Grenoble INP - Phelma

Système d'Exploitation et Programmation Concurrente
Rapport

Projet Client-Serveur

Étudiante :
Giulia DONINI BÜRKLE

Professeur :
Cyrille CHAVET

27 novembre 2024

Table des matières

1 Principes de Conception

1.1 Architecture Générale

Le programme client suit une architecture modulaire, où chaque composant a un rôle défini pour répondre aux besoins du projet. Les principaux modules et leurs fonctionnalités sont les suivants :

1. **Moniteur :**
   - Responsable de la connexion au serveur et de la synchronisation des processus.
   - Réceptionne les signaux indiquant l'arrivée de données (`SIGUSR1` pour voie 0 et `SIGUSR2` pour voie 1).
   - Détache la mémoire partagée et gère la déconnexion en fin d'exécution.
2. **Lecteur :**
   - Lit les données depuis les tampons circulaires de la mémoire partagée.
   - Transmet les données via un pipe au processus `Rédacteur`.
3. **Rédacteur :**
   - Regroupe les données lits par blocs de 5 avant de les transmettre.
   - Communique avec le `Driver` en veillant à respecter les priorités d'accès pour éviter des conflits.
4. **Driver :**
   - Affiche les données reçues dans l'ordre défini par les `Rédacteur`, pour chaque voie.

1.2 Protocole de Connexion

Le processus de connexion est géré dans la fonction `connectCL` du fichier `CL_moniteur.c`. Voici une description détaillée des étapes :

1. Génération de la clé pour la messagerie publique :
   - Une clé unique est générée à l'aide de la fonction `ftok`.

     ```
     key_t cle = ftok(CleServeur, C_Msg);
     ```

   - Cette clé est utilisée pour accéder à la file de messages IPC (Inter-Process Communication) partagée par le serveur.
2. Accès à la file de messages :
   - La fonction `msgget` ouvre ou crée la file de messages associée à cette clé. Les permissions 0666 permettent un accès en lecture et écriture.

     ```
     msg_id = msgget(cle, 0666|IPC_CREAT);
     ```

   - Si `msgget` échoue, le programme affiche une erreur et se termine.
3. Envoi du message de connexion :
   - Un message de type `CONNECT` est envoyé au serveur. Ce message contient le PID du client comme identifiant.

     ```
     sprintf(message.txt, "%d", cl_pid);
     msgsnd(msg_id, &message, L_MSG, 0);
     ```

4. Réception de la réponse du serveur :

— Le client attend un message en retour, de type correspondant à son PID. Ce message contient : Une clé pour accéder à la mémoire partagée si la connexion est acceptée ou une chaîne vide si le serveur est saturé.

```
msgrcv(msg_id, &message, L_MSG, cl_pid, 0);
```

— Si le serveur est saturé (message vide), le client affiche une erreur et se termine.

5. Création de la clé pour la mémoire partagée :
   — À partir de la réponse du serveur, une clé est générée pour accéder aux tampons en mémoire partagée.

```
shared_key = ftok(message.txt, C_Shm);
```

6. Envoi de l'acquittement (ACK) :
   — Un message de type `ACK` est envoyé au serveur pour confirmer que le client a bien reçu les informations nécessaires à l'accès.

```
ack_message.type = ACK;
snprintf(ack_message.txt, L_MSG, "%d", cl_pid);
msgsnd(msg_id, &ack_message, strlen(ack_message.txt) + 1, 0);
```

La déconnexion est gérée dans la fonction `disconnectCL`, exécutée avant la fin du programme client. Voici les étapes :

1. Préparation du message de déconnexion :
   — Un message de type `DECONNECT` est préparé, contenant le PID du client.

```
sprintf(message.txt, "%d", getpid());
```

2. Envoi du message au serveur :
   — Le client envoie ce message via la file de messages, informant le serveur qu'il souhaite mettre fin à la session.

```
msgsnd(msg_id, &message, L_MSG, 0);
```

3. Confirmation implicite de déconnexion :
   — Le serveur, à réception de ce message, supprime le client de sa liste active.
   — Aucune confirmation explicite n'est reçue côté client dans cette implémentation.

1.3 Moniteur

1.3.1 Création et Gestion des Pipes

Les pipes sont utilisés pour transmettre les données entre les processus internes du client (`Lecteur` et `Rédacteur`).

