GDB

Liste des commandes

gdb(1) permet de débugger vos programmes plus facilement en permettant d'analyser l'état durant l'exécution. Pour pouvoir analyser votre exécutable avec **gdb(1)**, vous devez ajouter les symboles de débuggage lors de la compilation en utilisant l'option `-g` de **gcc(1)**:

```
gcc -g gdb.c -o my_program
```

L'option -g de **gcc(1)** place dans l'exécutable les informations sur les noms de variables, mais aussi tout le code source.

Lancez gdb avec la commande `gdb my_program`. Ceci va vous ouvrir la console de `gdb` qui vous permet de lancer, le programme et de l'analyser. Pour démarrer le programme, tapez `run`. gdb va arrêter l'exécution au premier problème trouvé. Votre programme tourne encore pour l'instant. Arrêtez-le avec la commande `kill`.

Breakpoint

Pour analyser un programme, vous pouvez y placer des breakpoints. Un breakpoint permet de mettre en pause l'exécution d'un programme à un endroit donné pour pouvoir afficher l'état des variables et faire une exécution pas-à-pas. Pour mettre un breakpoint, vous avez plusieurs choix:

`break [function]` met en pause l'exécution à l'appel de la fonction passée en argument à la commande
`break [filename:linenumber]` spécifie le fichier du code source et la ligne à laquelle l'exécution doit s'arrêter
`delete [numberbreakpoint]` supprime le breakpoint spécifié

Note : Chaque breakpoint est caractérisé par un numéro. Pour obtenir la liste des breakpoints utilisés `info break`

Informations à extraire

Une fois un breakpoint placé, plusieurs informations peuvent être extraites via **gdb(1)** :

`print [variablename]` affiche la valeur de la variable dans son format de base. Il est possible de connaître la valeur pointée en utilisant `*` ainsi que l'adresse de la variable avec `&`.

```
Il est aussi possible de modifier une variable avec ``set variable [nom_variable] = [valeur]``.
De façon similaire avec ``print [nom_variable] = [valeur]``.
```

`info reg [registre]` affiche les informations sur tous les registres si aucun registre n'est explicitement spécifié. `info reg eax` donne le même résultat que `print $eax`.

```
Il est intéressant de noter qu'il est possible d'afficher une variable sous le format spécifié. Pour cela, remplacer ``pr
* ``p/x`` - affiche en format hexadécimal la variable spécifiée
* ``p/d`` - en format entier signé
* ``p/f`` - en format floating point
* ``p/c`` - affiche un caractère.
```

`backtrace` ou `bt` affiche la pile des appels de fonctions.

```
Il est possible de naviguer dans la pile des appels à l'aide de ``up`` et ``down``. Ces deux commandes montent et descend
```

`info frame` donne des informations sur la frame actuelle.
`list` affiche les lignes de codes entourant le break. On peut donc facilement voir le code posant un problème ou analyser le code avant de faire une avancée pas à pas.
`show args` affiche les arguments passés au programme.
`info breakpoints` affiche les breakpoints
`info diplays` affiche les displays
`info func [fonctionname]` affiche le prototype d'une fonction

Avancement de l'exécution

Quand vous avez acquis suffisamment d'informations sur le programme, vous avez plusieurs choix pour continuer son exécution :

`next` exécute la prochaine instruction de votre code source, mais sans rentrer dans des fonctions externes.
`step` exécute la prochaine instruction de votre code source, mais en entrant dans le code des fonctions appelées.
`continue` continue le reste de l'exécution jusqu'au prochain breakpoint.

Automatisation

Lors d'un débuggage long et fastidieux, il est parfois nécessaire d'exécuter certaines commandes à chaque breakpoint.

> `commands [numerobreakpoint]` définit une liste de commandes associées à un breakpoint. Celles ci seront exécutées quand on s'arrêtera sur ce breakpoint. Il suffit de taper les commandes à effectuer les unes après les autres et de terminer par `end`. Si vous ne fournissez pas de numéro, les commandes sont assignées au dernier breakpoint créé.
>
> `display [variablename]` affiche la variable à chaque breakpoint.

