особенности национальной отладки в UNIX

крис касперски ака мыщъх, no email

...отладка подобна охоте или рыбной ловле: те же эмоции, страсть и азарт. долгое сидение в засаде в конце концов вознаграждается. очередной невидимой миру победой....

Евгений Коцюба

первое знакомство в GDB (что-то вроде debug.com для MS-DOS только мощнее) вызывает у поклонников Windows смесь разочарования с отвращением, а увесистая документация вгоняет в глубокое уныние, граничащее с суицидом. отовсюду торчат рычаги управления, но нету газа и руля. не хватает только каменных топоров и звериных шкур. как юниксоиды ухитряются выжить в агрессивной среде этого первобытного мира – загадка.

введение – отладка в исторической перспективе

Несколько строчек исходного кода UNIX'а еще помнят те древние времена, когда ничего похожего на интерактивную отладку не существовало и единственным средством борьбы с ошибками был аварийный дамп памяти. Программистам приходилось месяцами (!) ползать по вороху распечаток, собирая рассыпавшийся код в стройную картину. Чуть позже появилась отладочная печать – операторы вывода понатыканные в ключевых местах и распечатывающие содержимое важнейших переменных. Если происходит сбой, простыня распечаток (в просторечии – "портянка") позволяет установить чем занималась программа до этого и кто ее так.

Отладочная печать сохранила свою актуальность и по сегодняшний день. В мире Windows она в основном используется лишь в отладочных версиях программы (см. листинг 1), и убирается из финальной (см. листинг 2), что не есть хорошо: когда у конечных пользователей происходит сбой, в руках остается лишь аварийный дамп, на котором далеко не уедешь. Согласен, отладочная печать кушает ресурсы и отнимает время. Вот почему в UNIX так много систем управления протоколированием – от стандартного `syslog`, до продвинутого Enterprise Event Logging'ra (http://evlog.sourceforge.net/). Они сокращают накладные расходы на вывод и журналирование, значительно увеличивая скорость выполнения программы.

Отладочная печать на 80% устраняет потребности в отладке, ведь отладчик используется в основном для того, чтобы определить как ведет себя программа в данном конкретном месте: выполняется ли условной переход или нет, что возвращает функция, какие значения содержатся в переменных и т. д. Просто влепите сюда `fprintf/syslog` и посмотрите на результат!

```
#ifdef __DEBUG__
      fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
#endif
```

Листинг 1 неправильный пример использования отладочной печати

```
if (__DEBUG__)
      fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
```

Листинг 2 правильный пример использования отладочной печати

Человек – не слуга компьютера! Это компьютер придуман для автоматизации человеческой деятельности (в мире Windows – наоборот!), поэтому UNIX "механизирует" поиск ошибок настолько, насколько это только возможно. Включите максимальный режим предупреждений компилятора или возьмите автономные верификаторы кода (самый известный из которых – LINT) и баги побегут из программы как мыщъх'и с тонущего корабля. (Windows-компиляторы тоже могут генерировать сообщения об ошибках, по строгости не уступающие gcc, но большинство программистов пропускает их мимо ушей. Культура программирования, блин!)

Пошаговое выполнение программы и контрольные точки останова в UNIX используется лишь в клинических случаях (типа трепанации черепа), когда все остальные средства оказываться бессильными. Поклонникам Windows такой подход кажется несовременным, ущербным и жутко неудобным, но это все потому, что Windows-отладчики эффективно решают проблемы, которые в UNIX просто не возникают. Разница культур программирования между Windows и UNIX в действительности очень и очень значительна, поэтому, прежде чем кидать камни в чужой огород наведите порядок у себя. "Непривычное" еще не означает "неправильное". Точно такой же дискомфорт ощущает матерый юниксоид, очутившийся в Windows.

