Déverminage Ada avec GDB 7.2 et 7.4
(par FSF et GNAT GPL 2012)

1) Introduction

L'installation de GDB 7.4 (avec GNAT GPL 2012) sur Mac est décrite sur Blady en page Créations. Cette version utilisée sur MacOS 10.8 doit être signée avec un certificat système car taslgated, l'utilitaire de contrôle d'accès aux processus, n'est lancé qu'avec l'option -s (autorisation pour les programmes signés, voir le fichier /System/Library/LaunchDaemons/com.apple.taskgated.plist). Le paragraphe 2 montre comment signer GDB.

D'autre part, GDB provoque l'affichage de lignes comme "BFD: /usr/lib/dyld(i386:x86-64): unable to read unknown load command 0x2a" à cause, semble-t-il, d'une incompatibilité avec le format Mach-O des exécutables MacOS 10.8 avec la bibliothèque BFD de GDB.

Le paragraphe 2 donne les instructions pour construire GDB 7.2 à partir de ses sources. Les paragraphes suivants vous indiquent l'utilisation courante de GDB.

2) Construction de GDB 7.2

Télécharger le fichier http://ftp.gnu.org/gnu/gdb/gdb-7.2.tar.bz2 sur le bureau du Mac, puis lancer le Terminal :

```
$ cd ~/Desktop
$ tar xzf gdb-7.2.tar.bz2
$ mkdir build-gdb
$ cd build-gdb
$ ../gdb-7.2/configure
$ make
```

Seulement, ce n'est pas fini car en utilisant directement GDB vous obtiendriez l'erreur :

```
(gdb) run
Starting program: stb
Unable to find Mach task port for process-id 349: (os/kern) failure
(0x5).
  (please check gdb is codesigned - see taskgated(8))
```

Il est nécessaire de changer les droits du programme sous MacOS 10.6 et 10.7 ou de créer une certification avec Trousseau d'accès sous MacOS 10.8. Voir http://sourceware.org/gdb/wiki/BuildingOnDarwin.

Sous MacOS 10.6 :

```
$ sudo chgrp procmod gdb
$ sudo chmod g+s gdb
```

Sous Mac OS 10.8, vous devez créer un certificat avec l'application "Trousseau d'accès" dans une session administrateur.

Dans le menu de l'application cliquer sur "Assistant de certification" -> "Créer un certificat...", puis donner un nom comme "gdb72-cert", garder le type d'identité 'Racine auto-signée', changer le type de certificat par 'Signature de code', cocher 'Me laisser ignorer les réglages par défaut' puis cliquer sur 'Continuer' et plusieurs fois sur 'Continuer' jusqu'à l'emplacement dans le trousseau d'accès puis modifier l'emplacement du certificat par 'Système' et cliquer sur 'Créer'. Une fenêtre demande de saisir le mot de passe administrateur puis cliquer sur 'Terminer'.

Ensuite dans la fenêtre de Trousseaux d'accès, sélectionner 'Système' et le certificat puis sélectionner le menu "Fichier" -> "Lire les informations". Dans la fenêtre qui apparaît déplier le paragraphe "Se fier" et sélectionner "Signature de code" à "Toujours approuver". Fermer la fenêtre. Une fenêtre demande de saisir le mot de passe administrateur. Si une petite croix blanche apparait dans un cercle bleu, c'est tout bon. Finalement, bien fermer 'Trousseaux d'accès'.

Il nous reste plus qu'à signer GDB avec ce certificat :

```
$ codesign -s gdb72-cert gdb
```

3) Utilisation de GDB

Le manuel de GDB est accessible dans la documentation de GNAT :

```
$ open /usr/local/gnat/share/doc/gnat/html/gdb.html
```

Une fois le code source Ada compilé avec l'indispensable option "-g", le dévermineur peut être lancé :
(Voir le code de l'exemple sur Blady à l'adresse :
http://blady.perso.wanadoo.fr/Ressources/exemples_gdb.tgz)

```
$ gnatmake -g figures
gcc -c -g figures.adb
gnatbind -x figures.ali
gnatlink figures.ali -g
$ gdb figures
...
```

GDB se termine avec l'instruction "quit" (q) :

```
(gdb) quit
```

4) L'affichage du code source

Le code source est affiché avec la commande "list" (l) :

```
(gdb) list
10
11          subtype Class is Instance'Class;
12          -- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut
l'instance
13          -- et tous ces descendants
14
15          procedure Positionne (Objet : in out Instance; X, Y : Coordonnee);
16          function RetourneX (Objet : in Instance) return Coordonnee;
17          function RetourneY (Objet : in Instance) return Coordonnee;
18        private
19          type Instance is tagged record
```

