UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS
UFR FACULTÉ DES SCIENCES
Parc Valrose – BP 71 – F 06108 NICE Cedex 2

Licence 3 d'Informatique
Environnement de Programmation 2005-2006
Roger Rousseau

Séance de TP n° 06 de la semaine du 10 mars 2006

Débogage de programmes avec DDD

Corrigé

Créez votre répertoire de séance pour le TP et placez-vous dessus. Récupérez dans votre répertoire de séance l'archive "`Fourniture6.tgz`", accessible via le site Wiki de cet enseignement ("`http://deptinfo.unice.fr/twiki/bin/view/Linfo/GLEnvtProg0506`") et déballez-la.
Ce répertoire contient :

- trois répertoires : 1.trier, 2.file1, 3.file1 et 4.file2 pour les questions suivantes.

1. Démonstration du fonctionnement de base du débogueur `ddd`

Placez-vous dans le répertoire `1.trier` et construisez l'exécutable `trier` avec le *Makefile*. Le programme `trier` prend des nombres entiers en argument, séparés par des espaces. Exécutez les expériences suivantes : (vous pouvez avoir des résultats différents sur votre machine, car ce programme est « bogué »)

```
~>trier 5 9 3
après tri:
3 5 9
```

ce qui a l'air de marcher... Mais avec d'autres jeux d'essai, en mettant 4 ou 5 arguments dont certains négatifs ou au contraire très grands (1 million), vous constaterez l'apparition de valeurs intruses dans les résultats.
En suivant la démo faîte par votre enseignant sur les fonctionnalités de base de `ddd` , vous trouverez la cause du problème en tapant les mêmes commandes que lui sur votre machine.
Lancez en parallèle la commande **`ddd trier &`** et suivez les instructions de votre enseignant. Cette démo est celle indiquée par Andreas Zeller dans son manuel sur `ddd` ("`http://www.gnu.org/manual/ddd`").

Réponse ⇒ *Suivre les explications du manuel de Zeller, chapitre 1. L'erreur est dans* `main` *:*
`shell\_sort(a, argc)`
au lieu de
`shell\_sort(a, argc`-1**`)`**.

2. Recherche d'une cause d'avortement de programme (abort)

Placez-vous dans le répertoire "`2.file1`". Ce répertoire contient le programme de test "`tfile1.c`" du type abstrait "`file1.c`" du TP précédent, mais avec une "bogue" qui produit une erreur d'adressage, détectée par le matériel, indiquée au noyau de votre système par une interruption matérielle qui prévient alors votre programme par un signal (`SIGSEGV`, *Segmentation Fault*). Si votre programme n'a pas de traitant (*handler*) pour ce signal, c'est le traitement par défaut qui est appliqué. Celui-ci est d'écrire, sur le répertoire courant, une image de la mémoire du processus fautif dans un fichier "`core`" (appelé aussi "`dump`") et un message d'avertissement du genre « Segmentation fault (core dumped) ».
L'absence des mots "`(core dumped)`" dans le message précédent indique qu'aucun fichier `core` n'a été produit, ce que vous pouvez vérifier avec la commande `ls`. La raison est que votre shell est dans un état où l'écriture des fichiers `core` est interdite. Tapez la commande `ulimit -c`. Elle affiche la taille maximum autorisée pour un fichier `core`. Si elle est nulle, cela interdit son écriture. Tapez la commande "`ulimit -c 1000000`" pour autoriser l'écriture de fichiers core jusqu'à 1 million d'octets et réexécutez le programme `tfile1`. Cette fois, vous obtenez le message "`segmentation fault (core dumped)`" et constatez l'apparition d'un fichier "`core`" [1]

[1] Sous Cygwin le fichier obtenu est "`tfile1.stackdump`", mais ce fichier destiné au logiciel DRWatson sous Windows n'est pas utilisable sous `ddd` .

Ce fichier est analysable par un débogueur, comme `adb` , `gdb` (débogueurs "ligne") ou `dbx` , `ddd` (débogueurs graphiques). `ddd` utilise `gdb` par défaut pour analyser les fichiers `core`. Lancez `ddd` en indiquant le fichier binaire du programme à analyser et le fichier `"core"` produit :

```
# Sous Linux , tapez la commande suivante avec complétion pour faire apparaître XXX :
~> ddd tfile1 -- core core.XXXX
# Sous Cygwin, tapez la commande suivante :
~> ddd tfile1.exe &
# Et lancez l'exécution sous ddd, car le fichier stackdump est destiné à Dr. Watson
```

La fenêtre du bas indique l'adresse du plantage dans le code source. Avec les commandes élémentaires de `ddd` , vous pouvez facilement :

- identifier que c'est la fonction `nbPlaces` qui plante ;
- examiner le contenu de la variable `fi` et constater qu'elle est nulle, ce qui explique l'erreur d'adressage mémoire lors du dérepérage du pointeur ;
- ensuite rechercher pourquoi ce pointeur est nul. Comme il indique l'adresse du descripteur de file, il faut examiner le fonctionnement de la routine `creer`. Pour cela placez un point d'arrêt *(breakpoint)* au début de cette routine. Exécutez en pas à pas son fonctionnement et suivez l'évolution des valeurs des variables `f` et `fi`. Vous constatez qu'on sort de la routine avant d'avoir appelé `malloc`. En examinant le code, vous trouverez pourquoi : une faute de frappe classique lorsqu'on programme en C, dénoncée par les livres sur les pièges de la programmation dans ce langage...