1. Création des pipes :
   — Deux pipes sont créés pour permettre la communication entre les processus.

```
int pipe1[2];
int pipe2[2];
int* pipe_st[2];
[...]
pipe(pipe1);
pipe(pipe2);
```

   — Les descripteurs des pipes sont stockés dans un tableau `pipe_st` pour permettre un accès structuré selon la voie.

```
pipe_st[0] = pipe1;
pipe_st[1] = pipe2;
```

2. Structure des pipes :
   — Chaque pipe est un tableau de deux descripteurs de fichiers :
   — `pipe_st[voie][0]` : utilisé pour la lecture.
   — `pipe_st[voie][1]` : utilisé pour l'écriture.

1.3.2 Gestion des Signaux

Les signaux sont utilisés pour informer le client de l'arrivée de nouvelles données dans les tampons circulaires de la mémoire partagée. Les fonctions `litbuf1` et `litbuf2` sont associées aux signaux `SIGUSR1` et `SIGUSR2` respectivement :

```
signal(SIGUSR1, litbuf1);
signal(SIGUSR2, litbuf2);
```

Et quand le client reçoit un signal, les fonctions sont appelés.

```
void litbuf1() {
    voie = 0;
    signal(SIGUSR1, litbuf1);
    printf("Signal recu par le main voie 0\n");
    lecteur(voie);
    redacteur(voie);
}
```

La fonction `pause()` suspend le programme jusqu'à ce qu'un signal soit reçu, et sa boucle est contrôlée par `nb_iteration`.

1.4 Lecteur

Le Lecteur lit les données dans les tampons circulaires de la mémoire partagée.
— Attache la mémoire partagée via la clé générée lors de la connexion (`shared_key`).

```
shared_mem_ptr = (BUF *)shmat(CLTshmid, NULL, 0);
```

— Accède aux tampons correspondant à la voie donnée (voie 0 ou 1), lit la valeur à l'index courant dans le tampon circulaire et écrit cette valeur dans le pipe.

```
int index = (shared_mem_ptr + voie)->n;
int data = (shared_mem_ptr + voie)->tampon[index];
write(pipe_st[voie][1], &data, sizeof(int));
```

1.5 Rédacteur

Le Rédacteur regroupe les données reçues du `Lecteur` et les transmet au `Driver`.

1. Gestion des buffers :
   — Maintient un buffer local pour chaque voie (`buffer1` et `buffer2`).
   — Lit les données depuis le pipe correspondant à la voie et les stocke dans le buffer.

```
read(pipe_st[voie][0], &buffer[voie][times[voie]], sizeof(int));
```

2. Envoi au `Driver` :
   — Une fois 5 données accumulées, elles sont transmises via la fonction `send_to_driver`.

```
if (times[voie] == 5) {
    send_to_driver(buffer[voie], 5, voie);
    times[voie] = 0;  // Réinitialisation du compteur
}
```

1.6 Driver

Le `Driver` gère l'affichage des données reçues par les rédacteurs.
— Reçoit les données sous forme d'un tableau (`data`), leur taille (`size`), et la voie associée (`voie`).

```
void send_to_driver(int *data, int size, int voie) {
    printf("[Driver (Voie %d)]: ", voie);
    for (int i = 0; i < size; i++) {
        printf("%d ", data[i]);
    }
    printf("\n");
}
```

2 Problèmes et solutions

2.1 Transfert de données entre processus

— **Problème :** Lors de la conception initiale, il était difficile de transmettre efficacement les données entre les différents processus du client. Plusieurs approches ont été testées, notamment l'utilisation de variables globales avec le mot-clé `extern`, de structures partagées, et de pointeurs. Ces méthodes introduisaient des conflits de synchronisation ou des incohérences dans les données transmises.
— **Solution :** La solution a été de mettre en place des pipes, un mécanisme IPC adapté à un transfert de données structuré entre les processus. Chaque voie a son propre pipe, permettant au `Lecteur` d'écrire les données lues et au `Rédacteur` de les lire sans interférences.

2.2 Lecture incorrecte dans la mémoire partagée

— **Problème :** Le client ne parvenait pas à récupérer les données depuis la mémoire partagée, et les valeurs obtenues étaient systématiquement à 0. Ce comportement était dû à l'utilisation d'une clé incorrecte lors de l'accès à la mémoire partagée avec `shmget`. La clé correcte, générée par le serveur et transmise au client via la file de messages, n'était pas utilisée correctement.
— **Solution :** Une vérification de la clé transmise par le serveur a été ajoutée dans la fonction de connexion. Le processus de génération et d'utilisation de la clé a été clarifié et simplifié pour garantir que le client utilise toujours la clé appropriée. Avec une variable globale, il était possible de récupérer la clé pour la mémoire partagée et l'utiliser dans l'accès.