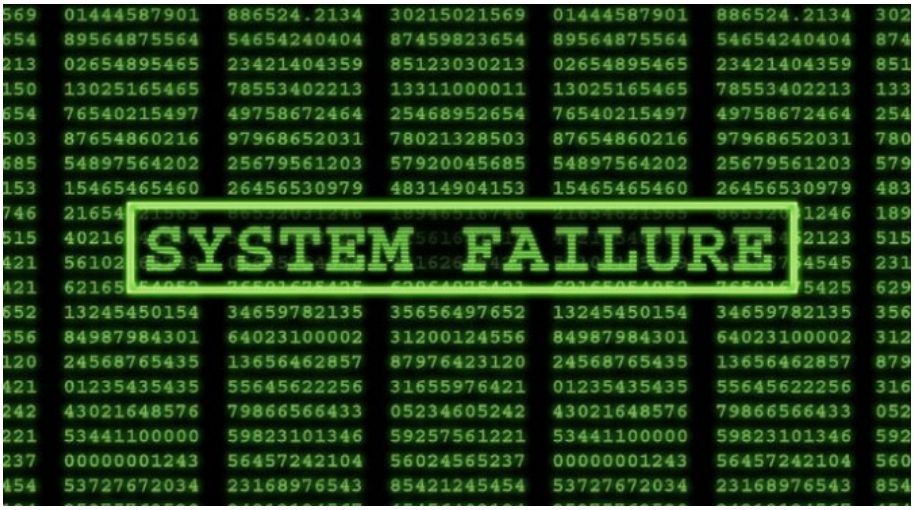
MONCHICOURT Baptiste SERNIT Jérémy

Rapport de Projet Système

CRÉATION D'UN SHELL EN LANGAGE C



TELECOM Nancy

Deuxième année (2017-2018)

Table des matières

Introduction

Le sujet de ce projet était l'implémentation d'un programme similaire à shell ou bash. Nous devions pouvoir exécuter des programmes, les principales fonctionnalités de rediriction, les pipe, l'utilisation d'un mode erreur et/ou historique, l'affichage d'une invite de commande et l'exécution de commandes en arrière plan.

1 Implémentation de l'algorithme

1.1 Saisie d'une commande

1.2 Segmentation et exécution d'une commande

Une fois la commande récupérée dans une chaîne de caractères on la segmente pour ensuite l'exécuter dans le bon ordre. La segmentation suit les étapes suivantes :

- appel récursif à gauche par sous-commandes séparées par ";" dans *execute* (on appelle *execute(sub_sentence, ...)* avec *sub_sentence* la première sous-commande, puis *execute(sentence+i+1, ...)* avec *sentence+i+1* qui pointe vers la suite de la commande) ;
- ensuite on parcourt de droite à gauche la sous-commande à la recherche des caractères spéciaux &, &&, |, ||, <, >, > > et on fait un appel récursif à droite cette fois-ci pour obtenir des sous-sous-commandes qu'on pourra exécuter avec *execvp* ;
- on transforme la chaîne de caractère en tableau de mots en utilisant les espaces comme séparateurs (remarque : il n'y a pas de caractère spécial à ce niveau de la segmentation, on reconnait donc une instruction du type ECHO A&ECHO B) ;
- on exécute la sous-sous-commande grâce à *execvp* en lui passant notre tableau en paramètre.

Les différentes fonctions utilisées pour ces étapes sont :

```
1 int execute(char*, int, int*, int, int*);
2 int testMethod(char*, char*, char*);
3 int isInString(char*, char);
4 int analyseInstruction(char*, int, int*, int, int*);
5 char** parseSentence(char*);
6 int execOperation(char**, int, int).
```

Pour l'exécution des commandes on vérifie qu'on n'est pas dans un cas particulier comme par exemple un *cd*, un *true* ou un *false*.

1.3 Logicals

Les deux fonctions dites "Logicals" sont celles correspondant aux symboles et ||. Elles sont respectivement implémentées par les fonctions *int executeIfFirstSucceeds(char*, char*, int, int*, int*);* et *int logicalOr(char*, char*, int, int*, int*);*.

On utilise un paramètre *int* returnCode* (pointeur vers un entier) pour savoir si on doit ou non exécuter le second argument, c'est-à-dire la commande à droite du symbole, en fonction du code de retour après l'exécution du premier arugment, c'est-à-dire la commande à gauche du symbole :

```
1  int executeIfFirstSucceeds(char* args1, char* args2, int errorMode, int* returnCode, int*
       pid) {
2    execute(args1, errorMode, returnCode, 1, pid);
3    if(*returnCode)
4      return analyseInstruction(args2, errorMode, returnCode, 1, pid);
5    else {
6      return 0;
7    }
8  }
9
10 int logicalOr(char* args1, char* args2, int errorMode, int* returnCode, int* pid) {
11   execute(args1, errorMode, returnCode, 1, pid);
12   if(!*returnCode) {
13     *returnCode = 1;
14     return analyseInstruction(args2, errorMode, returnCode, 1, pid);
15   } else return 0;
16 }
```

1.4 Pipes

Pour réaliser un pipe on réalise un *fork* et on redirige les entrées et sorties de manières à ce que la sortie du premier argument (la commande à gauche du symbole) arrive à l'entrée du second argument (la commande à droite du symbole).

