/project$ ps aux |head -n 1 ; ps aux |grep question_3_server
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
user        5008  0.0  0.0  18952  1544 pts/0    Sl+  22:29   0:00 ./bin/q3/question_3_
  ↪ server
```

```
user@03102025:~/cnam/nsy103/project$ cat /proc/5008/status
Name:   question_3_serv
Umask:  0002
State:  S (sleeping)
Tgid:   5008
Ngid:   0
Pid:    5008
PPid:   4503
TracerPid:  0
```

```
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 100 1000
NStgid: 5008
NSpid:  5008
NSpgid: 5008
NSsid:  4503
Kthread:    0
VmPeak:    18952 kB
VmSize:    18952 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      1544 kB
VmRSS:      1544 kB
RssAnon:          116 kB
RssFile:         1428 kB
RssShmem:           0 kB
VmData:    16608 kB
VmStk:       132 kB
VmExe:         4 kB
VmLib:      1596 kB
VmPTE:        52 kB
VmSwap:        0 kB
HugetlbPages:           0 kB
```

Pour la solution proposée dans la question 3, avec les threads on a ceci :

- 1 seul processus
- Une utilisation CPU et mémoire de 0.0%
- Une VmRSS de 1544 kB

Cette nouvelle approche supprime 2 processus. Pour la partie CPU, l'ordre de grandeur est trop petit pour être mesuré. Pour la partie mémoire on peu se baser sur le VmRSS. En effet, la solution avec les `forks` affiche 1632 kB mais il ne faut pas oublier qu'il y a 3 processus ! On peu donc estimer la mémoire utilisée à 3 * 1632 kB soit 4,8 MB. Pour la solution avec les threads on a 1544 kB soit 1,5 MB. D'après ces estimations, la solution avec les threads consomme donc 3,2 fois moins de mémoire.

Question 4

A présent, le processus serveur créé un thread pour chaque requête prélevée dans la file de message. Soit k requêtes reçues, vous aurez donc k threads qui s'exécutent en parallèle. Chaque thread traite la requête et renvoie la réponse.

Mettez en place la structure de votre application en considérant que serveur et client sont sur une même machine. Codez votre solution en prenant soin de documenter votre code.

Réponse

Cette question va impliquer de nombreux changement sur l'architecture du serveur. Nous utilions déjà des threads, ce qui facilite la tâche. La question à se poser est la suivante : comment créer dynamiquement des threads ? J'ai donc songé à un système en deux temps : Un premier

temps de lecture de la requête et un second de création du thread poour traiter la requête. Ma plus grosse difficultée a été qu'on ne peu passer qu'un seul argument à un thread. L'ennui est que j'ai besoin de lui passer la requête pour qu'il la traite et le `msgid` pour qu'il puisse y répondre. Il a donc fallu créer une structure dans laquelle on ajoute la requête et le `msgid`. On passe ensuite cette structure en argument au thread. Cette nouvelle solution est plus extensible en étant moins limité par le nombre de requêtes. On verra également l'utilisation de `detach` pour ne pas attendre la fin du thread avant de traiter de nouvelles requêtes. Le client demeure inchangé.

Serveur

```
// Définition des requêtes (générique)
typedef struct {
    long mtype;
    pid_t pid;
    int spectacle;
    int nb_places;
} Request;

// Définition des réponses de type LIST
typedef struct {
    long mtype;
    pid_t pid;
    Spectacle spectacles[MAX_SPECTACLE];
} Response_list;

// Définition des réponses de type RESERV
typedef struct {
    long mtype;
    pid_t pid;
    bool ack;
} Response_reservation;
```

Dans cette nouvelle version, le serveur interpète les requêtes de façon générique. Il ne fait plus la différence entre les requêtes de type `LIST` et les requêtes de type `RESERV`. On a donc une structure `"Request"` unique.

