道具としての Emacs (Gud/gdb 編)

藤原 誠

日本のインターネット人口のうち PC を使って接続している人は 2001 年に 2,500 万人などと言われています。つまりそれだけの人々が PC を使っている訳ですが、このうち、何人の人が Emacs や vi を使っているのでしょうか？ (何の根拠もないのですが) 多めに見て Emacs を使っている人が 10 万人と仮定すると、0.4 % になります。両方で 1% くらいということでしょうか。1000 人の中で Emacs が 4 人 vi が 6 人？ そのようなものでしょうか。あるいは最近は全体の PC 人口の方が増えているでしょうから割合はもっと減っているかも知れません (以上根拠のない仮の話ですので数字は忘れて下さい:-)。

さて、人から Emacs に誘われる場合、まずは**動く環境**を提供してもらう、手とり足とり**丁寧に教えて**もらう、こんな**素晴しい機能**があるよと見せてもらうことが考えられます。

Emacs を使い始めるにはどうしたらいいか

人に**動く環境**を作ってもらえれば、その方が幸運です。しかし、そうでなければ、自分で用意します。これには、次のような場合が考えられます。

- OS に初めから用意されているのでシェルで emacs と入力するか、画面でスタートメニューやアイコンを使って起動すれば良い (RedHat, Fedora Core)
- ports や pkgsrc で簡単設置 (FreeBSD, NetBSD)
- OS に初めから入っている。古めで端末表示しか出来ないが (`emacs -nw`)、とりあえず使い始められる (Mac OS X, Windows + cygwin)
- 21.3 は端末でしか動かないので、cvs で取って来て、patch も当てて `make bootstrap` (Mac OS X 10.3)
- 親切な人が実行形式を用意してくれているので、もらって来てダブルクリックすれば良い (Mac OS X, Zaurus)
- 名前は違うが機能は殆んど同じ Meadow というものがある。Netinstall で設置すれば気軽に使い始められる (Windows)
- 自分で `wget, tar, ./configure; make; make install` で用意する (Emacs-21 の場合にはその前に画像関係の準備が何点か必要)
- その OS での make は少し難しい (でもやる ?)

以上で具体的な URL などが必要な場合は次を参照して下さい。
http://emacs-21.ki.nu/how-to-start/

丁寧に教えてもらう、教えてあげる

これには例えば、次のような項目があります。

- 入力・編集方法など Emacs そのものの使い方
- 良く使われるモード、例えば dired (ディレクトリ・エディタ) の使い方
- 日本語入力の方法

Emacs は画面で説明を読めるので、自習でも使い方は覚えられると思います (ただ、英語が多いです)。C-h T と入力して始める個人指導 (tutorial) は日本語なので、これから始めるのも一つの方法です。

日本語入力については選択肢や、考慮点が多いこと、Emacs 以外での入力も、できれば同じような使い方にしたいなど、いろいろと考える必要があるのですが、**これだっ**[1]ていうのがないのが困るかも知れません。

1 筆者は tamago + FreeWnn だと思うのですが、今なら tamago + anthy-agent か anthy.el + anthy-agent かも知れません

初校 [原稿 $RCSfile: emacs.tex,v – $Date: 2004/07/26 14:17:49 (GMT), 組版: 2005/6/25 6:38]

素晴しい機能/道具としての Emacs

Emacs の用途としては、プログラムや文書の作成、HTML の作成、メールの読書き、などが考えられます。しかし、その他にも様々な Elisp パッケージが用意されています。これらを使うと、単に文書やプログラムを作るだけではなくて、**もっといろいろなこと**が出来ます。これが**道具としての Emacs** と言われる理由です。そのパッケージには、**Emacs に初めから付属**しているものと、自分で**後から追加**するものがあります。それも、Emacs の版が進むと、別だったものが付属になる場合や、OS によっては、初めから親切に入っているものなどがあります。これらの Elisp パッケージは、多くの場合、**外部のプログラムを起動**します。

今回は、特にそれらの中で、Emacs を使ったことがない人にも「へぇー、こんな機能があるのか、Emacs を使ってみようかな」と言われそうな、**おすすめの gdb** を紹介します。この Emacs の gdb 機能は初めから付属しています。そうして gdb という外部プログラムを起動します。この gdb はシェルから gdb と入力して使うことも出来るけれど、Emacs から使うと、もっと便利に使える、という訳です。

以下では Emacs-21.3 で動作確認をしていますが、この記事の後半には、21.4 での画面構成と操作も示しています。

Emacs から gdb を使う

gdb は GNU のデバッガです。自分で書いたプログラムや、人の作ったプログラムがうまく動かない時、Segmentation Fault してしまう時などに何が問題なのかを調べるのに使います。

gdb はシェルから起動して使う方法もあります。gdb と言えば、「core を見て、where や bt を入力して、問題を起こした場所を見つける道具」ということまでは御存知の方が多いかも知れません。その時は次のように引数として実行形式とコアファイルの名前を指定して起動します。

図 1 例題: factorial.c

```
#include <stdio.h>
main () {
  int y;
  int f = 1;
  printf("Hello World!\n");
  fflush(NULL);
  for ( y = 0 ; y < 11; y++) {
    f *= y;
    printf("Factorial %2x:  %8d\n", y, f);
    fflush(NULL);
  }
  return 0;
}
```

```
$ gdb /usr/bin/named /etc/named/name.core
...
..
(gdb) where
...（スタックの様子を表示）
```

他の人の作ったプログラムが Segmentation Fault してしまう場合があるとします。そのときに、自分で解決しなくても、上のようにして調べてこの様子を報告するだけで、他の人が問題を理解でき、解決が早くなります。これだけでも gdb はとても重宝します。

Emacs から起動した gdb を使って Segmentation Fault を調べる場合、特に大きなプログラムで、偉力を発揮します。そのいい例があるのですが、場合が特殊なのと、この記事の後に見ていただいた方がいいと思って、それはこの記事の**最後に付録**として付けてあります。

さて、Emacs と連係した gdb には、ソースコードを表示して、そのソースの実行中の行を示してくれる機能があります。これはすぐに試すことが出来、またわかりやすいと思うので最初に紹介します。

gdb の準備 (Makefile と factorial.c)

gdb を使うためには対象のプログラムを **-g を付けてコンパイル**します。これで、プログラムの中で使われている**名前** (symbol) **とアドレスの対応表**が生成され、それが実行形式に付属します[2]。

この -g を付ける一つの方法は、Makefile に設定しておくことです。これから入力しようとしている例題の factorial.c と同じディレクトリに Makefile という名前のファイルを用意しておきます。内容は次の一行です。

```
CFLAGS= -g
```

2 strip を使うと、それの削除も可能です

表 1 M-x の入力方法

ESC	x	ESC の後に x を入力
Alt	+ x	Alt と x を同時に入力
C-[	x	Ctrl と [を同時に入力。その後に x
⌘	+ x	Mac の ⌘ キーも Alt の代りに使えます

そうして、例えば図 1 のような factorial.c を用意します。これが今回の対象プログラムの例題です。この C のコードを Emacs で入力するか、もし慣れていないなら他のエディタで作っても構いませんが、その後で、Emacs でそのファイルを開いておきます。

コンパイルも Emacs で (M-x compile)

他のエディタで作った場合には、シェルで、例えば次のようにして Emacs を開きます。

```
$ emacs factorial.c
```

あるいは、Emacs で入力した場合にはその開いたままの窓で構いません。そうして、どちらの場合でも、次のように入力します。

```
M-x compile RET
```

ここで、M-x は ESC x と入力します。または Alt と x を同時に押します。あるいは Ctrl と [を同時に押した後に Ctrl を離してから x を押します (表 1 参照)。RET と書いてあるのは [改行] を入力します。そうすると、ミニバッファ[3]に次のように聞いて来ます。

```
Compile command: make -k
```

この行の最後に次のように factorial の字を補います。

```
Compile command: make -k factorial
```

これで改行を入力すると、画面が図 2 のようになります。下の窓の下から二行目のように cc の後に -g が付いているのを確認しておきます。

Makefile にもう一行加えて次のように書いておくと、

```
CFLAGS= -g
factorial:
```

M-x compile RET として

```
Compile command: make -k
```

と聞かれた時に、改行を入力するだけでコンパイルすることも出来ます。

3 Emacs の窓の下のにある操作入力用の窓です

図 2 コンパイル (M-x compile)

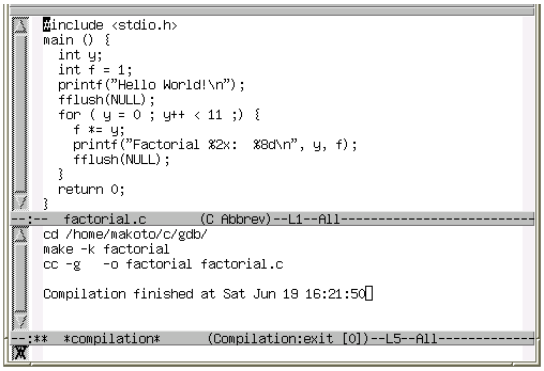


表 2 窓とバッファの操作

C-x o	反対側の窓に移る
mouse-1	クリックして窓とカーソル位置を選択
C-x 0	反対側だけの窓一つにする
C-x 1	自分だけの窓一つにする
C-x 2	窓を二つにする
C-x k RET	バッファを消す
C-x C-b	バッファ一覧を表示 (その後 f で選択)

gdb を起動 (M-x gdb)

カーソルを図 2 の下の窓に移動します。それには C-x o とするか、マウスで下の窓をクリックします。窓の操作は何かと必要になるので、少しだけ表 2 にまとめておきました。そうして、M-x gdb [改行] と入力して Emacs の gdb モードを開始します。

```
M-x gdb (改行)
```

すると最初に、どのように起動するか聞いて来ます。

```
Run gdb (like this): gdb
```

そこで次のように、factorial という名前を追加して[4]改行を入力します

```
Run gdb (like this): gdb factorial
```

(f だけを入力した後に [Tab] をすると factorial まで補ってくれるかも知れません)。これで、図 3 のような画面になり、下の窓に

```
(gdb)
```

と表示して gdb の操作入力待になります。コンパイルが済んだ後で emacs を起動したなどの場合、画面の上下が図 3 と逆になる場合もありますが、どちらでも同じ

4 Mac OS X や、後で説明する cvs 版の場合、Run gdb (like this): gdb --annotate=3 と聞いて来ます。それはそのままにして、後に同じように追加します

図 3 Emacs から M-x gdb で gdb を起動

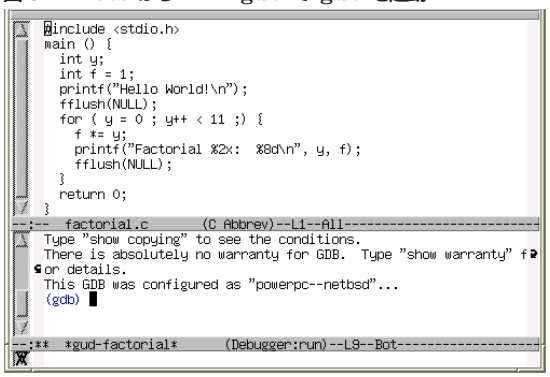
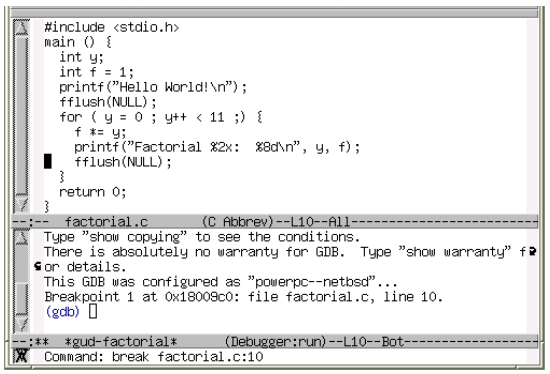


図 4 C-x 空白でブレークポイントを設定



です。

ブレークポイントを設定 (C-x 空白)

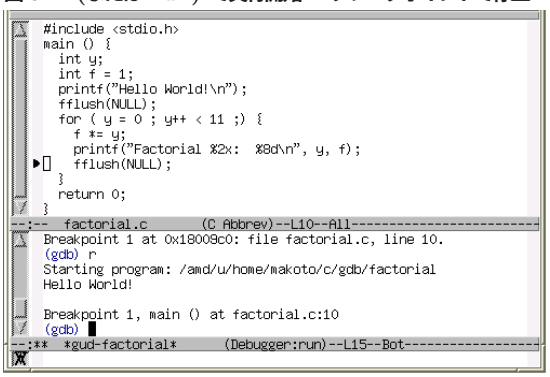
図 3 で run とすれば、いまコンパイルした factorial を実行してくれます。しかし何も考えないで run をすると最後まで走ってしまうだけで gdb を使っている意味がみえません[5]。そこで、実行前に少し準備します。

例えば、プログラムの実行途中で調べたいことがあるので、その前で止って欲しいとします。そこで停止点 (ブレークポイント) を設定します。

図 3 の二つ見えている窓のうち、ソースの方の窓 (この例では上側) に fflush(NULL); という行が二つあります。その二つ目の方 (図 4 の上の窓でカーソルがある行) にカーソルを持っていって **C-x [空白]** と入力します。これで画面が、図 4 のようになります。この下の窓の、下から二行目には、次のような表示があります。

5 もし Segmentation Fault するなら run だけでも意味があるのですが

図 5 r (または run) で実行開始 →ブレークポイントで停止




```
Breakpoint 1 at 0x18009c0: file factorial.c, line 10.
```

これは 1 番目のブレークポイントが factorial.c の 10 行目に設定され、この例ですと、その実際のアドレスは 0x18009c0 だと言っています。