List accepte comme argument le numéro d'une ligne de code ou une tranche de lignes de code :

```
(gdb) list 103
98            Positionne (Objet, X, Y); -- on utilise le constructeur h?rit? de Figure
99            Objet.R := R;
100         end Positionne;
101
102         function RetourneR (Objet : in Instance) return Figure.Coordonnee is
103         begin
104           return Objet.R;
105         end RetourneR;
106
107         function Aire (Objet : in Instance) return Figure.Surface is

(gdb) list 1,13
1       with Text_IO; use Text_IO;
2       procedure Figures is
3
4         package Figure is
5           type Coordonnee is range -100 .. 100;
6           subtype Surface is Float;
7           type Instance is tagged private;
8           -- remarquez le mot cl? "tagged" qui d?clare l'objet et "private" qui rend
sa structure
9           -- interne invisible
10
11          subtype Class is Instance'Class;
12          -- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut
l'instance
13          -- et tous ces descendants
```

List accepte comme argument le nom d'une procédure ou d'une fonction. Si plusieurs répondent à ce nom, un choix est proposé :

```
(gdb) list positionne
file: "figures.adb", line number: 26
file: "figures.adb", line number: 96
(gdb) list 22
21             X, Y : Coordonnee := 0;
22          end record;
23       end Figure;
24
25       package body Figure is
26          procedure Positionne (Objet : in out Instance; X, Y : Coordonnee) is
27          begin
28             Objet.X := X;
29             Objet.Y := Y;
30          end Positionne;
(gdb) list 96
91             -- l'initialisation par d?faut ? z?ro ce qui revient ? un Point ;-)
92          end record;
93       end Cercle;
94
95       package body Cercle is
96          procedure Positionne (Objet : in out Instance; X, Y, R :
Figure.Coordonnee) is
97          begin
98             Positionne (Objet, X, Y); -- on utilise le constructeur h?rit? de Figure
99             Objet.R := R;
100         end Positionne;
```

Les procédures et fonctions d'une unité Ada sont préfixées de son nom :

```
(gdb) list figures.allume
141      MaFigure  : Figure.Instance; -- instanciation de l'objet
142      MonPoint  : Point.Instance; -- instanciation de l'objet
143      MonCercle : Cercle.Instance; -- instanciation de l'objet
144      MonCarre  : Carre.Instance; -- instanciation de l'objet
145
146      procedure Allume (X, Y : Figure.Coordonnee) is
147      begin
148         Put_Line ("Allume");
149      end Allume;
150
```

Astuce : taper sur <retour-ligne> pour exécuter à nouveau la commande :

```
(gdb) <retour-ligne>
151       procedure Eteins (X, Y : Figure.Coordonnee) is
152       begin
153         Put_Line ("Eteins");
154       end Eteins;
155
156       procedure MonAffiche is new Point.Affiche (Allume, Eteins);
157
158     begin
159       Figure.Positionne (MaFigure, 10, 20);
160       Put_Line ("X : " & Figure.RetourneX (MaFigure)'Img);
```

Astuce : taper sur <tab> pour lister toutes les possibilités :

```
(gdb) list figures<tab>
figures                  figures.eteins
figures.allume           figures.figure.instance
figures.carre.aire       figures.figure.positionne
figures.carre.instance   figures.figure.retournex
figures.cercle.afficheaire  figures.figure.retourney
figures.cercle.aire      figures.figure.surface
figures.cercle.instance  figures.monaffiche
figures.cercle.positionne   figures.point.instance
figures.cercle.retourner
```

Le code source d'une unité instanciée à partir d'un générique est celui du générique :

```
(gdb) list figures.monaffiche
57          type Instance is new Figure.Instance with null record;
58          -- et le mot cl? "with" pour ajouter des champs, ici on n'ajoute rien pour
faire un point
59        end Point;
60
61        package body Point is
62          procedure Affiche (Objet : in Instance; EstVisible : Boolean) is
63          begin
64            if EstVisible then
65              Allume (RetourneX (Objet), RetourneY (Objet));
66            else
```

L'unité générique en elle même n'existe pas pour le dévermineur :

```
(gdb) list figures.point.affiche
Function "figures.point.affiche" not defined.
```

5) Les points d'arrêts

Le points d'arrêt pour interrompre l'exécution du programme sur une instruction sont positionnés de la même manière avec la commande "break" (b) :

(gdb) **break** figures.monaffiche
Breakpoint 1 at 0x100004671: file figures.adb, line 64.

