Rapport du mini-projet en C

Durant la création des différents programmes demandés lors de ce mini-projet, j'ai pu m’apercevoir de quelques points forts et faiblesses de ceux-ci. Tronquer fut compliqué pour ma part, si c’est la dernière ligne de la sortie, si elle n’était pas remplie, compté le nombre d’espaces à mettre pour ensuite écrire le texte sur la sortie standard. Des calculs assez obscurs à base de division et modulo sont les point faibles de celle-ci. Les faiblesses du premier programme, je dirai qu’allouer de la mémoire de cette manière est sûrement risqué pour les fuites de mémoire ou les exploitations de buffer overflow. Par exemple, lorsqu’on utilise la commande xxd sans argument, ce qui veut dire qu’on écrit directement dans la sortie standard stdin. On aura forcément une fuite de mémoire lorsqu’on enverra notre signal de fin. Comme autre faiblesse, il ne réimprime pas les caractères spéciaux dans la partie texte de la sortie.

Au niveau des points fort de celui-ci, il permet de savoir au bout de quel caractère on atteint 16 octets et nous affiche notre fichier en base 16 (hexadécimal) qui est plus pratique dans le développement.

Ensuite, pour le deuxième exercice, le point faible numéro 1, c'est qu’il y aura forcément des commandes qui retourneront toujours des différences de sortie alors que ce sont les mêmes commandes. Par exemple, si on fait « ./compare ls -li – ls -li » dans le dossier dans lequel les fichiers de la commande 1 et de la commande 2 sont créées. Vu qu’il y en a un qui sera écrit avant l’autre, car les processus n’exécute pas leurs programmes exactement en même temps.

Pour un second point faible, il utilise des commandes prédéfinies dont nous n’avons pas la main dessus si elles sont obsolètes ou contiennent un quelconque bug de retour. Et il demande plus de ressource que le premier, puisqu'il utilise plusieurs processus le long du programme.

Pour ses points fort, il est pratique pour comparer les sorties d’un code personnel avec un code officiel comme comparer l’exercice 1 avec le xxd du package xxd. Et second point fort, il utilise des fichiers temporaires uniques qui sont supprimées quand le programme s’arrête.

Pour les entrées-sorties, j'ai principalement utilisé du haut-niveau sauf lors de l’utilisation de mkstemp(3) pour le deuxième exercice. Vu que mkstemp(3) créer un fichier unique avec une template prédéfini, ouvre ce fichier et renvoi un descripteur de fichier ouvert en bas-niveau. J’ai dû utiliser close(2) qui est une fonction bas-niveau pour fermer les différents descripteurs ouvert grâce à mkstemp(3). Mais j’ai aussi utilisé du bas-niveau pour copier les descripteurs et rediriger la sortie avec dup2(2) dans le deuxième exercice.

Dans le deuxième exercice, les différentes commandes sont exécutées grâce à des fork(2) de processus. Effectivement, j'ai fork une première fois et fork une deuxième fois dans le processus fils pour fermer le descripteur secondaire pour ce processus, pouvoir rediriger la sortie de la première

commande grâce à dup2(2) vers le fichier A et l’exécuter. Ensuite, j'ai attendu que le processus petit-fils (celui citer ci-dessus) se termine et j’ai récupéré son statut pour voir s'il y avait une erreur et si oui le programme complet s’arrêtait ici, sinon je faisais exactement pareil mais j’inversais, donc je ferme le premier descripteur créer à partir de mkstemp(3), je redirige dans mon fichier B et exécute la seconde commande. Ensuite le processus père ferme les deux descripteurs correctement et vérifie si aucune erreur n’a été recensé, sinon on supprime les fichiers temporaires et arrête le programme ici.

On peut utiliser le programme du deuxième exercice pour vérifier si la sortie de notre programme (./xxd) est la même que le programme du package xxd (xxd), donc on exécute « ./compare xxd compare.c -- ./xxd compare.c » et on regarde si notre programme à trouver que la sortie était identique.