ENSEIRB-MATMECA

Projet S5

Filière Informatique

Shakespearian Monkeys

Auteurs :
M Enguerran Beauvillard
Mme Sonia Ouedraogo

Superviseur :
M Mathieu Faverge

14 Décembre 2018

Table des matières

1 Version de base (a.k.a achievement 0) 3
- 1.1 Objectifs . . . 3
- 1.2 Difficultés rencontrées . . . 3
- 1.3 Solutions . . . 3
  - 1.3.1 Choix de la structure file . . . 3
  - 1.3.2 Alternative au "malloc" . . . 4
  - 1.3.3 Gestion des singes . . . 4
  - 1.3.4 Implémentation des fonctions et analyse de complexité 5

2 Achievement 1 8
- 2.1 Objectifs . . . 8
- 2.2 Difficultés rencontrées . . . 8
- 2.3 Solutions . . . 8
- 2.4 Modifications des fonctions singes . . . 9

3 Mise en oeuvre et réalisation 11
- 3.1 Utilisation du "Header" . . . 11
- 3.2 Réalisation de la fonction principale : le jeu . . . 11
- 3.3 Conception du Makefile . . . 12
- 3.4 Description des tests de validation . . . 12

Introduction

Dans le cadre de ce projet, nous retraçons la démarche d'un groupe de singes cherchant à produire du texte. Ici nous nous basons sur un extrait de Shakespeare rédigé en anglais (indeed). Le groupe de singes étant organisé, des rôles leur sont associés, ainsi nous retrouvons parmis eux les singes lecteur, statisticien, imprimeur et écrivain. Dans un système de tours de jeux, les primates vont alterner entre travail et grève en fonction de l'avancée de leurs collègues. A chaque tour, un singe actif est appelé aléatoirement pour effectuer sa tâche. Le jeu se termine lorsqu'un certain nombre de tours est atteint ou si tous les singes se retrouvent en grève. Notre travail se décompose en deux parties, la version de base comportant seulement trois singes, et l'achievement 1 ajoutant un nouveau type de singe ainsi que des modifications pour les trois précédents.

Cadre de travail

La réalisation de ce projet s'est étalée sur huit semaines à raison de huit heures par semaine sous la tutelle d'un encadrant auquel il faut ajouter le temps de pratique libre non encadrée. Pendant les séances nous avons chercher à subdiviser le travail et ainsi réaliser des fonctions auxiliaires sur lesquelles nous nous appuierions pour le projet global. Nous avions aussi à notre disposition une plateforme en ligne (git) où nous pouvions mettre en commun le travail du binôme, et permettant aussi de rendre notre travail final. Par une série de tests la forge git nous permettait aussi de voir où nous en étions et si nous validions au fur et à mesure le travail demandé. Par ailleurs ce dernier devait être réalisé intégralement en langage C et était principalement axée sur l'utilisation des structures et des pointeurs.

1 Version de base (a.k.a achievement 0)

1.1 Objectifs

L'objectif premier est de lire les mots d'un fichier texte et pour ensuite les afficher dans la sortie standard tout en comptant entre autres les multiplicités de chacun de ses mots. Ainsi n'interviennent dans cette partie que les singes lecteur, statisticien et imprimeur. La réalisation de cette version peut être divisée en différentes parties. Dans un premier temps, il nous est demandé de récupérer un à un les mots du fichier texte fourni pour ensuite les traiter et notamment compter leur multiplicités. Enfin les mots doivent être affichés après le traitement, une fois de plus un à un. Ces trois étapes correspondent aux rôles des singes évoqués précédemment.

1.2 Difficultés rencontrées

Pour répondre à ces problématiques, il nous a fallu commencer par choisir comment implémenter la file qui allait être utilisée par les singes. La structure de file était en effet imposée, nous n'avions pas le choix que de l'utiliser, mais nous devions faire en sorte de l'implémenter de façon à réduire la complexité des fonctions enfiler et défiler. Ensuite nous avons dû faire un choix sur la structure à adopter pour caractériser les singes et gérer le travail de chacun. Par ailleurs, outre le fait que le choix du singe devant travailler à un tour devait être aléatoire, force est de constater que leurs travaux étaient aussi inter-dépendants. En effet, cela nous a confronté à la difficulté de faire savoir à un singe "x" qu'un singe "y" avait déjà travaillé (par exemple faire comprendre au singe imprimeur que le singe statisticien avait ou n'avait pas encore traité un mot).

1.3 Solutions

Les contraintes évoquées dans le paragraphe précédent nous ont donc menés à faire des choix particuliers d'implémentations que nous allons défendre ici. Puis nous détaillerons ensuite notre procédure de réalisation du travail des différents singes.

