TP de Programmation : debugger

Deboguer, instrumenter, analyser

1 - Utiliser `gdb`

`gdb` est un programme permettant d'analyser d'autres programmes. En particulier il permet d'observer finement l'exécution d'un programme et de trouver les erreurs qu'il contient (on parle alors de deboguage). Pour pouvoir utiliser `gdb` sur un programme écrit en C sous Unix, il faut obligatoirement compiler ce programme avec l'option `-g`.

`gdb` est très pratique par exemple pour repérer précisément les erreurs système classiques qui se produisent à l'exécution :

- Segmentation Fault : accès non autorise a une zone mémoire. Se produit lorsqu'on accède a de la mémoire qui n'est dans aucune des trois zones de notre programme (généralement lorsqu'on dépasse les bornes d'un tableau, ou que l'on déréférence un pointeur qui n'a pas été initialise).
- Bus error : comme précédemment mais en plus grave, dans ce cas la valeur d'un pointeur que l'on déréférence n'est pas une adresse valide du système.
- Floating exception : se produit lors d'un calcul impossible sur des nombres flottant (exemple division par 0).

Notons que certaines de ces erreurs ne surviennent que sur certains systèmes. Par exemple, pas de Bus Error ou de Floating Exception sur un x86 sous Linux (a priori la mémoire est complètement adressable sur 32 bits et les flottants ont des valeurs comme INF ou NAN possibles).

Questions :

- réccupérez les fichiers `essai_fap.c`, `fap_bug.c` et `fap.h`
- écrivez un `Makefile` permettant de compiler `essai_fap.c` en utilisant tous les fichiers sources sans oublier de compiler avec l'option `-g`
- lancez `essai_fap` et insérez un premier couple (valeur,priorité). Vous pouvez alors constater que ce programme est bogué, il produit une erreur mémoire classique. Nous allons utiliser `gdb` pour le déboguer.
- lancez `gdb` pour déboguer `essai_fap` de l'une des deux manières suivantes :
  - soit `gdb essai_fap`
  - soit `gdb` suivi de `file essai_fap`

  vous vous trouvez alors dans l'interpréteur de commandes de `gdb`. Saisissez `run` pour lancer le programme en cours de déboguage et insérez un couple, ceci devrait nous mener au "Segmentation fault" précédent. La différence est que `gdb` nous montre à quel endroit l'erreur s'est produite (fonction, paramètres, fichier et ligne). Pour avoir plus de détails, vous pouvez alors utiliser les commandes suivantes :
  - `list` pour afficher le code source du programme autour de l'erreur.
  - `up` et `down` pour remonter ou redescendre dans la pile d'appel des fonctions.
  - `quit` pour quitter `gdb`.

  Corrigez cette première erreur présente sur la ligne indiquée par `gdb`.

2 - Points d'arrêt et execution pas à pas

Si vous essayez un peu le programme `essai_fap`, vous pourrez assez rapidement constater qu'il reste encore des bugs. Commençons par nous attaquer à celui qui survient lors de l'extraction : essayez d'insérer un élément (`i 42 42`) et de l'extraire plusieurs fois ensuite (`e` sur une ligne, plusieurs fois). Vous pouvez constater que le même élément est extrait à chaque tentative, comme s'il n'avait pas été supprimé de la FAP. Pire, si essayez d'insérer ensuite un nouvel élément (`i 41 41`) et que vous essayez de supprimer plusieurs fois ce nouvel élément, le programme finit par planter dans la bibliothèque standard.

