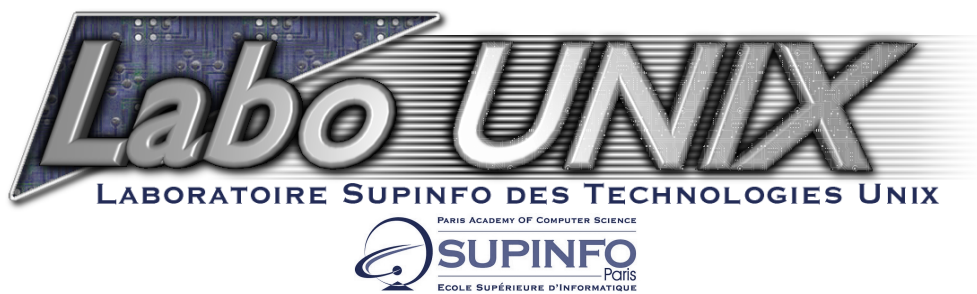


# Initiation au débogage sous Unix



Labo-Unix - <http://www.labo-unix.net>

2001-2002

## Table des matières

<b>Droits de ce document</b>	<b>3</b>
<b>Introduction</b>	<b>4</b>
<b>1 Les outils plus moins standards du C</b>	<b>5</b>
<b>2 Le poids lourd GDB</b>	<b>6</b>
2.1 Prise en main . . . . .	6
2.2 Exécution du programme au sein de gdb . . . . .	9
2.3 Gestion des points d'arrêts . . . . .	9
2.3.1 Mise en place . . . . .	9
2.3.2 Activation et désactivation . . . . .	11
2.3.3 Définir une condition d'arrêt . . . . .	12
2.3.4 Exécuter une série de commandes . . . . .	12
2.4 Méthodes pour avancer dans le programme une fois stoppé . . . . .	13
2.4.1 Continuer l'exécution du programme . . . . .	13
2.4.2 Exécution pas à pas . . . . .	14
2.5 Interception et traitement des signaux . . . . .	16
2.6 Considérations sur les programmes multitâches . . . . .	18
2.6.1 Les programmes utilisant le fork() . . . . .	18
2.6.2 Programme utilisant les pthreads . . . . .	19
2.7 Examiner le contenu de la pile . . . . .	21
2.8 Examiner le contenu de la mémoire . . . . .	23
2.9 Examiner le contenu des registres . . . . .	25
2.10 Informations concernant les fichiers source . . . . .	28
2.11 Modifier le comportement du programme . . . . .	28
2.12 Utiliser un fichier de type "core" . . . . .	30
<b>3 Electric Fence et les "malloc debuggers"</b>	<b>31</b>
<b>4 strace/ltrace</b>	<b>31</b>
4.1 strace . . . . .	31
4.2 ltrace . . . . .	32
<b>5 Profiling</b>	<b>33</b>
5.1 Qu'est-ce-que c'est ? . . . . .	33
5.2 gprof . . . . .	33
5.2.1 Préparation . . . . .	33
5.2.2 Comment interpréter les résultats . . . . .	34
5.3 gcov . . . . .	36
5.3.1 Préparation . . . . .	36
5.3.2 Comment interpréter les résultats . . . . .	36

<b>6</b>	<b>GNU Free Documentation License</b>	<b>39</b>
6.1	Applicability and Definitions . . . . .	39
6.2	Verbatim Copying . . . . .	40
6.3	Copying in Quantity . . . . .	40
6.4	Modifications . . . . .	41
6.5	Combining Documents . . . . .	42
6.6	Collections of Documents . . . . .	43
6.7	Aggregation With Independent Works . . . . .	43
6.8	Translation . . . . .	43
6.9	Termination . . . . .	44
6.10	Future Revisions of This License . . . . .	44
	<b>Références</b>	<b>45</b>

## **Droits de ce document**

Copyright (c) 2001 labo-unix.org

Permission vous est donnée de copier, distribuer et/ou modifier ce document selon les termes de la Licence GNU Free Documentation License, Version 1.1 ou ultérieure publiée par la Free Software Foundation ; avec les sections inaltérables suivantes :

- pas de section inaltérable

Une copie de cette Licence est incluse dans la section appelée GNU Free Documentation License de ce document.

“Lorsque à coder tu commenceras, la force du débogage et la maîtrise de toi tu trouveras”

Suivons cette phrase pleine de bon sens que nous a enseignée un grand maître dont nous préférerons taire le nom. En effet, lorsqu’on découvre les joies du codage, en général, la force obscure n’est pas loin et les bugs aussi divers soient-ils menacent. C’est alors qu’il devient intéressant de maîtriser un certain nombre de réflexes, d’en inventer d’autres, d’utiliser des outils adéquates... C’est ce à quoi nous allons essayer de vous introduire dans le document suivant. Ah, j’oubliais, les exemples seront donnés en “C”, mais les principes restent les mêmes pour la plupart des autres langages et “debuggers”.

Si vous vous retrouvez nez à nez avec des erreurs absolument inacceptables, que vous pensez que tout ou partie de ce cours n’est pas bon, envoyez un e-mail à [staff@labo-unix.org](mailto:staff@labo-unix.org).

Bien sûr nous restons ouverts à toutes remarques constructives ou non.

## 1 Les outils plus moins standards du C

Il y a quantités de bugs qui sont susceptibles d'être évités, ou du moins mieux cernés au sein même du code par le biais d'astuces, ou de fonctionnalités du langage prévues à cet effet.

Naturellement, lorsqu'on se retrouve face à un bug, nous sommes poussés à utiliser des repères pour déterminer ou affiner notre recherche. Les fonctions couramment utilisées sont "*printf*" et "*puts*" auxquelles on peut associer les macros suivantes :

*\_\_FILE\_\_* Donne le fichier dans lequel cette macro est appelée.

*\_\_FUNCTION\_\_* Donne la fonction dans laquelle cette macro est appelée.

*\_\_FUNC\_LINE\_\_* Donne la ligne de la fonction dans laquelle cette macro est appelée.

*\_\_LINE\_\_* Donne la ligne par rapport à l'ensemble du fichier source.

Voyons l'horrible fichier source de ce pgnfr (Programme Qui Ne Fait Rien) :

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

#if DEBUG
#define DEBUG_TAG printf ("\n[D] ("__FILE__") ["__FUNCTION__"] -- Line %d\n", __LINE__);
#else
#define DEBUG_TAG#endif

char * falc (char *arg1);

int main (int argc, char **argv)
{
    DEBUG_TAG;
    falc (argv[1]);
    DEBUG_TAG;

    return 0;
}

char * falc (char *arg1)
{
    char *buf;
    unsigned int taille_buf;

    DEBUG_TAG;
    taille_buf = strlen (arg1);
    DEBUG_TAG;
    buf = malloc (taille_buf);
    DEBUG_TAG;
    strncpy (buf, arg1, taille_buf);
    DEBUG_TAG;
    printf ("---> Le premier argument du programme est : %d\n", buf);

    return NULL;
}
```

Lors du lancement du programme nous devons obtenir quelque chose qui s'approche de ce qui suit :

```

<prompt> $ gcc -DEBUG -Wall -o exemple_1 exemple_1.c
<prompt> $ ./exemple_1 42
[D] (exemple_1.c) [main] -- Line 13
[D] (exemple_1.c) [falc] -- Line 25
[D] (exemple_1.c) [falc] -- Line 29
[D] (exemple_1.c) [falc] -- Line 33
[D] (exemple_1.c) [falc] -- Line 37
----> Le premier argument du programme est : 42
[D] (exemple_1.c) [falc] -- Line 41
[D] (exemple_1.c) [main] -- Line 15
<prompt> $ ./exemple_1
[D] (exemple_1.c) [main] -- Line 13
[D] (exemple_1.c) [falc] -- Line 25
segmentation fault (core dumped) ./exemple_1

```

Première remarque, le programme est tellement bien conçu qu’il plante, et ne se limite pas à cela, il nous crée un fichier “core”. Un fichier “core” est juste une image de la mémoire au moment où le programme a planté. L’intérêt est d’avoir une trace de ce qui s’est passé dans certaines conditions parfois difficiles à recréer. Nous verrons ultérieurement comment utiliser de tels fichiers. L’essentiel ici est de remarquer que la ligne de code incriminée est “*taille\_buf = strlen (arg1);*” et donc de faire un test sur la valeur de “*arg1*”.

## 2 Le poids lourd GDB

Bien sûr, le débogage tel que nous l’avons vu précédemment montre rapidement ses limites<sup>1</sup>. Il nous faut alors un outil digne de ce nom : “*Gnu DeBugger*”. Nous traiterons ici que de l’interface en mode console, mais il dispose tout de même d’interfaces graphiques telle que “*xxgdb*” ou bien mieux encore “*DDD*” (même s’il repose essentiellement sur *gdb*, il utilise aussi d’autres debuggers, le rendant ainsi très complet).

### 2.1 Prise en main

Avant de commencer à débogger en utilisant *gdb*, il faut préparer son programme. Pour ce faire, au moment de la compilation, il suffit de rajouter l’option “*-g*” à *gcc*. Si vous ne le faites pas, l’utilisation de *gdb* paraîtra tout de suite moins souple. Il est bon de noter que les options d’optimisation telle que “*-O*”, “*-O2*” et “*-O3*”, peuvent bouleverser le code et vous ne déboggerez peut-être pas ce que vous souhaitez. C’est pour ceci qu’il peut arriver que certains compilateurs ne puissent accepter à la fois les options d’optimisation et de débogage.

Ensuite, c’est fin prêt, il ne reste plus qu’à lancer *gdb* et lui associer ce que l’on souhaite débogger ; un fichier binaire, un processus en cours d’exécution ou encore un fichier “core”.

```

<prompt> $ ls
exemple_1  exemple_1.c  exemple_1.core
<prompt> $ gcc -g -Wall -o exemple_1 exemple_1.c
<prompt> $ gdb exemple_1
GNU gdb 4.16.1
Copyright 1996 Free Software Foundation, Inc.

```

<sup>1</sup>Même si son action est limitée, il est important de l’utiliser en complément d’outils de débogage pour mieux cerner l’origine du problème.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "sparc-unknown-openbsd2.9"...