```
void* thread_handler(void* arg) {
    // Strucutre pour les données reçues en argument
    struct {
        Request* req; // Déclaration du pointeur req de type Request
        int msgid; // Déclaration de msgid
    } *data = arg; // Déclaration du pointeur data avec le contenu de arg

    Request* req = data->req; // Le pointeur req contient data->req (la requête)
    int msgid = data->msgid; // msgid contient data->msgid (le msgid à utiliser)

    // Si le mtype est de type LIST
    if (req->mtype == TYPE_LIST) {
        // Log sur la console
        printf("[CONSULTATION] Client %i a effectué une demande de consultation.\n",req
  ↪ ->pid);
        Response_list resp; // Déclaration de la réponse de type list
        resp.mtype = req->pid; // resp.mtype est égal au pid contenu dans la requête
        resp.pid = getpid(); // resp.pid = pid du serveur

        sem_wait(&mutex); // Demande du mutex
```

```
        // Copie des données en mémoire dans la réponse
        memcpy(resp.spectacles, table_spectacles, sizeof(table_spectacles));
        sem_post(&mutex); // Relâchement du mutex
        // Envoi de la réponse sur la MSQ
        msgsnd(msgid, &resp, sizeof(Response_list) - sizeof(long), 0);
    } else if (req->mtype == TYPE_RESERV) { // Si la requête est de type RESERV
        Response_reservation resp; // Déclaration de la réponse de type reserv
        resp.mtype = req->pid; // resp.mtype est égal au pid fourni dans la requête
        resp.pid = getpid(); // resp.pid = pid du serveur
        // Relâchemen duu sémaphore
        sem_wait(&mutex);
        // Vérification des entrées :
        // Si le spectacle demandé existe, que lenb de places n'est pas égal à 0
        // et que il reste suffisament de places
        if (req->spectacle >= 0 && req->spectacle < MAX_SPECTACLE &&
            req->nb_places > 0 &&
            table_spectacles[req->spectacle].places >= req->nb_places) {
            // Décrémentation du nombre de places demandées
            table_spectacles[req->spectacle].places -= req->nb_places;
            // Sauvegarde dans le fichier DAT
            save_spectacles();
            // On définis le ACK à True
            resp.ack = true;
            // Log sur la console
            printf("[RESERVATION] Client %i réservé %d place(s) pour le spectacle %d.
↪ Places restantes : %d\n",
                   req->pid, req->nb_places, req->spectacle,
                   table_spectacles[req->spectacle].places);
        } else {
            // Si entrées non valides, ACK à False
            resp.ack = false;
            // Log de l'erreur dans la console
            printf("[RESERVATION] Impossible pour le spectacle %d et le client %i\n",
                   req->spectacle, req->pid);
        }
        // Relâchement du mutex
        sem_post(&mutex);
        // Envoi de la réponse sur la MSQ
        msgsnd(msgid, &resp, sizeof(Response_reservation) - sizeof(long), 0);
    }
    // Libération de l'espace mémoire utilisé
    free(req);
    free(data);
    // Fin du thread
    pthread_exit(NULL);
}
```

Il s'agit ici de la partie principale du serveur, le thread de traitement. C'est lui que le main va créer à chaque nouvelle requête. C'est dans ce thread que la requête sera traitée comme consultation ou comme réservation. Il n'y a plus de boucle infinie car ce thread est temporaire. On retrouve la structure évoquée plus haut. Le pointeur data permets d'accéder à la requête transmise par le main et au msgid généré par le main. Puisqu'on utilise un pointeur, il faut utiliser des `->` à la place des `.` pour accéder aux différents champs de la structure. Le reste de la logique reste identique à ce qui a été précédemment implémenté. Il n'y donc plus qu'un seul thread pour gérer les consultations et les réservations mais ce thread est appelé k fois en fonction de k requêtes émises par j clients. A la fin du thread, l'espace mémoire qui lui est dédié est libéré avec un `free()` sur `req` et un autre sur `data` avant de se terminer avec un `exit`.