実行開始 (r または run)

以上で準備が出来たので C-x o と入力するか、マウスでクリックして図 4 の下の窓の (gdb) の後にカーソルを置きます。そうして、次のように入力して

```
(gdb) run [改行]
```

実行を開始します。短く r [改行] とだけ入力しても同じことが出来ます (実際には下の窓では、カーソルはどこにあっても良く、改行を入力した時に、その左側に入力したい文字、この場合なら run、だけが書いてあれば充分です)。すると、図 5 のように

```
Starting program: /home/makoto/c/gdb/factorial
```

(など) と表示され、先ほど設定したブレークポイントまで進んで止ります。図 5 の上の窓の左側には、停止した行を示す三角の矢印が fflush(NULL); の行を指しています。

n または next で次の行を実行

図 5 で、下の窓にカーソルがある時に n または next を入力すると、上の窓の、矢印のある fflush(NULL) の行が実行され、その結果が図 6 のようになります。この中には次のように一行表示が出ています。

```
Factorial 1: 1
```

図 6 n (または next) で次の文を実行

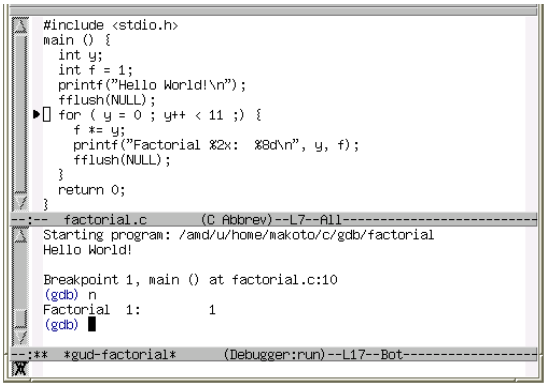
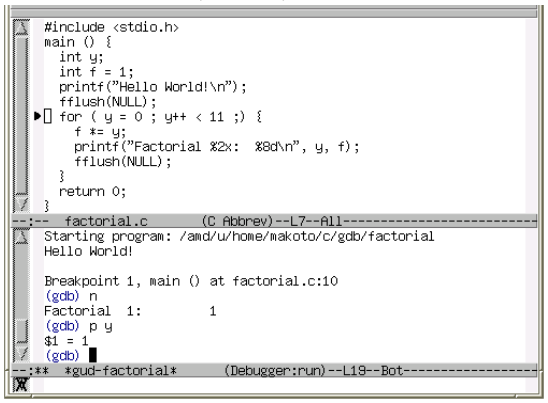


図 7 変数の値を表示 (print y)



そうして、図 6 の上の窓では、矢印が次に実行する for の行に移っています。

変数の値を表示 (print)

このように停止した時に、変数を値を表示させることも出来ます。変数の名前を指定して、

```
print y
```

あるいはもっと短く p y [改行] と入力します。そうすると、図 7 の下の窓のように、

```
$1 = 1
```

という形で指定した変数の値 (この例では y = 1) を表示[6] します。print y の代りに、

```
print "%0x", y (あるいは)
print/x  y     (あるいは)
    p/x  y
```

6 $1 のように表示されているのは、履歴変数で、後でこの内容を利用するためのものです

図 8 再び実行継続 (c または cont)

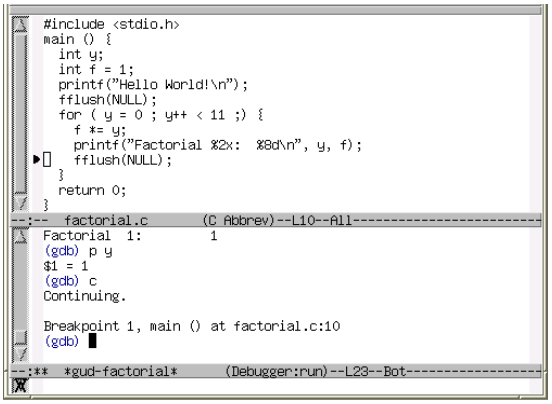
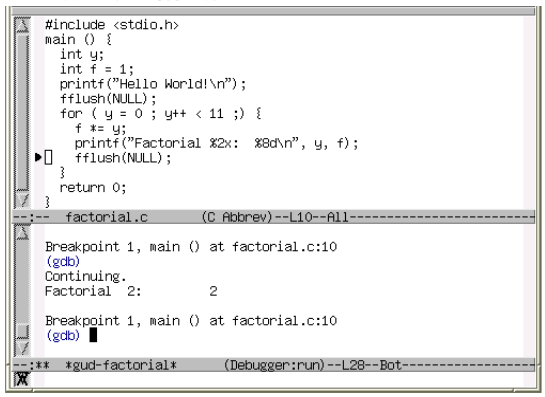


図 9 改行は前の操作の繰返し



のように入力すると 16 進表示にすることも可能です。

再び実行継続 (c または cont)

図 7 の下の窓で c (cont でも同じ) を入力すると、次の breakpoint まで実行します (図 8)。

改行は前の操作の繰返し

(gdb) の入力待では、単に [改行] を入力すると、直前に入力した操作を繰返します。例えば、図 8 の場合、直前入力は c でしたので、[改行] を入力すると実行を継続して、次の breakpoint まで進みます (図 9)。

図 8 と図 9 ではそれぞれの上の窓で、矢印が同じところを指しています。しかし

図 9 の下の窓には fflush(NULL) の行の実行結果の、

```
Factorial 2: 2
```

という行が表示されていて、一まわりして同じところに

図 10 ブレークポイントの表示と消去 (delete 1)

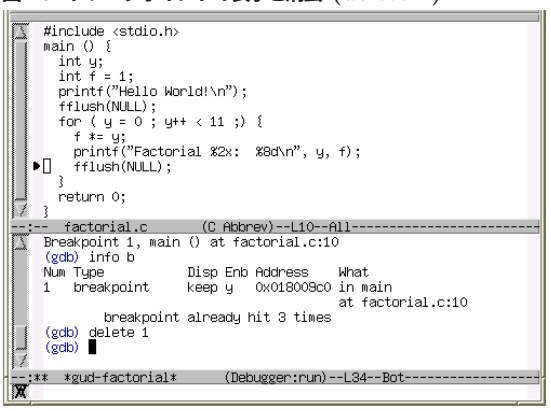
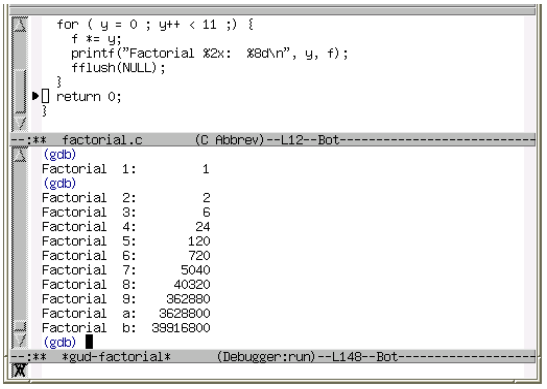


図 12 until を使ってループの中を効率的に通過



止っているのです。

ブレークポイント表示 (info b)

ブレークポイントはいくつでも設定出来ます。それで、いまどこにブレークポイントが設定されているか知りたいこともあるでしょう。その時には、(gdb) と表示されている入力待で info b と入力すると、図 10 の下の窓のように、breakpoint を表示します。この図の一部を拡大したものを図 11 に示します。

n の代りに u, until を使ってループを通過

図 6 で n を使って一行づつ実行する方法をみました。代りに、u (または until) を使うと、ループの中を効率的に通れます。ループを何回も繰返すのは一行づつ実行しなくてもいい、という場合、u を使います。u はループの中を初めて通る時には一行づつ実行します。しかし、二回目以降は、省略して、**ループの終了した次の行**まで進めます。図 12 では u の後に、改行を何回か入力して u を繰返したので、ループを抜けた return 文のところまで進んでいます。

r を入力すると初めからまた実行する

例えば、前の続きで r と入力してみます。すると、またプログラムの**初め**から、同じように実行して、一つ目のブレークポイントで止ります。何か間違えてもやり直しが効くことが解っていると、安心していろいろなことが試せます。とは言っても、今回の例題が画面に表示するだけのものだからそう言っていられます。**極端な例**として、調べているプログラムが、何かものを削るような工作機械の制御をしているとしたら、何回も起動する訳にはいかないでしょう。

ブレークポイント消去 (delete)

は図 11 の左端の Num の欄には breakpoint 番号が表示されています。

この番号を指定して delete (番号)、つまりここの例では delete 1 または短く d 1 と入力すると、この breakpoint を消せます (図 10 の下の窓)。

もう一度 c をして最後まで

さらに、c を入力すると、継続して実行しますが、今回はブレークポイントを消してしまったので、最後まで実行します (図 13)。これで対象プログラムが終了します。

help の表示

(gdb) の入力待で help と入力すると、図 14 のような表示になります。これは 更に help breakpoints な

図 11 ブレークポイント表示 (info b で breakpoint を表示)

```
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x018009c0 in main at factorial.c:10
```

図 13 実行継続 (c または cont) で最後まで

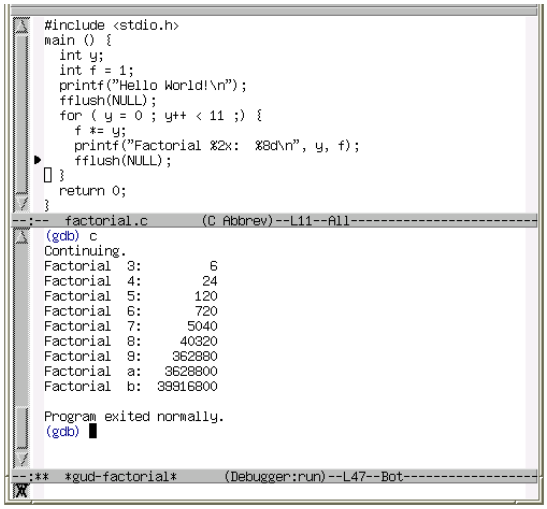
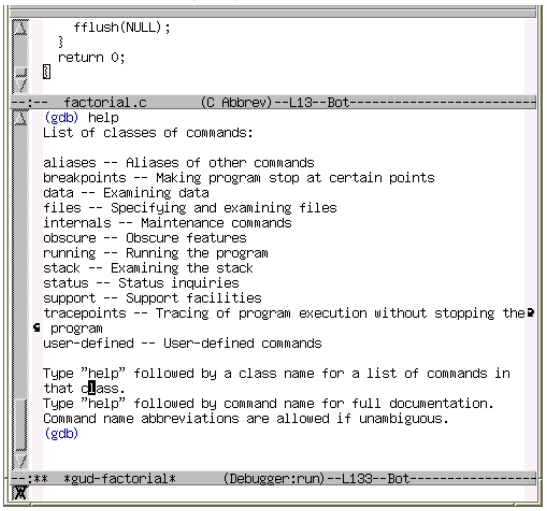


図 14 help の表示 (入口)



どと入力して本当の help を見て下さいというような表示内容です。例えば help running と入力すれば、実行の方法の説明が表示されます。

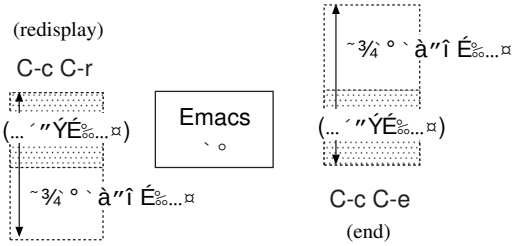
shell モードと同じ機能 (キー) も使えます

少し話が飛びますが、Emacs には gdb と見かけが少し似たシェルモードがあります。開始するのは M-x shell [改行] です。これは Emacs の中でシェルの操作をします。gdb では、このシェルモードのキーと同じ割当が使えます。たとえば**入力**時には M-p, M-n で以前

表 3 シェルモードと同じ出力表示の操作

C-c C-r	再表示	直前の表示の**先頭**を窓の**最初**に表示。最初から見直す
C-c C-e	表示の最後	直前の表示の**最後**が窓の**下端**になるように表示
C-c C-o	出力取消	表示を消す

図 15 シェルモードと同じ出力表示の操作 (窓高 < 操作表示)



に入力した操作を呼出してもう一度使えます。

(gdb) の実行の結果の**出力**表示については、表 3 のような操作があります。C-c C-r, C-c C-e は図 15 のように、直前の操作の表示が窓高よりも長い時に便利です。また表の最後の C-c C-o は直前の出力の表示を消します。例えば、(gdb) と表示されている時に、help info と入力すると、help が長々と表示されます。さて、その表示は見終った、もう要らないという時に、C-c C-o と入力[7]すると、次のようにその表示が消されます。もし消

```
(gdb) help info
*** output flushed ***
(gdb)
```

した後で、また見たいなと思った時には、C-_ (Ctrl と下線) または C-x u を入力して「やり直し」(undo) を使えば元に戻して見ることも出来ます。

表示されている内容を保存

今回の説明では、モード行に次のような字が見えていました (例えば図 14 の下端に見えます)。

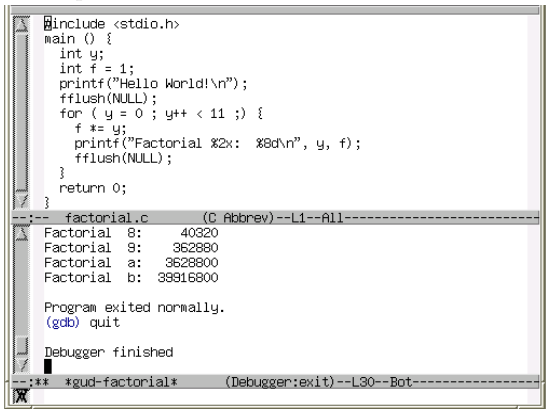
```
--:** *gud-factorial*  (Debugger:run)--L139--
```