Pour corrigez cela, nous allons à nouveau utiliser `gdb`. Après l'avoir lancé sur notre programme, placez un point d'arrêt au début de la fonction `extraire` (ligne 43) :

- `break extraire`
- ou

```
list extraire
break 43
```

Exécuter ensuite le programme pas à pas (ligne par ligne en descendant ou non dans les appels de fonction). Pour cela, vous avez à votre disposition tout un ensemble de commandes dont les plus communes sont :

- **Afficher des expressions**
  - `print expression` pour afficher l'expression demandée, ex: `print courant, print *courant, print courant->element, print courant->prochain, ...`
  - `display expression` ou `undisplay numero` pour afficher (ou ne plus afficher) de manière répétée une expression à chaque arrêt dans le programme.
- **Continuer l'exécution**
  - `step` exécute la prochaine ligne du programme et descend dans les appels de fonctions s'il y en a.

    - `next` exécute la prochaine ligne du programme sans descendre dans les appels de fonctions.
    - `cont` reprend l'exécution jusqu'au prochain point d'arrêt ou jusqu'à la fin du programme.
- **Modifier les points d'arrêt**
    - `list fonction/ligne` ou `break` pour afficher le code source ou les points d'arrêt.
    - `break ligne` ou `delete numéro` pour placer ou enlever des points d'arrêt.

Questions :

- En continuant l'exécution, trouvez et corrigez le bug de la fonction `extraire`.

3 - Génération de jeu de test, points d'arrêts conditionnels

Pour continuer notre séance de débogage, nous allons construire un fichier qui contiendra des commandes de manipulation de la fap. L'intérêt est que nous pourrons donner ce fichier en entrée à notre programme :

- cela nous évitera de saisir manuellement les entrées à chaque exécution ;
- cela nous permettra de stocker toutes les séquences d'entrées particulières que nous pourrions vouloir tester ;
- cela nous permettra de tester le programme sur de très grandes entrées, typiquement générées aléatoirement.

Pour ce TP, nous nous limiterons au cas du fichier d'entrées générées aléatoirement.

Questions :

- Dans cette partie, nous allons générer une séquence d'entrées aléatoires qui respecte le format de manipulation de la fap du programme que nous sommes en train de tester. Ecrivez un programme (appelé `essai` par exemple) qui affiche aléatoirement 1000 séquences parmi les deux séquences suivantes :

    - `i nombre_aléatoire nombre_aléatoire`
    - `e`

    avec pour proportions 2/3 de la première et 1/3 de la seconde (on veut plus d'insertions pour tester la libération tout à la fin). Ce sont respectivement les commandes utilisées par `essai_fap` pour insérer et extraire un couple dans la FAP. Votre programme se terminera par l'affichage du caractère 'q' sur une ligne. Vous pourrez vous aider des fonctions `random` et `srandom` pour produire les valeurs aléatoires (pour donner une graine à `srandom`, vous pouvez utiliser la valeur renvoyée par la fonction `getpid` qui varie à chaque nouvelle exécution). Remarque : ce programme doit être indépendant du reste, il n'a pas besoin d'utiliser les fonctions `inserer` et `extraire` présentes dans la FAP. Un affichage possible de ce programme sera :

    ```
    e
    i 333389410 430141989
    i 465984181 496654487
    i 329783298 1239883801
    i 1221951516 742964340
    e
    i 866399555 1204571199
    e
    i 965240854 81840503
    i 1026711895 1696625858
    ...
    q
    ```

    Essayez votre programme pour vérifier qu'il produit bien l'affichage prévu.
- utilisez ce programme pour exécuter `essai_fap` avec un jeu de test généré de la façon suivante :

    ```
    essai | ./essai_fap
    ```

    Un nouveau bug devrait apparaître et vous pouvez constater que, si vous exécutez cette ligne de commande plusieurs fois, le bug ne se produit jamais au même moment. Pour déboguer, le mieux est de travailler sur un jeu d'entrées fixe et un programme déterministe. Nous allons donc sauver un jeu d'entrées généré :

    ```
    ./essai >jeu_test.txt
    ```

    et n'utiliser que celui-ci par la suite :

    ```
    ./essai_fap <jeu_test.txt
    ```

- lancez la même exécution sous `gdb` de la manière suivante:

