5. Alternative : Communication par Files de Messages

Contrairement aux Tubes (Pipes), nous utilisons ici un objet du système d'exploitation appelé File de Messages.
Cela permet une communication plus souple.

1) Avantage de cette architecture

Persistance : La file existe dans la mémoire du noyau Linux. Même si le client s'arrête, les messages peuvent rester dans la file.

Multi-clients : Une seule file peut gérer plusieurs clients en même temps grâce au système de "Types".

2) Le choix des Structures

Pour que le système accepte nos messages, nous avons dû respecter une structure précise :

Le champ long type (Obligatoire) : Il doit être placé en premier dans la structure. C'est lui qui sert d'aiguillage.

Nos structures ne contiennent pas que le type, elles transportent les données métier :

MessageRequete : contient un int action (le choix), un int id_spec (le spectacle choisi) et un int nb_places.

MessageReponse : contient un tableau char texte[512] pour que le serveur puisse envoyer une réponse lisible à l'utilisateur.

TYPE_REQ (1) : Le serveur écoute uniquement ce type pour recevoir les commandes.

TYPE_RES (2) : Le client écoute uniquement ce type pour recevoir les réponses.

3) Les fonctions clés (Appels Système)

msgget (Instanciation de la file) :

On utilise une Clé unique (CLE_FILE). C'est comme un numéro de téléphone que le client et le serveur partagent pour se trouver.

Le serveur utilise le drapeau IPC_CREAT pour créer la file si elle n'existe pas.

msgsnd (Envoi - Utilisation de Pointeur) :

On donne l'adresse de notre structure (via un pointeur &req ou &res).
Le calcul de taille : On fait sizeof(msg) - sizeof(long).

La gestion des données côté Serveur :

Le serveur possède une instanciation d'un tableau de structures Spectacle en mémoire.
Ce tableau contient l'ID, le nom et le stock actuel.
C'est la base de données locale du serveur.
Chaque réservation (action 2) vient modifier directement ce tableau en soustrayant le nombre de places demandées.

Pourquoi ?

Parce que le système gère le long type tout seul, il ne veut connaître que la taille des données utiles qu'on a rajoutées derrière.

msgrcv (Réception - Bloquante) :

Cette fonction est bloquante. Le programme s'arrête et "dort" tant qu'un message du bon type n'est pas arrivé. C'est parfait pour économiser de la batterie/CPU.

msgctl (Nettoyage) :

À la fin du programme Serveur, on utilise cette commande pour détruire la file. Si on oublie, la file reste "fantôme" dans la RAM du PC.

Pourquoi avoir choisi cette méthode plutôt que le Pipe ?

Argument 1 (Indépendance) : "Avec les files de messages, le client et le serveur n'ont pas besoin d'être père et fils (pas besoin de fork). Ce sont deux programmes totalement indépendants."

Argument 2 (Tri sélectif) : "Grâce au long type, je peux envoyer 10 types de messages différents dans le même tuyau et chaque processus ne ramassera que ce qui le concerne. C'est beaucoup plus propre qu'un Pipe."

Argument 3 (Simplicité) : "Le système gère lui-même la séparation des messages. Dans un Pipe, si j'envoie deux messages d'affilée, ils peuvent se mélanger si je ne fais pas attention. Dans une file, chaque message est un paquet bien fermé."

scénario d'exécution :

“Qu'est-ce qui se passe si je lance le client en premier ? “

L'ordre d'exécution :

On lance le Serveur en premier pour qu'il crée la file avec IPC_CREAT.
On lance ensuite le Client. Si le client est lancé avant, le msgget échouera car la clé 12345 n'existera pas encore dans le système.

Que se passe-t-il si deux clients achètent la dernière place en même temps ?

Dans cette version simple, le serveur traite les messages un par un (séquentiellement) car il n'y a qu'une seule boucle while et un seul msgrcv. Donc, le premier message arrivé dans la file sera traité, le stock sera mis à jour, et le deuxième message verra qu'il n'y a plus de places. Il n'y a pas de conflit possible

4) Contenu des échanges

Ajouter les int et les char du texte (Exemple)

Architecture de communication (Files de Messages IPC) :