```
// Boucle principale
    while (1) {
        // Le pointeur req de type Request obtient dynamiquement un espace en mémoire
        Request* req = malloc(sizeof(Request));
        // Réception de n'importe quel message sur la MSQ
        msgrcv(msgid, req, sizeof(Request) - sizeof(long), 0, 0);

        // On crée une structure pour passer le msgid et la requête au thread
        // On ne peu passer qu'un seul argument au thread, on est donc obligés de faire
 ↪  ça
        struct {
            Request* req; // Pointeur req de type Request
            int msgid; // Déclaration de msgid
        } *data = malloc(sizeofdata));
```

```
        data->req = req; // On rempli le contenu de data req avec la requête reçue
        data->msgid = msgid; // On rempli le contenu de msgid-> msgid avec le msgid de
 ↪ la MSQ
        // Déclaration du thread
        pthread_t th;
        // Création du thread avec la fonction associée, la requête et le msgid en
 ↪ argument
    }
```

La boucle principale du serveur a également évoluée. Dans l'ancienne version, on avait l'appel de deux threads. Dans celle ci on a une boucle infinie de traitement. Dans cette boucle on retrouve la création du pointeur `req` basé sur la structure générique vu plus haut qui permettra la réception des deux types de requêtes. Cela est possible car les deux requêtes commencent de façon identitique. Sans cette condition, cela aurait été impossible. Ensuite le serveur écoute les messages sur la MSQ. Quand une nouvelle requête arrive, le serveur créer un autre pointeur basé sur une structure pouvant contenir la requête (`req`) et le `msgid`. Il passe ensuite dans ce pointeur les données de la requête et le msgid. Enfin, la boucle principale déclare puis créer le thread (en lui passant data en argument) avant de le "détacheré grâce à `detach`. Cette dernière action permets de ne pas attendre la fin du thread avant de relancer la boucle et donc, de répondre à de nouvelles requêtes.

Conclusion

Avec cette nouvelle architecture, il sera plus simple d'étendre et de gérer les requêtes clients. Grâce aux threads en parralèle (avec `pthread_detach` notamment), il est possible de gérer plusieurs clients en parralèle, sans blocages. La gestion de la mémoire est désormais optimisée car chaque allocation de mémoire pour un thread - et donc requête - est géré dynamiquement, ce qui optimise les ressources.

Question 5 : bonus

Transposez la dernière solution dans le cas où clients et serveurs ne sont plus sur la même machine.

Réponse

Pour répondre à cette question, nous allons devoir utiliser les sockets. Nous verrons comment les créer, les assigner et les utiliser. J'ai également amélioré la gestion des erreurs.

Serveur

```
#include <netinet/in.h> // Pour utiliser AF_INET et AF_INET6
#include <sys/socket.h> // Pour socket(), bind()...
#include <arpa/inet.h>  // inet_addr (pour convertir les ip)

// Définition du port pour le socket
#define PORT 5000
```

On commence par importer des nouvelles librairies qui vont permettre d'utiliser correctement les sockets. Après quelques recherches j'ai identifié que `socket.h` et `inet.h` étaient inclus dans `netinet.h`. Cela est propre à ma configuration. Je les ai inclus pour être exhaustif.

On définis le port (`5000`). Il sera utilisé plus tard dans le code au moment de la mise en écoute du serveur. C'est là que le client devra envoyer ses requêtes.

```
// Structure d'une requête par un client
typedef struct {
    int type;
    int spectacle;
    int nb_places;
} Request;
```

J'ai unifié les deux types des requêtes pour simplifier le code. Une requête de type consultation ou réservation aura donc la même structure. Le type permets de détromper les deux.

```
// Structure d'une réponse à une demande de consultation
typedef struct {
    Spectacle spectacles[MAX_SPECTACLE];
} Response_list;

// Structure d'une réponse à une demande de réservation
typedef struct {
    bool ack;
} Response_reservation;
```

Les structures de réponses ont été allégées. Puisque nous sommes avec des sockets, plus besoin de préciser le PID du client comme avant. La connexion établie s'assurer que l'information arrive au bon destinataire.

```
// Fonction de gestion des requêtes client
void* client_handler(void* arg) {
    int sock = *(int*)arg; // sock = l'argument passé à l'appel
    free(arg); // On libère l'espace utilisé par les arguments

    Request req; // Création de req, basé sur Request
```

Il s'agit ici de la fonction principale utilisée par les threads générés par le main. `client_handler` vient remplacer `thread_handler`. Comme avant, il récupère les arguments fournis (`arg`). Ici il s'agit du socket à aller lire. On libère directement la mémoire pour gagner du temps. Pas besoin d'attendre la fin du thread puisqu'on a récupéré l'information nécessaire dès le début. Ensuite on créé req, basé sur Request. On stockera le contenu du socket dedans. C'est la requête.