La liste des points d'arrêt est affichée avec la commande "info breakpoints" (i b) :

(gdb) **info breakpoints**
Num Type Disp Enb Address What
1 breakpoint keep y 0x0000000100004671 in figures.monaffiche
at figures.adb:64

De droite à gauche, se trouvent le numéro d'ordre, le type d'arrêt ("breakpoint" sur une instruction ou "watchpoint" lorsqu'une valeur définie a changée), le type d'action ("keep" reste actif, "del" effacé, "dis" désactivé) , l'activation courante ("y" ou "n"), l'adresse et la localisation de l'instruction où a été positionné le point d'arrêt.

Un point d'arrêt est désactivé avec la commande "disable breakpoints" (dis b) :

(gdb) **disable breakpoints 1**
(gdb) info breakpoints
Num Type Disp Enb Address What
1 breakpoint keep **n** 0x0000000100004671 in figures.monaffiche
at figures.adb:64

Le point d'arrêt est réactivé avec le commande "enable breakpoints" (en b) :

(gdb) **enable breakpoints 1**
(gdb) info breakpoints
Num Type Disp Enb Address What
Num Type Disp Enb Address What
1 breakpoint keep **y** 0x0000000100004671 in figures.monaffiche
at figures.adb:64

Le point d'arrêt est réactivé pour une seule fois avec le commande "enable breakpoints once" (en b o) :

(gdb) **enable breakpoints once 1**
(gdb) info breakpoints
Num Type Disp Enb Address What
Num Type Disp Enb Address What
1 breakpoint dis y 0x0000000100004671 in figures.monaffiche
at figures.adb:64

Un point d'arrêt est effacé avec la commande "delete breakpoints" (del br) :

(gdb) **delete breakpoints 1**
(gdb) info breakpoints
No breakpoints or watchpoints.

6) L'exécution du programme

L'exécution du programme est lancée avec la commande "run" (r) :

(gdb) **run**
Starting program: figures
<...>
X : 10
Y : 25
R : 30
Aire cercle : 2.82600E+03
Aire carré : 1.60000E+03
Eteins
[Inferior 1 (process 7512) exited normally]

L'exécution avec la commande "start" s'arrête sur la première instruction du programme en posant un point d'arrêt temporaire. En Ada, ce n'est pas forcément le début du programme principal car les élaborations d'objets Ada sont considérés comme des instructions.

(gdb) **start**
Temporary breakpoint 1 at 0x1000016e9: file figures.adb, line 19.
Starting program: figures
Temporary breakpoint 1, figures () at figures.adb:19
19 type Instance is tagged record

Les arguments du programme, si existants, sont ajoutés après la commandes "run" et "start" ou enregistrés avec la commande "set args".

(gdb) **run 10**
(gdb) **set args 10**

Positionnons le point d'arrêt pour examiner les entrailles de notre programme en cours d'exécution et relançons le programme :

```
(gdb) break figures.monaffiche
Breakpoint 2 at 0x100004671: file figures.adb, line 64.
(gdb) run
Starting program: figures
X :  10
Y :  25
R :  30
Breakpoint 2, figures.monaffiche (objet=..., estvisible=false)
   at figures.adb:64
64              if EstVisible then
```

La file d'appels est affichée avec la commande "backtrace" (bt) :

```
(gdb) backtrace
#0  figures.monaffiche (objet=..., estvisible=false) at figures.adb:64
#1  0x0000000100003636 in figures () at figures.adb:168
```

L'ajout d'un nombre va limiter l'affichage aux derniers appels si positif ou aux premiers appels si négatif :

```
(gdb) bt 1
#0  figures.monaffiche (objet=..., estvisible=false) at figures.adb:64
(More stack frames follow...)
(gdb) bt -1
#1  0x0000000100003636 in figures () at figures.adb:168
```

La file d'appels avec les variables locales est affichée avec la commande "backtrace full" (bt full) :

```
(gdb) bt full
#0  figures.monaffiche (objet=..., estvisible=false) at figures.adb:64
No locals.
#1  0x0000000100003636 in figures () at figures.adb:168
       mafigure = (x => 10, y => 20)
       monpoint = (x => 15, y => 25)
       moncercle = (x => 10, y => 10, r => 30)
       moncarre = (x => 15, y => 20, r => 40)
```