```
(gdb) quit
<prompt> $ vi pouet&
<prompt> $ ps
12211 p7  SNs      0:07.03 -zsh (zsh)
21212 p7  TN       0:00.15 vi pouet
42422 p7  RN+      0:00.01 ps
<prompt> $ gdb -q /usr/sbin/vi 21212
(no debugging symbols found)...
/home/moncompte/22049: No such file or directory.
Attaching to program /usr/bin/vi', process 21212
0x8118bf8 in ?? ()
(gdb) quit
<prompt> $ gdb -q exemple_1 exemple_1.core
```

```
Core was generated by exemple_1'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/libexec/ld.so...done.
Reading symbols from /usr/lib/libc.so.26.2...done.
#0  0x8091310 in strlen ()
(gdb) q
<prompt> $
```

Plusieurs remarques ; tout d'abord, lorsqu'on a assez de voir le message indiquant que gdb est un soft GPL (<http://www.gnu.org/licenses>), il suffit de passer en paramètre "-q" et c'est terminé.

Une fois l'invite de commande de gdb affichée, plusieurs choses ; tout d'abord, fonctionnalité absolument géniale, le support de la complétion<sup>2</sup>, et ce, non seulement sur les commandes mais aussi sur les variables, les fonctions et à peu près tous ce qui lui est accessible. Il vous permet également de garder le contact avec votre shell par le biais de la commande "*shell [commande à exécuter]*". Cependant, parceque vous aurez à le faire très souvent, la commande "*make*" est directement accessible par l'invite de commande de gdb. Tout comme le shell, vous retrouvez la commande "*apropos [un mot]*" qui permet de retrouver les commandes qui se rapportent à un mot particulier. De même, pour l'aide, vous avez à votre disposition "*help [une commande]*", qui donne une aide générale sur une commande, "*info*", qui donne des informations relatives au programme debuggé et "*show*", qui donne des informations relatives à gdb lui-même. Nous verrons par la suite des cas concrets de leur utilisation.

```
<prompt> $ gdb -q exemple_1
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
```

<sup>2</sup>Procédé consistant à compléter le nom d'une entité, en l'occurrence, en appuyant sur la touche de tabulation

```

user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
internals -- Maintenance commands

```

Type "help" followed by a class name for a list of commands in that class.  
 Type "help" followed by command name for full documentation.  
 Command name abbreviations are allowed if unambiguous.

```

(gdb)
(gdb) s<tab><tab>
search          set          show          source         stop
section         sharedlibrary si         step          symbol-file
select-frame   shell          signal        stepi
(gdb) show <tab><tab>
annotate                input-radix          remotewritesize
args                    language             solib-absolute-prefix
auto-solib-add         listsize             solib-search-path
check                   output-radix         stop-on-solib-events
commands                paths                symbol-reloading
complaints              print                targetdebug
confirm                 prompt               user
convenience             radix                values
copying                 remotebaud           verbose
demangle-style          remotebreak          version
directories              remotecache          warranty
editing                  remotedebug          watchdog
environment              remotedevice         width
gnutarget                remotelogbase        write
height                   remotelogfile
history                   remotetimeout
(gdb) show args

```

Arguments to give program being debugged when it is started is "".

```

(gdb) info <tab><tab>
address          dcache              line                source              udot
all-registers    display             locals              sources             variables
args             files                program             stack               warranty
breakpoints      float               registers           target              watchpoints
catch            frame                set                 terminal
common           functions            sharedlibrary       threads
copying          handle                signals              types
(gdb) info functions

```

All defined functions:

```

File exemple_1.c:
char *falc(char *);
int main(int, char **);

```

```

File /usr/src/gnu/egcs/gcc/libgcc2.c:
void __do_global_ctors();
void __do_global_dtors();
void __main();

```

Non-debugging symbols:

```

00002020 start
0000285c dlopen
00002898 dlclose
000028d0 dlsym
0000290c dlctl

```

```
00002964 dlerror
(gdb)
```

## 2.2 Exécution du programme au sein de gdb

Désormais, il nous faut exécuter le programme et lui indiquer où s'arrêter. Tout d'abord, la commande `"run [arguments]"` comme son nom l'indique lance le programme et permet de lui passer les arguments souhaités exactement à la manière dont vous le feriez au niveau de l'invite de commande du shell<sup>3</sup>. Dans le cas où on aurait besoin de relancer à plusieurs reprises le programme il est possible d'enregistrer une variable qui va contenir nos arguments et les placer automatiquement lorsqu'on utilisera la commande `"run"`. Pour ce faire, `"set args [liste des arguments]"`. Pour les voir `"show args"`. Ce procédé est valable pour initialiser/voir n'importe quelle variable d'environnement (`"set [variable d'environnement]=[valeur]"` et `show [variable]"`).

## 2.3 Gestion des points d'arrêts

### 2.3.1 Mise en place

Maintenant, il faut préciser où l'on souhaite s'arrêter dans le programme. Pour ce faire, nous avons à notre disposition 3 types de fonctions :

- `"breakpoint"` : permet de s'arrêter à une ligne, une fonction ou une adresse précise, un offset ...
- `"watchpoint"` : permet de s'arrêter lorsque la valeur d'une expression change.
- `"catchpoint"` : permet de s'arrêter lorsqu'un évènement particulier intervient.

Pour chacun de ces types de "points d'arrêt", il est possible d'ajouter des conditions dans le but d'affiner le débogage.

Dans la famille des `"breakpoints"`, nous avons :

- `"break [objet]"` : l'objet peut être une fonction, un offset (+offset, -offset), un numéro de ligne, un numéro de ligne dans un fichier ( fichier :numéro.de.ligne ), une adresse mémoire ... C'est la fonction la plus basique pour mettre un point d'arrêt.
- `"tbreak"` : bénéficie des mêmes propriétés que `"break"` mais permet de ne s'arrêter qu'une seule fois. Cette fonction s'avère particulièrement utile pour n'entrer qu'une seule fois dans une boucle.
- `"rbreak"` : place un `"breakpoint"` partout où la regexp<sup>4</sup> donnée en argument est validée.

Les `"watchpoints"` peuvent devenir utiles lorsqu'on ne connaît pas où et quand une expression va intervenir dans un programme, ou encore lorsqu'on ne connaît pas quand celle-ci va être modifiée. Ils se définissent comme suit :

- `"watch expression"` arrête le programme si l'expression est rencontrée.
- `"rwatch expression"` arrête le programme si l'expression est lue par le programme.
- `"awatch expression"` arrête le programme si l'expression est lue ou écrite par le programme.

Quant à eux les `"catchpoints"` peuvent servir à maintes reprises, lorsque le programme charge une bibliothèque en mémoire, lorsqu'un signal particulier lui a été envoyé ... Il

<sup>3</sup>Il est possible de rediriger la sortie du programme vers un autre terminal que celui utilisé par gdb par la commande `"tty nom_du_tty"`, ce qui s'avère très pratique pour éviter de polluer notre espace de débogage et de mieux voir les sorties du programme

<sup>4</sup>"regexp" est l'abréviation d'expression régulière. C'est un motif représentant une chaîne de caractère.

est cependant important de noter que beaucoup de fonctionnalités listées avec la commande “*help catch*” ne sont pas encore implémentées sur un certain nombre d’architectures ( sauf HP-UX ). Les “*catchpoints*” permettent de faire ceci avec les syntaxes citées ( nous ne listons ici que ceux gérés par les architecture de type x86 ) :

- “*catch throw*” capture les appels à la commande “*throw*” en C++.
- “*catch catch*” capture les exceptions en C++.

Tous les exemples de sorties de `gdb` au sein de cette partie font référence au fichier source de l’exemple précédemment étudié.

```
(gdb) rbreak falc*
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
char *falc(char *);
(gdb) b exemple_1.c:35
Breakpoint 2 at 0x2a84: file exemple_1.c, line 35.
(gdb) break 43
Breakpoint 3 at 0x2aa8: file exemple_1.c, line 43.
(gdb) info breakpoints
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x00002a74 in falc at exemple_1.c:31
2  breakpoint      keep y   0x00002a84 in falc at exemple_1.c:35
3  breakpoint      keep y   0x00002aa8 in falc at exemple_1.c:43
(gdb)
```

Une fois mis en places et utilisés ou non, il peut être utile d’enlever certains points d’arrêt. Pour ce faire il suffit d’utiliser “*delete*” ou “*clear*” comme suit (les “*id*” sont récupérés à l’aide de la fonction “*info breakpoints*” et font référence à la première colonne du résultat de cette commande) :

- “*clear*” efface tous les points d’arrêt de la pile d’exécution actuelle.
- “*clear fonction*”
- “*clear fichier :fonction*” enlève n’importe quel point d’arrêt à l’entrée de la fonction spécifiée.
- “*clear numéro\_de\_ligne*”
- “*clear fichier :numéro\_de\_ligne*” la même chose que ci-dessus mais au niveau d’une ligne.
- “*delete [breakpoints] [id1 id2 id3 ...]*” enlève des points d’arrêt spécifiés par leur numéros d’identifiant. Si aucun identifiant n’est donné, tous les points d’arrêts seront effacés.