1.3.1 Choix de la structure file

La structure de cellule qui nous était fournie était la suivante :

```
1 struct cell {
2     char word[MAX_WORD_LENGTH];
3     int noccs;
4     struct cell* next;
5 }
```

Pour notre première approche, nous avions implémenter la file à travers une structure contenant un pointeur noté first_cell (initialisé à **NULL**) vers la première cellule. Chaque cellule ayant un pointeur vers la cellule suivante, nous formions ainsi une structure chaînée. Ainsi il nous était facile de réaliser la fonction enfilage (puisqu'il suffisait de faire pointer la dernière cellule vers la nouvelle cellule) et la fonction de défilage (puisqu'il suffisait de modifier first_cell vers ma cellule suivante). Cependant, nous avons constaté qu'avec cette structure, la fonctions d'enfilage s'effectuait en complexité temporelle linéaire, puisqu'il fallait parcourir la file jusqu'à la dernière avant de pouvoir ajouter la nouvelle. Pour réduire cette complexité, nous avons décidé de modifier notre structure de file en y ajoutant un deuxième pointeur qui lui amène directement sur la dernière cellule. Avec une telle structure nous pouvions désormais enfiler en temps constant étant donné que le parcours de la file n'était plus nécessaire.

1.3.2 Alternative au "malloc"

Pour allouer un zone mémoire pour la création des structures, le sujet nous impose l'utilisation d'un tableau de structure (nommé "pool") dont nous définissons la taille au préalable. Ainsi il est primordial de vérifier la place restante dans la pool avant chaque ajout de cellule dans la file. Cette solution est certes efficace et contourne l'utilisation du malloc, mais nécessite aussi une bonne gestion de notre complexité spatiale, qui peut être très rapidement énorme et problématique si nous réservons plus d'espace qu'il n'en faut. A noter qu'une solution pour réduire cette complexité aurait été de recycler les cellules défilées, mais par manque de temps, ceci n'a pas été fait dans notre cas.

1.3.3 Gestion des singes

Pour ce qui est de la gestion de nos singes travailleurs, nous avons travailler à l'aide de la structure suivante :

```
1 struct monkeys {
2     int reader
3     int printer
4     int statistician
5     int number_active_monkeys
6 };
```

Les trois premiers entiers représentent l'état du singe associé, ayant la valeur 1 si le singe est actif, 0 si le singe est en grève. Nous avons choisi de faire une seule structure contenant tous les singes, car comme évoqué plus haut, les travaux des singes sont interdépendants. Ainsi un singe après son travail pourra réveiller un autre en changeant simplement la valeur du statut de celui-ci. Par ailleurs, au lieu de faire une fonction qui calcule à chaque fois le nombre de singe actifs nous avons choisi d'inclure un entier représentant ce nombre dans la structure des singes, et de simplement l'incrémenter ou la décrémenter lors du changement d'état d'un singe.

1.3.4 Implémentation des fonctions et analyse de complexité

Le singe lecteur

Le travail du singe lecteur est de lire le fichier texte à chaque fois qu'il est appelé à jouer, puis de compléter sa file avec une copie de ce mot tout en comptant le nombre de mots lus. Il part en grève si l'une des deux conditions suivantes et vérifiées : soit il arrive à la fin du fichier texte fourni, soit la file qu'il remplit ne peut plus être allongée (ce qui signifie que l'espace allouée dans la pool est épuisée). Ce qui suit est le prototype de la fonction du singe lecteur que nous avons mis en place.

```
1 void reader_work(struct queue* queue_read,
2                   struct monkeys* monkeys,
3                   FILE* file,
4                   int* number_word_read);
```

La complexité en temps de cette fonction est relativement faible voire constante étant donné qu'elle n'appelle que la fonction enfiler. Pour lire le fichier, nous avons préféré utiliser la fonction fscanf plutôt que fgetc car cette dernière lit un fichier caractère par caractère, ce qui n'est pas pratique ici. En

effet, il nous aurait alors fallu tester nous même la séparation entre chaque mot pour ensuite les reconstituer. Une méthode qui aurait donc mener à l'augmentation de la complexité temporelle, ce qui est loin d'être souhaitable. De plus l'avantage de fscanf est que la fonction lit non seulement le fichier mot par mot, mais en plus place ensuite le curseur au début du mot suivant, ce qui est préférable pour la prochaine étape de lecture. On teste en outre la fin du fichier avec le code de retour de fscanf. Par ailleurs concernant la complexité en espace de la fonction reader_work, elle était dans un premier temps conséquente (de l'ordre de la taille du fichier) puisqu'à chaque appel du singe lecteur nous créions une nouvelle chaîne de caractère dans laquelle nous copions le mot lu afin de l'enfiler. Pour remédier à cela, nous avons redéfini cette chaîne de caractère sous forme d'une variable statique. Ainsi à chaque appel du singe lecteur, il nous était possible de réutiliser cette même variable.

