Manuel Technique

Introduction

Ce manuel technique expose les solutions développées pour réaliser les différents modes de communication entre un serveur et ses clients dans le cadre d'une application de traitement matriciel. Le système repose sur une architecture évolutive, exploitant les mécanismes suivants :

- Mémoire partagée pour le transfert rapide des données.
- Sémaphores pour synchroniser les accès concurrents.
- Multithreading pour la gestion simultanée de plusieurs requêtes clients.

Architecture

Vue d'ensemble

1. **Client** :
   - Envoie des requêtes contenant les paramètres des matrices à générer et multiplier.
   - Récupère les résultats calculés par le serveur via la mémoire partagée.
2. **Serveur** :
   - Attend les requêtes des clients et les traite de manière asynchrone.
   - Utilise des threads pour paralléliser les calculs.
   - Gère des objets partagés pour échanger les données et les résultats.
3. **Mécanismes de communication** :
   - **Mémoire partagée** : Permet un échange rapide de données.
   - **Sémaphores** : Synchronisent les écritures et lectures entre clients et serveur.
   - **Multithreading** : Garantit une exécution fluide même avec plusieurs clients simultanés.

Détails d'implémentation

1. Mémoire partagée

La mémoire partagée est créée et gérée à l'aide des fonctions suivantes :

- **Création et ouverture** :

```
int shm_fd = shm_open("/shared_matrices", O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, SIZE);
void* ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

- **Structure des données partagées** : La mémoire contient les informations suivantes :

- ➉ Paramètres de la requête (PID client, dimensions des matrices).
- ➉ Matrices générées et leur produit.
- ➉ Indicateurs de statut (requête en attente, traitement terminé).

➉ **Libération** :

À la fin, le serveur libère les ressources :

```
shm_unlink("/shared_matrices");
```

2. Sémaphores

Les sémaphores sont utilisés pour gérer les accès concurrentiels à la mémoire partagée.

➉ **Création et initialisation** :

```
sem_t* sem_a = sem_open("/sem_a_ready", O_CREAT, 0666, 0);
sem_t* sem_b = sem_open("/sem_b_ready", O_CREAT, 0666, 0);
```

➉ **Utilisation** :

- ➉ Le client notifie le serveur que des données sont prêtes :

```
sem_post(sem_a);
```

- ➉ Le serveur attend une requête :

```
sem_wait(sem_a);
```

➉ **Libération** :

```
sem_close(sem_a);
sem_unlink("/sem_a_ready");
```

3. Multithreading

Le serveur crée un thread par requête client pour paralléliser les calculs.

➉ **Création de threads** :

```
pthread_t thread_id;
pthread_create(&thread_id, NULL, process_request, (void*)request_data);
```

➉ **Traitement dans un thread** : La fonction `process_request` effectue les opérations suivantes :

- ➉ Génération de matrices aléatoires.
- ➉ Calcul du produit matriciel.
- ➉ Écriture des résultats dans la mémoire partagée.

➉ **Synchronisation des threads** :

```
pthread_join(thread_id, NULL);
```

Fonctionnement

1. Initialisation

1. **Serveur** :
    - ➉ Crée la mémoire partagée et initialise les sémaphores.
    - ➉ Attend les requêtes entrantes.
2. **Client** :
    - ➉ Écrit une requête dans la mémoire partagée.
    - ➉ Signale au serveur que les données sont disponibles (via le sémaphore).

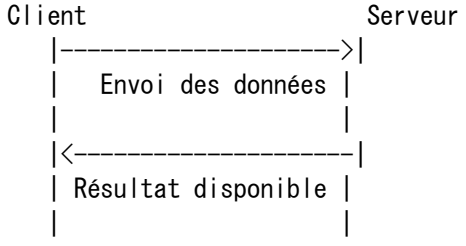
2. Communication

1. **Envoi des données** : Le client insère ses paramètres dans la mémoire partagée et active le sémaphore `sem_a_ready`.
2. **Traitement par le serveur** :
    - ➉ Le serveur démarre un thread pour traiter la requête.
    - ➉ Les matrices sont générées et multipliées.
    - ➉ Le résultat est écrit dans la mémoire partagée.
3. **Retour au client** : Une fois le calcul terminé, le serveur signale au client que le résultat est prêt (via `sem_b_ready`).

Gestion des erreurs

1. **Détection de mémoire manquante** : Si le serveur n'initialise pas la mémoire, le client affiche une erreur et se termine gracieusement.
2. **Synchronisation** : Les sémaphores préviennent les conflits d'accès concurrentiels.
3. **Nettoyage des ressources** : En cas de panne ou d'arrêt, des scripts peuvent être fournis pour nettoyer les segments de mémoire et les sémaphores orphelins.

Diagramme de séquence



Conclusion

Cette architecture, basée sur les mécanismes UNIX, permet une communication efficace et synchronisée entre clients et serveur. L’utilisation des sémaphores et de la mémoire partagée garantit des performances élevées tout en préservant l’intégrité des données.