    ```
    run <jeu_test.txt
    ```

    le programme s'interrompt alors lors de la `Segmentation Fault` à l'endroit où celle-ci se produit (boucle `while`) après avoir affiché une sortie qui ressemble au texte suivant :

    ```
    Aide :
    Saisir l'une des commandes suivantes

    i nombre priorite  :   inserer un nombre avec sa priorite
    e                  :   extraire le nombre de priorite maximale
    ```

```
v                 :   teste si la fap est vide
h                 :   afficher cette aide
q                 :   quitter ce programme
La fap est vide !
(1448575598,529893416) a ete insere
(1448575598,529893416) a ete extrait
La fap est vide !
La fap est vide !
La fap est vide !
(407554779,1310005120) a ete insere
(858057669,1120800437) a ete insere
(1426228474,42941353) a ete insere
(246382257,520846289) a ete insere
(312349583,1241263652) a ete insere
(779822535,449518726) a ete insere
(1719730016,76271212) a ete insere
(1426228474,42941353) a ete extrait
(237271553,1286694501) a ete insere
(1719730016,76271212) a ete extrait
(779822535,449518726) a ete extrait
(246382257,520846289) a ete extrait
(1821239730,584699683) a ete insere
(1821239730,584699683) a ete extrait
(279502296,81310861) a ete insere
(562963211,939368530) a ete insere
(279502296,81310861) a ete extrait
(562963211,939368530) a ete extrait
(858057669,1120800437) a ete extrait
(1254363091,464495613) a ete insere
(1254363091,464495613) a ete extrait
(312349583,1241263652) a ete extrait
(1556667732,975107965) a ete insere
(1556667732,975107965) a ete extrait
(237271553,1286694501) a ete extrait
(407554779,1310005120) a ete extrait
(268124953,1288857280) a ete insere

Thread 3 hit Breakpoint 1, inserer (f=0x100200050, element=855911404, priorite=2056228799) at fap_bug.c:33
33              while ((priorite >= courant->priorite) && (courant != NULL)) {
```

L'erreur n'est peut être pas très parlante en voyant juste la ligne, nous aimerions pouvoir nous arrêter au début du `while` lors du bon appel de la fonction `inserer` (lorsque l'élément à insérer est 855911404 par exemple) pour pouvoir exécuter la boucle pas à pas. Pour cela nous allons utiliser un point d'arrêt conditionnel :

```
break 33 if element == 855911404
```

Relancez alors le programme depuis le début avec `run`, il devrait maintenant s'arrêter au début de la boucle `while` de l'appel à `inserer` qui pose problème. Il ne vous reste plus qu'à exécuter pas à pas en observant et trouver l'erreur !

- Un autre exemple, le programme cesar.c va vous permettre de mettre en oeuvre les notions précédentes. Ce programme permet de coder un message textuel en décalant de manière circulaire les lettres majuscules dans l'alphabet d'une valeur donnée (codage dit de César). Il requiert deux arguments sur la ligne de commande, sous `gdb` vous pouvez les lui passer de la manière suivante :

  ```
  run 5 'COMMENT ALLEZ VOUS ?'
  ```

  Cela permet d'exécuter le programme en lui demandant de coder la chaîne `COMMENT ALLEZ VOUS ?` avec un décalage de 5. Ce programme comporte deux problèmes :
  - la valeur de décalage utilisée n'est pas la bonne (exécutez pas à pas jusqu'au calcul du décalage)
  - certaines lettres ne sont pas codées par une lettre (mettez un point d'arrêt qui se déclenche au moment du codage d'une de ces lettres)

4 - Valgrind

Il est parfois utile d'aller encore plus loin dans l'analyse d'un programme. Par exemple, nous pouvons souhaiter :

- analyser tous les accès mémoire pour déterminer s'il font partie d'une zone allouée ou non, si les variables ont été initialisées avant d'être lues, s'il n'y a pas d'accès à la pile en dehors du contexte courant, ...
- analyser les appels aux fonctions mais aussi les défauts de cache à tous les niveaux sans forcément devoir recompiler le programme analysé (utile lorsqu'on ne dispose pas des sources).
- ...