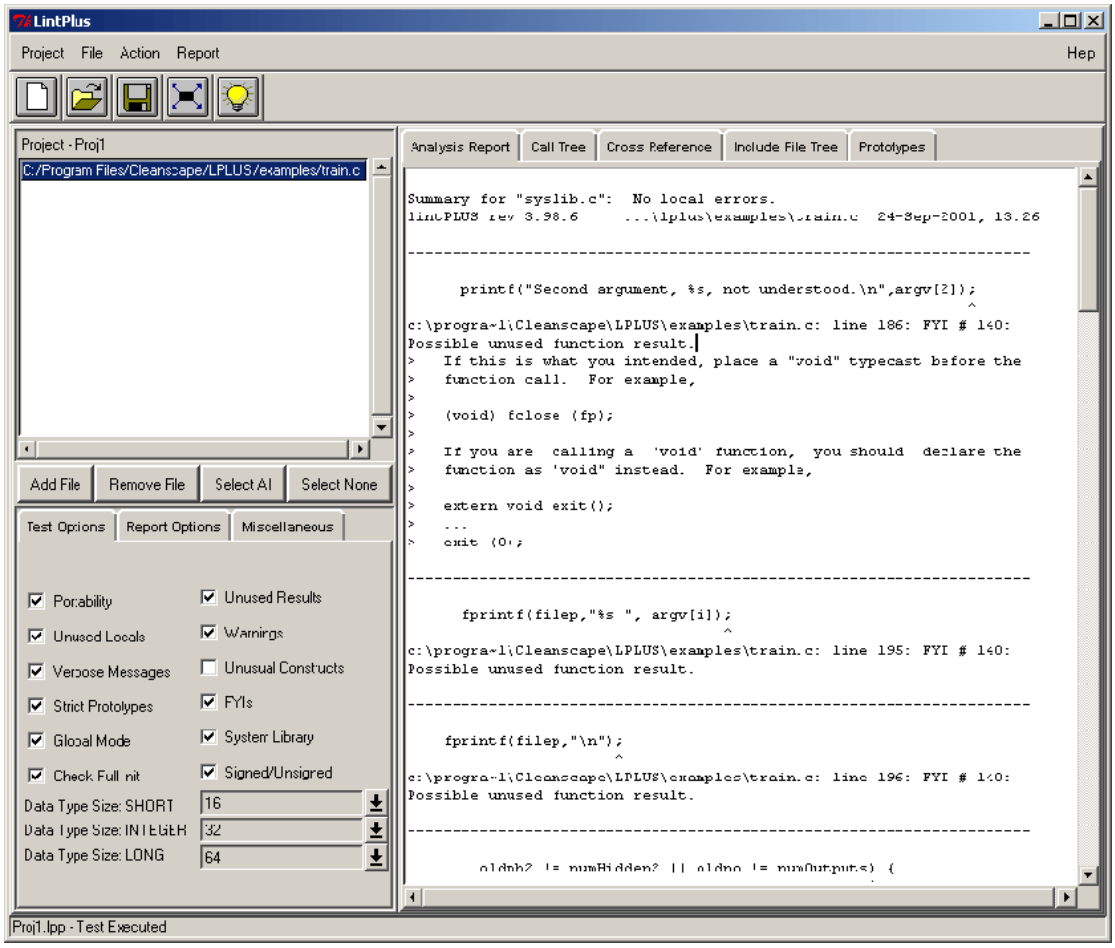

Рисунок 1 LINT в охоте на багов

ptrace – фундамент для GDB

GDB – это системно-независимый кросс-платформенный отладчик. Как и большинство UNIX-отладчиков он основан на библиотеке PTrace, реализующий низкоуровневые отладочные примитивы. Для отладки многопоточных процессов и параллельных приложений рекомендуется использовать дополнительные библиотеки, например, CTrace (http://ctrace.sourceforge.net/), а лучше – специализированные отладчики типа Total View (http://www.etnus.com), поскольку GDB с многопоточностью справляется не самым лучшим образом (см. врезку "поддержка многопоточности в GDB").

ptrace может: переводить процесс в состояние останова/возобновлять его выполнение, читать/записывать данные из/в адресное пространство отлаживаемого процесса, читать/записывать регистры ЦП. На i386 это – регистры общего назначения, сегментные регистры, регистры "сопроцессора" (включая SSE) и отладочные регистры семейства DRx (они нужны для организации аппаратных точек останова). В LINUX еще можно манипулировать служебными структурами отлаживаемого процесса и отслеживать вызов системных функций. В "правильных" UNIX'ах этого нет и недостающую функциональность приходится реализовывать уже в отладчике.