Le singe statisticien

Le rôle du singe statisticien quant à lui est de lire un par un les mots de la file du singe lecteur et de compter leur multiplicité. Ainsi son travail est quelque peu semblable à celui de son collègue lecteur en ce sens que lui aussi remplit une file avec des cellules contenant des mots. Mais une première différence est qu'il faut veiller au fait que le statisticien ne se contente pas d'enfiler tous les mots, en effet il regroupe les doublons (étant donné son but de compter le nombre d'occurrences de chaque mot). Pour cela, pour chaque mot traité, il a fallu d'abord vérifier qu'il n'était pas déjà présent dans la file du singe statisticien avant de faire appel à la fonction d'enfilage, ou dans le cas contraire simplement augmenter l'occurrence dans la cellule associée à ce mot. Pour effectuer cette fonctionnalité nous avons donc créé la fonction manage_queue_stat. La complexité en temps de cette fonction est linéaire ce qui est le cas idéale puisque dans tous les cas il nous faut parcourir la file.

```
1 void stats_work(struct queue* queue_stat,
2                 struct queue* queue_read,
3                 int* number_diff_word,
4                 struct monkeys* monkeys);
```

Une seconde contrainte s'impose dans le travail du singe statisticien qui le diffère du lecteur. En effet celui-ci lit les mots dans la file du singe lecteur

mais ne les en sort pas, il lui faut donc trouver un moyen d'avancer dans la file du lecteur pour à chaque fois lire le mot suivant. Cela entraîne que contrairement à la méthode utilisé par le lecteur, ici il ne sera pas possible de simplement lire le premier élément de la file. Une première implémentation pourrait être d'inclure dans les cellules du lecteur un élément (un entier par exemple que l'on modifierait) qui permettrait de marquer le passage du statisticien. Ainsi à chaque tour il aurait à parcourir la file jusqu'à la prochaine cellule non marquée. Cependant ce qui nous a découragé à utiliser une telle méthode est une fois de plus la complexité linéaire qu'elle aurait générée, car il s'avère possible de faire encore mieux. On a préféré utiliser plutôt un pointeur statique qui mémoriserait l'adresse de la prochaine cellule à analyser par le statisticien. Ainsi à chaque tour le singe n'a plus qu'à traiter directement la cellule pointée, puis à déplacer le pointeur sur la cellule suivante et se mettre en grève quand le pointeur est **NULL**, car cela indiquerait une file vide ou que le parcours de la file est tout simplement terminé.

Le singe imprimeur

Contrairement à ses collègues, l'imprimeur est le seul des trois primates à ne pas enfiler de cellules, mais plutôt à les défiler. Effectivement le rôle du singe imprimeur est, comme déjà mentionné plus haut, d'afficher sur la sortie standard les mots de la file du lecteur. Mais un élément vient un peu compliquer son travail : l'imprimeur ne peut afficher que les mots de la file du lecteur qui ont aussi déjà été traités par le statisticien. Si au préalable, l'idée d'un entier dans la structure cellule pour marquer le passage du statisticien avait été évincée, il se trouve que dans ce cas là elle s'avère très utile. Ainsi pour permettre à l'imprimeur de se repérer, nous avons quelque peu modifier les cellules en y introduisant un entier **status** auquel le statisticien donnerait la valeur 1 à chaque traitement. Il nous est donc maintenant possible d'identifier précisément les mots qui sont imprimables par le singe imprimeur. Dernière tâche assumée par ce travailleur acharné, il est aussi possible d'obtenir le nombre de mots imprimés grâce à l'entier **printer_counter**

```
1 void printer_work(struct queue* queue_read,
2                    int* printed_counter,
3                    struct monkeys* monkeys);
```

2 Achievement 1

2.1 Objectifs

Avec l'achievement 1 arrivent des modifications sur le travail de nos trois singes ainsi que l'apparition d'un quatrième primate ; le singe écrivain. En plus d'étudier le texte fourni originalement, il sera maintenant possible d'écrire du neuf.

