TELECOM Nancy deuxième année (2018-2019)

Rapport du projet Réseaux et Systèmes : Réalisation de *Rsfind*

Auteurs :
Barloy NATHAN,
David FORLEN

Enseignant encadrant :
Nicolas SCHNEPF

Table des matières

1 Introduction **2**

2 Développement de Rsfind **3**

2.1 Gestion des options 3

2.2 Remplacement de `fprintf` 3

2.3 Parcours récursif de l'arborescence 3

2.4 Gestion des pipes dans l'option `--exec` 4

2.5 Gestion du chargement dynamique de libmagic 5

2.6 Expressions régulières 5

3 Gestion de projet **6**

3.1 Temps de travail 6

1 Introduction

Ce rapport présente le travail fourni afin de réaliser un outil de recherche multi-critères de fichiers. Cet outil est matérialisé par la commande : `rsfind`.

2 Développement de Rsfind

2.1 Gestion des options

Les options de la commande `rsfind` sont parsées grâce à la librairie *getopt*. La présence des options et leur valeur sont conservée dans une structure `Option` dont le pointeur sera passé à toutes les fonctions ayant un comportement dépendant de ces options.

2.2 Remplacement de `fprintf`

La fonction `printWrite` a été écrite afin de se passer de `fprintf`. Le squelette de la fonction est tirée du code posté sur le site : *http://www.firmcodes.com/write-printf-function-c/*. Celui-ci a été cependant corrigé pour éviter les fuites mémoires et dépassements de lectures. L'usage simpliste des fonctions `puts` et `putchar` ont été remplacés par l'écriture dans un buffer vidé en sortie de la fonction `printWrite` pour minimiser le nombre d'écriture dans une des sorties. Comme c'est l'appel à la fonction `write` qui écrit ce qui est à écrire par `printWrite`, il a été possible de permettre de définir à l'appel de `printWrite` la sortie sur laquelle écrire.

La fonction `printWrite` utilise donc deux variables importantes : `buffer`, qui est la chaîne de caractère qui est construite au cours du parcours des arguments de la fonction. Et `bufferParcours` qui pointe sur le caractère de `buffer` où écrire le prochain caractère lu.

Un des problèmes engendrés par le fait qu'on ait choisi de construire nous même la chaîne de caractère `buffer` à écrire, est qu'il faut également gérer les cas où l'on rencontre un *%s* dans la chaîne à écrire. En effet, `buffer` est alloué avant le parcours des chaînes de caractères à remplacer dans les différents % que contient la chaîne de caractère à écrire. Chaque écriture dans buffer est donc précédé par l'appel de la fonction `preventOverflow` écrite pour l'occasion et réallouant au besoin `buffer` pour s'assurer qu'il n'y ait pas d'écriture hors zone mémoire réservée. Cette fonction gère également le changement de l'adresse pointée par `bufferParcours`, pour qu'il continue à pointer au bon endroit dans `buffer`, même si ce dernier est réalloué à une adresse mémoire différente.

2.3 Parcours récursif de l'arborescence

Pour simplifier les opérations qui allaient avoir lieu sur les fichiers une fois le parcours fait, on a créé une nouvelle structure, qui contient les informations nécessaires pour la suite. Pendant assez longtemps, cette structure ne marchait pas, bien que le squelette soit fonctionnel. En effet, on utilisait la fonction strcat de la librairie string pour concatener 2 chaînes entre elles. Le soucis est que la chaîne de départ n'est pas réallouée, et des erreurs survenaient si les chaînes devenaient trop longues (par exemple, "./tests/testsEnvironnement"devenait "./tests/testsEnvironnemenA"). Il a donc fallu recréer une fonction pour concatener, afin de ne pas avoir d'erreurs.

Il y a en fait 2 structures créées : une structure `file` représentant un fichier et une structure `directory` représentant un répertoire. Cette structure est créée de manière récursive : on parcours les éléments contenus dans un dossier, et on les ajoutes en tant que fils du `directory` (on sépare les fils `directory` des fils `file`). Lors de cette étape, avant d'ajouter un fichier dans l'arborescence, on vérifie que celui-ci vérifie les propriétés entrées dans les options : si ce n'est pas le cas, il ne sera pas ajouté. De plus, si un fichier est vide, il ne contient pas de fichier recherché, donc on ne

va pas non plus l'ajouter dans l'arborescence. Il en résulte que l'arborescence obtenue à la fin ne contient que les chemins qui nous intéresse.

Il suffit ensuite de parcourir cette arborescence et d'afficher son contenu de la manière voulue

Il y a aussi eu une erreur de compréhension du sujet : on pensait que le `rsfind` sans argument était équivalent à un `ls`, et devait donc afficher les éléments dans l'ordre alphabétique. Or il est censé être équivalent au `find`, et l'affichage est donc différent.

2.4 Gestion des pipes dans l'option `--exec`

Avant de gérer les pipes dans l'option `--exec`, on commence par écrire la fonction `parseExecArgs` pour parser les chaînes de charactères en commande exécutables par `execvp`. Cela permet de construire un `char** argv` qui est un tableau dont les cases pointent vers les chaînes de caractères (les mots) de la commande à exécuter.

Il suffit alors de parcours la chaîne de caractère `charArgsAvantTraitement` et lancer le parseur `parseExecArgs` sur la chaîne de caractère comprise entre le dernier pipe rencontré, et le pipe qu'on rencontre à l'instant. Cela permet de former le `char*** pargv` dont les cases pointent sur les `char** argv` qui contiennent les commandes qui étaient séparées par un pipe. Le schéma mémoire d'un exemple de cette opération est présenté à la figure 1.