Пример использования ptrace в своих программах приведен ниже:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

main()
{
        int pid;                        // pid отлаживаемого процесса
        int wait_val;                   // сюда wait записывает возвращаемое значение
        long long counter = 1;          // счетчик трассируемых инструкций


        // расщепляем процесс на два
        // родитель будет отлаживать потомка
        // (обработка ошибок для наглядности опущена)
        switch (pid = fork())
        {
                case 0:                 // дочерний процесс (его отлаживают)

                        // папаша, ну-как потрассируй меня!
                        ptrace(PT_TRACE_ME, 0, 0, 0);

                        // вызываем программу, которую надо отрассировать
                        // (для программ, упакованных Шифрой это не сработет)
                        execl("/bin/ls", "ls", 0);
                        break;

                default:                // родительский процесс (он отлаживает)

                        // ждем пока отлаживаемый процесс
                        // не перейдет в состояние останова
                        wait(&wait_val);

                        // трассируем дочерний процесс, пока он не завершится
                        while (WIFSTOPPED(wait_val) /* 1407 */)
                        {
                                // выполнить следующую машинную инструкцию
                                // и перейти в состояние останова
                                if (ptrace(PT_STEP, pid, (caddr_t) 1, 0)) break;

                                // ждем пока отлаживаемый процесс
                                // не перейдет в состояние останова
                                wait(&wait_val);

                                // увеличиваем счетчик выполненных
                                // машинных инструкций на единицу
                                counter++;
                        }
                }
                // вывод кол-ва выполненных машинных инструкций на экран
                printf("== %lld\n", counter);
}
```

Листинг 3 пример использования ptrace на Free BSD – подсчет кол-ва машинных команд в ls, для компиляции под LINUX замените PT_TRACE_ME на PTRACE_TRACEME, а PT_STEP на PTRACE_SINGLESTEP

>>> врезка ptrace и ее команды

В user-mode режиме доступа всего лишь одна функция – `ptrace((int _request, pid_t _pid, caddr_t _addr, int _data))`, но зато эта функция делает все! При желании вы можете за пару часов написать собственный мини-отладчик, специально заточенный под вашу проблему.

Аргумент `_request` функции ptrace важнейший из всех – он определяет что мы будем делать. Заголовочные файлы в BSD и LINUX используют различные определения, затрудняя

перенос ptrace-приложений с одной платформы на другую. По умолчанию мы будем использовать определения из заголовочных файлов BSD.

PT_TRACE_ME (в LINUX – PTRACE_TRACEME): переводит текущий процесс в состояние останова. Обычно используется совместно с `fork/exec`, хотя встречаются так же и самотрассирующиеся приложения. Для каждого из процессов вызов PT_TRACE_ME может быть сделал лишь однажды. Трассировать уже трассируемый процесс не получится (менее значимое следствие – процесс не может трассировать сам себя, сначала он должен расщепиться). На этом основано большое количество антиотладочных приемов, для преодоления которых приходился использовать отладчики, работающие в обход `ptrace` (см. статью "методология защиты в мире UNIX"). Отлаживаемому процессу посылается сигнал, переводящий его в состояние останова из которого он может быть выведен командой PT_CONTINUE или PT_STEP, вызванных из контекста родительского процесса. Функция `wait` задерживает управление материнского процесса до тех пор, пока отлаживаемый процесс не перейдет в состояние останова или не завершиться (тогда она возвращает значение 1407). Остальные аргументы игнорируются.