この *gud-factorial* が今回の gdb のバッファの名前[8]です。このバッファに表示されている内容を後でゆっくり見たいので保存したい、ということがあるかも知れません。その時には、そのバッファにカーソルがある時に、

7 Ctrl を押したまま c と o を順に押してから最後に Ctrl を離します

8 gud という名前は Grand Unified Debugger の略です。Gud は gdb 以外のデバッガも利用出来ます。

図 16 quit で終了



C-x C-w と入力すると次のように聞いて来ます。

```
Write file: ~/c/gdb/
```

(~/c/gdb/ の部分には、実際には現在のディレクトリが表示されます) ここで保存したい名前を入力して改行します。

gdb の終了は quit

gdb を終了するには quit を入力します (図 16 の下の窓)。さらに、この下側の Debugger finished と表示されている窓が不要なら、C-x k [改行] でバッファを捨てます。あるいは C-x 0 を入力すると、カーソルのある方の窓を消して、一つの窓になります。C-x 0 で表示を消した場合には、バッファは残っているので、次に M-x gdb と入力すると、またその画面を表示します。

namazu-2.0.13-1 を使って

以上紹介した方法は、Emacs/gdb の初歩ともいうべきものでした。しかし実際にはもう少しこみ入った操作や準備が必要になるはずです。そのような例として、全文検索ソフトウェアの namazu-2.0.13-1.tar.gz を取上げて見ます[9]。namazu は動かすために、いくつか別のソフトウェアを用意する必要があります。しかし既に入っている場合や、ports/pkgsrc から入れてある場合には、必要なものは用意されていることになりますので、それらをそのまま利用出来ます。あるいは、Namazu 以外の他のものを取上げても似たような操作になりますので、ここは読むだけでも参考にして下さい。

図 17 namazu を configure;make

```
$ wget -q http://www.namazu.org/ ⇒
       stable/namazu-2.0.13-1.tar.gz
$ tar zxf namazu-2.0.13-1.tar.gz
$ cd  namazu-2.0.13
$ ./configure CFLAGS="-g -O2"
$ make
```

⇒ の字は紙面の都合での折返しを表わしています。実際にはここは一行です (以下同様)。この図の場合は間に空白なしの一行です。

configure で CFLAGS に -g を付ける

gdb で調べるためには、コンパイル時に -g が必要です。ここでは図 17 の下から 2 行目のようにして、configure の時に CFLAGS に -g を指定して、その後に make します。

-g と -O は同時に指定出来ない場合もあります (gcc 以外)。その理由もあって、-O は指定しない方が良い、という話もあります。確かに -O2 を付けておいて n, next を使うと、実行順序が前後してしまいます。しかし、実情はこれを付けたものが使われるのが普通なので、ここでは付けています。

mknmz で索引を作っておく

namazu を実行するには予め mknmz を使って索引を作ってあることが前提です。もし作ってあれば、それを使えばいいのですが、ない場合もあるでしょうから、同じディレクトリにある doc を対象に索引を作って見ます。namazu の make をしたのと同じディレクトリのシェルで図 18 のように操作します。まず mkdir index で索引を置くディレクトリを作っておきます (この index の名前は特に意味はありません。名前が他と重ならなければ、何でも構いません)。

今回は実験ですので make するだけで make install はしません。make install せずに make した位置で実行するには少しおまじないが必要です。図 18 の 2 行目で、環境変数の pkgdatadir に 'pwd' を使って現在のディレクトリを設定しているのはそのためです。

また日本語を検索できるようにするには、mknmz の実行時に環境変数 LANG などに ja で始まる値を設定[10] しておきます (図 18、3 行目)。

9 http://www.namazu.org/

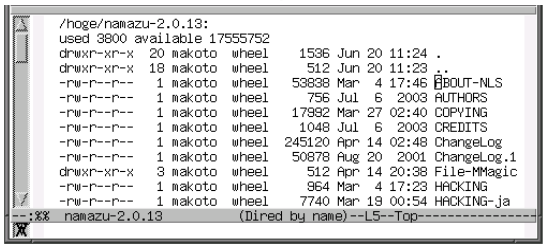
10 http://www.ki.nu/software/namazu/tutorial/lang.html

図 18 (図 17 の続き) mknmz を実行して索引を作っておく

```
$ mkdir index
$ env  pkgdatadir=‘pwd‘ \
             LANG=ja_jp.eucJP  \
       scripts/mknmz -O index doc
Looking for indexing files...
（以下 20 数行略）
```

\ はシェルの継続行を表わします。その字を使わず、代りに空白でつないで一行で入力しても構いません。

図 19 emacs . で dired が開く



次に gdb の中で namazu を起動しますが、その前にもう少し準備をしておきます (Emacs → gdb → namazu)。

~/.emacs に一行追加

gdb の中で、実行時の引数の指定に日本語を使えるように ~/.emacs の中に次のおまじないを書いておきます。

```
(modify-coding-system-alist ’process "gdb"
                  ’(euc-japan . euc-japan))
```

ここでは二行に分けて書いてありますが、続けて一行に書いても同じです。これで **emacs から起動する gdb という名前のプロセス**の、入出力の符号体系を euc-japan に設定します。OS によっては sjis あるいは utf-8 にする必要があります (Mac OS X 10.3 の場合は sjis です)。~/.emacs を変更した場合には emacs を再起動するか、追加部分を M-x eval-region などで有効にしておきます。この設定はそのまま ~/.emacs に残しておいて大丈夫だと思います。

emacs の中で gdb を起動

さて以上の準備が出来たら、やはり make したところと同じディレクトリで emacs を起動します。引数にピリオド (.) を指定します。

```
$ cd /hoge/namazu-2.0.13
$ emacs .
```

図 20 M-x gdb → (gdb) 入力待

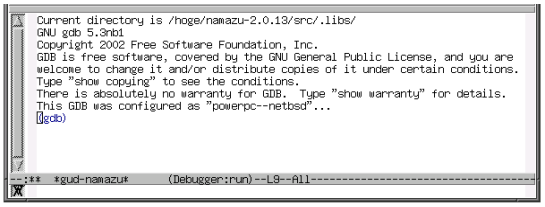


表 4 break (または b) の書式

break	行番号	表示したソースの行番号にブレークポイントを設定
break	関数名	関数の入口にブレークポイントを設定

これで図 19 のように dired の画面になりますが、ここではそれは利用しません。最初の時 (図 3) と同じように

```
M-x gdb RET
```

と入力すると、次のように聞いて来ます。

```
Run gdb (like this): gdb
```

ここで次のように、コンパイルして出来ている実行形式を指定します。

```
Run gdb (like this): gdb src/.libs/namazu
```

これで改行を入力すると emacs の画面の中で図 20 のように (gdb) の入力待になります。

入口にブレークポイントを設定 (b main)

最初の例の図 4 では、ソースを表示している時に **C-x 空白**でブレークポイントの設定をしました。そうでなくて、(gdb) と表示されている入力待でも設定出来ます。表 4 のように b (または break) の後に、行番号や関数名を指定しても、ブレークポイントを設定出来ます。

(Emacs からの使い方に限らないのですが) 例えば list main とすると、main 関数の前後を表示します。

```
(gdb) list main
    （前略）
290     int
291     main(int argc, char **argv)
292     {
    （後略）
```

この時に表示された 291 行目という数字を使って、

```
(gdb) break 291
```

などとすることも可能です。

普通は実行を開始する入口の名前が main() と分っています。これを利用して、run と入力する前に、以下のようにb main と入力して、関数名を指定して breakpoint を設定します。

```
(gdb) b main
Breakpoint 1 at 0x18073d0: ⇒
           file namazu-cmd.c, line 294.
```

以下で (gdb) の入力待になっているのは emacs のバッファの中での操作です。

実行開始 (run) →共有ライブラリがない?

おそらく、この後に単に run とすると、次のように

```
(gdb) run
Starting program: ⇒
   /hoge/namazu-2.0.13/src/.libs/namazu
Shared object "libnmz.so.7" not found

Program exited with code 01.
```

共有ライブラリがないと言われてしまいます。これは make install 前なので止むを得ません。

そこで環境変数を設定します。次のように pwd を使うとカレントディレクトリを表示します。それを見る

```
(gdb) pwd
Working directory ⇒
           /hoge/namazu-2.0.13/src/.libs.
```

と、ここでは .libs になっています (実は pwd の値は図 20 の一行目にも表示されています)。そこでここからの相対パスを使って、共有ライブラリを探す環境変数 LD_LIBRARY_PATH を設定します。それには set env を使って、次のように指定します (短い env の代りに environment と入力しても同じです)。

```
(gdb) set env LD_LIBRARY_PATH ../../nmz/.libs
```

本当に実行開始 (run), main の前で停止

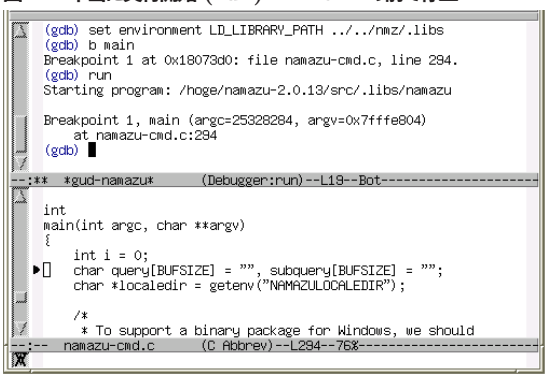
この後に、図 22 の上から 4 行目のように run と入力します。run は何回入力しても最初から実行するので、前の操作に続けて構いません。main 関数にブレークポイントが設定してあるので、これで図 22 のように入口、つまり main という名前の関数で止ります。正確には、それに対応したアドレスにある命令を実行する前です。図 22 の上の窓は (大きくすると) 図 21 のように表示されています。

図 21 ブレークポイントを設定したので main の入口で止まる

```
(gdb) run
Starting program: /hoge/namazu-2.0.13/src/.libs/namazu

Breakpoint 1, main (argc=25328284, argv=0x7fffe804) at namazu-cmd.c:294
```

図 22 本当に実行開始 (run) → main の前で停止



引数なしで n 20 (next 20 回)

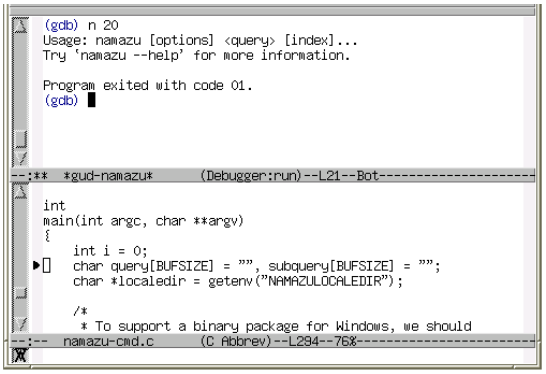
next は「次の文を実行」します。next は単に n でも有効です。例えば引数を付けて、n 20 と入力すると、next を 20 回入力したのと同じになります。入口 (main) で止っているところから、この n 20 を入力してみます。実は今は (namazu の) 引数を指定していないので、簡単な help メッセージを表示して終了してしまいます (図 23)。

実行時の引数の指定 (set args)

そこで help 表示で終わらないように namazu の実行に引数を指定します。普通のシェルなら、次のように入力するところです。

この場合の意味は、-l (list の意) でファイルの名前だけを表示する、「安定」という字を検索する、索引の置いてある場所は ../../index という名前、という指定です。さて、この引数の部分だけを set args を使って指定します。例えば、次のようになります。(もし日本語入

図 23 引数なしで n 20 (next 20 回) → Usage:




```
$ namazu  -l 安定 ../../index
```

力の方法が分らない場合には安定の字の代りに mission とでも入力しておいて下さい)。

引数を設定したので run すると (正常終了)

breakpoint は消しておいて、これで、run してみます。いままでの設定をまとめて示すと、次のとおりです。

```
(gdb) delete breakpoints
(gdb) set env LD_LIBRARY_PATH ../../nmz/.libs
(gdb) set args -l 安定 ../../index
```

これで run すると、次のように表示されます。

```
(gdb) run
Starting program: /hoge/namazu-2.0.13/src/ ⇒
                .libs/namazu -l 安定 ../../index
/hoge/namazu-2.0.13/doc/ja/tutorial.html

Program exited normally.
```

この下から 2 行目の ja/tutorial.html で終っている行は、gdb の操作の表示に namazu の実行結果が混って表示されています。この結果は、そのファイルに「安定」という文字が見つかった、という意味で表示されています。

ソースをながめてブレークポイントを設定

前の項では delete breakpoints を使ってブレークポイントを全て消してしまいました。そこでもう一度、続

```
(gdb) set args -l 安定 ../../index
```

きの (gdb) の入力待で

```
(gdb) b main
```

と入力しておきます。そうして run で初めから実行します。これで以前の図 22 と同じように止ります。

この main で止ったところで、このソースである main() を定義している namazu-cmd.c が表示されます。さらに少しソースをながめていると、同じ namazu-cmd.c の 369 行目付近の次の行がどうも**本格的な処理**らしい、と目を付けます。

```
if (namazu_core(query, subquery) ⇒
                        == ERR_FATAL) {
```

そこで、この行にカーソルを置いて **C-x 空白**しておきます。その時には Breakpoints の設定の様子が図 24 のように表示されます。

これで c すると、b main で停止していたところから再開して、いま設定したブレークポイントの namazu_core の行で止ります (実行の操作を何か間違えてしまったら、run から始めれば、何回でも同じことが出来ます)。

step を使うと、関数呼出の中まで入る

いままで実行の制御には、開始 (run)、継続 (cont)、次行 (next) 等を使って来ましたが、ここでは **step の s** を使います。この s は関数呼出 (サブルーチン) の中に入って行きます。その結果、図 25 のように namazu.c の中の namazu_core の入口で止っている表示になります。