```
(gdb) b 42
Breakpoint 1 at 0x2aa8: file exemple_1.c, line 42.
(gdb) b 43
Note: breakpoint 1 also set at pc 0x2aa8.
Breakpoint 2 at 0x2aa8: file exemple_1.c, line 43.
(gdb) b 44
Breakpoint 3 at 0x2abc: file exemple_1.c, line 44.
(gdb) b 45
Note: breakpoint 3 also set at pc 0x2abc.
Breakpoint 4 at 0x2abc: file exemple_1.c, line 45.
(gdb) i b
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x00002aa8 in falc at exemple_1.c:42
2  breakpoint      keep y   0x00002aa8 in falc at exemple_1.c:43
3  breakpoint      keep y   0x00002abc in falc at exemple_1.c:44
4  breakpoint      keep y   0x00002abc in falc at exemple_1.c:45
(gdb) l 44
39      strncpy (buf, arg1, taille_buf);
```

```

40
41     DEBUG;
42
43     printf ("----> Le premier argument du programme est : %s\n", buf);
44
45     DEBUG;
46
47     return NULL;
48 }
(gdb) clear exemple_1.c:42
Deleted breakpoint 1
(gdb) i breakpoints
Num Type          Disp Enb Address      What
2  breakpoint      keep y   0x00002aa8 in falc at exemple_1.c:43
3  breakpoint      keep y   0x00002abc in falc at exemple_1.c:44
4  breakpoint      keep y   0x00002abc in falc at exemple_1.c:45
(gdb) delete 3
(gdb) i breakpoints
Num Type          Disp Enb Address      What
2  breakpoint      keep y   0x00002aa8 in falc at exemple_1.c:43
4  breakpoint      keep y   0x00002abc in falc at exemple_1.c:45
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) i b
No breakpoints or watchpoints.
(gdb)

```

### 2.3.2 Activation et désactivation

Bien sûr, pour garder une grande souplesse et éviter les tâches trop répétitives, il est également possible de seulement désactiver les points d'arrêts qui ne nous intéressent pas avec les commandes “*enable*” et “*disable*” :

“*disable [breakpoints] [id1 id2...]*” Désactive les points d'arrêts spécifiés, et tous si aucun identifiant n'est passé en paramètre.

“*enable [breakpoints] [id1 id2...]*” Réactive les points d'arrêt désactivés.

“*enable [breakpoints] once id1 id2...*” Active les points d'arrêts jusqu'à ce qu'ils soient rencontrés une fois, et les désactive ensuite.

“*enable [breakpoints] delete id1 id2...*” cf ci-dessus mais sont effacés.

On peut particulièrement prêter attention aux champs “Disp” et “Enb” qui vont modifier leur état suivant les commandes appliquées.

```

(gdb) b 31
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
(gdb) b 43
Breakpoint 2 at 0x2aa8: file exemple_1.c, line 43.
(gdb) delete 2
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x00002a74 in falc at exemple_1.c:31
2  breakpoint      keep y   0x00002aa8 in falc at exemple_1.c:43
(gdb) disable 2
(gdb) i b
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x00002a74 in falc at exemple_1.c:31
2  breakpoint      keep n   0x00002aa8 in falc at exemple_1.c:43
(gdb) enable once 2

```

```
(gdb) enable delete 1
(gdb) i b
Num Type           Disp Enb Address      What
1  breakpoint      del  y   0x00002a74 in falc at exemple_1.c:31
2  breakpoint      dis  y   0x00002aa8 in falc at exemple_1.c:43
(gdb)
```

### 2.3.3 Définir une condition d'arrêt

Comme nous l'avons évoqué au début, il est également possible, une fois les points d'arrêt fixés, de leur attribuer des conditions pour affiner le débogage. Cela se fait par le biais de la commande "*condition*" à laquelle on passe l'identifiant du point d'arrêt considéré et, bien évidemment, la condition.

```
(gdb) l 39
34
35     buf = (char *) malloc (taille_buf);
36
37     DEBUG;
38
39     strncpy (buf, arg1, taille_buf);
40
41     DEBUG;
42
43     printf ("----> Le premier argument du programme est : %s\n", buf);
(gdb) b 39
Breakpoint 1 at 0x2a94: file exemple_1.c, line 39.
(gdb) condition 1 taille_buf > 10
(gdb) i b
Num Type           Disp Enb Address      What
1  breakpoint      keep y 0x00002a94 in falc at exemple_1.c:39
    stop only if taille_buf > 10
(gdb)
```

### 2.3.4 Exécuter une série de commandes

De même il est possible d'exécuter une liste de commande à chaque fois le point d'arrêt rencontré. Imaginons que nous voulions regarder le contenu du second argument de la fonction "*strncpy*" ainsi que la taille du buffer lorsqu'on s'arrête à la ligne 39. Pour ce faire, il suffit de procéder comme suit :

```
(gdb) l 39
34
35     buf = (char *) malloc (taille_buf);
36
37     DEBUG;
38
39     strncpy (buf, arg1, taille_buf);
40
41     DEBUG;
42
43     printf ('`----> Le premier argument du programme est : %s\n', buf);
(gdb) b 39
Breakpoint 1 at 0x2a94: file exemple_1.c, line 39.
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just ``end``.
>silent
>printf "Argument 2 : %s\n", buf
```

```
>printf "Argument 3 : %d\n", taille_buf
>end
(gdb) run une_chaine
Starting program:
/un_rep/pouet/exemple_1 une_chaine
Argument 2 :
Argument 3 : 10
(gdb)
```

Pour retirer une liste de commandes associées à un point d'arrêt :

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>end
```

Maintenant que faire lorsqu'on est arrêté pour poursuivre nos investigations dans un programme "buggé" ?

## 2.4 Méthodes pour avancer dans le programme une fois stoppé

Comme pour les thèmes précédemment abordés nous le laissent supposer, une multitude de fonctions sont disponibles afin d'indiquer comment continuer l'exécution du programme. Par exemple, si nous souhaitons faire du pas à pas à, exécuter un nombre précis d'instructions assembleur, une ligne d'instructions dans le langage utilisé dans le fichier source...

### 2.4.1 Continuer l'exécution du programme

Le but des fonctions citées ci-après, est de continuer l'exécution du programme comme il le ferait normalement jusqu'au prochain point d'arrêt ou jusqu'à la fin du programme (qui peut être une erreur). L'argument qui peut leur être passé consiste à ignorer "nb\_ignorer" fois le point d'arrêt sur lequel on est arrêté lors de la suite de l'exécution.

Pour ce faire deux commandes :

- "continue [nb\_ignorer]"
- "fg [nb\_ignorer]"

Dans notre fichier d'exemple précédent :

```
(gdb) b 31
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
(gdb) b 35
Breakpoint 2 at 0x2a84: file exemple_1.c, line 35.
(gdb) b 39
Breakpoint 3 at 0x2a94: file exemple_1.c, line 39.
(gdb) i b
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x00002a74 in falc at exemple_1.c:31
2  breakpoint      keep y   0x00002a84 in falc at exemple_1.c:35
3  breakpoint      keep y   0x00002a94 in falc at exemple_1.c:39
(gdb) run une_string
Starting program:
/repalc/exemple_1 une_string

Breakpoint 1, falc (arg1=0xf7fffa6c "une_string") at exemple_1.c:31
31      taille_buf = strlen (arg1);
(gdb) run une_string
Starting program:
```

```
/repalc/exemple_1 une_string

Breakpoint 2, falc (arg1=0xf7fffa6c "une_string") at exemple_1.c:35
35      buf = (char *) malloc (taille_buf);
(gdb) c 3
Will ignore next 2 crossings of breakpoint 2. Continuing.

Breakpoint 3, falc (arg1=0xf7fffa6c "une_string") at exemple_1.c:39
39      strncpy (buf, arg1, taille_buf);
(gdb) c
Continuing.
----> Le premier argument du programme est : une_string

Program exited normally.
(gdb)
```

#### 2.4.2 Exécution pas à pas

Dans ce contexte, nous allons voir qu'il est non seulement possible d'exécuter une ligne du fichier source, mais aussi une instruction assembleur, de ne pas exécuter des parties de code ne contenant pas de symboles de débogage ...

Tout d'abord, "*step*" permet d'exécuter la ligne de code courante et de s'arrêter à la suivante. C'est pour ça que lorsqu'on souhaite débogger une fonction ne comportant pas de symboles de débogage, l'ensemble de la fonction est exécutée et non pas une seule ligne du code source. Si toutefois on souhaite s'arrêter à la première ligne d'une telle fonction, "*set step-mode on*" va nous le permettre. Il est possible de passer en argument le nombre de lignes que doit exécuter gdb. Dans le même ordre d'idée, "*next*" permet d'exécuter une ligne de code mais cette fois ne rentre pas dans les fonction. Elle les exécute et poursuit à la ligne qui les suit.

```
(gdb) b main
Breakpoint 1 at 0x2a0c: file exemple_1.c, line 18.
(gdb) run une_string
Starting program:
/repalc/exemple_1 une_string

Breakpoint 1, main (argc=2, argv=0xf7fff9dc) at exemple_1.c:18
18      falc (argv[1]);
(gdb) step
falc (arg1=0xf7fffa6c "une_string") at exemple_1.c:31
31      taille_buf = strlen (arg1);
(gdb) s
35      buf = (char *) malloc (taille_buf);
(gdb) c
Continuing.
----> Le premier argument du programme est : une_string

Program exited normally.
(gdb) run une_autre_string
Starting program:
/repalc/exemple_1 une_autre_string

Breakpoint 1, main (argc=2, argv=0xf7fff9d4) at exemple_1.c:18
18      falc (argv[1]);
(gdb) next
----> Le premier argument du programme est : une_autre_string
21      return 0;
(gdb) n
```

```

22     }
(gdb) n
0x20a0 in start ()
(gdb)
Single stepping until exit from function start,
which has no line number information.

```

```

Program exited normally.
(gdb)

```

Lorsqu'on souhaite continuer le programme jusqu'à la fonction dans laquelle nous trouvons se termine, nous pouvons utiliser "*finish*" qui en plus affichera la valeur de retour si il y en a une.

```

(gdb) b 31
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
(gdb) run chaine_de_caractere
Starting program:
/repalc/exemple_1 chaine_de_caractere

Breakpoint 1, falc (arg1=0xf7fffa64 "chaine_de_caractere") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) finish
Run till exit from #0  falc (arg1=0xf7fffa64 "chaine_de_caractere")
    at exemple_1.c:31
----> Le premier argument du programme est : chaine_de_caractere
main (argc=2, argv=0xf7fff9d4) at exemple_1.c:21
21     return 0;
Value returned is $1 = 0x0
(gdb) c
Continuing.