L'exécution reprend avec la commande "step" (s) pour exécuter l'instruction suivante en entrant éventuellement à l'intérieur d'une fonction ou d'une procédure appelée :

```
(gdb) step
67              Eteins (RetourneX (Objet), RetourneY (Objet));
```

L'exécution reprend avec la commande "next" (n) pour exécuter l'instruction suivante en restant au même niveau de la procédure ou de la fonction courante :

```
(gdb) next
Eteins
69          end Affiche;
```

L'exécution reprend avec la commande "finish" (f) pour exécuter toutes les instructions suivantes jusqu'à la fin de la procédure ou de la fonction courante :

```
(gdb) finish
#0  figures.monaffiche (objet=..., estvisible=false) at figures.adb:69
69          end Affiche;
```

L'exécution reprend avec la commande "continue" (c) pour exécuter toutes les instructions suivantes jusqu'à la fin du programme ou la rencontre d'un point d'arrêt :

```
(gdb) continue
Continuing.
[Inferior 1 (process 7534) exited normally]
```

7) Les informations du programme

Reprenons notre examen en positionnant un point d'arrêt sur la première instruction de notre programme principal et lançons le programme à nouveau :

```
(gdb) break 159
Breakpoint 3 at 0x100002fcb: file figures.adb, line 159.
(gdb) run
Starting program: figures
Breakpoint 3, figures () at figures.adb:159
159       Figure.Positionne (MaFigure, 10, 20);
```

La procédure courante est affichée avec la commande "frame" (fr) :

```
(gdb) frame
#0  figures () at figures.adb:159
159         Figure.Positionne (MaFigure, 10, 20);
```

Les informations de la procédure courante sont affichées avec "info frame" :

```
(gdb) info frame
Stack level 0, frame at 0x7fff5fbffa80:
 rip = 0x100002fcb in figures (figures.adb:159); saved rip 0x1000010ce
 source language ada.
 Arglist at 0x7fff5fbffa70, args:
 Locals at 0x7fff5fbffa70, Previous frame's sp is 0x7fff5fbffa80
 Saved registers:
  rbx at 0x7fff5fbffa48, rbp at 0x7fff5fbffa70, r12 at 0x7fff5fbffa50,
  r13 at 0x7fff5fbffa58, r14 at 0x7fff5fbffa60, r15 at 0x7fff5fbffa68,
  rip at 0x7fff5fbffa78
```

La liste des variables locales est affichée avec la commande "info locals" :

```
(gdb) info locals
mafigure = (x => 0, y => 0)
monpoint = (x => 0, y => 0)
moncercle = (x => 0, y => 0, r => 0)
moncarre = (x => 0, y => 0, r => 0)
```

Le type d'une variable est obtenue avec la commande "whatis" :

```
(gdb) whatis mafigure
type = figures.figure.instance
```

Continuons dans notre programme :

```
(gdb) step
figures.figure.positionne (objet=..., x=10, y=20) at figures.adb:28
28              Objet.X := X;
```

L'ensemble des arguments d'une procédure ou d'une fonction est affichée avec la commande "info args" :

```
(gdb) info args
objet = (x => 0, y => 0)
x = 10
y = 20
```

Les variables sont affichées individuellement sans retour à la ligne avec la commande "output" (out) :

```
(gdb) output x
10
```

Les variables sont aussi affichées individuellement avec retour à la ligne et historisation avec la commande "print" (p) :

```
(gdb) print x
$1 = 10
```

8) Les exceptions

Utilisons le programme de l'article sur les exceptions Ada.
(Voir le code de l'exemple sur Blady à l'adresse :
http://blady.perso.wanadoo.fr/Ressources/exemples_gdb.tgz)

```
$ gnatmake -g stb
gcc -c -g stb.adb
gnatbind -x stb.ali
gnatlink stb.ali -g
$ gdb stb
```

