Quel est ce Pokemon ?
Rapport

Antoine BERTHION (000566199) Romain CROUGHS (000572261)
Lucas VAN PRAAG (000569535)

Année 2023-24

1 Introduction

Dans ce projet, nous devions faire de la recherche concurrente d'images à partir d'une image passée en paramètre, qui sera comparée avec des images qui seront passées dans l'entrée standard. L'enjeu principal du projet est donc de créer deux processus, qui peuvent communiquer avec le processus père afin d'effectuer la tâche précédemment énoncée.

Mais alors comment faire gérer et faire communiquer ces processus ?

Les systèmes d'exploitation modernes nous offrent beaucoup de méthodes pour coordonner nos processus. Dans ce rapport, nous décrirons comment nous avons imaginé les processus et les protocoles de communication entre les processus. Enfin, nous parlerons des scripts bash qui sont utiles au bon fonctionnement de notre programme.

2 Programme `img-search`

Ce programme est le coeur du projet. Il permet de récupérer les images à comparer, créer les processus fils qui vont effectuer la recherche concurrente et enfin de récupérer les résultats des processus fils.

2.1 Récupération des paramètres

Nous avons deux types de paramètres différents a récupérer :

— Image que l'on souhaite comparer avec la banque d'images qui est passée en argument du programme.

— Banque d'images qui est passée dans l'entrée standard.

Premièrement, récupérer l'image passée dans les paramètres est une tâche basique, car elle se trouve dans notre argument `argv[]` à l'index 1 (l'index 0 étant le nom de notre application).

Deuxièmement, pour récupérer l'entrée standard, nous pouvons utiliser l'appel systeme `int fgets(char buf, int buf_size, stdin)` qui va lire l'entrée standard ligne par ligne et mettre le contenu dans notre buffer et retourner `1`. Dans le cas où nous tombons sur un `EOF`, il renvoie simplement 0. Pour récupérer toutes les lignes de notre entrée standard, nous pouvons mettre cette fonction dans une boucle `while` et, par effet de bord, la boucle va se bloquer quand le fichier sera terminé.

2.2 Création des processus enfant

Pour gérer la recherche concurrente des images, nous devons créer deux processus qui vont avoir un fonctionnement similaire, mais va rechercher des images différentes qui lui seront envoyées par le processus père via un pipe anonyme.

2.3 Fonctionnement du père

Notre processus père a une responsabilité : transmettre les paramètres entrés par l'utilisateur à ses deux enfants. Pour avoir une charge de travail équitable sur les deux processus enfants, nous tenons un compteur de combien d'images ont été entrées dans l'entrée standard. Si ce compteur est pair, il envoie l'image dans le pipe du premier enfant, dans le cas inverse, il l'envoie à l'autre enfant.

Cependant, lors de l'écriture dans les pipes, nous avons fait face à la problématique suivante. La comparaison de distance entre deux images est un processus long. Au contraire, récupérer les images dans l'entrée standard ne l'est pas. Cela ne pose pas de problèmes quand nous utilisons une entrée standard qui est reliée au clavier de l'utilisateur, mais quand nous faisons un pipe entre nos processus (avec l'opérateur | en bash) le processus père écrit plus vite que le processus enfant ne peut le lire. Pour remédier à cela, nous avons opté pour une solution en signaux. Lorsque le père a écrit dans les deux pipes, au début de son exécution, il se met en pause avec un signal `SIGSTOP` qu'il s'envoie à lui même. Une fois qu'un des processus fils a fini son calcul il débloque l'exécution du père avec le signal `SIGCONT`. Cela permet aux pipes de ne jamais être surchargés.

2.4 Fonctionnement des processus enfants

Le processus enfant va tout d'abord lire ce qu'il y a dans son pipe. Vu que le read est bloquant, nous n'avons qu'à attendre que le père écrive quelque chose pour commencer l'exécution. Une fois qu'une image est recue, nous allons créer un nouveau processus et modifier son fil d'exécution avec l'appel système `execlp` car nous avons veillé a ce que notre programme `img-dist` soit dans les variables `PATH` lors de l'execution de notre programme. Nous pouvons alors attendre qu'il se finisse avec la fonction `wait()` pour récupérer sa valeur de retour (qui est la distance entre l'image que l'on recherche et l'image de la banque d'image). Finalement, nous pouvons stocker ce résultat dans la mémoire partagée qui sera détaillé dans le chapitre suivant.