はじめには図 6 で、next を使いました。この n (next) は一つの関数の中で、さらに呼んでいる関数の中に**入ることなく**次の行にいきます。一方 s (step) を使うと、呼ばれている関数の中に入って行きます。

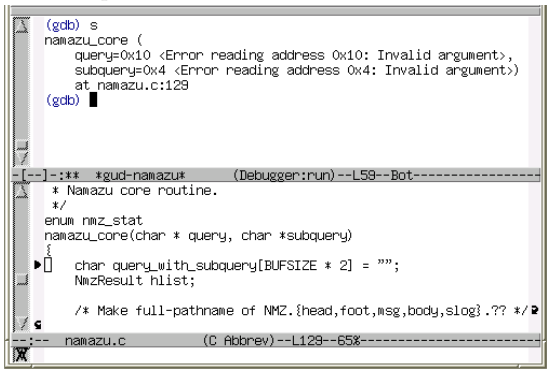
backtrace, bt 呼出関係の階層表示

(gdb) の入力待で help と入力するとその表示の中に次のような stack という字があります (図 14 の下の窓

図 24 namazu_core の行にもブレークポイントを設定

```
Breakpoint 1, main (argc=25328284, argv=0x7fffe808)
    at namazu-cmd.c:294
Breakpoint 2 at 0x1807504: file namazu-cmd.c, line 369.
```

図 25 step で関数 namazu_core に入ってとまる



の 10 行目)。

```
stack -- Examining the stack
```

これは「help の次に stack と入力すれば、スタックの調べ方についてもっと詳しい説明があります」という意味です。では、と help stack と入力するとその中に次の

```
bt -- Print backtrace of all stack frames
```

字があります。これは backtrace あるいは短く bt です。この bt を使うと、実行途中に、呼出関係を記憶しているスタック (stack frame) から情報を取出して表示し、どこから呼ばれているかの様子を見せてくれます。

例えば、**何げなくソースを見て**いて、この関数は、どのように呼ばれるのかなというような時があると思います。例えば、**C-x C-f** と入力して

```
Find file: /hoge/namazu-2.0.13/src/.libs/
```

と表示されるのを .lib/ の部分を消して、

```
Find file: /hoge/namazu-2.0.13/src/output.c
```

のように変更して改行すると、output.c が開きます。その output.c の中の 229 行目付近の

```
static void
print_hitnum_each (struct nmz_hitnumlist *hn)
{
```

が気になったとします。その時には、次の方法で

```
(gdb) delete breakpoints
```

いままでの設定を消してから、その 229 行目で C-x 空白 を使ってブレークポイントを設定します。

```
Breakpoint 1 at 0x18035f8: file output.c, ⇒
                                    line 229.
```

実行 (run) に必要な次のような設定はしてあったとして

```
(gdb) set env LD_LIBRARY_PATH ../../nml/.libs
(gdb) set args -l 安定 ../../index
```

(あるいは後から設定しても同じです)、次のように入力します。

```
(gdb) run
```

すると、上の 229 行目の print_hitnu_each に設定した breakpoint で止ります。ここで **bt と入力**します。これで停止している時の呼出関係 (backtrace) を表示します。実際の表示例は図 26 のようになります。この図は、下の方から上の方に向って、順に呼出されている様子が引数の値とともに表示されています。これらの情報はスタックに記憶しているので、スタックを表示などと呼ばれることもあります。

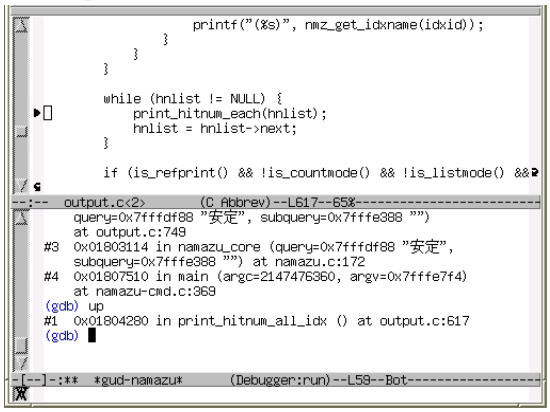
up down で階層を上下してソースを表示

ここで **up** や **down** と入力すると、呼出関係の階層を上下してそれに対応するソース画面を表示します。例えばここで up と入力した画面を 図 27 に示します。上の窓で矢印が指しているのは図 26 の `#1` の行にあった、関数 print_hitnum_each を呼出している部分です。こ

図 26 bt (backtrace) で呼出されている様子を表示 (呼出関係の階層表示)

```
Breakpoint 6, print_hitnum_each (hn=0x7fffdf88) at output.c:230
(gdb) bt
#0  print_hitnum_each (hn=0x7fffdf88) at output.c:230
#1  0x01804280 in print_hitnum_all_idx () at output.c:617
#2  0x018046f4 in print_result (hlist=
      {num = 1, stat = SUCCESS, data = 0x1825040},
    query=0x7fffdf88 "安定", subquery=0x7fffe388 "")
    at output.c:749
#3  0x01803114 in namazu_core (query=0x7fffdf88 "安定",
    subquery=0x7fffe388 "") at namazu.c:172
#4  0x01807510 in main (argc=2147476360, argv=0x7fffe7f4)
    at namazu-cmd.c:369
(gdb)
```

図 27 up で一つ上の関数を表示



れは output.c の 617 行であり、図 26 では、#1 に相当します。

この時にこのようにいろいろな画面が表示されても、今の実行で停止している状態には変化はありません。例えば up した後に n とすると、(up する前に) 最後に止って指していた行を実行します。

現在の関数を抜けるまで実行 (finish)

図 12 でループを抜ける u, until を紹介しました。これに似たものに finish があります。これは今実行中の **関数が終ったところ** (return 文の次) まで進めます。もうここは見たから次に進みたい、とか、間違って深く入ってしまったけれど早く抜け出したい、というような時に使います。finish と入力して、その後に bt で呼出関係を表示している様子を図 28 に示します。図 26 と比べると、#0 のところが消えています。他も数字が一つづつ繰上って、例えば#1 のところが一行だけ進んで #0 になっています。

図 28 現在の関数を抜けるまで実行 (finish) した時の bt

```
(gdb) finish
Run till exit from #0  print_hitnum_each (hn=0x7fffdfa8)  at output.c:230
print_hitnum_all_idx () at output.c:618
(gdb) bt
#0  print_hitnum_all_idx () at output.c:618
#1  0x018046f4 in print_result (hlist=
      {num = 1, stat = SUCCESS, data = 0x1825040},
    query=0x7fffdfa8 "安定", subquery=0x7fffe3a8 "")   at output.c:749
#2  0x01803114 in namazu_core (query=0x7fffdfa8 "安定",
    subquery=0x7fffe3a8 "") at namazu.c:172
#3  0x01807510 in main (argc=2147476392, argv=0x7fffe80c)  at namazu-cmd.c:369
(gdb)
```

図 29 ~/.gdbinit に設定する内容 (例)

```
cd  /hoge/namazu-2.0.13
exec src/.libs/namazu
file src/.libs/namazu
set environment LD_LIBRARY_PATH nmz/.libs
set args -l 安定 index
break main

# Local Variables:
# coding: euc-japan
# End:
```

(以上は、中に漢字があるので、euc-japan で保存しておきます)

実行しないで抜ける (return)

finish と似たものに return があります。これは関数を抜ける (終了する) のですが、現在の行から抜けるまでの、残りの文は実行しません。ただし、

(gdb) return 値

のように入力して、戻り値を指定出来ます。

変数の変化を監視する watch

watch を使うと変数が変化した時を教えてくれます。また display を使うと、ブレークポイントで止るたびに値を表示します。

設定を ~/.gdbinit に書ける

今回 gdb の起動時に同じことを何回か入力しました。そのような設定を ~/.gdbinit あるいは現在のディレクトリ、通常はソースのあるディクトリの .gdbinit に書いておけます。例えば、今回の場合、図 29 のように書いておきます。注釈文は # で始めます。図の欄外にも書きましたが、中に漢字があるので符号体系 (coding-system) を euc-japan にしておきます。これを保存する時に、モード行に漢字コードが、次のように E: になっ

ていればいいのですが、

```
-[--]E:** ~/.gdbinit  ...
```

もしそうなっていなかったら、

```
C-x RET f                          （または）
M-x set-buffer-file-coding-system RET
    （のどちらかの後に）
euc-japan RET
```

としてから C-x C-s などで保存しておきます。これで、

```
M-x gdb RET
```

と入力して、

```
Run gdb (like this): gdb
```

と聞かれた場合に、単に改行するだけで準備が出来、run と入力するだけで main で止まるところまで行けます。

ソースを消してしまった時 f または frame

何かの理由でソース側の表示を消してしまうことがあるかも知れません。そのような時には、

```
(gdb) f（改行）
```

と入力すれば、いま停止している部分を探して表示してくれます。

next step stepi の違い

以上では一行を実行するのに n(next), s(step) を使いましたが、nexti, stepi というものもあります。これは アセンブラ (機械語) の行単位で実行してくれる操作です。ソースがない実行形式をデバッグするのに役立ちます。

gdb のまとめ

表 5 にここまでの操作の一覧を示します。本稿では cont, next などの長い名前の操作を使いました。これはシェルで使う gdb の操作と同じ文字でした。しかしその代りに C-c C-r, C-c C-n などのキーも使えます。それも表に加えてあります。

表 5 gdb のまとめ (M-x gdb で起動した後)

操作	短縮	キー割当	意味・機能
【実行・停止】			
run	r		最初から実行
cont	c	C-c C-r	実行継続
break	b	C-c C-b	ブレークポイントの設定
ソース側で		C-x 空白	(同上)
tbreak		C-c C-t	一回だけのブレークポイント
delete	d	C-c C-d	ブレークポイント消去
next	n	C-c C-n	矢印行を実行
step	s	C-c C-s	ソース一行の実行
nexti	ni		(機械語一行)
stepi	si	C-c TAB	(機械語一行)
until	u		ループ抜まで実行
finish		C-c C-f	関数抜けまで実行
return			残りを実行せず抜ける
【スタック】			
backtrace	bt		(呼出) 階層表示
up		C-c <	(呼出関係) 浅 ↑
down		C-c >	(呼出関係) 深 ↓
【変数】			
print	p	C-c C-p	変数値の表示
watch			変数の変化を知らせる
display			変数を停止毎に表示
【ソース】			
list	l		ソース表示 (行番号)
frame	f		ソース表示 (窓)
【実行環境】			
set		set args 等	さまざまな設定
exec-file	exec		実行形式ファイル指定
symbole-file	symbol		シンボルファイル指定
file			実行形式+シンボル
core			コアファイル指定
dir			ソース探索指定追加
path			実行形式探索指定
【説明・情報】			
info	i		情報
help	h		説明
【その他】			
改行			直前操作の繰返し

キー割当を使うと、その操作は (gdb) の画面に表示されないので、画面がややすっきりします。

21.3.50 cvs 版の場合

以上、Emacs の中で GNU gdb を使う方法を見て来ましたが、実は Emacs の次の版では、画面が変ります。

Emacs 21.3 の次の版は 21.4 です。そのための開発中の版を匿名 cvs から取得して make bootstrap して使うことが出来ます[11]。それには図 30 のような方法があります。筆者は、今回 21.3 の方も残して使えるようにしておきたかったので、./configure の行を次のよう

```
  (図30の cd emacs の続き)
% mkdir /export/emacs
% ./configure --prefix=/export/emacs
% make bootstrap
  (以下図30に同じ)
```

に少し変更して普通でない位置に入るように指定しました。この場合なら、起動は次のようになります。

```
$ /export/emacs/bin/emacs
```

また Fedora Core1 以降などの新しいカーネルを使っている場合、実行形式の配置が変更されています。このため、そのままでは make bootstrap の最中の temacs から emacs を作るところで、Segmentation Fault します。etc/PROBLEMS に書いてあるのですが、そういう時には、

```
$ setarch i386 ./configure
$ setarch i386 make bootstrap
```

とやると、うまく行きます[12]。

このようにして作った emacs の版名は 21.3.50 です。cvs から make した時には、いつもこの版名で、変化はしません。ですので、何か問題がある時などは、この版名でなく、いつ cvs co して来た版なのかを明らかにしておいた方が話が通じます。

cvs 版を更新するには

make をしてから、しばらく時間が経って、新しい版に更新する時には、次のようにしてから、

```
$ cd emacs
$ cvs update -dP -C
```

11 ただし、文字通り開発中なので、動作しないものがあったり、仕様が変更になる可能性があるのを了承してお読み下さい。

12 実は筆者もかなりなやんだのですが、藤島裕士さんに教えていただき解決しました。

図 30 cvs 版を make bootstrap する

```
    (sh/bash の場合)
$ CVS_RSH=ssh
$ export CVS_RSH

   (csh/tcsh の場合)
% setenv CVS_RSH ssh

   (以下共通)
% cvs -d :ext:anoncvs@subversions.gnu.org: ⇒
              /cvs-latest/emacs  co  emacs
% cd emacs
% ./configure
% make bootstrap
% su
Password:
# make install
```

⇒ この図の折返しは空白なしでつなぎます。

```
$ make
  (または)
$ make clean && make bootstrap
```

します。cvs update に -C を付けておくと、自分で変更したファイルがある場合に、それを名前を変えて保存し、必ず更新してくれます。これはとても便利なのです。しかし、cvs が少し古い場合、例えば、Mac OS X, 10.3[13] またはそれ以前の版では、そのままでは、 -C は使えないので、入力しません。

cvs 版が make 出来ないこともある

cvs 版は**日々変更される**ので、時には make 出来ないこともあります。その時には co と emacs の間に日付を指定する文字列をはさみます。具体的には、図 30 の下から 7-8 行目を

```
% cvs -d :ext:anoncvs@subversions.gnu.org: ⇒
  /cvs-latest/emacs  co -D 2004-07-01 emacs
```