```
// Lecture des données tant que le client est connecté
    while (read(sock, &req, sizeof(req)) > 0) {

        if (req.type == TYPE_LIST) { // Si le type des "LIST"
```

```
            Response_list resp; // Création de resp, basé sur Response_list

            sem_wait(&mutex); // Demande du mutex
            // Copie des données en mémoire dans la réponse
            memcpy(resp.spectacles, table_spectacles, sizeof(table_spectacles));
            // Relâchement du mutex
            sem_post(&mutex);
            // Ecriture de la réponse sur le socket
            write(sock, &resp, sizeof(resp));
        }
```

Le traitement est semblable a ce qui a été vu précédement. La différence majeure est que le thread reste ouvert tant que la connexion avec le client est active. Cela signifie que ce thread traitera toutes les requêtes d'un client donné tant que sa session est active. Ensuite on vérifie le type, si c'est une consultation on renvoie le contenu de la mémoire tout en utilisant correctement les sémaphores. J'ai supprimé les logs de consultation car dans une dynamique de serveur en réseau et avec un traitement multi-threads, on imagine une forte sollicitation et donc un nombre de logs démesuré pour la consultation.

```
// Si le type est "RESERV"
        else if (req.type == TYPE_RESERV) {
            Response_reservation resp; // Création de resp, basé sur Response_
 ↪ reservation
            resp.ack = false; // Par défaut le ACK est à False
            // Demande du mutex
            sem_wait(&mutex);
            // Vérification des entrées
            if (req.spectacle >= 0 &&
                req.spectacle < MAX_SPECTACLE &&
                req.nb_places > 0 &&
                table_spectacles[req.spectacle].places >= req.nb_places) {
                // Décrémentation du nombre de places
                table_spectacles[req.spectacle].places -= req.nb_places;
                // Sauvegarde des données
                save_spectacles();
                // Le ACK passe à True
                resp.ack = true;
                // Affichage d'un message de log
                printf("[RESERV] - %d places réservées pour le spectacle %d.\n"
                       "%d places restantes pour le spectacle %d\n",
                      req.nb_places, req.spectacle, table_spectacles[req.spectacle].
 ↪ places, req.spectacle);
                // Relâchement du mutex
                sem_post(&mutex);
                // Ecriture de la réponse sur le socket
                write(sock, &resp, sizeof(resp));
                }
            }
        }
```

La partie réservation reste semblable. Elle conserve le message de log car j'ai jugé qu'il était important de pouvoir tracer le nomnbre de places restantes pour un spectacle donné. On pourrait imaginer ajouter l'heure pour suivre l'évolution des réservations à travers le temps.

```
    // Fermeture du socket
    close(sock);
```

```
    // Message de log
    printf("[INFO] - Client déconnecté\n");
    return NULL;
}
```

Enfin le thread se termine proprement en fermant sa connexion au socket et en affichant un log sur la déconnexion du client. Le thread se termine quand le client se déconnecte.

```
// Création d'un socket
    // Protocole Ipv4, mode stream (TCP)
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
    perror("[INFO] - Erreur dans la création du socket");
    exit(1);
    }
    // Déclaration de la structure qui contient les infos du serveur
    struct sockaddr_in addr;
    // Précise IPv4
    addr.sin_family = AF_INET;
    // Mets le port d'écoute sur 5000
    addr.sin_port = htons(PORT);
    // Le serveur écoute sur toutes les interfaces ).addr.sin_addr.s_addr = INADDR_ANY;
```

Dans la boucle principale, la lecture des données n'a pas changé. Ce qui a beaucoup évolué en revanche, c'est la gestion des communications. La partie MSQ a été entièrement supprimée au profit des sockets. On commence par créer le socket avec son descripteur associé (`server_fd`). Ensuite on vérifie qu'il n'y a pas d'erreur à la création en regardant le code de retour avec un `if`.

Une fois le socket créé, on déclare une structure pour ajouter les informations du serveur. On y précise donc qu'on utilise IPv4, le port sur lequel écouter (défini en haut du fichier) et sur quelle interface écoute (toutes dans notre cas).

