UFR de **mathématique** et d'**informatique**
Université de Strasbourg

Files chaînées dynamiques

Cette séance de Travaux Pratiques fait suite à la première séance portant sur les files FIFO (*First In First Out*). Il s'agit cette fois-ci d'implémenter ces files sous la forme de chaînes. Vous trouverez sur Moodle une archive contenant un certain nombre de fichiers pour vous aider à démarrer. Après l'avoir téléchargée :

```
$ tar -xf sda1-tp2.tar.gz
$ cd sda1-tp2
```

Les seuls fichiers qu'il est nécessaire de modifier sont les fichiers `file_chaine.c` et `file_chaine_optimisee.c`. Le fichier `Makefile` contient les règles nécessaires pour compiler deux programmes interactifs respectivement intitulés `chaine` et `chaine-optimisee`.

```
$ make
$ ./chaine # ou ./chaine-optimisee
```

Le comportement de ces deux programmes est identique à celui des programmes du TP1. Pour rappel, le programme crée une file vide puis attend des instructions de la forme :

- `aide` : Affiche une aide.
- `ajoute <x>` : Ajoute l'entier x à la file.
- `suppr` : Affiche l'entier en tête de file et le supprime de la file.
- `etat` : Affiche "vide" ou "non-vide" selon l'état de la file, et sa taille si elle n'est pas vide.
- `affiche` : Affiche le contenu de la file.
- `quit` : Détruit la file et termine le programme.

Une fois de plus, vous devrez proposer deux implémentations différentes. On choisit cette fois-ci la même interface par mutation à apparence fonctionnelle dans les deux cas. C'est donc le même fichier `main.c` et surtout le même header (voir le fichier `file.h`) qui sont utilisés pour les programmes `chaine` et `chaine-optimisee` :

```c
struct file;                          /* Déclaration de la structure pour une file. */
typedef struct file file;             /* Typedef pour une file. */
file* file_nouv();                    /* Nouvelle file vide. */
file* file_adjq(file* f, S x);        /* Ajoute un élément x en queue. */
file* file_supt(file* f);             /* Supprime l'élément en tête. */
S file_tete(const file* f);           /* Retourne l'élément en tête. */
bool file_estvide(const file* f);     /* Teste si une file est vide. */
size_t file_taille(const file* f);    /* Retourne le nombre d'éléments. */
S file_ieme(const file* f, size_t i); /* Retourne le i^ième élément. */
void file_detruit(file* f);           /* Détruit une file. */
```

1 Première implémentation de files chaînées

Modifiez le fichier `file_chaine.c` (et uniquement celui-ci) pour implémenter les files chaînées. Le programme `chaine` produit par le `Makefile` utilisera cette implémentation. La structure proposée (voir également la Figure 1) est la suivante :

```
struct file
{
  /** \brief Élément. */
  S element;
  /** \brief Chaînon suivant. */
  struct file* suivant;
};
```

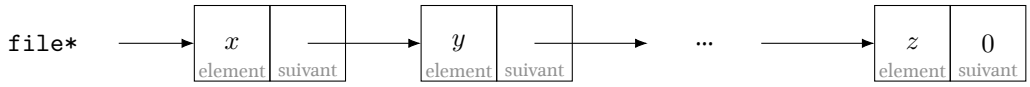

FIGURE 1 – Représentation graphique d'une file chaînée

Lorsque l'on représente une file sous la forme de structures chaînées, il est d'usage de représenter une file vide par un pointeur nul. Voir également la Figure 2 pour l'ajout en queue et la Figure 3 pour la suppression de la tête.

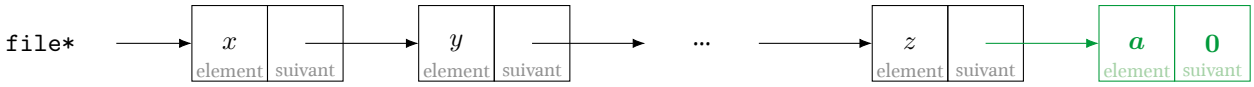

FIGURE 2 – Ajout de l'élément a en queue de file.

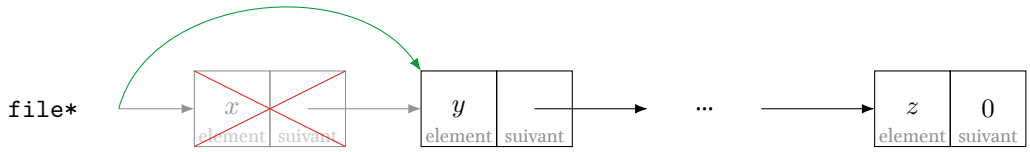

FIGURE 3 – Suppression de la tête

2 Comparaison avec le TP1

Le défaut de l'implémentation précédente est qu'il faut souvent parcourir complétement la chaine de structures créée en mémoire, ce qui peut être relativement lent. Le but de cet exercice est de faire des mesures de performance afin de mettre ceci en évidence. Commencez par créer un fichier texte `bigtest` [1] qui contient les commandes à passer pour :

- ajouter 10000 éléments à la file
- supprimer ces éléments
- tout en demandant un "état" **après chaque commande** (!)