のようにして、確実に make 出来ると分っている日付を指定します。これを取消すには、つまり、このように「最新でないという指定」をした後に、それを取消したい時には -A を付けて update します。

```
$ cd emacs
$ cvs update -dP -C -A
```

cvs 版の gdb は五種類の窓を表示

このようにして出来た版の場合、~/.emacs に次のよ

```
(setq gdb-many-windows t)
```

13 cvs-1.10 が入っています

図 31 cvs 版 (21.4 対応) で起動すると

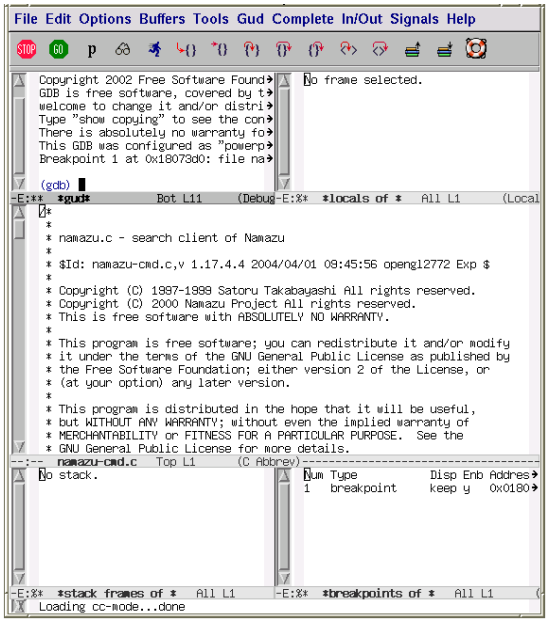


図 32 gdb 画面構成 (cvs 版)

操作卓 (console)	変数 (locals)
ソース画面	
スタック	ブレークポイント

うに書いておいて、

```
M-x gdb RET
```

と起動すると、図 31 のように、**五分割の画面**[14] になります。これで、gdb をもっと視覚的に使うことが出来ます。この五つの窓は図 32 のような構成になっています。この中で、**操作卓**と**ソース画面**の二つは、前に紹介した 21.3 で表示されていたものと同じです。それ以外の三つが新しく表示されるようになっています。次のように入力すると、以前のような二分割の画面に戻ります。

```
M-x gdb-many-windows RET
```

同じことをもう一度入力すると、またこの五分割になりま

す。もし画面が乱れてしまったら、次のようにすれば、

14 変数 gdb-use-inferior-io-buffer の値を nil 以外に設定すると六分割になります。(後述)

表 6 ツールバーのアイコン

	break	ブレークポイントの設定
	remove	ブレークポイントの解除
	print	変数の値を表示
	watch	変数を監視
	run	最初から実行
	until	ループを抜けるまで実行
	cont	実行再開
	step	ソース一行の実行 (関数内容)
	next	矢印行を実行
	finish	関数を抜けるまで実行
	si	機械語一行 (サブルーチン入)
	ni	機械語一行 (同上無)
	up	(呼出関係) 浅↑
	down	(呼出関係) 深↓
	help	説明

開始の五分割に戻せます。

```
M-x gdb-restore-windows RET
```

この新しい構成では、これら他にも、Input/Output、レジスタ、アセンブラ、スレッド等の窓 (buffer) も表示出来ます (詳しくは後述)。

図 31 の上にはツールバーがあり、アイコンが見えています。その意味などを表 6 に示します。この表の中の最初にある STOP  が以前の **C-x 空白**と同じです。カーソルをソースの中に置いて、このアイコンをクリックするとブレークポイントがその行に設定されます。そうして、ブレークポイントが設定された行には、**赤い点の印**が付きます。

変数名の上にカーソルを置いて watch  をクリックすると、その変数を監視するようになります。ただし、

図 33　ブレークポイント窓

表 7　ブレークポイント窓の表示項目

項目	意味
Num	ブレークポイント番号
Type	種別 (breakpoint/watchpoint)
Disp	停止した後の措置、保存 (keep) 消去 (del)
Enb	有効 (y) 無効 (n)
Address	メモリの番地
What	ソース名、行数

```
Watching expressions requires gdb 6.0 onwards
```

つまり式の形で監視するなら gdb-6 以降を使って下さい、と言われるかも知れません[15]。gdb-6 を入れておけば、これは避けられるので、そうするのも一つの方法です。執筆時点では gdb-6.1.1 が公開されています。筆者は (NetBSD/pkgsrc の devel/gdb6 で) make して gdb-6.1 を入れました。

ブレークポイント窓

図 31 などの右下に表示されているのが図 33 のようなブレークポイント窓です。表示項目の意味を表 7 に示します。この窓で breakpoint と書いてある行の上にカーソルを置いて**改行**を入力すると、そのブレークポイントに対応した**ソースを表示**します。d を使うと、ブレークポイントを削除します。以上のキー割当は図 34 の左側に示しました。

実はまだ説明していませんでしたが、ブレークポイントは、停止位置を記憶したまま、その**有効・無効化の切換**が出来ます (これは以前の版でも同じです。その場合は enable/disable (番号) と入力します)。無効化すると、停止位置は記憶したまま、一時的に通過するように設定できます。今説明している版では、図 33 のブレークポイント窓で、該当行にカーソルを置き、空白を入力すると、有効・無効化出来ます (gdb-toggle-breakpoint)。有効な場合には赤い点で、無効な場合には灰色の点で示します[16]

15 Mac OS X で 10.3 試した時には gdb は 5 でしたが何も言われませんでした

図 34　ブレークポイントのキー割当 (左) とメニュー (右)

機能	キー
対応ソースを表示	改行
削除	d
有効・無効 (反転)	空白

(注)

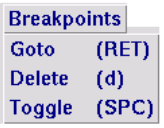


図 35　~/.gdbinit に一行追加

```
cd  /hoge/namazu-2.0.13
exec src/.libs/namazu
file src/.libs/namazu
set environment LD_LIBRARY_PATH nmz/.libs
set args -l 安定 index
break main
break print_hitnum_each

# Local Variables:
# coding: euc-japan
# End:
```

ブレークポイント窓にカーソルがある時にはメニューバーが次のようになっています。

その中の Breakpoints を選ぶと、図 34 の右側のような項目が表示されます。ここから選んで上に説明した機能を操作することも出来ます。

スタック窓 (bt, backtrace を常に表示)

スタック窓の表示例は図 36 の左下を見て下さい。この画面は、前に使った namazu-2.0.13-1 をもう一度例題にしています。まず以前に使った~/.gdbinit(図 29) に一行追加して、図 35 のようにします。下から四行目が追加した行です。これで、print_hitnum_each にも停止点を設定します。そうして、次のように入力して

```
M-x gdb RET
```

質問が来たら、最後の語は消して単に改行します。

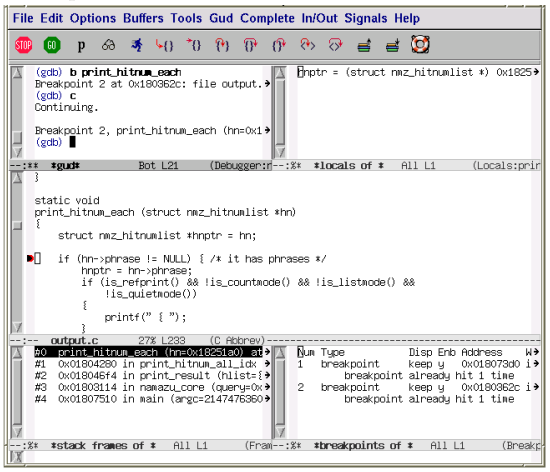
```
Run gdb (like this): nmz-config
                     ↓
Run gdb (like this): (改行)
```

これで、五分割の画面になります。次に r を入力します。

```
(gdb) r
```

16 (注) この段落に説明している機能は、2004-06-27 頃から動くようになりました (gdb-ui.el -r 1.21)。emacs をそれ以前に checkout した場合には動きません

図 36 print_hitnum_each で止っている様子



(あるいは 5 つ目の走るアイコン  をクリックしても同じです) これで main で止ります。さらに:

```
(gdb) c
```

のようにするか、7 番目のアイコン  を使って実行を再開すると、print_hitnum_each で停止します。その止ったところの画面が図 36 です。bt と入力するまでもなく、左下にスタックの様子が表示されています。

図 36 の (gdb) の入力待で up や down と入力すると、左下の黒く反転した行が上下し、ソースの窓の方はそれに対応した部分が表示されます。ツールバーのボタン   を使っても同じことが出来ます。

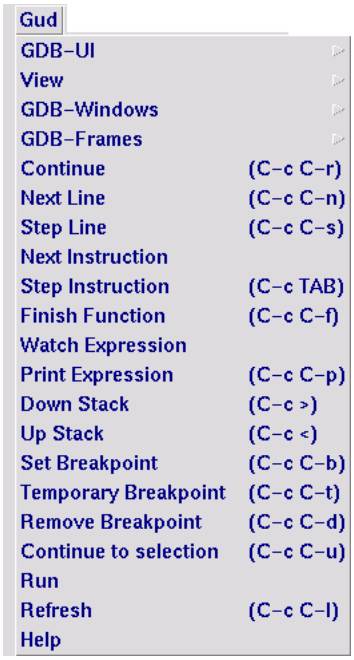
あるいは、このスタック窓にカーソルがあれば、そこで改行をすると、それだけで、対応するソースが中央の窓に表示されます。

Gud のメニューバー (1)

Gud という名前は、少し前に脚注でも紹介しましたが、Grand Unified Debugger の意味です。gdb 以外のデバッガも使えるので、そのような名前になっています。さて、図 36 などのようにカーソルが左上の窓 (*gud*) にある時に、画面の上のメニューバーを見ると、図 37 のように項目が表示されています。このうちの一番左の Gud にはその下の図 38 のような選択があります。この中を順に少し見てみます。

図 37 Gud のメニューバー (の一部)

図 38 メニューバー Gud の項目



メニューバー → Gud → GDB-UI

図 38 の一つ目は GDB-UI, User Interface です。

この中には次の二つの副メニューがあります。

- Display other windows
  これを選ぶと、五分割にするか二分割にするかを反転します。四角のボタンが図のように押されていると五分割になります。この Display other windows と書いてある字の上にカーソルを持って行くと、全体が少し前に出ますが、そこでクリックすると、メニューが消えて、もう一度見に行くと、四角のボタンの状態が反転しています。
- Restore window layout
  これは M-x gdb-restore-window RET と同じで、画面を初期値に戻します。戻した時に二分割になるか

五分割になるかは、上の Display other Windows のボタンの状態によります。

メニューバー → Gud → View(ソースか機械語か)

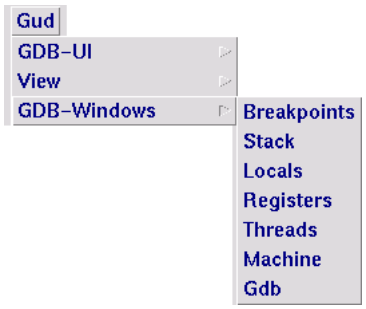
図 38 の二番目の View には Source と Machine の

選択があります。ここではこの二つのうちどちらかを選んで、それを中央の窓に表示します。

- Source (gdb-view-source-function)
  直前に何かソースを表示していればそれを、もし表示していなければプログラム入口のファイルを表示します。
- Machine (gdb-view-assembler)
  機械語のバッファを表示します。実行形式を逆アセンブルして表示します。

メニューバー → Gud → GDB-Windows,Frames

図 38 の 3,4 番目の GDB-Windows と GDB-Frames は良く似ていて、それぞれ、新しい窓や、枠を開けます。GDB-Windows の方は中央の窓を使います。GDB-Frames の方は、新たに枠を開けます。開けるバッファの種類は表 8 に示しましたが、枠も窓の場合も同じです。

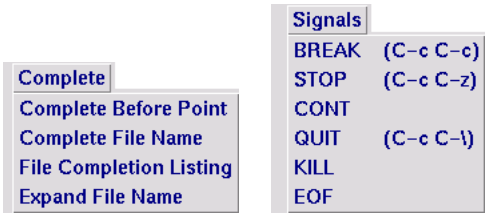


これらの中で、初めの三種類と、最後の Gdb は五分割窓なら、初めから開いています。中央の三種類はこのメニューから開けることになります。これらについては、後の「その他のバッファ」で少しだけ説明を追加します。

表 8 GDB-Windows,Frames で開けられる窓・枠の種類

Breakpoints	ブレークポイント	
Stack	スタック枠	(開始画面)
Locals	変数	
Registers	レジスタ	
Threads	スレッド	(メニューから開ける)
Machine	機械語	
Gdb	(gdb) 操作待	(開始画面)

図 39 補完とシグナルのメニュー



Gud のメニュー (図 38 の残り)

図 38 の Gud のメニューの五番目の Continue 以下の残りの項目は、前半の 21.3 版で説明した機能と同じです。

このメニューですが、カーソルが Gud バッファ以外にある時には、キーの割当のプリフィックスが C-c から C-x C-a に変っていますが、どちらも使えます。

最後から四つ目の Continue to selection という項目は u, until と同じものです。

Gud のメニューバー (2)