On exécute la commande à gauche du symbole avec notre fonction *execute* qui prend une chaîne de caractères, et celle à droite du symbole avec un *execvp* après l'avoir parsée. C'est ce qui permet de faire la récursivité à droite sur nos sous-commandes.

1.5 Redirections

Pour les trois symboles de redirection on fait appel à la même fonction afin de factoriser le code, on applique celle correspondant au symbole entré en faisant une simple disjonction de

cas. Il nous suffit alors de faire un *fork* et réaliser les redirections d'entrée et sortie adéquates, puis à exécuter l'argument numéro un, c'est-à-direla commande à gauche du symbole.

1.6 Mode non-intéractif

Le mode non-intéractif est très similaire au mode interractif. La différence principale est qu'il n'affiche pas la phrase de prompt lorsqu'il détecte que l'entrée n'est pas un terminal.

1.7 Mode erreur

Le mode erreur a été un mode assez simple à implementer puisque nous retournions déjà un code erreur pour chaque fonction qui faisait arrêter ou non le programme complet. En n'utilisant pas le mode erreur, le programme affichait seulement un message d'erreur mais lors de l'utilisation du mode erreur, il renvoit une valeur de sorte à arrêter le programme comme avec la fonction exit.

1.8 Readline

Le mode readline utilise la même fonction que la fonction de lecture du terminal standard à quelques différences près, l'affichage du prompt et l'ajout à l'historique.

1.9 Extension facultative / Parser

Lors de la lecture des commandes, nous avons pu implémenter en même temps le parser des différentes commandes et mots clés. Il est alors possible de lire des commandes même si elle n'ont pas d'espace ou trop d'espace.

2 Difficultés rencontrées

2.1 Saisie d'une commande

Cette partie du programme a été plutot facile à implémenter et nous n'avons rencontré aucune difficulté. Nous avons passé un peu plus de temps à pouvoir lire les options.

2.2 Segmentation et exécution d'une commande

La principale difficulté a été de choisir une bonne stratégie de segmentation. La récursivité à gauche pour les " ;" était assez intuitif tandis que celle à droite pour les symboles spéciaux l'était bien moins, et cela nous a obligé à revoir totalement notre code une fois avoir décidé d'utiliser cette méthodologie.

Bien sûr, le C est un langage délicat et nous avons dû nous y reprendre à plusieurs reprises pour avoir un code fonctionnel sans problème de pointeur ou autre.

2.3 Logicals

Nous avions mal interprété l'action réalisée par ces fonctions dans un premier temps, et nous avons par conséquent dû nous y reprendre à plusieurs fois avant d'implémenter des fonctions qui renvoient bien le résultat attendu.

Aussi, la gestion d'un code de retour pour vérifier si la commande suivante devait ou non être exécutée a été un obstacle sur lequel nous avons un peu buté avant de décider d'ajouter un paramètre à nos fonctions pour les gérer plus facilement.

2.4 Pipes

Nous n'avons pas rencontré de difficulté particulière à part le fait que nous avons pas mal de temps à reussir à la faire fonctionner correctement. Aussi lors des tests, il y avait des fuites de fd.

2.5 Redirections

Lors de l'implémentations des redirections, nous faisions trop de fork à cause de l'execution de chaque sous commande, sans nous en rendre compte dans un premier temps. Après une analyse systématique du déroulement de l'algorithme, nous sommes parvenus à identifier le problème. Sa résolution s'en est suivie rapidement.

2.6 Mode non-intéractif

Malgré le fait que le mode non-intéractif et le mode interactif est similaire au niveau de l'implémentation. Les 2 modes ne donnaient pas toujours les mêmes résultats, surtout lors de l'utilisation des fonctions fork.

2.7 Readline

L'utilisation de la bibliothèque readline est plutôt simple. Notre difficulté avec cette fonctionnalité à implémenter a été le fait de devoir utiliser la bibliothèque dynamiquement grâce aux 2 fonctions dlopen et dlsym qui ont posé problème. Finalement, nous avons réussi à utiliser ces 2 fonctions et readline fonctionnait.

Conclusion

En conclusion, nous avons reussi à implémenter une bonne partie des fonctionnalités demandées : parser, redirections, pipes, readline, mode erreur, lecture d'un fichier, mode interractif ou non interractif, l'enchainement non fonctinnel des commandes. La seule fonctionnalité totalement non fonctionnelle est le retour en avant plan d'une commande auparavant executée en arrière plan. En ce qui concerne le nombre d'heures passées par chacun de nous est d'environ 60 heures puisque nous travaillions principalement ensemble.