Valgrind est une collection d'outils d'analyse de programme permettant de satisfaire ces souhaits. Pour ce TP, nous l'utiliserons uniquement pour analyser la correction de notre utilisation du tas, c'est le mode de fonctionnement par défaut de `valgrind`, pour l'utiliser il suffit de l'invoquer avec un exécutable comme argument :

```
valgrind ./essai_fap
```

Questions :

- En utilisant `valgrind` vous devriez encore trouver deux bugs :

- des accès en dehors des cellules (avons nous alloué la bonne taille ?)
- des fuites mémoire (la fap est-elle détruite correctement ?)

Corrigez ces erreurs.

5 - Instrumentation

Maintenant, nous aimerions connaitre l'évolution de la taille de notre FAP, en particulier sa taille maximale. Pour cela nous allons instrumenter notre code afin d'ajouter un traitement lors du passage dans les fonctions `inserer/extraire` effectués par le programme.

Questions :

- écrivez deux fonctions `instrumentation_inserer` et `instrumentation_extraire` dans lesquelles vous compterez le nombre d'éléments en cours et maintiendrez la taille maximale vue jusqu'alors avant d'appeler les fonctions `inserer` et `extraire`. Vous pourrez stocker ces informations dans des variables globales. Evidemment ces fonctions devront avoir les même paramètres que les fonctions `inserer` et `extraire` (pour pouvoir les appeler).
- pour que ce soit nos fonctions qui soient appelée au lieu des fonctions `inserer/extraire` habituelles, nous allons utiliser le préprocesseur **sans qu'il soit nécessaire de modifier directement le code de la fap ou du main**:
  - placez dans un nouveau fichier d'entête des directives de la forme :

    ```
    #define inserer instrumentation_inserer
    ```

    ceci aura pour effet de remplacer les `inserer` par des `instrumentation_inserer` lors de la compilation. Faites de même pour `extraire`

  - modifiez votre `Makefile` pour compiler le code du main avec l'option `-include` de gcc pour inclure votre fichier d'entête.

    Attention : il faudra inclure votre fichier d'entête lors de la compilation de tous les fichiers source utilisant `inserer` ou `extraire` mais pas lors de la compilation des fichiers contenant la fap ou les fonctions `instrumentation_inserer` et `instrumentation_extraire` (si vous pensiez le contraire, relisez cette partie, quelque chose a du vous échapper).

  - ajoutez un affichage dans `instrumentation_inserer` lorsque le max change.
  - observez l'exécution, vous devriez maintenant voir l'évolution de la taille max de la FAP, tout ça sans avoir touché au code source existant.

Remarque : ceci est aussi réalisable via `valgrind`, mais nous avons choisi cette solution "manuelle" pour des raisons pédagogique (compréhension du processus de compilation).

6 - Pour aller plus loin, profiling

Lorsque l'on souhaite quantifier les passages dans les différentes fonctions d'un programme et le temps passé à l'intérieur de chacune d'elles on utilise un outil conçu pour ça : le profiler. Le profiler sert donc à comprendre d'où peuvent venir des problèmes de performances d'un programme et sert à déterminer quelles sont les parties coûteuses en temps de calcul.

Questions :

- pour utiliser le profiler `gprof` vous devez compiler votre programme avec l'option `-pg` (aussi bien pour la traduction que pour l'edition de liens).
- exécutez votre programme, un fichier `gmon.out` est alors créé.
- utilisez `gprof` en lui donnant le nom de votre programme en argument. Une série de statistiques s'affiche alors (temps moyen passé dans chaque fonction, nombre de passages, arbre d'appels, ...).
- déterminez le temps moyen passé dans les fonctions `inserer` et `extraire`. Proposez une autre implémentation de la fap pour laquelle `inserer` soit plus efficace au détriment de l'extraction.