```
// Associe le socket créé à une IP et un port
    if (bind(server_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
    perror("[INFO] - Erreur dans le bind");
    close(server_fd);
    exit(1);
    }
```

On associe le socket aux informations qu'on vient de déclarer. Pour cela on utiliser `bind()`. On vérifie également que `bind()` n'a pas rencontré d'erreurs.

```
// Bascule le serveur en écoute
    listen(server_fd, 10);
```

On mets le serveur sur écoute avec `listen()`.

```
// Initialise le sémaphore
    sem_init(&mutex, 0, 1);

    // Message de log
    printf("[INFO] - Serveur TCP en écoute sur le port %d\n", PORT);
```

Le sémaphore est initialisé avant d'afficher un log précisant le port utilisé.

```
    // Boucle infinie
    while (1) {
```

```
        // Attente d'un client (mode bloquant)
        int client_fd = accept(server_fd, NULL, NULL);
        // Attribution dynamique de mémoire pour le descripteur du client
        // Comme vu en q4, on doit utiliser cela pour passer
        // les informations au thread
        int* pclient = malloc(sizeof(int));
        // On défini le descripteur créé avec ce qu'on a reçu sur le socket
        *pclient = client_fd;

        // Déclaration du thread
        pthread_t th;
        // Création du thread avec pclient en argument
        pthread_create(&th, NULL, client_handler, pclient);
        // pthread_detach pour ne pas attendre la fin
        pthread_detach(th);
    }
}
```

Enfin, la boucle infinie du serveur se lance. Dans cette boucle, le serveur commence par attendre la connexion d'un client. Cette attente est bloquante, il ne passe pas à la suite tant qu'il n'a pas reçu une connexion. Il enregistre ensuite le retour de `accept()` dans `client_fd`. La fonction `accept()` va retourner un nouveau descripteur de socket. Il s'agit du descripteur associé à la connexion établie avec le client. Le thread l'utilisera donc pour communiquer avec le client distant.

Pour pouvoir passer cette information au thread, il faut passer par un pointeur. Ce dernier se voit attribuer un espace mémoire de façon dynamique grâce à `malloc()`.

Pour terminer, la boucle créé le thread "th" avec `pthread_create` en précisant `client_handler` comme fonction à lancer et le contenu du pointeur `* pclient` - à savoir `pclient` - comme argument. La boucle se détache du thread via `pthread_detach` avant de se relancer et d'attendre une nouvelle connexion.

Client

```
#include <sys/socket.h> // Pour socket(), bind()...
#include <arpa/inet.h>  // inet_addr (pour convertir les ip)

// Port utilisé pour le socket
#define PORT 5000
```

Similaire au serveur.

```
// Structure d'une requête au serveur
typedef struct {
    int type;
    int spectacle;
    int nb_places;
} Request;

// Structure d'une réponse à une demande de consultation
typedef struct {
    Spectacle spectacles[MAX_SPECTACLE];
} Response_list;

// Structure d'une réponse à une demande de réservation
typedef struct {
```

```
    bool ack;
} Response_reservation;
```

Similaire au serveur.

```
// Boucle principale
int main() {

    int choice;

    // Définition du socket
    // Ipv4, mode stream (TCP)
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("[INFO] - Erreur dans la création du socket");
        exit(1);
    }

    // Déclaration de la structure qui contient les infos du serveur
    struct sockaddr_in serv;
    // Précise IPv4
    serv.sin_family = AF_INET;
    // Mets le port d'écoute sur 5000
    serv.sin_port = htons(PORT);

    // Conversion de l'IP fournie en texte
    // On écrit cette IP dans la structure qui contient les infos du serveur
    inet_pton(AF_INET, "IP_DU_SERVEUR", &serv.sin_addr);