2.5 Stockage des resultats dans la mémoire partagée

Pour stocker tous les résultats qui ont été calculés, nous pouvons les mettre dans un tableau dans la mémoire partagée. Cependant, nous devons veiller aux problèmes de concurrence, car nous ne savons pas quand les données vont être écrites par les deux processus fils. Pour éviter tous ces problèmes de concurrence, nous avons décidé d'assigner deux places complètement différentes dans la mémoire, afin d'éviter tous les problèmes qui pourraient être causés par la concurrence des recherches. En pratique, un enfant écrit et lit uniquement sa partie de la liste partagée et l'autre ne lit et écrit uniquement dans l'autre partie. Nous avons alors un tableau d'entier qui est de taille 2, et deux tableaux de caractère. Le tableau d'entier contiendra la distance la plus petite trouvée par chaque processus, et les tableaux de caractère le chemin correspondant à cette valeur. Une fois que toutes les recherches sont terminées, le processus père va comparer les deux éléments de la liste d'entiers, et le plus petite valeur sera le resultat de retour. La valeur par défaut de cette liste d'entier est -10, ce qui permet de savoir si aucune image n'a été entrée.

2.6 Gestion des signaux

Enfin, pour bien coordonner nos processus, nous devions avoir une gestion habile des signaux.

En ce qui concerne la gestion des signaux du fils, nous masquons les signaux qui peuvent bloquer son exécution afin de ne pas bloquer le calcul de la distance entre deux images. Nous allons alors uniquement traiter le signal `SIGUSR1` qui sera masqué pendant l'execution de `img-dist` et qui sera géré a la fin du calcul. Cela permet de gérer nous même les signaux et de ne pas avoir de arrêts avec des `SIGINT` qui pourraient survenir a des moments imprévus.

En ce qui concerne le processus père, il va devoir gérer ses propres signaux et envoyer des signaux a ses enfants si besoin. Si il recoit une `SIGINT`, il va envoyer une signal `SIGUSR1` à ses enfants comme mentionné dans le paragraphe ci-dessus, et puis terminer sa propre execution. Il va avoir le même mecanisme s'il reçoit un signal `SIGPIPE` car, comme décrit dans les consignes, nous voulons que notre programme se termine si ce genre de signal se produit.

Enfin, nous utilisons les signaux pour savoir quand nous pouvons écrire dans les pipes. Une fois que le processus père a écrit dans les deux pipes, reliés a ses deux enfants, il se met en pause. Quand un enfant a libéré son pipe, il envoie un signal `SIGCONT` à son père afin que ce dernier se débloque et écrive l'adresse suivante dans le pipe. Cette pause se fait grace à la fonction standard `pause()` qui se met en pause un fil d'exécution et qui reprend cet exécution quand une signal est traité par un handler.

3 Script bash `list-file`

Le fonctionnement de ce programme est assez simple car il repose sur une fonction déjà existante en bash, le programme `find`. Nous allons lui ajouter les paramètres `-maxdepth 1` afin qu'on ne cherche pas les images dans les sous-dossiers qui peuvent se trouver dans le dossier de recherche. Également, nous allons inclure le paramètre `-type f` qui signifie que nous ne voulons que afficher les fichiers et non les dossiers. Enfin, le paramètre `-name ".*"` afin de ne pas inclure les fichiers masqués (ce paramètre n'est pas vraiment obligatoire mais il permet de rendre le programme plus robuste). Nous avons également veillé à informer l'utilisateur s'il manque un paramètre dans son appel au script.

4 Scipt bash `launcher`

Finalement, nous devions créer un script qui permette de lancer notre programme. Nous avions donc deux modes à gérer, comme il est décrit dans l'énoncé.

Premièrement, nous avons du modifier la variable `PATH`. Pour ce faire, nous devons comprendre comment fonctionne cette variable. Cette variable contient des adresses de fichiers contenant des programmes qui sont séparés par des :. Si nous voulons ajouter notre programme, il nous suffit d'ajouter notre chemin à cette variable, nous avons alors la commande `PATH="$PATH:$PWD/img-dist/"`. Nous incluons `PWD` car nous voulons le chemin absolu de notre pogramme.

Pour gérer le mode interactif, nous devions écouter le `stdin`, ajouter le préfix qui a été entré, et puis le rediriger vers notre programme `img-search`. L'écoute de l'entrée peut se faire grace a la fonction read de bash, qui mis dans un boucle while nous permet d'écouter jusqu'à ce que l'utilisateur fasse un Ctrl+D. Nous pouvons alors `echo` l'entrée standard, et l'envoyer dans un pipe à notre programme. Nous devons veiller à ce qu'il y ait bien un seul et unique passage à la ligne (notamment a la fin car echo génère un passage a la ligne).

En ce concerne le mode automatique, nous allons utiliser notre script `list-file` et rediriger la sortie de ce script vers l'entrée de notre programme `img-search`, avec l'opérateur `|`.

5 Conclusion

Pour conclure, nous avons pu, à travers ce projet, comprendre toutes les méthodes de communication entre processus. Entre les pipes utilisés pour communiquer du père vers les enfants, de la mémoire partagée pour enregistrer les résultats ou encore les signaux pour synchroniser les processus. Nous avons également pu aborder les possibles problèmes de concurrence dans la mémoire partagée.