PT_ATTACH (в LINUX – PTRACE_ATTACH): переводит в состояние останова уже запущенный процесс с заданным pid, при этот процесс-отладчик становится его "предком". Остальные аргументы игнорируются. Процесс должен иметь тот же самый UID, что и отлаживающий процесс и не быть setuid/setduid процессом (или отлаживаться root'ом).

PT_DETACH (в LINUX – PTRACE_DETACH): прекращает отладку процесса с заданным pid (как по PT_ATTACH, так и по PT_TRACE_ME) и возобновляет его нормальное выполнение. Все остальные аргументы игнорируются.

PT_CONTINUE (в LINUX – PTRACE_CONT): возобновляет выполнение отлаживаемого процесса с заданным pid без разрыва связи с процессом-отладчиком. Если `addr == 1` (в LINUX – 0), выполнение продолжается с места последнего останова, в противном случае – с указанного адреса. Аргумент `_data` задает номер сигнала, посылаемого отлаживаемому процессу (ноль – нет сигналов).

PT_STEP (в LINUX – PTRACE_SINGLESTEP): пошаговое выполнение процесса с заданным pid'ом: выполнить следующую машинную инструкцию и перейти в состояние останова (под i386 это достигается взводом флага трассировки, хотя некоторые "хакерские" библиотеки используют аппаратные точки останова). BSD требует, чтобы аргумент `addr` был равен 1, LINUX хочет видеть здесь 0. Остальные аргументы игнорируются.

PT_READ_I/PT_REEAD_D (в LINUX – PTRACE_PEEKTEXT/PTRACE_PEEKDATA) чтение машинного слова из кодовой области и области данных адресного пространства отлаживаемого процесса соответственно. На большинстве современных платформ обе команды полностью эквивалентны. Функция `ptrace` принимает целевой `addr` и возвращает считанный результат.

PT_WRITE_I/PR_READ_D (в LINUX – PTRACE_POKETEXT, PTRACE_POKEDATA) запись машинного слова, переданного в `_data`, по адресу `addr`.

PT_GETREGS/PT_GETFPREGS/PT_GETDBREGS (в LINUX – PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_GETFPXREGS): чтение регистров общего назначения, сегментных и отладочных регистров в область памяти процесса-отладчика, заданную указателем `_addr`. Это системно-зависимые команды, приемлемые только для i386 платформы. Описание регистровой структуры содержится в файле <machine/reg.h>.

PT_SETREGS/PT_SETFPREGS/PT_SETDBREGS (в LINUX – PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_SETFPXREGS): установка значения регистров отлаживаемого процесса путем копируя содержимое региона памяти по указателю `_addr`.

PT_KILL (в LINUX – PTRACE_KILL): посылает отлаживаемому процессу сигнал sigkill, который делает ему харакири.

>>> врезка поддержка многопоточности в GDB

Определить поддерживает ли ваша версия GDB многогопоточность или нет можно при помощи команды "`info thread`" (вывод сведений о потоках), а для переключений между потоками используйте "`thread N`".

Если поддержка многопоточности отсутствует обновите GDB до версии 5x или установите специальный патч, поставляемый вместе с вашим клоном UNIX или распространяемый отдельно от него.