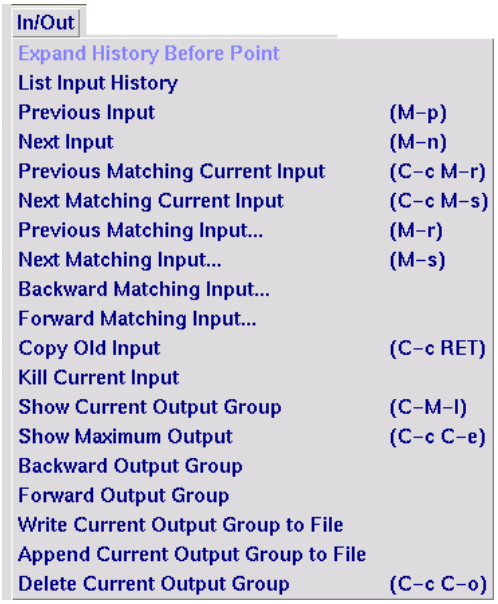
カーソルが左上の *gud* バッファにある時には、画面上部のメニューバーには次の選択があります。このうち

の最初の Gud は上で説明したので、残りを (前後しますが) Complete Signals In/Out の順に見ます。

補完 (Complete)

図 39 の左側は (gdb) の入力待での操作の補完です。それぞれ上から、カーソル左側の補完、ファイル名の補完、ファイル名の補完候補の一覧を表示、ファイル名の展開をしてくれます。

図 40 In/Out のメニュー



シグナル (Signals)

図 39 の右側にあるように、いくつかの種類のシグナルを対象プログラムに送ります。例えば STOP で止めて CONT で継続のようにも使えます。

入出力 (In/Out)

Gud ツールバーの三番目の In/Out のメニューを図 40 に示しています。入出力というのは、*gud* バッファでの (gdb) の入力待で何かを**入力**する時に助けてくれる機能と、表示された**出力**を操作するという意味です。

入力の方は、一番良く使う M-p, M-n (直前に入力した操作を表示) などを初めとして、シェルモードの時の履歴の機能と同じものなどがメニューに見えています。

出力については、例えば中央付近やや下の C-c C-e や、最後の C-c C-o の表示削除は前に表 3 で説明しました。

メニューの下の方に Group という字がありますが、これは「一回ごとのシェルの操作の表示」という意味です。

watch と speedbar(変数の値を監視)

ツールバー (表 6) の watch のアイコン を使うか、図 37 の 11 番目にある Watch Expression を使って変数を監視するように設定をすると、speedbar の形式の枠[17]が開きます。図 41 の左側に開いているのが speedbar の枠です。これは中央の窓で、変数名 i にカーソルを置いて、 をクリック、同様に argc と argv にカーソルを置いてクリックして開いた画面です。

図 41 Watch expression → speedbar 画面

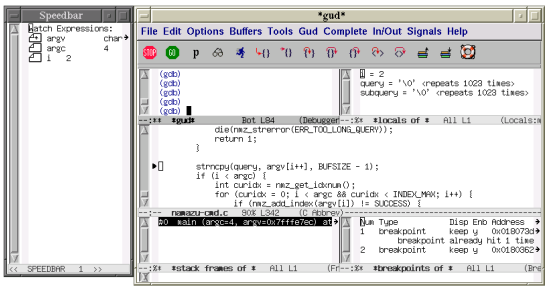
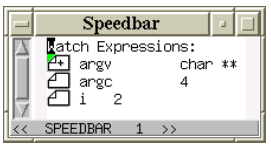
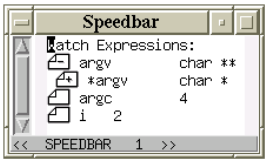
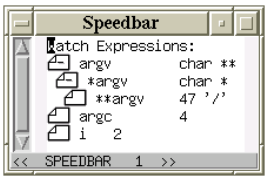


表 9 speedbar で変数の構造を表示

	argv の左側の には + の字があります。ここにマウスを持っていくと、角の色が変ります。
	その を mouse-2 で一回クリックすると *argv という行が増えます (少し時間がかかります)。
	さらにもう一度 *argv の左側の をクリックすると、**argv の行が増えます。

もし対象が構造体やポインタの場合、それを木構造で表示します。表 9 に例を示します。木構造になっている場合には + の印があるので解ります。mouse-2(中ボタン) を使うと、それを開けたり閉じたり出来ます。表 9 の例だと argv がそれです。

構造体の入口にカーソルがある場合には、D で表示を消せます。単純な変数を表示している場合には、改行か

17 Emacs の通常の操作で、M-x speedbar RET と入力すると、ファイルの一覧を表示して操作出来る枠が開きます。それと同じ形です

図 42 gdb 画面構成 (6 分割)

操作卓 (console)	変数 (locals)
ソース画面	**I/O バッファ**
スタック	ブレークポイント

mouse-2 を入力すると次のようにミニバッファで聞

```
New Value:
```

いてきます。ここで新しい値を入力すると、それが変数の値に反映されます。

変数の値が変更されていたかどうかを font-lock-warning-face の色で示してくれます。これは変数 gdb-show-changed-value が nil 以外に設定されている場合 (初期値) です。

さらに speedbar の表示を **関数::変数名** の形式にも出来ます。C の場合に限るのですが、gdb-use-colon-colon-notation を nil 以外に設定します。ただし、複合文になっている場合、うまく働かないので、この変数の初期値は nil になっています。

その他の窓 (Buffer)

ブレークポイント窓、スタック窓については説明しましたが、その他にも表示出来る窓 (Buffer) の種類がいくつかあります。

入出力窓 (Input/Output Buffer)

変数 gdb-use-inferior-io-buffer (初期値 nil) を nil 以外に設定すると、デバッグ対象プログラムの入出力と gdb とのやりとりを分けられます。その時に入出力に使うのがこのバッファです。gdb の応答とプログラムの表示が混らずに、見やすくなります。その場合には、画面の構成は図 42 のようになります。

このような画面にするには ~/.emacs に次のように書いておきます。

```
(setq gdb-many-windows t)
(setq gdb-use-inferior-io-buffer t)
```

このバッファでは、シェルモードと同じ C-c C-o なども使えます。

変数窓 (Locals Buffer)

変数を表示します。単純な変数の場合には値を表示しますが、構造体の場合には、その構造だけを表示します。それらの値を表示するには speedbar の窓を使うのは、前に示しました。

レジスタ窓 (Registers Buffer)

CPU の中のレジスタの値を表示します。図 45 のような表示になります。表示方法などは少し先の項でみます。

アセンブラ窓 (Assembler Buffer)

スタックで示されている部分の機械語を逆アセンブルして表示します。いまどこで停止しているかを表示したり、ブレークポイントの設定・解除が可能です。ソースを表示している場合と同じように外縁にブレークポイントの印を示します。ソースのないライブラリの中に入ってしまった場合や、si を使って、機械語のステップ動作をすると、その表示になります。あるいは、この窓を表示するもう一つの方法はツールバーの
GUD → GDB-Windows あるいは GDB-Frames
を使います。

アセンブラ記述を用意しておくとソース窓に表示

次のような hello.c を用意します (今回は /home/username/c/ の下に作ったと仮定しています)。

```
#include "stdio.h"
main (){
 printf( "Hello World\n");
}
```

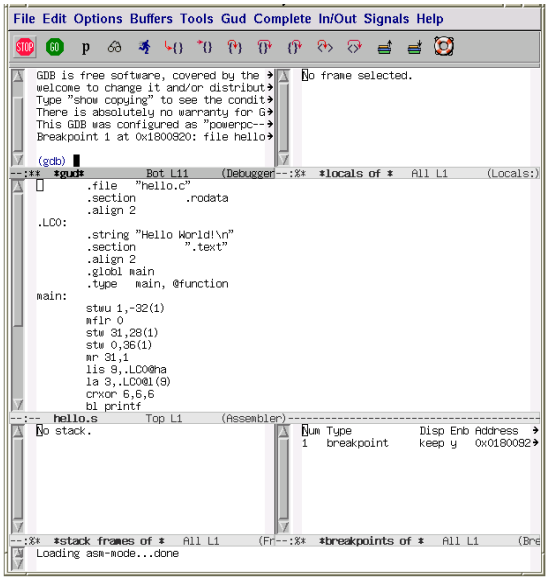
hello.c から次のようにして hello.s と hello を作ります。-s を付けるとアセンブラ記述 (hello.s) を作って、残します。

```
$ cc -s hello.c
$ cc -g -o hello hello.s
```

hello.c が置いてあるディレクトリの.gdbinit に次の四行だけを書いておきます。cd の先は自分の都合に合せて、つまり hello.c などが置いてある位置に変更して下さい。

```
cd /home/username/c
file hello
```

図 43 ソース窓にアセンブラ（hello.s で起動したところ。中央）




```
exec hello
b main
```

前に出て来たものと同じですが、~/.emacs に次のような設定をしておきます。

```
(setq gdb-many-windows t)
```

Emacs の中で gdb を起動します。

```
M-x gdb RET
Run gdb (like this): gdb --annotate=3
```

これで起動画面は図 43 のようになります。中央にアセンブラのソースが表示されています。ただし、この図は PowerPC の場合のものです。

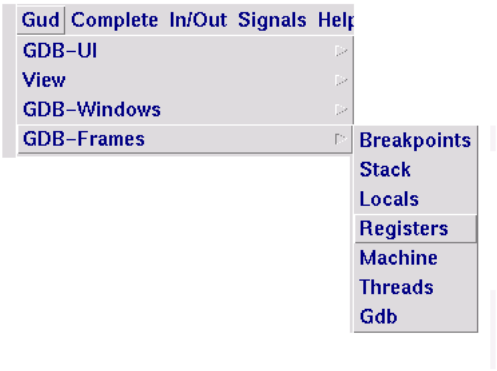
レジスタ窓を表示するには

ツールバーのメニューから図 44 のように Gud → GDB-Frames → Registers を選びます。
これで枠が一つ開きます。ただし、最初は次のような字だけが表示されます。

```
The program has no register now.
```

(gdb) で r (改行) して実行します。すると、その枠に図 45 のようなレジスタ一覧が表示されます。枠の大きさは調整する必要があるかも知れません。また CPU によって内容は違います。この図では、左から、レジスタ名、その内容、その内容が指している番地の内容となっています。

図 44 メニューからレジスタ窓を開ける



この中の下から 8 番目にある $pc はプログラムカウンタ、つまり実行する命令のアドレスを指しているレジスタです。

(gdb) で si と入力をするか、ツールバーの右から五番目の  をクリックすると、一命令づつ実行します。その時に、レジスタの内容が変化して行きます。

Thread バッファ

対象プログラムのスレッドを表示します。カーソルを持っていって**改行**を入力すると、そのスレッドを**現在のスレッド**として、ソースの窓にも、それに対応する部分を表示します。Mouse-2 でも同じように操作出来ます。

エラーメッセージ・問題など

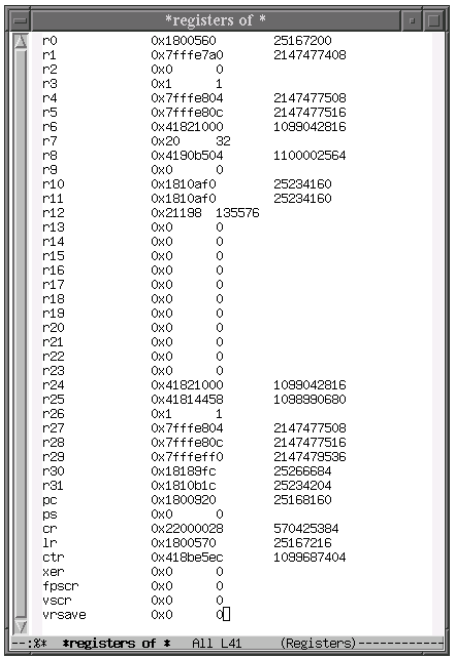
終了したはずなのに already running under gdb

emacs から gdb を起動して、一度 (gdb) の窓で quit で終了した後に M-x gdb とやると、次のような字が表示されるかも知れません。

```
This program is already running under gdb
```

もう gdb は終了しているはずなのに、と思われますが、バッファが残っていると、まだ実行中と認識してしまいます。このような時には *gud で始まる名前のバッファにカーソルを持っていき、C-x k (kill-buffer) します。その後に M-x gdb すれば大丈夫のはずです。

図 45 レジスタ窓 (Registers), PowerPC の場合



ツールバーからアイコンが消えてしまった

21.3.50 の場合ですが、ファイルを探したりしていて *gud-hoge* のバッファを表示から消してしまったときには、ツールバーのアイコンが消えてしまいます。そのような時には*gud で始まる名前のバッファを表示すれば、出てきます。具体的には

```
C-x b *gud [Tab] RET
```

とします。これで [Tab] での補完が効かない時には、本当に消えてしまったのでしょう。そのような時には M-x gdb から開始します。

gdb 一般の話を少し

Emacs から使う gdb に限った話ではなくて、gdb 一般に共通なことを少し紹介します。

.gdbinit の例は emacs/src/.gdbinit にも

今回の 21.3.50 を組立てるために emacs を cvs から持ってきているとすると、その中の emacs/src/.gdbinit にかなり色々書いたもの (400 行以上) が置いてあります。そういうものもあるのだということでちらっと見て下さい。

分散しているソースを見る時に必要な dir

Mozilla とか XFree86 など大きなソースを見ようとすると、*.c, *.h などが分散しているので少し工夫が必要です。そのような場合には、例えば、dir を使って、どこにあるかを教えます。

```
(gdb) dir directory_name
```

とすればいいのですが、いくつも手で加えるのは大変です。そこで、一例ですが、次のようにして一覧を作ります。

```
$ find . -name \*.c -or -name \*.h | \
  perl -nle 's/\/[^\/]*.[ch]$//;    \
            print "dir $_ "   ;' | \
  sort -u > ! /tmp/list
```