```

```

Program exited normally.
(gdb)

```

Lorsqu'on souhaite continuer jusqu'à un point précis, nous pouvons utiliser "*until emplacement*", ou l'emplacement est défini exactement de la même façon que les points d'arrêt.

```

(gdb) b 31
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
(gdb) run toto_en_short
Starting program:
/repalc/exemple_1 toto_en_short

Breakpoint 1, falc (arg1=0xf7fffa64 "toto_en_short") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) until exemple_1.c:43
falc (arg1=0xf7fffa64 "toto_en_short") at exemple_1.c:43
43     printf ("----> Le premier argument du programme est : %s\n", buf);
(gdb) c
Continuing.
----> Le premier argument du programme est : toto_en_short

```

```

Program exited normally.
(gdb)

```

Si nous nous retrouvons confronté au problème de voir comment fonctionnent des programmes, bibliothèques ou autres n'ayant aucun symboles de débogage, les commandes "*stepi*" et "*nexti*" s'avèrent particulièrement utiles. Elles permettent toutes deux de se limiter à l'exécution d'une instruction assembleur. Tout comme "*step*" et

“*next*”, “*stepi*” n’exécute qu’une seule instruction et s’arrête à celle qui suit et “*nexti*” fait de même sauf dans le cas d’un appel de fonction. Dans ce cas, elle exécute toute la fonction et s’arrête à la ligne qui suit l’appel. Si nous souhaitons exécuter “*nb*” fois l’une ou l’autre de celles-ci, il suffit de passer “*nb*” en paramètre. Pour une question de clareté, il est intéressant de taper la séquence “*display/i \$pc*” dont vous comprendrez le sens ultérieurement, si ce n’est déjà le cas.

```
(gdb) b 31
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
(gdb) run toto_en_slip
Starting program:
/repalc/exemple_1 toto_en_slip

Breakpoint 1, falc (arg1=0xf7ffa64 "toto_en_slip") at exemple_1.c:31
31      taille_buf = strlen (arg1);
(gdb) si
0x2a78 31      taille_buf = strlen (arg1);
(gdb) display/i $pc
1: x/i $pc 0x2a78 <falc+12>:   call 0x4094 <_DYNAMIC+148>
(gdb) si
0x2a7c 31      taille_buf = strlen (arg1);
1: x/i $pc 0x2a7c <falc+16>:   nop
(gdb)
0x4094 in _DYNAMIC ()
1: x/i $pc 0x4094 <_DYNAMIC+148>:   save %sp, -96, %sp
(gdb) ni
----> Le premier argument du programme est : toto_en_slip

Program exited normally.
(gdb)
```

## 2.5 Interception et traitement des signaux

Les signaux en question sont ceux qui nous permettent de dialoguer avec nos processus. Pour en avoir la liste sous l’invite du shell il suffit de taper “*kill -l*” et pour avoir la liste de ceux qui peuvent être gérés par gdb, “*info signals*”. Dans cette liste, nous pouvons indiquer comment gdb doit réagir sur tel ou tel signal par le biais de la commande “*handle signal réaction*” où “réaction” fait partie de la liste suivante :

- “*nostop*” lorsque le signal est intercepté, gdb nous l’indique, mais continue l’exécution du programme.
- “*stop*” stoppe le programme lorsque le signal est intercepté.
- “*print*” affiche un message lorsque le signal est intercepté.
- “*noprint*” n’affiche rien et ne stoppe pas le programme.
- “*pass*”
- “*noignore*” le signal peut être intercepté par le programme
- “*nopass*”
- “*ignore*” ce signal est masqué par gdb.

```
(gdb) b main
Breakpoint 1 at 0x2a0c: file exemple_1.c, line 18.
(gdb) run chaine_test
Starting program:
/repalc/exemple_1 chaine_test
```

```
Breakpoint 1, main (argc=2, argv=0xf7fff9dc) at exemple_1.c:18
18          falc (argv[1]);
```

```
(gdb) shell ps
  PID TT  STAT      TIME COMMAND
14367 p7  INs       0:15.76 -zsh (zsh)
31065 p7  TN        0:00.83 gdb -q exemple_1
26728 p7  SN+       0:00.45 gdb -q exemple_1
21075 p7  TNX       0:00.13 /repalc/exemple_1 chaine_test
 2890 p7  RN+       0:00.11 ps
 8909 p8  IWNs      0:00.39 -zsh (zsh)
(gdb) shell kill 21075
(gdb) c
Continuing.
```

```
Program received signal SIGTERM, Terminated.
main (argc=2, argv=0xf7fff9dc) at exemple_1.c:18
18          falc (argv[1]);
(gdb) c
Continuing.
```

```
Program terminated with signal SIGTERM, Terminated.
The program no longer exists.
(gdb) run une_2eme_chaine
Starting program:
/repalc/exemple_1 une_2eme_chaine
```

```
Breakpoint 1, main (argc=2, argv=0xf7fff9d4) at exemple_1.c:18
18          falc (argv[1]);
```

```
(gdb) i signal SIGTERM
Signal      Stop      Print    Pass to program Description
SIGTERM    Yes       Yes      Yes          Terminated
(gdb) handle SIGTERM nopass
Signal      Stop      Print    Pass to program Description
SIGTERM    Yes       Yes      No           Terminated
(gdb) shell ps
  PID TT  STAT      TIME COMMAND
14367 p7  INs       0:15.76 -zsh (zsh)
31065 p7  TN        0:00.83 gdb -q exemple_1
26728 p7  SN+       0:00.53 gdb -q exemple_1
11874 p7  TNX       0:00.13 /repalc/exemple_1 une_2eme_chaine
26296 p7  RN+       0:00.09 ps
 8909 p8  IWNs      0:00.39 -zsh (zsh)
(gdb) shell kill 11874
(gdb) c
Continuing.
```

```
Program received signal SIGTERM, Terminated.
main (argc=2, argv=0xf7fff9d4) at exemple_1.c:18
18          falc (argv[1]);
(gdb) c
Continuing.
----> Le premier argument du programme est : une_2eme_chaine
```

```
Program exited normally.
(gdb)
```

## 2.6 Considérations sur les programmes multitâches

Comme il y a deux façons de rendre un programme multitâches, il y a également deux modes de traitement.

### 2.6.1 Les programmes utilisant le fork()

Pour celà il n’y a pas de solution réellement pratique sauf sur HP-UX 11.x. Il va falloir regarder le PID du processus “forké” et dire à gdb de s’y attacher, et ce avant que celui-ci n’exécute la suite de ses instructions. Nous allons alors être contraint d’utiliser une astuce. Par exemple, la fonction “sleep()” devrait nous en donner le temps.

Voici le fichier source que nous allons prendre en compte pour l’exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

char * falc (char *arg1);

int main (int argc, char **argv)
{
    int  etat = 0;
    pid_t pid_enfant;

    puts (``C'est parti !!!'');

    pid_enfant = fork();

    if (!pid_enfant)
        falc(argv[1]);
    else
        wait(&etat);

    return 0;
}

char * falc (char *arg1)
{
    char *buf;
    unsigned int taille_buf;

    sleep (20);

    taille_buf = strlen (arg1);

    buf = (char *) malloc (taille_buf);

    strncpy (buf, arg1, taille_buf);

    printf (``----> L'argument du prog. est : %s\n'', buf);

    return NULL;
}
```

}

Voici donc un exemple de débogage d'un programme utilisant la fonction "fork()" :

```
(gdb) run blablatrucpouet&
Starting program:
/repalc/exemple_2 blablatrucpouet&

Program exited normally.
Cannot access memory at address 0x4014.
(gdb) C'est parti !!!

(gdb) shell ps
  PID TT  STAT      TIME COMMAND
 2410 p7  SN+      0:00.30 gdb -q exemple_2
 23971 p7  SN       0:00.05 /repalc/exemple_2 blablatrucpouet
  9287 p7  SN       0:00.01 /repalc/exemple_2 blablatrucpouet
 18146 p7  RN+     0:00.11 ps
  8909 p8  IWNs    0:00.39 -zsh (zsh)
(gdb) attach 9287
Attaching to program
/repalc/exemple_2', process 9287
Reading symbols from /usr/libexec/ld.so{\ldots} done.
Reading symbols from /usr/lib/libc.so.26.2{\ldots} done.
0x8064910 in nanosleep ()
(gdb) b 34
Breakpoint 1 at 0x2ac8: file exemple_2.c, line 34.
(gdb) c
Continuing.

Breakpoint 1, falc (arg1=0xf7fffa64 "blablatrucpouet") at exemple_2.c:34
Source file is more recent than executable.
34      taille_buf = strlen (arg1);
(gdb) c
Continuing.
----> L'argument du prog. est : blablatrucpouet

Program exited normally.
(gdb)
```

## 2.6.2 Programme utilisant les pthreads

Dans ce cas, gdb est un peu plus à l'aise. Tout d'abord, lorsqu'on s'arrête dans un "thread", tous les autres s'arrêtent également. Pour naviguer d'un "thread" à l'autre, "thread [id\_thread]" est particulièrement pratique. De même, si on ne souhaite mettre un point d'arrêt que sur l'un d'eux, "break emplacement thread id\_thread condition" (par défaut, lorsqu'on utilise "break", le point d'arrêt est mis sur tous les "threads").

Soit le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int pthread_falc (char *arg);
```

```

int main (void)
{
    pthread_t pth_id1, pth_id2;

    pthread_create (&pth_id1, NULL, (void *) pth_falc, "PTH_1");
    pthread_create (&pth_id2, NULL, (void *) pth_falc, "PTH_2");

    pthread_join (pth_id1, NULL);
    pthread_join (pth_id2, NULL);

    return 0;
}

int pth_falc(char *arg)
{
    sleep(2);
    printf ("--> %s <---\n", arg);

    pthread_exit(0);
}