    // Connexion au serveur
    if (connect(sock, (struct sockaddr*)&serv, sizeof(serv)) < 0) {
        perror("[INFO] - Erreur dans le connection au serveur");
        exit(1);
    }

    printf("[INFO] Connecté au serveur\n");
```

Ci-dessus, la boucle principale du client. De nouveau, on constate que la partie MSQ a été remplacé par une partie socket. On commence, comme pour le serveur par créer le socket en vérifiant les erreurs. Ensuite on entre les mêmes informations que le serveur dans la structure.

On converti l'adresse IP ajoutée en dur sous forme de texte en un format binaire, lisible par les fonctions socket. Pour cela on utilise `inet_pton()`. Ici j'ai écris `"IP_DU_SERVEUR"` mais il faut évidemment remplacer cela par une adresse IP réelle. Personellement j'ai utilisé un VPS loué chez un hébergeur pour effectuer mes tests et après avoir ouvert le port sur le serveur, cela a fonctionné.

Enfin, avant d'afficher un message de log, on se connecte au serveur via `connect()`. On lui précise le socket à utiliser (`sock`), la structure à utiliser (`serv`) et sa taille (celle de serv).

```
/ Consultation
        if (choice == 1) {
            // Remplissage de la requête de consultation
            request_list.type = TYPE_LIST;
            request_list.spectacle = 0;
            request_list.nb_places = 0;

            // Ecriture sur le socket
```

```
            write(sock, &request_list, sizeof(Request));

            // Lecture de la réponse sur le socket
            read(sock, &response_list, sizeof(Response_list));
```

Une fois le menu passé, pas beaucoup de changement ormis le fait qu'il faut désormais utiliser la requête unique et remplir les champs inutilisés par 0. On utilise `write()` et `read()` pour directement écrire et lire sur le socket. Cela ressemble fortement aux tubes puisque l'on utilise également un descripteur.

```
else if (choice == 2) {
            int index, nb;
            printf("Numéro du spectacle : ");
            scanf("%d", &index);
            printf("Nombre de places : ");
            scanf("%d", &nb);

            // Remplissage de la requête utilisateur avec les entrées
            request_reservation.type = TYPE_RESERV;
            request_reservation.spectacle = index;
            request_reservation.nb_places = nb;

            // Ecriture sur le socket
            write(sock, &request_reservation, sizeof(Request));

            // Lecture de la réponse du serveur
            read(sock, &response_reservation, sizeof(Response_reservation));
```

Même chose pour la réservation.

```
    // Fermeture du socket
    close(sock);
    // Message de log
    printf("[INFO] - Déconnexion du client\n");
    return 0;
```

Enfin, en cas de break (case 0), le client ferme proprement sa connexion au socket, affiche un log et se ferme.

Conclusion

En somme, cette question bonus a permis de procéder à une réelle avancée technique. En utilisant les sockets on se rapproche encore plus d'un cas réel et plausible. On pourrait imaginer développer et déployer ce type de serveur dans une petite entreprise ou pour de la gestion personelle. L'utilisation des sockets permets une communication à distance sur un même réseau ou à travers Internet. J'ai personellement essayé en passant par Internet à l'aide d'un VPS loué. Les structures ont également été unifiées et donc simplifiées. Cela simplifie le traitement notamment au moment de créer le thread. Cela réduit également le nombre de lignes de code. Etant donné que le thread reste en place tant que le client est connecté, on a une persistance de session. Cela pourrait être intéressant pour des cas futurs (un système de panier par exemple). On a également améliorer la gestion des erreurs en se concentrant sur les fonctionnalités les plus critiques.

Pour aller plus loin

Je pense que ce projet est déjà très complet et permets de bien étudier les communications intra-processus, inter-processus et inter-machines. J'ai découvert de nombreux concepts comme les sémaphores, les sockets, les MSQ, les tubes (nommés ou non), la gestion de la mémoire en C, les pointeurs, les threads... Pour aller plus loin j'ai recensé une liste de choses à développer en plus. Cela serait envisageable avec plus de temps et un besoin concret pouvant justifier les exigences listées.

Cette liste est non exhaustive.

- Tracer l'IP du client dans un fichier de log avec l'horaire. | Il faudrait utiliser `accept()` différement avec les arguments 2 et 3.
- Ajouter la date dans les logs.
- Stocker les logs.
- Gérer les signaux pour couper proprement le serveur.
- Utiliser des fichiers HEADER (.h) pour simplifier la compilation.
- Chiffrer la connexion socket avec TLS/SSL