GDB x Valgrind

Options de compilation pour le débogage

Pour pouvoir convenablement utiliser gdb et valgrind, il faut que le programme ait été compilé avec `gcc -g -O0` :

- `-g` : Compile en laissant des portes permettant à gdb d'accéder à l'état du programme en temps réel.
- `-O0` : Désactive les optimisations du compilateur. Ceci permet de faire que les instructions machine du programme que l'on va déboguer soient les plus proches possibles du code source.

Commandes gdb de base

`q`, `quit` : Quitte la session gdb.

`!<commande shell>` : Exécute la commande shell écrite après le point d'exclamation.

`tui enable` : Active le mode "tui" qui affiche en temps réel la ligne en train d'être exécutée par gdb en écran splitté.
Quand ce mode est activé, "Ctr-X, O" permet de basculer entre la fenêtre où le code source s'affiche et la fenêtre de la console gdb.

`Ctrl-L` : Ce raccourci clavier permet de rafraichir l'affichage de gdb quand il part en couille (ça peut arriver, surtout si l'on ne redirige pas la sortie standard du programme)

`tty /dev/pts/<numéro_de_terminal>` : Redirige dans un autre terminal la sortie standard du programme qui est en train d'être débogué.
Pour connaître le numéro d'un terminal, il suffit d'utiliser la commande bash `tty` dans le terminal dont on veut connaître le numéro.

Gestion des breakpoints

`break <nom_de_fonction>`, `break <numéro_de_ligne>` : Ajoute un breakpoint au programme pour que gdb s'arrête dessus lors de l'exécution de la commande `run`.

`watch <expression C>` : Ajoute un breakpoint qui met en pause l'exécution du programme dès que l'évaluation de l'expression C change de valeur.
Très utile pour détecter à quel moment une variable change de valeur ou à quel moment un invariant de boucle est rompu.

`info breakpoints` : Affiche la liste de tous les breakpoints.

`disa <numéro d'un breakpoint>` : Désactive un breakpoint.

`ena <numéro d'un breakpoint>` : Active un breakpoint.

`delete <numéro d'un breakpoint>` : Supprime un breakpoint.

Reprendre l'exécution

`run` : Lance le programme et s'arrête au premier breakpoint rencontré.

`next`, `n` : Passe à la ligne suivante.

`step`, `s` : Passe à l'étape suivante (plus précis que `next`).

`finish` : Continue jusqu'à ce que la fonction courante se termine.

`until <nom_de_la_fonction>`, `until <numéro_de_ligne>` : Continue jusqu'au début de la fonction ou jusqu'à la ligne.

`continue`, `c` : Continue l'exécution jusqu'au prochain breakpoint.

États

`print <expression C>`, `p <expression C>` : Évalue et affiche l'expression C.
Peut être utilisé pour afficher les valeurs des variables internes du programme ou pour modifier

leurs valeurs.

`display <expression C>` : Évalue et affiche l'expression C à chaque fois que gdb s'arrête après un `next` ou un `step`.

Contexte

`list`, `l` : Affiche les 10 prochaines lignes du fichier.

`list <numéro_de_ligne>`, `l <numéro_de_ligne>` : Affiche les lignes au voisinage du numéro de ligne donnée en argument.

`where`, `backtrace`, `bt` : Affiche la pile des frames et la ligne en cours.

`up` : Déplace l'analyseur dans la frame de dessus.

`down` : Déplace l'analyseur dans la frame du dessous.

Surveiller les allocations dynamiques de mémoire

Modifier le comportement de malloc et free dans gdb

Lancer gdb avec la variable d'environnement `MALLOC_PERTURB_` permet de changer le comportement des fonctions `malloc` et `free`.

Par exemple, si on lance `MALLOC_PERTURB_=0x12345 gdb ./mon_programme_tout_bugué`, alors :

- `malloc` écrit `0x12345` dans chaque octet qu'il alloue.
- `free` écrit le complément à 2 de 0x12345 dans chaque octet qu'il libère.

Modifier le comportement de malloc et free dans valgrind

Lancer valgrind avec les arguments `--malloc-fill=<hexnumber>` et `--free-fill=<hexnumber>` permet de perturber le fonctionnement de `malloc` et `free` au niveau de valgrind.

Utiliser gdb et valgrind ensemble

Il est possible d'entrer dans un programme avec gdb à partir de la première erreur détectée par valgrind lors de son exécution.

Pour cela, il faut deux terminaux :

- Terminal 1 : lancer le programme avec `valgrind --vgdb-error=1 ./mon_programme_tout_bugué`
- Terminal 2 :
    - lancer `gdb`
    - une fois dans gdb, utiliser la commande `target remote | vgdb --pid=<pid du processus valgrind>`

Après cela, il est possible d'explorer le programme dans le terminal 2 avec gdb, à partir de l'instant de la première erreur.

Pour arrêter le programme :

- Terminal 2 : quitter gdb avec la commande `q`
- Terminal 1 : tuer le processus valgrind avec `kill -KILL <pid du processus valgrind>`