```

Voici ce que pourrait donner une séance de débogage dessus :

```

(gdb) b pth_falc
Breakpoint 1 at 0x804859a: file test.c, line 28.
(gdb) run
Starting program: /repa/c/exemple_3
[New Thread 1024 (LWP 19135)]
[New Thread 2049 (LWP 19136)]
[New Thread 1026 (LWP 19137)]
[New Thread 2051 (LWP 19138)]
[Switching to Thread 1026 (LWP 19137)]

Breakpoint 1, pth_falc (arg=0x8048620 "PTH_1") at test.c:28
28      sleep(2);
(gdb) n
[Switching to Thread 2051 (LWP 19138)]

Breakpoint 1, pth_falc (arg=0x8048626 "PTH_2") at test.c:28
28      sleep(2);
(gdb) i threads
* 4 Thread 2051 (LWP 19138) pth_falc (arg=0x8048626 "PTH_2") at test.c:28
  3 Thread 1026 (LWP 19137) pth_falc (arg=0x8048620 "PTH_1") at test.c:29
  2 Thread 2049 (LWP 19136) 0x400f42e0 in __poll (fds=0x804b5f0, nfd=1,
    timeout=2000) at ../sysdeps/unix/sysv/linux/poll.c:52
  1 Thread 1024 (LWP 19135) 0x4006116a in __sigsuspend (set=0xbffffa2c)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:54
(gdb) p arg
$1 = 0x8048626 "PTH_2"
(gdb) thread 3
[Switching to thread 3 (Thread 1026 (LWP 19137))]
#0 pth_falc (arg=0x8048620 "PTH_1") at test.c:29
29      printf ("--> %s <---\n", arg);
(gdb) p arg
$2 = 0x8048620 "PTH_1"
(gdb) b 29 thread 4
Breakpoint 2 at 0x80485a7: file test.c, line 29.
(gdb) c

```

```

Continuing.
[Switching to Thread 2051 (LWP 19138)]

Breakpoint 1, pth_falc (arg=0x8048626 "PTH_2") at test.c:28
28      sleep(2);
(gdb) c
Continuing.
--> PTH_1 <---

Breakpoint 2, pth_falc (arg=0x8048626 "PTH_2") at test.c:29
29      printf ("--> %s <---\n", arg);
(gdb) thread 1
[Switching to thread 1 (Thread 1024 (LWP 19135))]
#0  0x4006116a in __sigsuspend (set=0xbffffa2c)
    at ../sysdeps/unix/sysv/linux/sigsuspend.c:54
54      ../sysdeps/unix/sysv/linux/sigsuspend.c: No such file or directory.
(gdb) c
Continuing.
[Switching to Thread 2051 (LWP 19138)]

Breakpoint 2, pth_falc (arg=0x8048626 "PTH_2") at test.c:29
29      printf ("--> %s <---\n", arg);
(gdb) c
Continuing.
--> PTH_2 <---
Ack! Thread Exited event.  What do I do now???
(gdb)

```

Il est intéressant de noter que les politiques de “scheduling<sup>5</sup>” des threads pouvant être différentes, il peut arriver que lorsqu’on exécute une instruction dans un “thread”, il puissent y en avoir plusieurs d’exécutées dans d’autres.

## 2.7 Examiner le contenu de la pile

La pile est un emplacement mémoire réservé lors de l’initialisation d’un programme. Elle va lui servir de zone d’échange pour les passages de paramètres, l’exécution de certaines séquences de code ... Par exemple, lors d’un appel de fonction, la pile va stocker, l’adresse de retour de la fonction appelante, les arguments classés par ordre inverse de leur entrée et les variables locales. Ceci sera valable à chaque fois que le programme va accéder à une fonction. Notre pile va donc être fragmentée. Ces fragments de piles, vont être qualifiés de “stack frames”, ou “frames” dans gdb. Cette approche va nous paraître un peu plus claire dans les exemples qui vont suivre.

Gdb va nous permettre de voir une partie du contenu de la pile en tapant la commande “*bt [-][décal\_frame]*” où *décal\_frame* sera une valeur indiquant si on souhaite se déplacer de + ou - *décal\_frame* par rapport à la “frame” courante. Si nous lui passons en paramètre l’argument “*full*”, gdb va nous retourner l’ensemble des variables locales. Nous allons également pouvoir nous déplacer dans ces “frames” en utilisant “*frame argument*” où l’argument peut être une adresse ou un nombre (“*select-frame argument*” est la version silencieuse). De même nous avons “*up nb*” et “*down nb*” permettant de monter ou descendre de “*nb*” frames (“*up-silently arg*” et “*down-silently arg*” correspondant aux équivalents “silencieux”).

<sup>5</sup>Le “scheduling” est la manière dont le système va ordonner l’exécution des processus en leur affectant des priorités plus ou moins importantes

Nous pouvons également récupérer des informations sur la frame courante en tapant “frame”, sur une “frame” à une adresse particulière avec “i f addr”, et des informations sur les variables locales avec “*info locals*”.

Pour la sortie écran de ce qui suit nous avons utilisé “exemple\_1.c”

```
(gdb) b falc
Breakpoint 1 at 0x80484ba: file exemple_1.c, line 31.
(gdb) run tareumenshort
Starting program: /tmp/exemple_1 tareumenshort

Breakpoint 1, falc (arg1=0xbffffdbb "tareumenshort") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) bt
#0  falc (arg1=0xbffffdbb "tareumenshort") at exemple_1.c:31
#1  0x80484a7 in main (argc=2, argv=0xbffffcd4) at exemple_1.c:18
#2  0x400382eb in __libc_start_main (main=0x8048490 <main>, argc=2,
    ubp_av=0xbffffcd4, init=0x8048308 <_init>, fini=0x804854c <_fini>,
    rtdl_fini=0x4000c130 <_dl_fini>, stack_end=0xbffffccc)
    at ../sysdeps/generic/libc-start.c:129
(gdb) bt full
#0  falc (arg1=0xbffffdbb "tareumenshort") at exemple_1.c:31
    buf = 0xbffffc68 "\210\202\003@\002"
    taille_buf = 1073831440
#1  0x80484a7 in main (argc=2, argv=0xbffffcd4) at exemple_1.c:18
No locals.
#2  0x400382eb in __libc_start_main (main=0x8048490 <main>, argc=2,
    ubp_av=0xbffffcd4, init=0x8048308 <_init>, fini=0x804854c <_fini>,
    rtdl_fini=0x4000c130 <_dl_fini>, stack_end=0xbffffccc)
    at ../sysdeps/generic/libc-start.c:129
    ubp_av = (char **) 0xbffffcd4
    fini = (void (*)()) 0x40015d64 <_dl_debug_mask>
    rtdl_fini = (void (*)()) 0x1
    ubp_ev = (char **) 0xbffffcd8
(gdb) select-frame 0
(gdb) up
#1  0x80484a7 in main (argc=2, argv=0xbffffcd4) at exemple_1.c:18
18     falc (argv[1]);
(gdb) down
#0  falc (arg1=0xbffffdbb "tareumenshort") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) frame
#0  falc (arg1=0xbffffdbb "tareumenshort") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) info locals
buf = 0xbffffc68 "\210\202\003@\002"
taille_buf = 1073831440
(gdb) n
35     buf = (char *) malloc (taille_buf);
(gdb)
39     strncpy (buf, arg1, taille_buf);
(gdb)
43     printf ("----> Le premier argument du programme est : %s\n", buf);
(gdb) info locals
buf = 0x80496f0 "tareumenshort"
taille_buf = 13
(gdb) c
Continuing.
----> Le premier argument du programme est : tareumenshort
```

```
Program exited normally.
(gdb)
```

## 2.8 Examiner le contenu de la mémoire

Pour parvenir à un débogage efficace, il faut désormais connaître quelles sont les valeurs des variables à un instant donné. Pour ce faire, un certain nombre de formats et de commandes permettent de voir celles-ci à différents niveaux ; de l'assembleur aux niveaux du langage utilisé pour constituer le programme. En voici les principales :

- `"print [/format] [expression]"`
- `"x [/[nombre]format] [expression]"`
- `"display [/format] [expression]"`

"x" permet un affichage plus bas niveau que "print". L'expression peut être une simple variable ou une expression complexe. Les méthodes d'accès aux variables sont du même type que ceux définis pour les points d'arrêts vus précédemment (Exemple : "print fonction : :variable"), et il faut que les variables auxquelles nous souhaitons accéder soient visibles de la fonction. Comme vous pouvez le remarquer, il est possible de passer un format en paramètre pour les trois commandes. Voici une liste des formats pris en compte :

- x - regarder les bits de la valeur en tant qu'entier et l'affiche en hexadécimal.
- d - affiche en tant qu'entier en décimal signé.
- u - affiche en tant qu'entier en décimal non signé.
- o - affiche l'entier en octal.
- t - affiche l'entier en binaire.
- c - affiche l'entier en tant que caractère constant.
- f - affiche la valeur en tant qu'un "float".

En plus de la méthode d'affichage, nous pouvons forcer gdb à considérer une variable ou zone mémoire comme étant d'un type particulier comme ceci : `"print type expression"`. Si vous utilisez la commande "x", on peut indiquer que l'on veut répéter "nombre" de fois l'opération. Il est possible de faire de même avec "print" comme ceci : `"print expression@nombre"`.

```
(gdb) b falc
Breakpoint 1 at 0x80484ba: file exemple_1.c, line 31.
(gdb) run pouet_zob
Starting program: /tmp/exemple_1 pouet_zob

Breakpoint 1, falc (arg1=0xbffffdbf "pouet_zob") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) n
35     buf = (char *) malloc (taille_buf);
(gdb)
39     strncpy (buf, arg1, taille_buf);
(gdb) p buf
$1 = 0x80496f0 ""
(gdb) n
43     printf ("----> Le premier argument du programme est : %s\n", buf);
(gdb) p buf
$2 = 0x80496f0 "pouet_zob"
(gdb) p /x buf
$3 = 0x80496f0
(gdb) p /x buf@4
$4 = {0x80496f0, 0xbffffc6c, 0x80484a7, 0xbffffdbf}
```

```
(gdb) p /x *buf@4
$5 = {0x70, 0x6f, 0x75, 0x65}
x/4t *buf@4
0x80496f0:      0110010101110101011011101110000      0110111101111010010111101110100
                00000000000000000000000001100010      000000000000000000000100100001001
(gdb) x/10i falc
0x80484b4 <falc>:      push   %ebp
0x80484b5 <falc+1>:     mov    %esp,%ebp
0x80484b7 <falc+3>:     sub    $0x18,%esp
0x80484ba <falc+6>:     add    $0xffffffff4,%esp
0x80484bd <falc+9>:     mov    0x8(%ebp),%eax
0x80484c0 <falc+12>:    push  %eax
0x80484c1 <falc+13>:    call  0x8048370 <strlen>
0x80484c6 <falc+18>:    add    $0x10,%esp
0x80484c9 <falc+21>:    mov    %eax,%eax
0x80484cb <falc+23>:    mov    %eax,0xffffffff8(%ebp)
(gdb) c
Continuing.
----> Le premier argument du programme est : pouet_zob

Program exited normally.
(gdb)
```

Comme on peut le voir dans l'exemple précédent, "x" est plus habilité à consulter l'état de la mémoire en certains points. Cette commande comporte quelques format autres que ceux cités précédemment :

- i - affiche une instruction en langage machine
- s - affiche une chaîne de caractères
- b - affiche un "byte"
- h - affiche un "half-word"
- w - affiche un "word"
- g - affiche huit "bytes"

Soulignons que lorsqu'on modifie le format d'affichage, il reste valable au prochain appel de la commande.