Gestion des Signaux

En plus des breakpoints, **gdb(1)** interrompt l'exécution du programme en cours lorsqu'il intercepte certains signaux d'erreurs comme les signaux `SIGSEGV` et `SIGINT`. **gdb(1)** permettra alors de corriger plus facilement certaines erreurs comme les erreurs de segmentation ou les problèmes de deadlocks.

Il est possible de gérer le comportement de **gdb(1)** lorsque des signaux sont interceptés. Tout d'abord, la commande `info signals` permet d'afficher la liste des signaux reconnus par **gdb(1)** ainsi que la façon dont il les traite (par exemple interrompre le programme en cours ou non). On peut changer la façon de traiter un signal avec la commande `handle [SIGNAL] [HANDLING...]` où `[SIGNAL]` est le signal à intercepter (son numéro ou son nom complet) et `[HANDLING]` la façon de traiter ce signal par **gdb(1) [1]**. Par exemple, la commande `handle SIGALRM stop print` permet d'interrompre le programme et d'afficher un message quand gdb intercepte le signal `SIGALRM`.

Localiser un signal

Avec **gdb(1)**, il est possible de localiser un signal et de débugger certaines erreurs comme une erreur de segmentation. En effet, lorsque **gdb(1)** interrompt le programme en cours après l'interception d'un signal d'erreur comme `SIGSEGV`, il est possible de trouver la ligne du programme à laquelle le signal a été intercepté en tapant le mot-clé `where` une fois le programme interrompu (il est cependant nécessaire d'avoir compilé le programme avec l'option `-g` de `gcc` pour trouver la ligne précise). Ensuite, grâce aux commandes expliquées plus tôt, il est possible de vérifier les valeurs des variables lors de l'interception du signal pour trouver l'origine du problème.

En plus de localiser facilement les erreurs de segmentation dans un programme, vous pourrez analyser plus aisément les problèmes de deadlock des threads. En effet, lorsque le programme est lancé sur le shell et que vous remarquez un deadlock, vous pouvez appuyer sur `CTRL + C` pour lancer le signal `SIGINT` au programme. Cela permettra de trouver les endroits où bloquent les différents threads du programme à l'aide des commandes décrites dans la section de débuggage des threads ci-dessous.

Extraction de code assembleur

> `disas` affiche le code assembleur
>
> `disas /m blah` met en correspondance le code assembleur et le code source

Pour arrêter la console de gdb, tapez `quit`.

Illustration avec des exemples

Premier programme

> Le premier programme est **`src/calc.c`**. Compilez-le et exécutez le pour vous apercevoir que le programme est erroné. A priori vous avez peu, ou pas, d'informations sur l'erreur. Lancez donc gdb à l'aide de `gdb calc` puis lancez le programme avec `run`.

```
Program received signal SIGFPE, Arithmetic exception.   => Exception arithmetique
0x0000000000400553 in calc (a=165, b=4) at calc.c:10    => Dans la fonction calc du fichier calc.c à la ligne 10

10                      res = (a*5 -10) / (b-i);        => Affichage de la ligne problématique
```

> Le premier réflexe doit être `list` pour observer le code. Puisque le problème vient de la ligne 10 dans la boucle, nous allons nous arrêter à la ligne 10 avec `break 10` et relancer le programme. Le programme va s'arrêter avant le début de la boucle. Utilisez `print a` et `print b` pour connaître les arguments reçus par calc.

```
Il est intéressant de noter une particularité du language C par rapport à java : une variable déclarée n'est pas initialis
```

> Puisque le problème vient du calcul arithmétique, placez un break sur cette ligne pour pouvoir observer à chaque itération les variables. `break 9` puis `commands` qui permet d'automatiser des commandes. Nous rajouterons comme commandes :

```
* ``echo i : ``
* ``print i``
* ``echo b : ``
* ``print b``
```

```
* ``echo numerateur : ``
* ``print a*5 -10``
* ``echo denominateur : ``
* ``print b-i``
* et enfin ``end`` pour terminer la liste de commandes.
```

Il ne reste plus qu'à avancer avec `continue` pour aller de breakpoint en breakpoint et d'observer les variables pour comprendre le problème. On va pouvoir deviner que le problème vient d'un dénominateur nul. Pour résoudre ce problème, il faut passer une valeur plus grande que 6 à calc lors de son appel depuis la fonction main. `list main` suivi de plusieurs `list` permet de visualiser la main. On peut repérer l'appel de la fonction calc à la ligne 18.