Réponse ⇒ *Il suffit de suivre les indications. L'erreur est dans le programme "`file1.c`", dans la fonction `creer` :*
if (`nbPlaces` <0);return `NULL`;
au lieu de
if (`nbPlaces` <0)return `NULL`;.

3. Recherche d'une cause de mauvais fonctionnement détecté par un programme de test

Placez-vous maintenant dans le répertoire `"3.file1"`. Ce répertoire contient encore une fois une implémentation du type abstrait `File` avec un programme de test simplifié, `tfile1` sans utiliser `check` . Le programme `file1.c` contient une "bogue" qui fait fait échouer le programme de test, mais sans plantage et donc sans production d'un fichier `core`. Avec `ddd` , il faut trouver la bogue, la corriger et vérifier que les tests sont alors sans erreur.
Exécutez le *Makefile* et repérez le numéro de ligne de l'échec (30). Lancez `ddd` sur le programme exécutable `"tfile1"`. Affichez les numéros de ligne (Source/Display Line Numbers). Placez un point d'arrêt sur la ligne incriminée (30 et lancez l'exécution. Examinez les variables qui sont à l'origine de l'échec (`sor.info` vaut "e2" au lieu de "e1"). Le problème vient donc du dernier appel à sortir.
Placez un point d'arrêt sur l'appel à `sortir` (ligne d'au dessus) et relancez l'exécution. Avancez d'un pas pour entrer dans sortir. Exécutez ensuite sortir en mode pas à pas. Affichez la variable interne `fi` et en double-cliquant sur sa représentation, faite apparaître sa valeur en mode vertical (avec les noms des champs). Vous trouvez que la file a trois places, trois éléments et les curseurs `entree==2` et `sortie=0`. Affichez le contenu de la mémoire `fi->mem[sortir]` (tapé dans le tampon d'entrée, après l'avoir vidé par ( ) . Vous trouvez `"e2"`, la valeur qui est sortie. Donc `sortir` fonctionne apparemment de façon correcte et le problème doit venir de l'état de la représentation de la file `fi`. Il faut donc inspecter le fonctionnement de `entrer`.
Abandonnez l'exécution (kill) et rechargez le source `"tfile1.c"` (menu File) pour placer un point d'arrêt sur le premier appel à entrer (ligne 26). Relancez l'exécution, puis entrez en pas à pas dans la routine `entrer`. Avec des doubles clics, dépliez le contenu de `fi`. Vous voyez apparaître le contenu de `FileImpl` avec ses champs. Si vous double-cliquez sur `mem`, vous affichez `fi->mem[0]`. Pour afficher les autres valeurs du tableau `mem`, il faut, soit expliciter les adresses, car `mem` est déclaré comme un pointeur dans le programme[2] Affichez donc `fi->mem[1]` et `fi->mem[2]`, soit indiquer la tranche du tableau que l'on veut afficher : `fi->mem[0..2]`.
Les champs `info` ont des adresses nulles (en rouge). Continuez l'exécution jusqu'à la fin de la routine (until). Lorsque vous quiterez la routine `entrer`, l'affichage des variables disparaitra. Mais dès que vous entrerez à nouveau dedans, il réapparaitra. Vous pouvez ensuite poursuivre l'exécution en pas à pas et observer les changements de valeur des champs de `FileImpl`. En réfléchissant aux valeurs que devraient avoir ces différents champs, vous trouverez facilement quel champ est incorrect et donc quelle instruction de la routine entrer est incorrecte. Corrigez l'erreur, relancez le test et concluez.

[2] Avec un langage à objets, on n'aurait pas cet inconvénient, car tout objet est typé et le système peut donc afficher son contenu comme il convient. En C, il faut aider le débogueur.

Réponse ⇒Il suffit de suivre les indications. L'erreur est dans le programme "file1.c", dans la fonction `entrer` *:*
`fi->entree =(fi->entree+1)% (fi->nbElements);`
au lieu de
`fi->entree =(fi->entree+1)% (fi->nbPlaces);`.

4. Pour les plus courageux

Placez-vous dans le répertoire `"4.file2"` et exécutez le *Makefile*. Ce répertoire contient la version 2 de la file, comme dans le TP de synthèse, mais avec un petit programme de test et deux erreurs d'implémentation à trouver avec `ddd` . Cette expérience devrait vous convaincre qu'il ne faut utiuliser un débogueur que lorsqu'il n'y a aucune autre solution. L'utilisation d'assertions exécutable évite le débogage...

Réponse ⇒Il y a deux bogues dans le programme "file2.c" :

- *La première erreur est dans la fonction* `estCorrect` *:* `if (i>dico.nb)`
  au lieu de
  `if (i<dico.nb)`.
- *La deuxième erreur est dans la fonction* `trierPRIORITE` *:* `fi->mem[e].priorite=MAXTICKET;`
  au lieu de
  `fi->mem[e].priorite=MINPRIORITE;`.

Remarque. Le problème posé par le débogage d'un programme de test piloté par `check` est compliqué par l'ajout des primitives de check qui sont censées être fiables et sans effet sur la cause du problème. Si les codes source de `check` sont accessibles, ils seront montrés par `ddd` comme les autres routines du programme à tester, ce qui nécessite de les sauter lors des exécutions en pas à pas. Une solution à ce problème est d'écrire un programme de test sans `check`, comme ci-dessus ou de changer la définition des macros de check pour les rendre inoffensives.