Finalement, "display" a la même vocation que "print" à l'exception près qu'à chaque arrêt du programme, la ou les expressions souhaitées seront affichées. Comme les autres commandes du même type, il est possible d'activer ou de désactiver un "display" par "enable display id\_display", et "disable display id\_display" ou de l'éliminer complètement par "undisplay id\_display" ("id\_display" est obtenu par "info display").

```
(gdb) b falc
Breakpoint 1 at 0x80484ba: file exemple_1.c, line 31.
(gdb) run zobby_la_mouche
Starting program: /tmp/exemple_1 zobby_la_mouche

Breakpoint 1, falc (arg1=0xbffffdb9 "zobby_la_mouche") at exemple_1.c:31
31      taille_buf = strlen (arg1);
(gdb) display buf
1: buf = 0xbfffc68 "\210\202\003@\002"
(gdb) n
35      buf = (char *) malloc (taille_buf);
1: buf = 0xbfffc68 "\210\202\003@\002"
(gdb)
39      strncpy (buf, arg1, taille_buf);
1: buf = 0x80496f0 ""
```

```
(gdb)
43     printf ("----> Le premier argument du programme est : %s\n", buf);
1: buf = 0x80496f0 "zobby_la_mouche"
(gdb)
----> Le premier argument du programme est : zobby_la_mouche
47     return NULL;
1: buf = 0x80496f0 "zobby_la_mouche"
(gdb)
48     }
1: buf = 0x80496f0 "zobby_la_mouche"
(gdb)
main (argc=2, argv=0xbffffcd4) at exemple_1.c:21
21     return 0;
(gdb)
0x80484b0      22     }
(gdb)
0x400382eb in __libc_start_main (main=0x8048490 <main>, argc=2,
    ubp_av=0xbffffcd4, init=0x8048308 <_init>, fini=0x804854c <_fini>,
    rtdld_fini=0x4000c130 <_dl_fini>, stack_end=0xbffffccc)
    at ../sysdeps/generic/libc-start.c:129
129     ../sysdeps/generic/libc-start.c: No such file or directory.
(gdb)
```

Si vous n'aimez pas l'affichage créé par `print`, il est possible de le modifier légèrement. La liste de ce qui est modifiable est accessible par "`show print`". Pour activer il suffit de faire "`set print option on`" et pour désactiver "`set print option off`". Une autre petite chose qui peut être utile est que "`print`" conserve un historique de ses différents appels, visible par "`show values`". Ainsi il est possible de rappeler une des valeurs comme suit : "`print $id_historique`", ou "`print $$n-ième_val`" pour accéder à la n-ième valeur depuis la fin.

## 2.9 Examiner le contenu des registres

Pour vraiment bien comprendre quel peut être l'origine d'un bug, surtout lorsqu'on travaille en assembleur, il est intéressant d'avoir la possibilité de consulter l'état des registres à un instant donné. Il suffit alors d'utiliser la commande "`info registers`" ou "`info all-registers`" pour pouvoir consulter tous les registres (même ceux de gestion des nombres à virgule flottante). Si vous n'êtes intéressé que par le contenu d'un seul, "`print $registre6`", ou par le contenu des registres de gestion des chiffres à virgule flottante, "`info float`".

Voici un exemple sur architecture x86 :

```
(gdb) b falc
Breakpoint 1 at 0x80484ba: file exemple_1.c, line 31.
(gdb) run blobby
Starting program: /repa/c/exemple_1 blobby

Breakpoint 1, falc (arg1=0xbffffdc2 "blobby") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) info all-registers
eax             0xbffffcd8      -1073742632
ecx             0x1             1
edx             0xbffffdc2      -1073742398
ebx             0x40126a58      1074948696
```

<sup>6</sup>Par convention, le registre pointant sur l'adresse de la prochaine instruction à exécuter est nommée "pc". Pointer sur "eip" reste toutefois faisable.

```

esp          0xbffffc34      0xbffffc34
ebp          0xbffffc4c      0xbffffc4c
esi          0x40015d64      1073831268
edi          0xbffffcd4      -1073742636
eip          0x80484ba       0x80484ba
eflags      0x282      642
cs          0x23      35
ss          0x2b      43
ds          0x2b      43
es          0x2b      43
fs          0x0      0
gs          0x0      0
st0         0      (raw 0x00000000000000000000)
st1         0      (raw 0x00000000000000000000)
st2         0      (raw 0x00000000000000000000)
st3         0      (raw 0x00000000000000000000)
st4         0      (raw 0x00000000000000000000)
st5         2992     (raw 0x400abb00000000000000)
st6         1006712931 (raw 0x401cf004e18c00000000)
st7         854      (raw 0x4008d580000000000000)
fctrl       0x37f      895
fstat       0x0      0
ftag        0xffff     65535
fiseg       0x23      35
fioff       0x808c6cd     134792909
foseg       0x2b      43
fooff       0xbffffda8     -1073742424
fop         0x35d      861
(gdb) p/x $eax
$1 = 0xbffffcd8
(gdb) c
Continuing.
----> Le premier argument du programme est : blobby

Program exited normally.
(gdb)

```

### Le même exemple sur architecture sparc V8 :

```

(gdb) b falc
Breakpoint 1 at 0x2a74: file exemple_1.c, line 31.
(gdb) run blobby
Starting program: /repalc/exemple_1 blobby

Breakpoint 1, falc (arg1=0xf7fffa6c "blobby") at exemple_1.c:31
31      taille_buf = strlen (arg1);
(gdb) info all-registers
g0          0x0      0
g1          0x8058000     134578176
g2          0x1000000     16777216
g3          0x81c06000    -2118098944
g4          0x0      0
g5          0x0      0
g6          0x316e     12654
g7          0xffffffff    -1
o0          0x1      1
o1          0x4000     16384
o2          0x40400083    1077936259
o3          0x1      1
o4          0x2098     8344

```

```

o5          0xffffffff          -1
sp          0xf7fff8a8          -134219608
o7          0x2bb8    11192
10          0xfa0bd798          -99887208
11          0xf8228800          -131954688
12          0x0            0
13          0x7            7
14          0x8            8
15          0x300    768
16          0xfa103000          -99602432
17          0x80ac060          134922336
i0          0xf7fffa6c          -134219156
i1          0xf7fff9e0          -134219296
i2          0xf7fff9dc          -134219300
i3          0xfa104fb0          -99594320
i4          0x3            3
i5          0x1014    4116
fp          0xf7fff918          -134219496
i7          0x2a1c    10780
f0          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f1          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f2          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f3          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f4          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f5          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f6          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f7          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f8          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f9          -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f10         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f11         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f12         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f13         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f14         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f15         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f16         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f17         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f18         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f19         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f20         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f21         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f22         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f23         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f24         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f25         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f26         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f27         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f28         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f29         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f30         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
f31         -NaN(0x7ffffff)      (raw 0xffffffff)      -NaN(0xffffffffffffff)
y          0xfd000000          -50331648
psr        0x40900081          1083179137
wim        0x0            0
tbr        0x0            0
pc         0x2a74    10868
npc        0x2a78    10872
fpsr       0x0            0

```

```

cpsr          0x0          0
(gdb) p/x $g1
$1 = 0x8058000
(gdb) c
Continuing.
----> Le premier argument du programme est : blobby

Program exited normally.
(gdb)

```

## 2.10 Informations concernant les fichiers source

Voici quelques fonctions permettant de garder un œil sur les sources et ce qu'elles sont devenues suite à la compilation :

- “*list*” - liste le contenu du fichier source.
- “*list id\_ligne*” - liste le contenu à partir de “*id\_ligne*”.
- “*list id\_ligne1, id\_ligne2*” - liste de “*id\_ligne1*” à “*id\_ligne2*”.
- “*show listsize*” - indique le nombre de lignes à afficher courant.
- “*set listsize [nombre]*” - initialize le nombre de lignes à afficher à “*nombre*”.
- “*search [regex]*” - recherche l'occurrence de “*regex*” qui est une expression régulière après la ligne courante.
- “*reverse-search [regex]*” - l'inverse de ci-dessus.
- “*info line id\_ligne*” - pour connaître l'adresse de début et de fin d'une ligne.
- “*disassemble adresse*”
- “*disassemble début\_plage fin\_plage*” - désassemble la zone mémoire spécifiée.
- “*whatis [expression]*” - pour connaître le type d'une expression.
- “*ptype [expression]*” - pour avoir une description de l'expression (très sympa sur des structures).
- “*info types [regex]*” - avoir les informations sur le type pour les occurrences de l'expression régulière.
- “*info scope*” - liste toutes les variables pour une étendue donnée.
- “*info source*” - donne des informations sur le fichier source courant.
- “*infos sources*” - donne des informations sur l'ensemble des fichiers source.
- “*info function[s] [regex]*” - donne des informations sur les fonctions dont l'occurrence de “*regex*” est validée.
- “*info variables*” - liste toutes les variables déclarées à l'extérieur de la fonction.
- “*info share*” - liste les bibliothèques utilisées.