Positionnons un point d'arrêt sur la première exception survenant avec la commande "break exception" ou "catch exception" et lançons le programme :

```
(gdb) list
2       procedure P1 is
3       begin
4       raise Constraint_Error;
5       end P1;
6       procedure P2 is
7       begin
8       P1;
9       end P2;
10       begin
11       P2;
(gdb) break exception
Catchpoint 1: all Ada exceptions
(gdb) run
Starting program: stb
Catchpoint 1, CONSTRAINT_ERROR at 0x0000000100001211 in stb.p1 () at
stb.adb:4
4               raise Constraint_Error;
```

```
(gdb) backtrace
#0  <__gnat_debug_raise_exception> (e=0x100011b20) at s-excdeb.adb:43
#1  0x000000010000371d in <__gnat_raise_nodefer_with_msg> (e=<optimized
out>)
    at a-except.adb:895
#2  0x0000000100003b19 in ada.exceptions.raise_with_location_and_msg (
    e=0x100011b20, f=<optimized out>, l=<optimized out>, c=<optimized out>,
    m=<optimized out>) at a-except.adb:1092
#3  0x0000000100003714 in <__gnat_raise_constraint_error_msg> (
   file=(system.address) 0xf, line=15, column=24,
    msg=(system.address) 0x100011888) at a-except.adb:885
#4  0x0000000100003bc7 in <__gnat_rcheck_CE_Explicit_Raise> (
   file=(system.address) 0x100011b20, line=15) at a-except.adb:1154
#5  0x00000001000019ee in stb.p1 () at stb.adb:4
#6  0x0000000100001a13 in stb.p2 () at stb.adb:8
#7  0x00000001000019fa in stb () at stb.adb:11
```

Le programme s'est arrêté sur la première occurrence d'une exception. La commande "backtrace" affiche l'origine de l'exception dans la procédure P1 du programme STB.

La commande "break exception" ou "catch exception" suivi du nom d'une exception permet de n'arrêter le programme que sur l'exception définie dans le programme modifié STBH :

```
$ gnatmake -g stbh
gcc -c -g stbh.adb
gnatbind -x stbh.ali
gnatlink stbh.ali -g
$ gdb stbh
(gdb) list 1
1       procedure STBH is
2        procedure P1 is
3        begin
4        raise Constraint_Error;
5        exception
6          when others =>
7            null;
8       end P1;
9        procedure P2 is
10       begin
(gdb)
11       P1;
12       raise Storage_Error;
13       end P2;
14       begin
15       P2;
16       end STBH;
```

(gdb) **break exception storage_error**
Catchpoint 1: `storage_error' Ada exception
(gdb) run
Starting program: stbh
Catchpoint 1, STORAGE_ERROR at 0x00000001000019dd in stbh.p2 () at stbh.adb:12
12 raise Storage_Error;
(gdb) cont
Continuing.

raised STORAGE_ERROR : stbh.adb:12 explicit raise
[Inferior 1 (process 8582) exited with code 01]
(gdb) del 1

Le programme ne s'est arrêté que sur l'exception nommée "Storage_Error" dans la procédure P2 du programme STB.

La commande "break exception unhandled" permet d'arrêter le programme sur l'occurrence d'une exception qui n'est pas récupérée au même niveau que son occurrence :

(gdb) **break exception unhandled**
Catchpoint 2: unhandled Ada exceptions
(gdb) run
Starting program: stbh
Catchpoint 2, unhandled STORAGE_ERROR at 0x00000001000019dd in stbh.p2 ()
at stbh.adb:12
12 raise Storage_Error;

Le programme ne s'est pas arrêté sur la première exception car celle-ci a été récupérée. Il s'est ainsi arrêté sur la seconde dans la procédure P2 du programme STB car cette dernière est non récupérée.

GNAT possède une option de visualisation de la pile des appels lors de la levée d'une exception (-bargs -E), par contre les lignes du code source ne sont pas affichées, ce sont les adresses (avec MacOS à partir de 10.7 nous ajoutons une option supplémentaire pour ne pas avoir des adresses indépendantes -largs -Wl,-no_pie). la commande "info line" va nous aider à retrouver les lignes du code source:

$ gnatmake -f -g stb -bargs -E -largs -Wl,-no_pie
gcc -c -g stb.adb
gnatbind -E -x stb.ali
gnatlink stb.ali -g -Wl,-no_pie
$ ./stb
Execution terminated by unhandled exception

```
Exception name: CONSTRAINT_ERROR
Message: stb.adb:4 explicit raise
Call stack traceback locations:
0x1000019e8 0x100001a0d 0x1000019f4 0x1000019b2
12:35:28 exemples_gdb $ atos.sh stb 0x1000019e8 0x100001a0d 0x1000019f4
0x1000019b2
stb__p1.2350 (in stb) (stb.adb:4)
stb__p2.2352 (in stb) (stb.adb:8)
_ada_stb (in stb) (stb.adb:11)
main (in stb) (b~stb.adb:142)
$ gdb stb
(gdb) info line *0x1000019e8
Line 4 of "stb.adb" starts at address 0x1000019d6 <stb__p1+12>
  and ends at 0x1000019ea <_ada_stb>.
(gdb) info line *0x100001a0d
Line 8 of "stb.adb" starts at address 0x100001a07 <stb__p2+15>
  and ends at 0x100001a0f <stb__p2+23>.
(gdb) info line *0x1000019f4
Line 11 of "stb.adb" starts at address 0x1000019ee <_ada_stb+4>
  and ends at 0x1000019f6 <_ada_stb+12>.
(gdb) info line *0x1000019b2
Line 142 of "b~stb.adb"
  starts at address 0x1000019af <main+87> and ends at 0x1000019b4 <main
+92>.
```