Supprimez les anciens break avec `delete [numerobreakpoint]` le numéro du breakpoint est connu via `info break`. Rajoutez un break à la ligne 18, `break 18` et lancez le programme. `set variable m = 10` pour assigner la valeur 10 à la variable m. Puis continuez l'exécution du programme. Celui se terminera normalement puisque il n'y a plus de division par zéro.

Deuxième programme

Le deuxième programme est appelé **`src/recursive.c`**. Celui ne présente aucun bug et se déroulera normalement. Toutefois, il est intéressant d'utiliser **gdb(1)** pour bien comprendre les différents contextes au sein d'un programme. Mettez un break sur la fonction `factTmp` avec `break factTmp` et ajoutez automatiquement à ce breakpoint la commande `backtrace`, via `commands`. Ensuite, lancez le programme. `backtrace` vous permet de visualiser les appels de fonction effectués. Nous pouvons voir que la fonction `factTmp` a été appelée par `factTerminal`, elle même appelée par la fonction `main`.

```
#0  factTmp (acc=1, nbr=6) at recursive.c:8
#1  0x000000000040057d in factTerminal (a=6) at recursive.c:17
#2  0x0000000000400598 in main (argc=1, argv=0x7fffffffe1b8) at recursive.c:23
```

Essayez d'afficher les variable `globalVar` puis `localVar`. Vous remarquerez qu'il n'est pas possible d'afficher `localVar` puisque cette variable ne fait pas partie de l'environnement contextuel de factTmp. Pour afficher cette variable, il faut remonter la liste des appels. `up` permettra de remonter les appels pour pouvoir afficher `localVar`. Une fois la variable affichée, redescendez avec `down` et continuez 4 fois le programme après le breakpoint. Vous remarquerez que la liste des appels s'allonge à chaque appel récursif, ce qui est tout à fait normal.

Naviguez dans les appels récursifs de factTmp en affichant les valeur de `globalTmp`, `tmp`, `acc` et `nbr`. Il est important de bien comprendre que la variable statique `globalTmp` est commune à tous les appels de la fonction `factTmp` et un changement de cette variable dans un des appels récursifs modifie la variable des autres appels. A contrario, la variable local ainsi que les arguments sont propres à chaque appel.

Vous pouvez maintenant terminer le programme.

Troisième programme

Le troisième programme est **`src/tab.c`**. Compilez-le. Ce programme s'exécute correctement, et pourtant, il y contient une erreur. Lancez le programme avec gdb et mettez un breakpoint sur la première instruction, à savoir la ligne 9. Pour comprendre un problème sans savoir où commencer, il est utile de suivre l'évolution des variables.

```
Il est important de savoir que ``print``, ainsi que ``display``, supportent les expressions telles que :
        * tab[1], tab[i],...
        * &i, *i,...
```

Avancez instruction par instruction, avec `step` ou `next` et portez attention aux valeurs de tab[i] par rapport à i. Une fois le problème trouvé avec gdb, solutionnez le.

Plus d'informations sur **gdb(1)** peuvent être trouvées sur:

https://www.cprogramming.com/gdb.html
https://developer.ibm.com/articles/l-gdb/
https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/gdb.html

Débuggage des threads avec GDB

gdb(1) est aussi utile pour débugger des programmes avec des threads. Il permet de faire les opérations suivantes sur les threads:

Recevoir une notification lors de la création d'un nouveau thread.
Afficher la liste complète des threads avec `info threads`.
Placer un breakpoint dans un thread. En effet, si vous placez un breakpoint dans une certaine fonction, et un thread passe lors de son exécution à travers ce breakpoint, `gdb` va mettre l'exécution de tous les threads en pause et changer le contexte de la console **gdb(1)** vers ce thread.
Lorsque les threads sont en pause, vous pouvez manuellement donner la main à un thread en faisant `thread [thread_no]` avec `thread_no` étant l'indice du thread comme indiqué par `info threads`

D'autres commandes pour utiliser **gdb(1)** avec les threads:

https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html

Footnotes

[1] Une liste plus complète des mots-clés utilisables pour modifier le comportement de gestion des signaux peut-être consultée ici : **ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_38.html** .