## 2.11 Modifier le comportement du programme

Une fois le “bug” cerné, modifier certaines variables ou registres peut nous permettre d'économiser pas mal de temps surtout dans le cadre de gros projets longs à compiler.

Tout d'abord, “*set [type]var\_ou\_registre*”, permet de modifier la valeur d'une variable ou d'un registre, et si nécessaire de lui indiquer le type qu'on souhaite utiliser pour faire l'affectation.

Une fois la valeur d'une variable ou registre modifiée il peut être intéressant de ne recommencer l'exécution du programme qu'à partir d'un endroit particulier du programme, ce que permet “*jump ligne\_ou\_adresse*”, ou encore d'envoyer un signal au processus courant par “*kill nom\_ou\_numero\_du\_signal*”.

```
(gdb) b falc
Breakpoint 1 at 0x80484ba: file exemple_1.c, line 31.
(gdb) run chaine_originelle
Starting program: /reपालc/exemple_1 chaine_originelle

Breakpoint 1, falc (arg1=0xbffffdb7 "chaine_originelle") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) n
35     buf = (char *) malloc (taille_buf);
(gdb)
39     strncpy (buf, arg1, taille_buf);
(gdb)
43     printf ("----> Le premier argument du programme est : %s\n", buf);
(gdb)
----> Le premier argument du programme est : chaine_originelle
47     return NULL;
(gdb) set buf="nouvelle_chaine"
(gdb) jump 43
Continuing at 0x80484f9.
----> Le premier argument du programme est : nouvelle_chaine

Program exited normally.
(gdb) run zob

Breakpoint 1, falc (arg1=0xbffffdc5 "zob") at exemple_1.c:31
31     taille_buf = strlen (arg1);
(gdb) p $eax
$3 = -1073742616
(gdb) set $eax=42
(gdb) p $eax
$4 = 42
(gdb) info registers
eax             0x2a         42
ecx             0x1          1
edx             0xbffffdc5   -1073742395
ebx             0x40126a58   1074948696
esp             0xbffffc44   0xbffffc44
ebp             0xbffffc5c   0xbffffc5c
esi             0x40015d64   1073831268
edi             0xbffffce4   -1073742620
eip             0x80484ba    0x80484ba
eflags         0x286        646
cs              0x23         35
ss              0x2b         43
ds              0x2b         43
es              0x2b         43
fs              0x0          0
gs              0x0          0
fctrl           0x37f        895
fstat           0x0          0
ftag            0xffff       65535
fiseq           0x23         35
fioff           0x808c6cd    134792909
foseg           0x2b         43
fooff           0xbffffda8   -1073742424
fop             0x35d        861
(gdb)
Continuing.
```

```
----> Le premier argument du programme est : zob
```

```
Program exited normally.
(gdb)
```

## 2.12 Utiliser un fichier de type “core”

Un fichier “core” est en fait une photographie de la mémoire et des registres au moment où le programme a effectué une violation d’accès. Ils interviennent en général lorsqu’on utilise des fonctions sur chaîne réclamant un ‘ 0’ à la fin du buffer, ou lorsqu’on tente de mettre trop de données dans un emplacement mémoire trop petit. Les fichiers “core” ne sont pas créé à chaque fois malheureusement<sup>7</sup>. Leur utilisation est la même que si vous utilisiez le programme en cours d’exécution.

```
<prompt> $ gdb -q exemple_1 exemple_1.core
Core was generated by exemple_1'.
Program terminated with signal 11, Segmentation fault.
#0  0x10de8 in strlen ()
(gdb) bt
#0  0x10de8 in strlen ()
#1  0x21b8 in falc (arg1=0x0) at exemple_1.c:31
#2  0x211c in main (argc=1, argv=0xf7ffa04) at exemple_1.c:18
(gdb) frame
#0  0x10de8 in strlen ()
(gdb) f 1
#1  0x21b8 in falc (arg1=0x0) at exemple_1.c:31
31      taille_buf = strlen (arg1);
(gdb) p arg1
$1 = 0x0
(gdb) p arg1
$1 = 0x0
(gdb) info registers
g0          0x0          0
g1          0x42c         1068
g2          0x10ba8       68520
g3          0xf7ffa55      -134219179
g4          0x0          0
g5          0x0          0
g6          0x0          0
g7          0x2020       8224
o0          0x0          0
o1          0xf7ffa08      -134219256
o2          0xf7ffa04      -134219260
o3          0x33d20       212256
o4          0x33d20       212256
o5          0x32          50
sp          0xf7fff940      -134219456
o7          0x2114         8468
10          0x171bc         94652
11          0x107e0        67552
12          0x2098         8344
13          0x0          0
14          0x1          1
15          0x80          128
```

<sup>7</sup>Si vous n’en avez jamais, vérifiez que la taille limite du fichier “core” est suffisamment importante avec la commande *limit*. Pour la modifier, il va probablement falloir faire *limit coredumpsize tailleM*

```

16          0xfa103000          -99602432
17          0x0              0
i0          0x0              0
i1          0xf7fffa08          -134219256
i2          0xf7fffa04          -134219260
i3          0x33d20    212256
i4          0x33d20    212256
i5          0x32             50
fp          0xf7fff940          -134219456
i7          0x2114    8468
y          0x0              0
psr         0x40901085          1083183237
wim         0x0              0
tbr         0x0              0
pc          0x21b8    8632
npc         0x10dec    69100
fpsr        0x0              0
cpsr        0x0              0
(gdb) quit
<prompt> $

```

### 3 Electric Fence et les “malloc debuggers”

“Electric Fence”, développé par Bruce Perens, est une bibliothèque utile pour déboguer les problèmes issues des allocations dynamiques de mémoire. En effet, il peut arriver qu’en dépassant les buffers qu’on a précédemment alloués, ou encore qu’on a voulu libérer plusieurs fois de suite le même emplacement mémoire, que le programme ne plante pas, ou du moins, pas tout de suite. Dans ces cas là, “efence” va faire planter l’application au bon moment. Son utilisation est très simple, il faut recompiler avec “-lefence” puis activer les variables d’environnement dont on a besoin. Elles sont au nombre de trois :

**EF\_ALIGNMENT** valeur en bytes de l’alignement des allocations.

**EF\_PROTECT\_BELOW** place des pages innaccessibles derrière chaque zone allouée.

**EF\_PROTECT\_FREE** pour détecter les problèmes de “double free”.

```

<prompt> $ gcc -Wall -lefence -o prog prog.c
<prompt> $ export EF_PROTECT_FREE=1
<prompt> $ ./prog
...

```

“Electric Fence”, bien que très connu, n’est pas le seul, il existe aussi “njamd”, “yam-d”, “cmmalloc”...

## 4 strace/ltrace

### 4.1 strace

Strace est un utilitaire disponible sous GNU/Linux dont le but est d’intercepter et d’afficher les différents appels systèmes et signaux qu’utilisent ou que génèrent certaines applications. Il affiche à l’utilisateur les noms des fonctions ainsi que leurs arguments tel qu’il aurait pu les remplir.

Exemple :

```

<prompt> $ strace ./exemple_1 chaine_de_caractere
execve("./exemple_1", [ "./exemple_1", "chaine_de_caractere"], [/* 17 vars */]) = 0
brk(0)                                = 0x80496e4
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=16293, ...}) = 0
old_mmap(NULL, 16293, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                               = 0
open("/lib/libc.so.6", O_RDONLY)      = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\264\323"... , 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=4783716, ...}) = 0
old_mmap(NULL, 1116516, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x4001b000
mprotect(0x40122000, 39268, PROT_NONE) = 0
old_mmap(0x40122000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_FIXED, 3, 0x106000) = 0x40122000
old_mmap(0x40128000, 14692, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x40128000
close(3)                               = 0
munmap(0x40017000, 16293)               = 0
getpid()                               = 31460
brk(0)                                = 0x80496e4
brk(0x804970c)                         = 0x804970c
brk(0x804a000)                         = 0x804a000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40017000
ioctl(1, SNDCTL_TMR_TIMEBASE, {B38400 opost isig icanon echo ...}) = 0
write(1, "----> Le premier argument du pro"... , 65----> Le premier argument
du programme est : chaine_de_caractere) = 65
munmap(0x40017000, 4096)               = 0
_exit(0)                               = ?
<prompt> $

```

Il permet un certain nombre d'opérations pour n'afficher que certains types d'appels, ou signaux, pour suivre les enfants du processus, formater l'affichage...

## 4.2 ltrace

Ltrace se focalise quand à lui sur les bibliothèques appelées pendant l'exécution d'un programme. Comme "strace", il permet de placer des filtres, de modifier le format d'affichage et d'afficher les différents appels systèmes. Pour cette dernière capacité, "strace" permet d'avoir un meilleur résultat à l'affichage que "ltrace".

Exemple :

```

<prompt> $ ltrace ./exemple_1 pouet
__libc_start_main(0x08048490, 2, 0xbffffcb4, 0x08048308,
0x0804854c <unfinished ...>
__register_frame_info(0x080495ec, 0x080496cc, 0xbffffc58, 0x4004a138,
0x40126a58) = 0x40127740
strlen("pouet")                          = 5
malloc(5)                                = 0x080496f0
strncpy(0x080496f0, "pouet", 5)          = 0x080496f0
printf("----> Le premier argument du pro"...----> Le premier argument du
programme est : pouet)                    = 51
__deregister_frame_info(0x080495ec, 0x4000b8d9, 0x400163e4, 0x400164f0, 7) = 0x080496cc
+++ exited (status 0) +++
<prompt> $

```

## 5 Profiling

### 5.1 Qu'est-ce-que c'est ?

Le “profiling” consiste à obtenir le profil d'un programme afin de déterminer où peuvent être ses faiblesses en temps CPU, en temps passé par fonction. . . Avec ces outils, on rentre plutôt dans une idée d'optimization. Pour ce faire nous avons à notre disposition deux outils dans le monde GNU, **gprof** et **gcov**.