```
(gdb) info threads
(gdb)
```

Листинг 4 отладка многопоточных приложений не поддерживается

```
info threads
  4 Thread 2051 (LWP 29448) RunEuler (lpvParam=0x80a67ac) at eu_kern.cpp:633
  3 Thread 1026 (LWP 29443) 0x4020ef14 in __libc_read () from /lib/libc.so.6
* 2 Thread 2049 (LWP 29442) 0x40214260 in __poll (fds=0x80e0380, nfds=1, timeout=2000)
  1 Thread 1024 (LWP 29441)  0x4017caea in __sigsuspend (set=0xbffff11c)
(gdb) thread 4
```

Листинг 5 отладка многопоточных приложений поддерживается

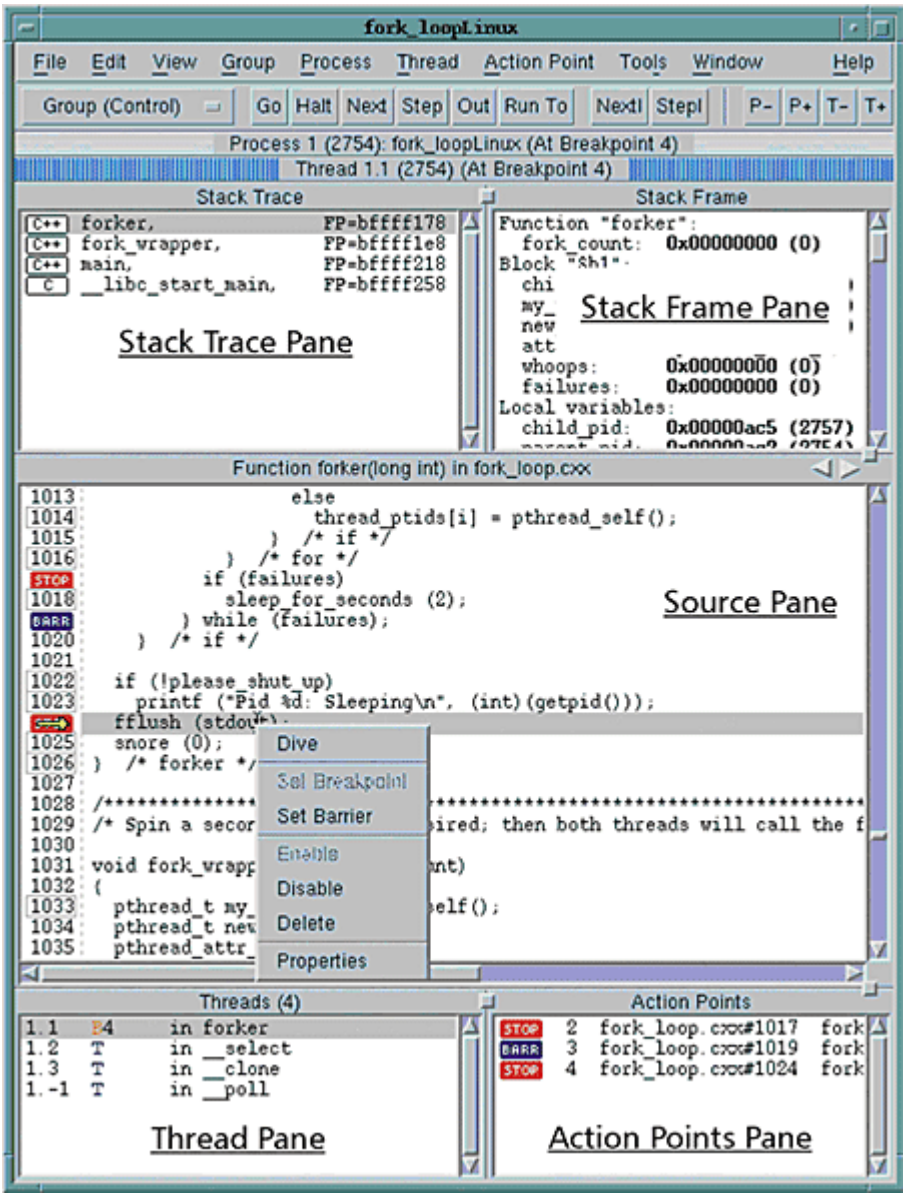

Рисунок 2 внешний вид отладчика Total View, специализирующихся на параллельных приложениях

краткое руководство по GDB

GDB – это консольное приложение, выполненное в классическом духе командной строке (см. рис 3). И хотя за время своего существования, GDB успел обрасти ворохом

красивых графических морд (см. рис 4, 5), интерактивная отладка в стиле TD в мире UNIX крайне непопулярна. Как правило, это удел эмигрантов с Windows-платформы, сознание которых необратимо искалечено идеологией "окошек". Грубо говоря, если TD – слесарный инструмент, то GDB – токарный станок с программным управлением. Когда-нибудь вы полюбите его…

