ULB UNIVERSITÉ LIBRE DE BRUXELLES

INFO F-201

Quel est ce Pokémon ?

Projet 1
Système d'exploitation

Elias Mennane (544445)

Deni Shkembi (507393)

Luis Brunard (577721)

2023

Contents

1 Processus multiples

1.1 Pourquoi ?

L'adoption de processus multiples dans ce programme, créant deux processus fils à partir du processus parent, répond à des besoins spécifiques. Cette approche permet de répartir le travail pour gagner du temps, sécuriser les espaces mémoire et assurer une fermeture propre du programme.

1.2 Exploitation d'un Algorithme Externe

Le programme utilise un programme externe (`img-dist`) pour effectuer la comparaison d'images. En lançant des processus distincts, chaque processus peut appeler cet algorithme externe de manière indépendante, évitant ainsi tout problème de concurrence ou de conflit entre les différentes instances de l'algorithme.

1.3 Distribution de la Charge de Travail et Comparaison Asynchrone

L'utilisation de deux processus fils permet de distribuer la charge de travail entre eux, chaque processus traitant alternativement une image différente, de manière asynchrone, depuis le `stdin`. Cela peut être particulièrement efficace pour traiter de grandes quantités de données de manière équilibrée et pour réduire le temps total d'exécution.

1.4 Gestion des Erreurs et Terminaison Propre

La création de processus distincts renforce la gestion des erreurs et des interruptions. En cas d'anomalie ou d'interruption, tel qu'un signal SIGINT, le programme peut interrompre correctement les processus enfants, prévenant ainsi les problèmes de gestion des ressources inachevées. En cas d'erreur telle que SIGPIPE, le programme peut interrompre spécifiquement le processus utilisant le tube (pipe) à l'origine de cette erreur de manière adéquate.

1.5 Flexibilité

L'utilisation de processus rend le programme plus flexible. Par exemple, si le programme doit être exécuté sur un système avec plusieurs cœurs, il peut exploiter efficacement ces ressources pour accélérer la comparaison d'images.

2 Pipe

2.1 Pourquoi ?

L'intégration de tubes (pipes) dans ce programme favorise une communication efficace et unidirectionnelle entre les processus. Cela facilite la transmission des chemins d'images, contribuant ainsi à la mise en œuvre réussie de la comparaison d'images en parallèle. L'utilisation de pipes anonymes s'avère largement adéquate, compte tenu de la relation de parenté entre les processus et de la limitation de la taille des messages à un maximum de 999 caractères.

2.2 Communication Unidirectionnelle

Deux pipes, `fd1` et `fd2`, sont utilisés pour établir une communication unidirectionnelle entre le processus parent et les processus fils. Ces pipes sont créés à l'aide de la fonction `pipe`, permettant ainsi au processus parent d'envoyer des données de manière efficace. Chaque processus ferme les extrémités du pipe qui ne sont pas utilisées. Par exemple, le processus parent ferme les extrémités de lecture (READ) des pipes fd1 et fd2, et les processus fils ferment les extrémités d'écriture (WRITE) des pipes dont ils n'ont pas besoin.

2.3 Transmission Asynchrone

Les pipes facilitent une communication asynchrone entre les processus, permettant chaque processus fils à lire les chemins des images dès qu'ils sont transmis dans le pipes par le processus parent via STDIN. Toutefois, le processus fils ne peut accéder aux données suivantes qu'une fois le traitement du chemin actuel est terminé. Cette approche favorise une exécution plus efficiente du programme en évitant les attentes superflues.

2.4 Problèmes rencontrés

Lors de l'utilisation des pipes, nous nous sommes rendu compte qu'il était important de fermer chaque pipe des autres processus dans chacun des fils. Sinon les processus fils ne s'arrêtent jamais

```
pipe(fd1);
pipe(fd2);
first_son = fork();
if(first_son != 0){
    second_son = fork();
    if (second_son != 0){} // Process Pere
    else{ // Process Fils 2
        close(fd1[READ]); //Bien fermer le pipe fd1 dans les 2 sens
        close(fd1[WRITE]);
    }
}
else{ // Process Fils 1
    close(fd2[READ]); //Bien fermer le pipe fd2 dans les 2 sens
    close(fd2[WRITE]);
}
```

3 Mémoire Partagée

3.1 Pourquoi ?

L'utilisation de la mémoire partagée est essentielle dans la résolution de ce projet. Cependant, une attention particulière a été accordée à la prévention des problèmes de concurrence, gérées via des sémaphores. La mémoire partagée, représentée par la structure `struct shared_memory`, est utilisée pour stocker les résultats de la comparaison entre les images. Cette approche facilite l'échange efficient de données entre les processus fils, évitant ainsi la nécessité de communications plus complexes.