La plus importante d'entre elles est le fait qu'à partir de maintenant, le singe statisticien doit, en plus de compter les mots, compter et retenir ceux qui le suivent. Cette modification permet la création d'un quatrième type de singe, le singe écrivain

2.2 Difficultés rencontrées

La majeure difficulté rencontrée vient de la nouveauté apporté au singe statisticien. Effectivement, celui-ci, en plus de compter les mots, doit maintenant compter et retenir ceux qui les suivent. Cette modification nous a forcé à chercher une méthode adaptée au problème. Évidemment en plus de cela il a aussi fallu modifier les fonctions des autres singes pour qu'eux aussi répondent à ce qui était maintenant demandé.

2.3 Solutions

Plus qu'une modification sur la fonction du singe statisticien, cette nouveauté nous a surtout amené à modifier la structure de cellule que nous utilisions dans l'achievement 0. Maintenant chaque cellule comporte une file composée de tous les mots successeurs du mot associée à cette cellule. Globalement cela nous donne donc une file composée de cellules contenant elles-mêmes chacune une file.

```
1  struct cell{
2    char word[MAX_WORD_LENGTH+1];
3    int noccs;
4    int number_foll_word;
   struct queue queue_follower;
   struct cell* next;
5    };
```

2.4 Modifications des fonctions singes

Évidemment l'ajout d'un nouveau singe a entraîné l'ajout d'un entier lui étant associé dans la structure **monkeys**.

Le singe écrivain

Le rôle du singe écrivain est de produire du nouveau texte à partir d'un mot choisi aléatoirement parmi ceux traités par son collègue statisticien. A chaque nouveau tour il continue d'écrire en choisissant (toujours aléatoirement) parmi les successeurs du mot précédent. De plus il termine ses phrases s'il ne trouve pas de successeurs à son mot ou s'il obtient un nombre aléatoire de probabilité 0.1. Cette gestion de la fin des phrases et de la ponctuation nous a d'ailleurs imposé un entier statique identifiant la longueur de la phrase en cours d'écriture, en effet il était par exemple imposé de ponctuer les phrases ne contenant qu'un seul mot par un point d'exclamation. Ce singe est inactif lors des cents premiers tours et n'entre en grève que dans le cas où il n'y a plus de mots dans la file du statisticien.

```
1 void writer_work(struct queue* queue_stat,
2                   struct queue* queue_writer,
3                   int* number_diff_word,
4                   struct monkeys* monkeys);
```

La complexité de cette fonction était dans le pire des cas linéaire puisque si le mot aléatoire choisi était tout au bout de la file du statisticien, il fallait donc la parcourir entièrement.

Le singe lecteur

Le singe lecteur est le seul singe a ne subir aucun changement lors de l'achievement 1.

Le singe statisticien

Par rapport à la version de base, le statisticien de l'achievement 1 lit les mots de la file du lecteur mais pour chaque mot il enfile aussi ses successeurs. Ainsi dans sa file, on doit retrouver un mot et chaque mot qui le suit à chacune de ses apparitions dans le fichier. Pour implémenter cela, nous avons inclue

dans chaque cellule une file **queue_follower** qui contiendrait les successeurs. Il a fallu donc introduire une deuxième fonction auxiliaire par rapport à la version de base :

```
1 void manage_successor_queue(struct queue* queue_stat,
2                              char* precedent_word,
3                              char* next_word);
```

Cette fonction prend en paramètre deux mots, cherche la file de successeurs du premier et y ajoute le second. Elle est de complexité en temps linéaire dû à la fonction de recherche (cette dernière aurait pu être moins coûteuse si nous avions dès le début des files doublement chaînées). Pour permettre au singe statisticien d'enfiler et les mots et leurs successeurs,nous avons utilisé la méthode suivante : **lire un mot X dans la file de lecteur, l'enfiler sur la file du statisticien et si le mot X a un précédent Y dans la file du lecteur (autrement dit si il n'est pas le premier mot de la file), l'enfiler également comme successeur du mot Y dans la file du statisticien**. Mais comment savoir si un mot a un précédent ET surtout comment avoir accès à ce précédent ? Nous avions pensé à modifier nos files pour utiliser des files doublement chaînées, on aurait ainsi un pointeur vers la cellule précédente. Mais cela n'aurait résolu qu'en partie notre problème, car en effet dans l'achievement 1 le statisticien doit défiler au fur et à mesure qu'il traite une cellule du lecteur. C'est pourquoi nous avions finalement opter pour l'usage d'une chaîne de caractère statique, initialisé avec "-" et a chaque tour contient le mot que le statisticien vient de lire chez le lecteur. Ainsi il n'avait qu'à vérifier que cette chaîne était différente de "-" et alors faire appel a la fonction manage_successors_queue Globalement la complexité en temps de la fonction stats_work est donc cette fois ci linéaire.