### 5.2 gprof

#### 5.2.1 Préparation

Tout comme gdb, gprof a besoin d'un certain nombre de choses au moment de la compilation pour lui indiquer ce qu'il doit faire. La première étape va donc consister à recompiler les fichiers sources avec l'option “-pg” en plus de l'option “-g” si on souhaite avoir un profil par ligne du code source. La deuxième étape consiste à lancer l'exécutable ; il va alors créer un fichier “gmon.out” que gprof va utiliser. Et finalement, il suffit de lancer gprof.

Les trois étapes que nous venons de citer sont reprises dans cet exemple :

```
<prompt> $ cat profiling.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int fonction1(int chiffre);
int fonction2(int i);
int fonction3(int chiffre);

int main (int argc, char **argv)
{
    unsigned int chiffre;

    chiffre = 0;

    if (argc < 2)
    {
        puts ("profiling chiffre");
        return 1;
    }

    chiffre = atoi (argv[1]);
    puts ("1");
    fonction1(chiffre);
    puts ("2");
    fonction2(chiffre);
    puts ("3");
    fonction3(chiffre);
    puts ("");

    return 0;
}
```

```

int fonction1(int chiffre)
{
    int i;

    for(i=0; i < chiffre; i++) {}
    return 0;
}

int fonction2(int i)
{
    while (i--) {}
    return 0;
}

int fonction3(int chiffre)
{
    int i = 0;

    while (i++ < chiffre) {}
    return 0;
}
<prompt> $ gcc -Wall -g -pg -o profiling profiling.c
<prompt> $ ./profiling 5000000000
1
2
3

<prompt> $

```

### 5.2.2 Comment interpréter les résultats

```

<prompt> $ gprof -b profiling
Flat profile:

```

Each sample counts as 0.01 seconds.

%	cumulative	self	calls	self	total	name
time	seconds	seconds		ms/call	ms/call	
33.35	21.50	21.50	1	21500.00	21500.00	fonction3
33.34	42.99	21.49	1	21490.00	21490.00	fonction1
33.31	64.46	21.47	1	21470.00	21470.00	fonction2

Call graph

granularity: each sample hit covers 4 byte(s) for 0.02% of 64.46 seconds

index	% time	self	children	called	name
[1]	100.0	0.00	64.46		<spontaneous> main [1]
		21.50	0.00	1/1	fonction3 [2]
		21.49	0.00	1/1	fonction1 [3]
		21.47	0.00	1/1	fonction2 [4]
-----					
[2]	33.4	21.50	0.00	1	main [1] fonction3 [2]
-----					
[3]	33.3	21.49	0.00	1	main [1] fonction1 [3]

```
-----
                21.47    0.00    1/1    main [1]
[4]    33.3    21.47    0.00    1    fonction2 [4]
-----
```

Index by function name

```
    [3] fonction1                [4] fonction2                [2] fonction3
<prompt> $
```

Le premier tableau appelé “Flat Profile” indique le temps total passé par chaque fonction lors de l’exécution.

**% time** Indique le pourcentage sur l’intégralité de l’exécution, passée dans dans cette fonction.

**cumulative seconds** Indique le nombre total de secondes passées dans cette fonction additionné au temps des fonctions précédentes.

**self seconds** Indique le temps passé dans le programme seulement pour cette fonction.

**self ms/call** Représente le temps moyen passé dans la fonction par appel.

**total ms/call** Représente le temps moyen passé dans la fonction ainsi que ses descendant par appels.

Le second tableau est appelé “Call Graph” et représente le temps passé par chaque fonctions et chacun de ses enfants. La signification des champs reste sensiblement la même que pour le “Flat Profile” si ce n’est “called” qui indique combien de fois la fonction a été appelée. Les signes “<cycle nb >” dans le champ “name” et “+” dans le champ “called” pouvant intervenir dans certains résultats de profils, indiquent que nous avons affaire à une fonction récursive ou à des fonctions qui s’appellent les unes les autres formant une boucle.

Le modèle qui suit est une autre forme de présentation. Celle-ci se fait ligne par ligne de code, mais l’approche générale reste la même que ceux vus précédemment.

```
<prompt> $ gprof -b -l profiling
Flat profile:
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
33.34	21.49	21.49				fonction1 (profiling.c:44)
29.87	40.74	19.25				fonction3 (profiling.c:60)
29.13	59.52	18.78				fonction2 (profiling.c:51)
4.17	62.22	2.69				fonction2 (profiling.c:53)
3.48	64.46	2.25				fonction3 (profiling.c:62)
0.00	64.46	0.00	1	0.00	0.00	fonction1
0.00	64.46	0.00	1	0.00	0.00	fonction2
0.00	64.46	0.00	1	0.00	0.00	fonction3

Call graph

granularity: each sample hit covers 4 byte(s) for 0.02% of 64.46 seconds

index	% time	self	children	called	name
		0.00	0.00	1/1	main (profiling.c:24) [34]

```

[6]      0.0    0.00    0.00      1      fonction1 [6]
-----
          0.00    0.00      1/1          main (profilng.c:28) [36]
[7]      0.0    0.00    0.00      1      fonction2 [7]
-----
          0.00    0.00      1/1          main (profilng.c:32) [38]
[8]      0.0    0.00    0.00      1      fonction3 [8]
-----

```

Index by function name

```

[6] fonction1                [3] fonction2 (profilng.c:51)
[7] fonction2                [2] fonction3 (profilng.c:60)
[8] fonction3                [1] fonction1 (profilng.c:44)
[4] fonction2 (profilng.c:53) [5] fonction3 (profilng.c:62)

```

Une fonctionnalité qui peut être utile est celle disponible en passant l'argument “-s”. Cela permet de lancer plusieurs fois le programme et d'additionner les nouveaux résultats avec les anciens, nous permettant ainsi d'augmenter la qualité des statistiques.

```

<prompt> $ ./profilng 50000000
<prompt> $ gprof -b -s profilng gmon.sum
...
<prompt> $ gprof -b -s profilng gmon.sum
...
<prompt> $ gprof -b profilng gmon.sum
...

```

## 5.3 gcov

### 5.3.1 Préparation

Comme gprof, il est nécessaire de rajouter les options “-ftest-coverage” et “-fprofile-arcs”, de lancer le programme, puis gcov :

```

<prompt> $ gcc -ftest-coverage -fprofile-arcs -Wall -o profilng profilng.c
<prompt> $ ./profilng 50000000
1
2
3

<prompt> $

```

### 5.3.2 Comment interpréter les résultats

```

<prompt> $ gcov -b profilng.c
 92.59% of 27 source lines executed in file profilng.c
 93.33% of 15 branches executed in file profilng.c
 53.33% of 15 branches taken at least once in file profilng.c
 92.86% of 14 calls executed in file profilng.c
Creating profilng.c.gcov.
<prompt> $ cat profilng.c.gcov
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int fonction1(int chiffre);
int fonction2(int i);

```

```

int fonction3(int chiffre);

int main (int argc, char **argv)
{
    1      unsigned int chiffre;

    1      chiffre = 0;

    1      if (argc < 2)
branch 0 taken = 100%
    {
        #####      puts ("profiling chiffre");
call 0 never executed
        #####      return 1;
branch 0 never executed
    1      }

    1      chiffre = atoi (argv[1]);
call 0 returns = 100%

    1      puts ("1");
call 0 returns = 100%

    1      fonction1(chiffre);
call 0 returns = 100%

    1      puts ("2");
call 0 returns = 100%

    1      fonction2(chiffre);
call 0 returns = 100%

    1      puts ("3");
call 0 returns = 100%

    1      fonction3(chiffre);
call 0 returns = 100%

    1      puts ("");
call 0 returns = 100%

    1      return 0;
branch 0 taken = 100%
    1      }
call 0 returns = 0%

int fonction1(int chiffre)
{
    1      int i;

    1      for(i=0; i < chiffre; i++){
branch 0 taken = -2%
branch 1 taken = 100%
branch 2 taken = -2%
    }

    1      return 0;
branch 0 taken = 100%

```

```

    1    }
call 0 returns = 0%

int fonction2(int i)
{
    1    while (i--){
branch 0 taken = -2%
branch 1 taken = 100%
branch 2 taken = -2%
    }
    1    return 0;
branch 0 taken = 100%
    1    }
call 0 returns = 0%

int fonction3(int chiffre)
{
    1    int i = 0;

    1    while (i++ < chiffre) {
branch 0 taken = -2%
branch 1 taken = 100%
branch 2 taken = -2%
    }
    1    return 0;
branch 0 taken = 100%
    1    }
call 0 returns = 0%
call 1 returns = 100%
```

Chaque nombre à gauche d'une ligne indique combien de fois celle-ci a été exécutée. Le pourcentage affiché est le nombre de fois où la fonction s'est terminée divisé par le nombre de fois où elle a été exécutée. Il est intéressant de noter que si des cas particuliers pouvait intervenir, les valeurs contenu dans le fichier de profil, sont cumulatives. Si on souhaite relancer l'application sans enlever les fichiers "\*.da", les valeurs seront remises à jour. Là encore cela permet d'avoir une meilleure qualité des statistiques.

## 6 GNU Free Documentation License

Version 1.1, March 2000

Copyright copyright 2000 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### 6.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format,  $\LaTeX$  input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

## 6.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 6.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material

on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 6.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version :

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.

- Preserve the section entitled “History”, and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled “Acknowledgements” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another ; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 6.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you in-

clude in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

## 6.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 6.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

## 6.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the

original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## 6.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 6.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ADDENDUM : How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page :

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation ; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST" ; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

## Références

[gdb] <http://www.gnu.org/software/gdb/gdb.html>

[strace] <http://www.liacs.nl/~wichert/strace/>

[ltrace] <http://packages.debian.org/unstable/utils/ltrace.html>

[ddd] <http://www.gnu.org/software/ddd/index.html>

[efence] <http://perens.com/FreeSoftware/>