これらの各行は次のようになっています。

1 *.c と *.h を全て探して
2 それらのあるディレクトリの文字列だけを抜出して
3 前に dir という文字を付けて
4 sort -u で重複を省いて保存

そうして、この出来た /tmp/list を .gdbinit の中に取込むというのも一つの方法です。

-g を付ける方法

gdb を使うには -g を付ける必要があります。オープンソース・ソフトウェアでは、make する時には多くの場合 -g を付けてあります。そうして make install する時に strip か install -s を使って、そのシンボル表を消すようにしています。ですから、そのような場合なら、make したままの状態で使えば、-g の心配は要らないことになります。

あるいは、そうなっていない大きなソフトウェアですと、どこに -g を書けば効果があるのか分るのに時間がかかる場合があります。

一例として XFree86 の場合を挙げます。

XFree86 の場合: Imakefile の 9 行目にある

```
#define PassCDebugFlags ⇒
        CDEBUGFLAGS="$(CDEBUGFLAGS)"
```

の行の最後に、次のように -g を付けます。

表 10 Gud で使える様々なデバッガ

名前	言語	その他
gdb	*	GNU debugger
dbx	*	Solaris 2
xdb	*	HPUX
sdb	*	SVR4 (除 Solaris 2)
perldb	Perl	Gud から perl -d で起動
jdb	Java	
pdb	Python	
bashdb	shell	

(*) は C, C++, Fortran, Pascal

```
#define PassCDebugFlags ⇒
          CDEBUGFLAGS="$(CDEBUGFLAGS) -g"
```

gdb 以外のデバッガを使う

ここまでの話は Gud (Grand Unified Debugger) から gdb を起動する場合でした。そこで、その他のデバッガについても少しだけ紹介します。Gud では、表 10 のようなものが使えます。とは言っても始めの四つは似たようなもので、OS 毎に利用出来るデバッガが違っているのに対応しています。残りの三つは言語毎のものです。これらの 起動は表 10 の中の左端の欄にある「名前」を使って、全て次のようにして起動します。

```
M-x 名前 RET
```

Perl デバッガ perldb

表 10 の中で perldb を少し試して見ます。ここからの話は 21.3 でも 21.3.50 でも同じです。

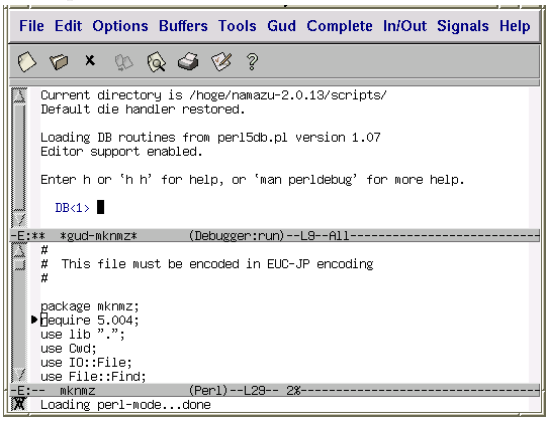
例題としては前半に使った namazu-2.0.13-1 の mknmz を使います。mknmz は既に索引が作ってあると何もしてくれないので、索引を作る先をもう一つ (index2) 用意します。また emacs 自体の起動に環境変数を付けておきます。以下は図 17 の続きです。

```
$ cd /hoge/namazu-2.0.13
$ mkdir index2
$ env pkgdatadir=‘pwd‘ LANG=ja_JP.eucJP emacs
```

これで emacs が起動したら、次のように perldb を起動します。

```
M-x perldb RET
```

図 46 perldb を起動した画面にはソースを表示



これで、次のように聞いてきます。

```
Run perldb (like this): perl -e 0
```

次のように直します。

```
Run perldb (like this): perl -d scripts/mknmz ⇒
                                -O ../index2 ../doc
```

perldb は起動すると、mknmz が置いてある scripts のディレクトリに cd するので、引数の指定は、そこから見た位置にしておきます。

perldb を起動した画面にはソースを表示

先ほどの続きで、改行を入力すると図 46 のような画面になります。モード行を見れば、mknmz の 29 行目を表示しているということが分ります。perldb の場合は gdb と違って run しなくても画面を表示します。r は run でなくて return の意味になっています。

mknmz の中には subroutine main という行があるので、例えば、次のように main というサブルーチンにブレークポイントを設定します。c を入力すると、その

```
DB<1> b main
DB<2> c
```

main まで行けます。その時の画面が図 47 です。

この後は、n(次の行) s(次の行、サブルーチン内に入る) を使って動作を追って行けます。次のように namazu_core にブレークポイントを設定しておいて c の後に止った画面が図 49 です。

図 47 main で止る

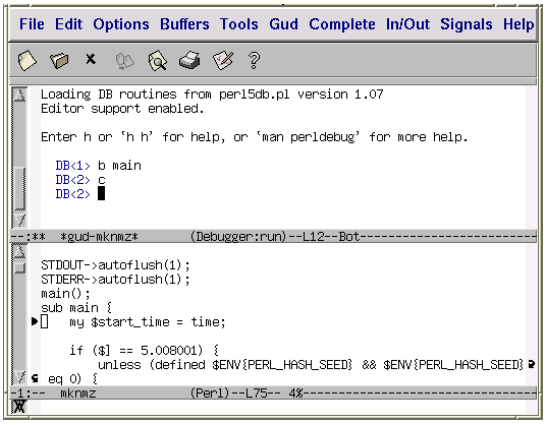


表 11 T で表示する時の呼出条件 (戻り型)

$	スカラー
@	リスト
.	戻り値を利用しない (void)

```
DB<2> b namazu_core
DB<3> c
```

スタックの状態を表示するのは T です。図 49 で止っている時に T を入力した時の表示を図 48 に示します。この中の $ @ . などの意味は表 11 に示しました。

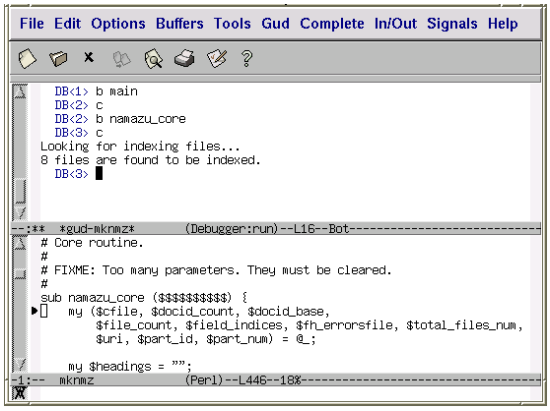
変数の値は次のようにして表示出来ます。

```
DB<3> p $variable
```

DB <n>の入力待で Perl 文と perldb 語

この DB <n> で入力待になっている時には perl の文もそのまま入力出来ます。そのため、ここでの操作は (perl では使われていない) n,s,r などの一文字だけに

図 49 namazu_core で止っている



なっています。表 12 に良く使う操作をまとめておきました。これらはシェルで次のように入力した場合と共通です。

```
$ perl -d perl_script_name
```

perl の文を直接入力する例として、次のような方法で、環境変数 PWD の値を調べられます。

```
DB<4> print $ENV{'PWD'}
/hoge/namazu-2.0.13/scripts
```

DB <4> のように表示されている番号は履歴番号なので、上のような操作をした後なら、!4 とすると、その番号で使った操作をもう一度繰返せます。

以上説明していることは Emacs の perldb モードに限った話ではなく、シェルから起動した場合も同じです。Emacs らしいことと言えば停止したソースを表示していること、C-x 空白でブレークポイントを設定出来るこ

図 48 T でスタックの状態 (呼出関係) を表示

```
  DB<3> T
$ = mknmz::namazu_core('/hoge/namazu-2.0.13/scripts/../doc/en/manual.html',
   0, 0, 0, ref(HASH), ref(IO::File), 8, undef, undef, undef)
   called from file `/hoge/namazu-2.0.13/scripts/mknmz' line 213

@ = mknmz::process_file('/hoge/namazu-2.0.13/scripts/../doc/en/manual.html',
   0, 0, 0, ref(HASH), ref(IO::File), 8)
   called from file `/hoge/namazu-2.0.13/scripts/mknmz' line 131

. = mknmz::main() called from file `/hoge/namazu-2.0.13/scripts/mknmz' line 73
  DB<3>
```

(形式は次の通りです)

```
戻り型 = サブルーチン名 (引数) called from file `ファイル名' line 番号
```

表 12 perldb の操作 (一部)

s	一文実行
n	次の文を実行 (サブルーチンには入らない)
改行	直前の n または s を繰返す
R	最初から実行しなおし
r	現在のサブルーチンから抜けるまで実行
c	実行を続ける
t	trace モードを反転
b	行番号でブレークポイントを設定
L	ブレークポイントを表示
T	スタックの backtrace を表示
p 式	式の値を表示
S [正規表現]	(正規表現に一致する) サブルーチンを表示
q	終了
h	英語での説明を表示 (130 行くらい)

とくらいです。C-c C-o は使えますが、相変らず便利だと思います。

```
  DB<3> h
*** output flushed ***
  DB<3>
```

t の 「trace モードを反転」を入力してから実行すると、一行毎の実行の様子を表示するようになります。シェルの -xv のようなものです。もう一度 t とすると、trace モードが off になります。

初期設定は ~/.perldb に書ける

初期設定を ~/.perldb に書くことが出来ます。ここにも perldb 語と perl 文の両方が書けるので、ここでの例に使った設定なら、次のように書いておくことも出来ます。対話的な入力と違うのは、行末のセミコロンが必要なことです。

```
$ENV{'pkgdatadir'} = "/hoge/namazu-2.0.13/";
$ENV{'LANG'}       = "ja_JP.eucJP";
b load "/hoge/namazu-2.0.13/scripts/mknmz";
```

英文での説明は man perldebug

perldb の使い方については、シェルで次のようにす

```
$ man perldebug
```

ると英語ですが 800 行くらいの説明が表示されます。それを見れば、ここでの説明は本当にその一部だけということが分ると思います。

図 51 gdb make と入力 ?

```
$ cd /usr/local/src/emacs
$ gdb -n -q make
(no debugging symbols found)...Using host  ⇒
                      libthread_db library ⇒
              "/lib/tls/libthread_db.so.1".
(gdb) info exec
Undefined info command: "exec".  ⇒
                                 Try "help info".
(gdb) info files
Symbols from "/usr/bin/make".
Local exec file:
 `/usr/bin/make', file type elf32-i386.
  (以下略)
```

この中では、-n は .gdbinit を読まない、-q は著作権などの表示をしない指定です。

【付録】 Segmentation Fault を調べる

CVS 版 Emacs と Linux の新カーネル

cvs 版の Emacs を Linux の新しいカーネル、例えば Fedora Core 1 などで make bootstrap すると、途中で Segmentation Fault と言って止ってしまいます。これを回避する方法は etc/PROBLEMS に書いてあるのですが、もし自分でどの辺が問題かを探すとしたらどのようにするか、実は **Emacs/gdb を使うと、とても簡単に**問題点を絞れる、という話を紹介します。

cvs から取って来て Seg-fault まで

gdb は 21.3 と次の版でかなり違うので、では、と 21.3.50 を作ってみようとする時の話です。しかし 新しい kernel を使っている Linux の機械では Exec-shield という機能が用意されています。これが有効になっている場合に emacs の make は少し問題があります。図 50 のように cvs co で取って来て、./configure の後 make bootstrap すると、その図の下から四行目の (2) のように Segmentation fault してしまいます。

gdb を使って調べようかと思いますが、最後に入力したのは make という字なので gdb make するのかなと、図 51 のようにしてみます。しかし、このように gdb make としたのでは、うまく行きません。図 51 の最後で info files として、その表示の最初にあるように、これでは make そのものをデバッグしようとしてしまいます。

図 50 cvs から取って来て Segmentation fault まで (Fedora Core など)

```
        $ export CVS_RSH=ssh
        $ cvs -d :ext:anoncvs@savannah.gnu.org:/cvsroot/emacs co emacs
           (略)
        $ cd emacs
        $ ./configure
           (略)
        $ make bootstrap
           (略)
        Finding pointers to doc strings...done
(1) →   Dumping under names emacs and emacs-21.3.50
(2) →   make[1]: *** [bootstrap-emacs] Segmentation fault
        make[1]: Leaving directory `/usr/local/src/emacs/src'
        make: *** [bootstrap] Error 2
        $
```

図 52 temacs を手で起動してみると

```
$ cd src
$ ./temacs --batch --load loadup bootstrap
  (略)
Finding pointers to doc strings...done
Dumping under names emacs and emacs-21.3.50
Segmentation fault
$
```

temacs 実行中に Segmentation Fault ?