Le singe imprimeur

Avec l'arrivée de l'achievement 1 le singe imprimeur possède maintenant sa propre file commune avec l'écrivain. En plus de cela, la différence notable avec la version précédente du primate est que lorsque celui-ci traite un mot, il se contente de l'imprimer sans l'enlever de la file. Pour palier à cette restriction nous avons donc utiliser la même méthode que celle utilisée pour le statisticien ; en utilisant un pointeur qui mémorise sa position dans la file de l'écrivain. Ainsi il ne parcours pas toute la file ce qui fournit une complexité en temps constant.

3 Mise en oeuvre et réalisation

3.1 Utilisation du "Header"

Les différentes fonctions évoquées précédemment ont été implémentées dans des fichiers .c différents. Pour gérer le travail des singes (permettre à l'un d'entre eux d'avoir accès aux données des autres) et par la suite réaliser la fonction principale du jeu, nous avons utiliser des fichiers .h composés des prototypes des fonctions pour chacun des fichiers .c. Cela permet notamment d'obtenir des fichiers plus clair pour la compréhension de la structure de notre projet.

3.2 Réalisation de la fonction principale : le jeu

Que ce soit pour la version de base ou le premier achievement, la fonction principale du jeu est codée dans un fichier .c incluant tous les fichiers .h des fonctions auxiliaires. Ici nous détaillerons sa terminaison sa correction et sa complexité.

Terminaison

Deux éléments peuvent mettre fin au travail des singes : l'échéance du nombre de tours de jeu ou la mise en grève de tous nos primates. Même en supposant que tous les singes ne vont jamais en grève (ce qui n'est d'ailleurs pas le cas) on est sûr que la fonction se termine puisque le nombre de tours maximale est constant et la variable turn augmente à chaque tour de jeu, ainsi MAX_TURN-turn est strictement décroissante et à valeurs positives donc minorée.

Correction

Ce qui est attendu à la fin du main, c'est le bilan du jeu à savoir le nombre de mots lus, les mots imprimés et leur nombre, le nombre de mots différents. . .Nous avions tester le jeu pour différents fichiers.txt et lancer avec différentes valeur du seed pour s'assurer que ces données étaient exactes. Ayant tester au préalable la correction des fonctions auxiliaires, la fonction main n'a été qu'une mise en commun.

Complexité

Nous avions dit que la complexité en temps de reader_work était constant, celle de stat_work linéaire, celle de printer_work constant ete celle de writer_work linéaire. On est déduit donc que pour les deux achievements la complexité en temps est globalement polynomiale d'ordre 2 dépendant de la taille de la pool initiale. En espace la complexité est proportionnelle à l'espace allouée dès le début par la pool.

3.3 Conception du Makefile

Pour compiler l'entièreté de notre projet et gérer les dépendances inter-fonctions nous avons utiliser un fichier de type makefile. Ainsi pour chaque .c il existe une règle générant le .o associé, .o qui seront utilisés pour générer le binaire final.

3.4 Description des tests de validation

Des tests en ligne sur la forge nous permettait d'avancer dans le projet en respectant les contraintes. Ces tests étaient au nombre des 8. Le test **make** vérifiait la compilation du code. En suite s'en sont suivis six tests qui vérifiaient les résultats attendus. D'une part le fichier texte ne devait être attendu forcément à la première place dans les paramètre. Pour palier à cela, nous avons utiliser la fonction parse_op qui renvoyait la position du premier argument non optionnel. Ensuite l'appel du programme sans paramètre ainsi qu'avec un fichier vide ne devait pas générer de code d'erreur. Nous avons vérifié ces points à travers des assertions dans le code. Un autre test vérifiait la correction du code en s'assurant par exemple que le number_word_read était sans erreur. Pour finir une commande test nous permettait de vérifier la correction de nos fonctions. Nous avons pu tester les fonctions principales comme enfiler, défiler, le déplacement des pointeurs ...

Conclusion

Ce travail a été l'occasion d'apprendre non seulement des connaissances dans le domaine informatique, que ce soit en langage C ou en manipulation de git, mais aussi d'apprendre à travailler en binôme. La répartition des tâches ainsi que le travail en parallèle est effectivement l'un des points clés de ce projet. Nous avons appris de nos erreurs et il est fort à penser que le travail accompli ne sera que plus conséquent pour le prochain projet, puisque celui-ci pourra s'effectuer sur des bases plus solides.