9) le multi-tâche

Utilisons le programme de l'article sur les tâches Ada.
(Voir le code de l'exemple sur Blady à l'adresse :
http://blady.perso.wanadoo.fr/Ressources/exemples_gdb.tgz)

```
$ gnatmake -g taches
gcc -c -g taches.adb
gnatbind -x taches.ali
gnatlink taches.ali -g
$ gdb taches
```

Positionnons un point d'arrêt sur la première ligne de code avec la commande "break" et lançons le programme :

```
(gdb) list taches
1       with Text_IO; use Text_IO;
2       procedure Taches is
3
4       Valeur : Natural := 0;
5
6       task type Compte;
```

```
7       task body Compte is
8        begin
9          loop
10           Valeur := Valeur + 1;
(gdb)
11           delay 0.5;
12         exit when Valeur > 99;
13         end loop;
14       end Compte;
15
16     task type Affiche;
17     task body Affiche is
18       begin
19         loop
20           Put_Line (Natural'Image(Valeur));
(gdb)
21           delay 1.0;
22         exit when Valeur > 99;
23         end loop;
24       end Affiche;
25
26     C : Compte;
27     A : Affiche;
28
29     begin
30     Put_Line ("Compte et affiche concurremment.");
(gdb)
31     end Taches;
(gdb) b 30
Breakpoint 1 at 0x361c: file taches.adb, line 30.
(gdb) run
Starting program: /Users/pascal/Documents/Programmation/test3/taches
 1
[New Thread 0x1503 of process 3557]
[New Thread 0x1603 of process 3557]
Breakpoint 1, taches () at taches.adb:30
30     Put_Line ("Compte et affiche concurremment.");
```

Le programme est arrêté. Nous pouvons examiner les tâches lancées avec la commande "info tasks" :

```
(gdb) info tasks
  ID       TID P-ID Pri State                Name
*  1 100803200       31 Runnable             main_task
   2 100804000    1  31 Delay Sleep           c
   3 100807800    1  31 Delay Sleep           a
```

Nos deux tâches "c" et "a" (ID 2 et 3) sont bien là puisque lancées à lors de l'élaboration. La troisième "main_task" (ID 1) n'est autre que le programme principal considéré naturellement comme une tâche. L'asterix (*) indique la tâche inspectée.

La commande affiche dans l'ordre son numéro interne à GDB, son numéro propre à GNAT, sa parenté (numéro interne GDB), sa priorité, son état courant : inactive (Unactivated), en exécution (Runnable), en attente (accept, delay, entry, child, select, delay, rendez-vous, ...) ou terminée (Terminated), son nom.

Des informations détaillée sur une tâche sont affichées avec la même commande complétée du numéro de la tâche :

```
(gdb) info tasks 2
Ada Task: 0x100804000
Name: c
Thread: 0x1903
LWP: 0
Parent: 1 (main_task)
Base Priority: 31
State: Delay Sleep
```

Basculons sur l'inspection de la tâche 2 avec la commande "task" qui sans paramètre donne la tâche courante :

```
(gdb) task
[Current task is 1]
(gdb) task 2
[Switching to task 2]
0x00007fff889d20fa in __psynch_cvwait ()
   from /usr/lib/system/libsystem_kernel.dylib
(gdb) bt
#0  0x00007fff889d20fa in __psynch_cvwait ()
   from /usr/lib/system/libsystem_kernel.dylib
#1  0x00007fff889f4f89 in ?? () from /usr/lib/system/libsystem_c.dylib
```

Malheureusement, un point d'arrêt arrête l'ensemble des tâches.

Pascal Pignard, décembre 2005, février-mars 2006, décembre 2010, février 2011, septembre-décembre 2012.