Pour ce faire, vous pouvez par exemple utiliser les commandes `printf(1)`, `seq(1)` et la structure de contrôle `for` de votre shell :

```
$ rm -rf bigtest
$ for i in $(seq 10000); do printf "ajoute ${i}\netat\n" >> bigtest; done
$ for i in $(seq 10000); do printf "supprime\netat\n" >> bigtest; done
```

1. La création de ce fichier peut prendre beaucoup de temps sur un système de fichiers distribué. Dans ce cas, remplacez le chemin `bigtest` par le chemin `/tmp/$USER-bigtest` (la plupart des systèmes d'exploitation récents placent `/tmp` dans la mémoire vive).

La commande `time(1)` permet de mesurer le temps d'exécution d'un programme. Recompilez les programmes du TP1 en fixant `TAILLE_MAX` à `50000` puis utilisez la commande `time(1)`.
Note : l'option `-C` de la commande `make(1)` de l'éxecuter depuis un autre répertoire.

```
$ make -C <répertoire du TP1> clean
$ make -C <répertoire du TP1> CPPFLAGS='-DTAILLE_MAX=50000'
$ time <répertoire du TP1>/mutation bigtest >/dev/null
$ time ./chaine bigtest >/dev/null
```

Que constatez-vous ?

3 Optimisation

Les files avec tableau de taille statique ont l'avantage du temps d'exécution très rapide en comparaison aux files chaînées, en particulier en raison de la nécessité de parcourir entièrement les structures chaînées en mémoire lors de certains appels. Mais elles présentent l'inconvénient d'être limitées en taille, ce qui peut être problématique lorsqu'on souhaite écrire un programme générique, dont le nombre d'éléments à stocker dans la file n'est pas borné à priori.

Le but de cet exercice et de créer une structure chaînée optimisée pour les appels à `file_taille()` et à `file_adjq()`. La structure proposée est la suivante (voir également Figure 4) :

```
struct chainon
{
  /** \brief Élément de type S */
  S element;
  /** \brief Pointeur sur le suivant */
  struct chainon* suivant;
};
typedef struct chaine chainon;

struct file
{
  /** \brief Tête de file chainée */
  chainon* tete;
  /** \brief Queue de file chainée */
  chainon* queue;
  /** \brief Nombre d'éléments dans la file */
  size_t taille;
};
```

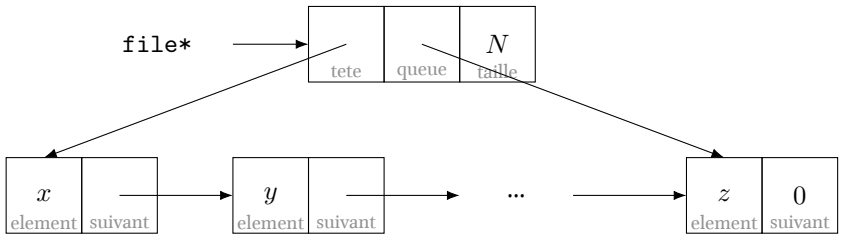

FIGURE 4 – Représentation graphique d'une file chaînée optimisée

Cette fois-ci, la file pointe vers une structure intermédiaire qui contient des pointeurs aussi bien vers la tête que vers la queue de la file. Modifiez le fichier `file_chaine_optimisee.c` (et uniquement celui-ci) pour implémenter les files de cette manière. Comparez ensuite cette implémentation aux implémentations précédentes comme dans l'exercice précédent.

A Types opaques

En observant le code fourni, vous constaterez que la structure `struct file` est défnie dans les fichiers `file_chaine.c` et `file_chaine_optimisee.c` tandis que le fichier `file.h` ne contient que les lignes suivantes :

```
struct file;
typedef struct file file;
```

Il s'agit d'un ***type opaque***. Le choix de l'implémentation se fait dans le fichier `Makefile`, à l'édition de liens, en fonction du fichier objet fourni :

```
chaine: main.o file_chaine.o
chaine-optimisee: main.o file_chaine_optimisee.o
```

Procéder ainsi présente plusieurs avantages :
— le code client ne peut pas modifier directement la structure.
— le code client est forcé d'utiliser les fonctions associées à la structure.
— *changer d'implémentation* ne nécessite de remplacer qu'un seul fichier `.c` (ou fichier objet `.o`).

B Fuites mémoires

Pour rappel ou information, vous pouvez vérifier l'absence de fuites mémoires à l'aide d'outils tels que `valgrind(1)` (voir Section B.1) ou bien en activant des options d'instrumentation (voir Section B.2).

B.1 Valgrind

Compilez vos programmes avec les informations de *debugging*, puis éxécutez `valgrind(1)` en passant en paramètre le programme cible et ses options :

```
$ make -B CFLAGS="-Og -ggdb"
$ valgrind ./chaine <tests/jeutest1.txt
$ valgrind --leak-check=full ./chaine <tests/jeutest1.txt
```

B.2 Instrumentation

Compilez vos programmes en ajoutant l'option d'instrumentation `-fsanitize=address` puis éxécutez votre programme de la manière classique :

```
$ make -B CFLAGS="-fsanitize=address" LDFLAGS="-fsanitize=address"
$ ./chaine <tests/jeutest1.txt
```