そこで、何を実行しようとした時に Segmentation Fault したのかが必要になります。幸い、図 50 (2) の `Segmentation Fault` と表示されている一行前に `Dumping under names emacs and emacs-21.3.50` という字が見えます (1)。

emacs の make は、まず素の temacs を作ります。次にその temacs を起動して良く使う elisp を全て読込みます。そうして、その読込んだ状態のメモリイメージを emacs という名前で書き出します (undump)。普段の実行時には、この undump 形式を使うことで、起動時に、いつも使う elisp を読んで評価 (load) する時間を節約しています。

このことを知っていれば、この Seg-fault は、temacs が出来ていて、それから emacs を作っている途中で落ちたのでは ? という推測がつきます。

そこで、ではその dump の具体的な方法はどうなっているのかなと、 src/Makefile の中を見ると、265 行目付近に次のような行があります。

```
./temacs --batch --load loadup bootstrap
```

で、試しに、シェルからこれをそのまま入れて見ます。すると、図 52 のようにやはり同じように落ちるので、これを Emacs/gdb で調べればいいなということが分ります。

Emacs/gdb 準備 (.gdbinit と ~/.emacs)

何度も試すことを考えて、図 52 で temacs の起動に使った引数を設定しておきます。それには、 set args を使います。次のようにして、実は既にある emacs/src/.gdbinit に一行加えます。

```
$ cd emacs/src
$ echo  \
  'set args  --batch --load loadup bootstrap' \
  >> .gdbinit
```

図 52 で試した文字列に比べて./temacs という字が消してあること、両端に ' があること、>> を使って、「最後に追加」の指定をしていること、等に注意して下さい。

それと、新しい gdb の機能が使えるように、次のような 三行だけの ~/.emacs-gdb (名前は何で構いません。次で指定します) を作っておきます。

```
(global-font-lock-mode t)
(setq gdb-many-windows t)
(setq gdb-use-inferior-io-buffer t)
```

別に作ってある emacs 21.3.50 を起動

さて、いよいよ emacs から gdb を起動します。ここで使う emacs は**実は今作ろうとしている** 21.3.50 です。その版の方がずうっと素適・効率的だからです。もし Fedora Core 等で試す時には、図 53 のようにして、setarch i386 を使って、動く emacs を一度作っておいて下さい。

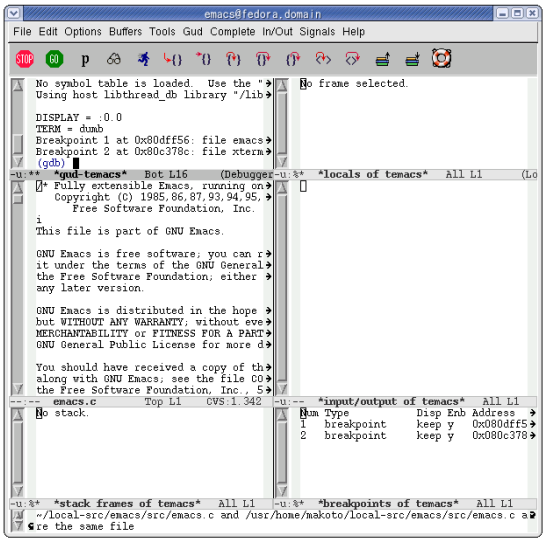
さて、次のように、先ほど用意した ~/.emacs-gdb を指定して起動します。

図 53 別途、予め 21.3.50 が使えるようにしておく

```
$ cd /usr/local/src
$ cvs -d :..(略)...  co emacs
$ cd emacs
$ setarch i386 ./configure
$ setarch i386 make bootstrap
$ sudo make install
$ make clean
```

上の作業で/usr/local/bin に入ります。$PATH には /usr/bin が先に来るようにしておきます。21.3.50 を起動するには明示的に /usr/local/bin/emacs と入力して起動します。

図 54 gdb を起動したところ




```
$ cd src
$ /usr/local/bin/emacs -q -l ~/.emacs-gdb -f gdb
```

ここで引数の意味は次の通りです。

-q	~/.emacs を読みません
-l ~/.emacs-gdb	設定はこの名前のファイルを読みます (load します)
-f gdb	今回は最初に M-x gdb としたいのですが、その代りに初めから gdb 関数を起動します

これで次のように聞いてくるので、改行を入力します。

```
Run gdb (like this): gdb --annotate=3 temacs
```

すると、画面が、図 54 のようになります。右下にはブレークポイントが設定されていますが、これは実は emacs に含まれている .gdbinit に設定されていたもので、今回自分で設定したものではありません。そうして、この図 54 の左上の (gdb) の窓で

```
(gdb) run
```

図 55 run と入力した後 Segmentation Fault で停止

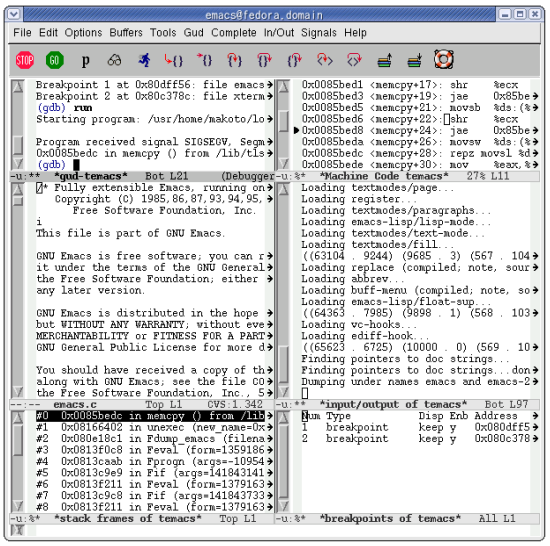


表 13 画面構成

(gdb) 窓	機械語窓 (逆アセンブル)
ソース窓 (emacs.c)	I/O buffer (Loading..)
スタック窓	ブレークポイント窓

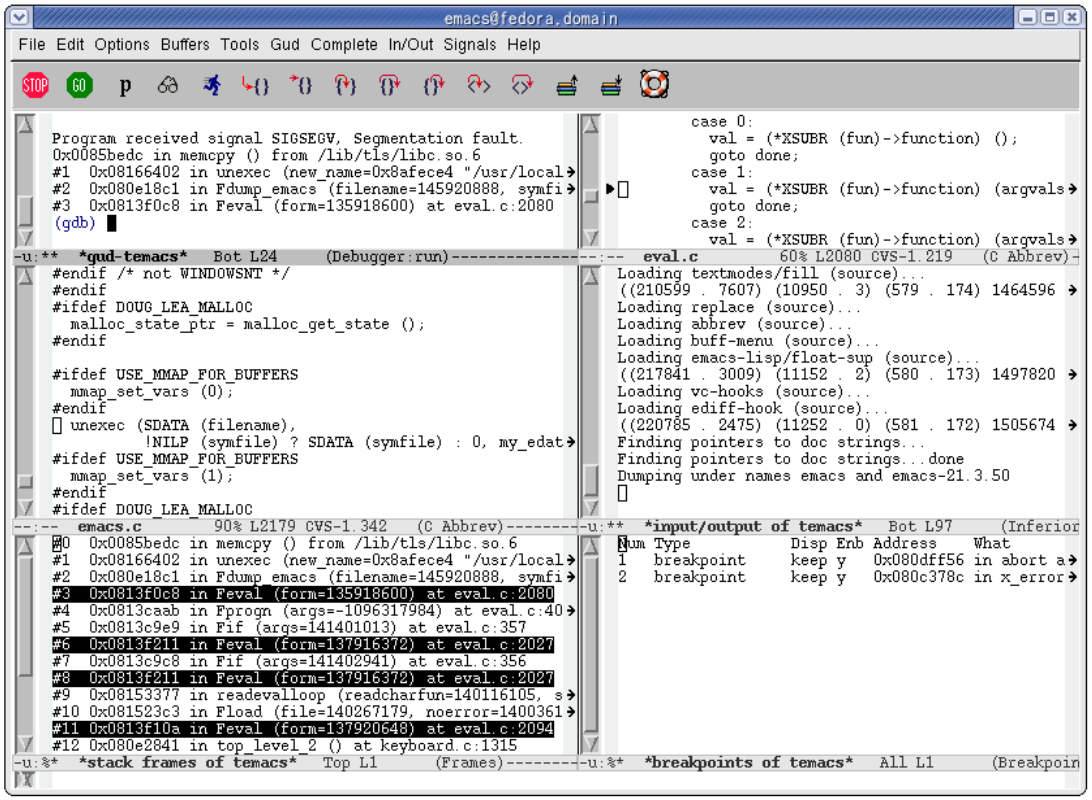
と入れてから改行すると実行が始まり、図 55 のように右中央に temacs からの出力が表示されます。そうして、左上の窓を見ると、SIGSEV の信号を受け取ったと言って止まっています。この時の画面の構成は表 13 のようになっています機械語窓には、止った memcpy のある付近を逆アセンブルして表示し、その中の止ったアドレスを三角矢印で指して[18]います。

Segmentation Fault を gdb で調べるときに良く使う方法は where, bt, backtrace などを入力して、停止した時のスタックの情報を表示するものです。それと同じものが図 55 の左下の スタック窓に表示されています (少し大きい字のものも後で図 56 で示します)。

これの意味を読みとるのは、普通なら少し時間がかかると思いますが、**ここからが emacs/gdb のうれしい部分**です。

18 良く見ると、この場合は二行ずれています

図 56 eval.c の場合



スタック窓の反転行を上下する

図 55 は、どこで止ったかを表示していますが、右下のスタック窓を使うと、全体のプログラムの中のどこまで進んで停止しているのかを見ることが出来ます。この窓に表示されているのは bt などで表示するもの同じで、下の方が呼出元です。上に行くにしたがって子供の呼出しになります。この窓で反転表示されている行が、ソース表示されているという意味です。この反転行を上下すると、それに対応してソースの該当行を画面に表示してくれます。例えば、図 55 の左下窓の #0 が反転しています。スタックの表示を上げる[19]と、この反転行が一つ下がります。それにはいくつか方法があります。その方法を表 14 に示します。

19 「子として呼ばれる方を下」と表現するとします。その時に「スタックを上げる」というよりは「呼出関係を上って行く」(戻って行く) と考える方が、正しいと思うのですがスタックの表示は上下逆になっています。そのため、このように表現しています

表 14 スタックの表示を上げる方法 (どれか一つ)

1	をクリック、これで、スタック表示 (画面内容) が上に上ります。注目点 (反転行) の方は下がります
2	カーソルを左下の窓の #1 の行に持っていって、改行
3	左上の (gdb) の窓で up と入力
4	C-c < と入力

図 57 表 14 のどれかを使ってスタックを上げた時

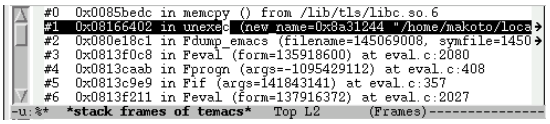
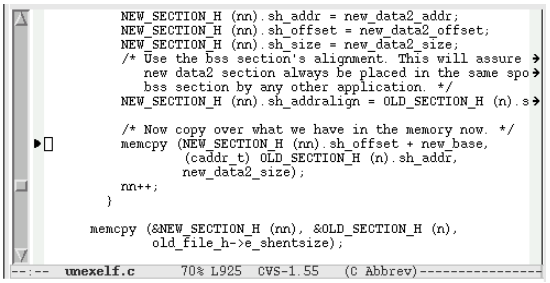


表 14 の 1 2 3 4 のうちのどれかをすると、スタック窓が図 57 のようになります。この時に、左中央のソース窓の方は、図 58 のように、それに相当する unexelf.c の 925 行目を表示します。

このまま表 14 にある操作のどれかをさらに二回すると、図 56 のように反転行がスタック窓の #3 に移ります。その時のソースは eval.c なのですが、これは #6 #8

図 58 ソース窓では unexelf.c: の 925 行目を表示



も共通なので、それらもまとめて反転表示されます。この時には、右上の窓が空いていたので、ソースは、その窓に表示されています。

このようにして、スタック窓を使って見るソースを指定を上下して、どのような文脈 (呼出関係) で問題が起きているかを調べることが出来ます。実はこのスタックを上下する時に、うっ、という感じで時間がかかります (1,2 秒のことです)。しかし、この時間が、ああ探してくれているんだなぁという、かえって有難さを感じさせます。

Emacs/gdb を使うと、停止位置と呼出関係が分る

今回の問題は、memcpy を呼出す時の引数の指定が適切でないため、呼ばれた方で、コピーをする時に不正なアドレスを指定して Segmentation Fault が起きているということまでは推測できました。

まとめ

- Emacs には Grand Unified Debugger (Gud) というデバッガを起動する道具 (Elisp) がある
- Gud は gdb などいくつかのデバッガが使える
- 基本的な機能は実行中のソースを表示してくれること
- gdb に限っては 21.4 版から複数分割の窓が開いて便利に使える
- 今回は 21.3 と 21.4 版の gdb、 それに perldb の機能の一部を紹介した
- 付録として Linux の新しい kernel で 21.3.50 の temacs が Segmentation Fault するが、その箇所をわかりやすく調べる方法を紹介した

おわりに

駆足で Gud と新しい Emacs/gdb の紹介をしてきました。雰囲気はおわかりいただけたでしょうか。あるいは実際に動かしてみて試していただけたでしょうか。

Gud/gdb などを使うと、いわゆるデバッグをするだけではなく、**実際に動かして**、どこに飛ぶのか、どこに戻るのか、変数の値はどうなっているのか等を実際に**ソースを見ながら**、調べることが出来ます。

ソースを見る時には、中に書いてあるコメントが役に立ちます。ここに紹介した方法を使うと、実行する順にそのコメントを読んでいけるので、それだけでも理解が一層深まります。**ソースが公開されているソフトウェアは本当に素晴しい**と実感されるのではないでしょうか？

(gdb) 参考文献・URL

- 中屋鋪恭子,
  プログラマーのためのデバッガの基礎知識,
  UNIX MAGAZINE 2004.4 (p31-53)
- Richard M.Stallman and Roland H.Pesche 著,
  (株) コスモプラネット訳,
  GDB デバッギング入門, アスキー出版, 1999
  (GDB 4.17 対応なので少し古いですが書籍としては貴重です)
- GDB-5 説明書 日本語訳
  http://www.asahi-net.or.jp/~wg5k-ickw/ ⇒
  html/online/gdb-5.0/gdb-ja_toc.html
- 工藤智行著,
  UNIX プログラミングの道具箱,
  Part 6 デバッグ, p215-271
  技術評論社, 2004/07
- 藤原 誠著,
  便利なツール Emacs らくらく入門
  技術評論社, 2004/07

(ふじわら・まこと (株) 絹)