3.2 Choix de `mmap` sur `shm`

Le programme utilise la fonction `mmap` plutôt que les appels système traditionnels tels que `shmget` et `shmat` pour créer la mémoire partagée. La raison de ce choix réside dans la simplicité et la portabilité offertes par `mmap`. `mmap` permet de mapper une région de la mémoire directement dans l'espace d'adressage du processus, offrant une interface plus intuitive et un code plus lisible.

3.3 Paramètres de `mmap`

La fonction `mmap` est utilisée avec les paramètres appropriés tels que `NULL` pour laisser le noyau choisir l'adresse de départ, `sizeof(struct shared_memory)` pour spécifier la taille de la région à mapper, `PROT_READ | PROT_WRITE` pour autoriser la lecture et l'écriture, et `MAP_SHARED | MAP_ANONYMOUS` pour créer une mémoire partagée et anonyme.

3.4 Sécurité et Simplicité

L'utilisation de `mmap` dans ce contexte offre une solution simple et sécurisée. Elle élimine la nécessité de gérer explicitement des clés de mémoire partagée, simplifiant ainsi la mise en œuvre. De plus, la gestion des erreurs est plus directe avec `mmap`.

4 Gestion de signaux

4.1 Pourquoi ?

La gestion des signaux dans ce programme offre un mécanisme robuste pour gérer les interruptions utilisateur (`Ctrl+C`) et les erreurs de communication (`SIGPIPE`), assurant ainsi une terminaison propre et une exécution sécurisée.

4.2 Signal SIGINT (`Ctrl+C`)

La fonction `SIGINT_HANDLE` est définie pour gérer le signal `SIGINT`, généralement déclenché par l'utilisateur avec `Ctrl+C` En cas de réception de ce signal, la modification de la variable `keep_running` à zéro, informe le processus parent de demander aux enfants de s'arrêter et d'attendre la fin de leurs tâches en cours.

4.3 Signal SIGPIPE

La fonction `SIGPIPE_HANDLE` est définie pour gérer le signal `SIGPIPE`. En cas de réception de ce signal, la variable `keep_running` est mise à zéro, indiquant au père de s'arrêter et la fonction envoie un signal `SIGTERM` aux processus fils à l'aide de la fonction `kill`. Cela permet une gestion propre des interruptions potentielles dues à des erreurs de communication.

4.4 Sécurité et Robustesse

L'utilisation des signaux offre robustesse et gére efficacement les interruptions. La gestion appropriée des signaux assure une terminaison propre des processus fils, évitant tout problème de ressources inachevées.

5 Sémaphores

5.1 Utilisation des Sémaphores

Le programme utilise des sémaphores pour synchroniser l'accès concurrentiel à la mémoire partagée entre les processus. L'utilisation des sémaphores dans ce programme permet de protéger la mémoire partagée, garantissant un accès cohérent et sûr aux données partagées entre les processus concurrents.

5.2 Sémaphore `sem`

Le sémaphore `sem` est initialisé avec la valeur 1 à l'aide de la fonction `sem_init`. Cette valeur indique qu'un seul processus à la fois peut accéder à la section critique protégée par le sémaphore.

5.3 Protection de la Mémoire Partagée

Avant d'accéder à la mémoire partagée, les processus utilisent les fonctions `sem_wait` et `sem_post` pour respectivement acquérir et libérer le sémaphore. Ceci garantit que seule une instance de processus à la fois peut entrer en section critique.

5.4 Sécurité et Cohérence des Données

L'utilisation des sémaphores assure une manipulation sûre des données partagées. La section critique protégée par le sémaphore permet d'éviter les conflits d'accès concurrentiel, assurant ainsi la cohérence des données partagées.

5.5 Simplification de la Synchronisation

Les sémaphores simplifient la synchronisation entre les processus en fournissant un moyen efficace de gérer l'accès concurrentiel. Cela contribue à la robustesse du programme.