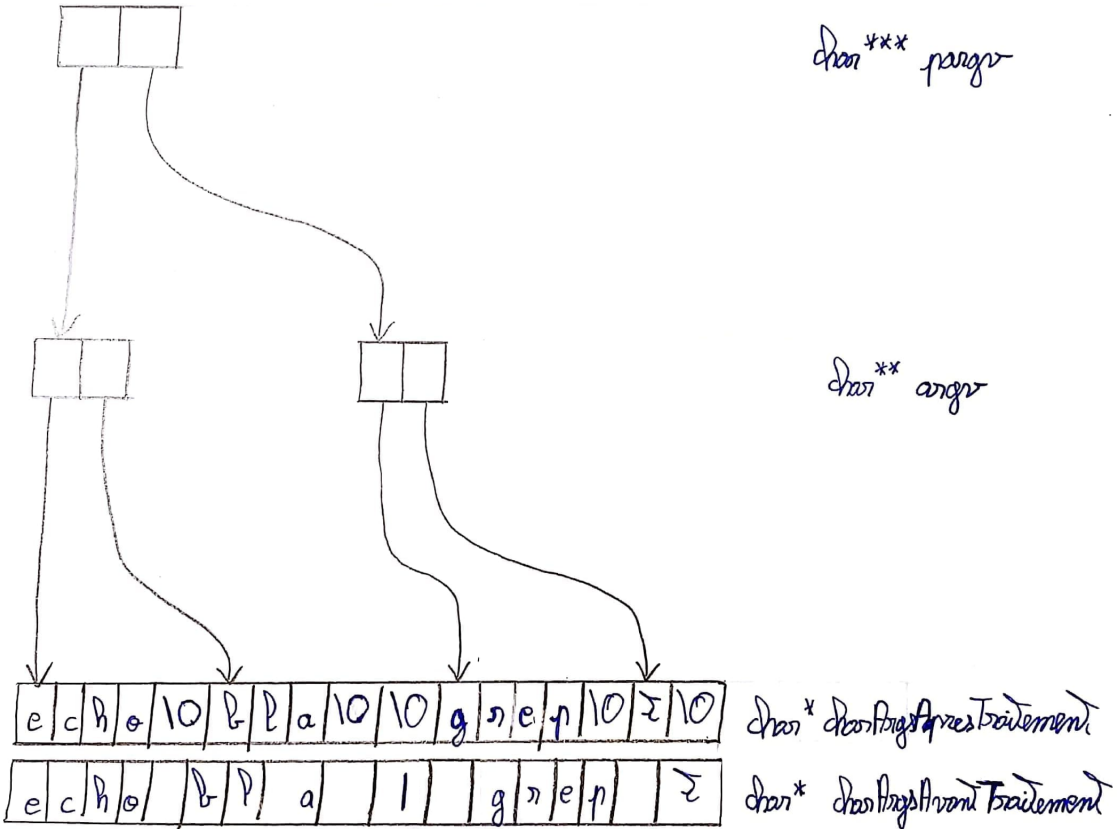

Figure 1: Schéma mémoire d'une char*** pargv

L'exécution de la commande contenue dans l'option `--exec` est gérée à l'aide d'un seul tube. En effet, lors du parcours des commandes séparées d'un pipe, on branche l'entrée standard du processus forké à la sortie standard de la commande précédente (la sortie du tube, ou l'entrée standard de notre programme, si la commande est la première à être exécutée). On branche la sortie standard de ce processus à la l'entrée du tube s'il y a d'autres commandes à exécuter ensuite, on la branche à la sortie standard du programme sinon.

2.5 Gestion du chargement dynamique de libmagic

Pour ne charger la librairie *libmagic* en mémoire que lorsque `rsfind` est appelé avec l'option `-i`, on utilise la librairie *libpcre*. Celle ci nous sert donc à charger toutes les fonctions de *libmagic* dont on se sert dans le programme. Les pointeurs vers ces fonctions sont stockés dans une structure `symbolsLibMagic*` dont le pointeur sera passé à la fonction nécessitant l'usage des fonctions de cette librairie.

2.6 Expressions régulières

Pour reconnaître les expressions régulières, on va créer une nouvelle structure qui représente un élément d'une expression régulière, et qui a comme frère un autre élément, qui représente la suite de l'expression. On doit donc créer une fonction qui va parser une chaîne de caractères pour créer une structure précédemment décrite, et une fonction pour regarder si une chaîne de caractère qui correspond a une expression régulière.

Il est difficile de prendre en compte toutes les éléments pouvant être présents dans les expressions régulières, notamment les caractères d'échappement, comme '.' ou '*', qui ont une déjà une signification.

3 Gestion de projet

3.1 Temps de travail

Le tableau suivant recense le nombre d'heures de travail passées par chacun sur les principales tâches identifiées au cours du projet.

Tâche	David	Nathan
Gestion du code et de l'environnement de développement	1	
Etape 1 : parseur	2,5	
Structure d'arborescence des fichiers		5
Etape 2 : parcours d'un dossier		2
Etape 3 : parcours récursif d'un dossier		3
Etape 4 : listing détaillé		1
Etape 5 : Recherche de texte	2	
Etape 6 : Recherche d'image	3	
Etape 7 : Exec	11	
Tests	6	4
printWrite sans utiliser fprintf	3	
Extension 1 : librairie chargée dynamiquement	3	
Extension 2 : expressions régulières		7
Corrections de bugs divers	4	6
Rédaction du rapport	4	1
Total (en heure)	37,5	30

Figure 2: Répartition du travail