```
#
# gcc -g debug_demo.c -o debug_demo
# gdb -q debug_demo
(gdb) b main
Breakpoint 1 at 0x80484ca: file debug_demo.c, line 13.
(gdb) r
Starting program: /root/debug_demo

Breakpoint 1, main (argc=1, argv=0xbfbffc68) at debug_demo.c:13
13              int a; int b;b  = 1;
(gdb) display/i $pc
1: x/i $eip  0x80484ca <main+6>:        movl   $0x1,0xfffffff8(%ebp)
(gdb) n
15              for (a = 0; a < 6; a++)
1: x/i $eip  0x80484d1 <main+13>:       movl   $0x0,0xfffffffc(%ebp)
(gdb) n
16                      foo(a, b);
1: x/i $eip  0x80484e0 <main+28>:       add    $0xfffffff8,%esp
(gdb) s
foo (a=0, b=1) at debug_demo.c:6
6               c = a + b;
1: x/i $eip  0x804849a <foo+6>: mov    0x8(%ebp),%eax
(gdb) p b
$1 = 1
(gdb)
```

Рисунок 3 внешний вид отладчика GDB

Для отладки на уровне исходных текстов программа должна быть откомпилирована с отладочной информацией. В gcc за это отвечает ключ "`-g`". Если отладочная информация недоступна, GDB будет отлаживать программу на уровне дизассемблерных команд.

Обычно имя отлаживаемого файла передается в командной строке ("`gdb filename`"). Для отладки активного процесса укажите в командной строке его ID, а для подключения коры (core dump) воспользуйтесь ключом "`-core==corename`". Все три параметра можно загружать одновременно, попеременно переключаясь между ними командной target. "`target exec`" переключается на отлаживаемый файл, "`target child`" на приаттаченный процесс, а "`target core`" на дамп коры. Необязательный ключ "`-q`" подавляет вывод копирайта.

Загрузив программу в отладчик мы должны установить точку останова. Для этого служит команда "`break`" (она же "`b`"). "`b main`" устанавливает точку останова на функцию `main` языка Си, а "`b _start`" – на точку входа в ELF-файл (впрочем, в некоторых файлах она называется по другому). Можно установить точку останова и на произвольный адрес: "`b *0x8048424`" или "`b *$eax`". Регистры пишутся маленькими буквами и предваряются знаком доллара. GDB понимает два "общесистемных" регистра: `$pc` – указатель команд и `$sp` – стековый указатель. Только помните, что непосредственно после загрузки программу в отладчик никаких регистров у нее еще нет и они появляются только запуска отлаживаемо процесса на выполнение (команда "`run`", она же "`r`").

Отладчик самостоятельно решает какую точку останова установить – программную или аппаратную и лучше ему не препятствовать (команда принудительной установки аппаратной точки останова – "`hbreak`" – работает не на всех версиях отладчика, в моей она не работает точно). Точки останова на данные в GDB называются "точками наблюдения" – watch point. "`watch addr`" вызывает отладчик всякий раз, когда содержимое `addr` изменяется, а "`awatch addr`" – при чтении/записи в `addr`. Команда "`rwatch addr`" реагирует только чтение, но работает не во всех версиях отладчика. Просмотреть список установленных точек останова/наблюдения можно командой "`info break`". Команда "`clear`" удаляет все точки останова, "`clear addr`" – все точки останова, установленные на данную функцию/адрес/номер строки. Команды "`enable`"/"`disable`" позволяют временно включать/отключать точки останова. Точки останова поддерживают развитый синтаксис условных команд, описание которого можно найти в документации. Команда "`continue`" ("`c`") возобновляет выполнение программы, прерванное точкой останова.

Команда "next N" ("n N") выполняет N-следующих строк кода без входа, а "step N" ("s N") со входом во вложенные функции. Если N не задано по умолчанию выполняется одна строка. Команды "nexti/stepi" делают тоже самое, но работает не со сроками исходного текста, а с машинными командами. Обычно они используются совместно с командой "display/i $pc" ("x/i $pc"), предписывающей отладчику отображать текущую машинную команду. Ее достаточно вызывать один раз за сеанс.

Команда "jump addr" передает управление в произвольную точку программы, а "call addr/fname" – вызывает функцию fname с аргументами! Этого нету даже в soft-ice! А как часто оно требуется! Другие полезные команды: "finish" – продолжать выполнение до выхода из текущей функции (соответствует команде soft-ice "P RET"), "until addr" ("u addr") – продолжать выполнение пока указанное место не будет достигнуто, при запуске без аргументов – остановить выполнение при достижении следующей команды (актуально для циклов!), "return" – немедленное возращение в дочернюю функцию.

__соответствие GDB и soft-ice команд

Команда "print выражение" ("p выражение") выводит значение выражения (например, "p 1+2"), содержимое переменной ("p my_var"), содержимое регистра ("p $eax") или ячейку памяти ("p *0x8048424", "p *$eax"). Если необходимо вывести несколько ячеек – воспользуйтесь командой "x/Nh addr", где N – кол-во выводимых ячеек. Ставить символ звездочки перед адресом в этом случае не нужно. Команда "info registers" ("i r") выводит значение всех доступных регистров. Модификация содержимого ячеек памяти/регистров осуществляется командой "set". Например, "set $eax = 0" записывает в регистр eax ноль. "set var my_var = $ecx" присваивает переменной my_var значение регистра ecx, а "set {unsigned char*}0x8048424=0xCC" записывает по байтовому адресу 0x8048424 число 0xCC. "disassemble _addr_from _addr_to" выдает содержимое памяти в виде дизассемблерного листинга, формат представления которого определяется командой: "set disassembly-flavor".

Команды "info frame", "info args", "info local" отображают содержимое текущего фрейма стека, аргументы функции и локальные переменные. Для переключения на фрейм материнских функций служит команда "frame N". Команда "backtrace" ("bt") делает тоже самое, что и "call stack" в Windows-отладчиках. При исследовании дампов коры она незаменима.

Короче говоря, приблизительный сеанс работы с GDB выглядит так: грузим программу в отладчик, даем "b main" (а, если не сработает, то "b _start"), затем "r", после чего отлаживаем программу по шагам: "n"/"s", при желании задав "x/i $pc" чтобы GDB показывал что у нас выполняется в данный момент. Выходим из отладчика по "quit" ("q"). Описание остальных команд – в документации. Теперь, по крайней мере, вы не заблудитесь в ней.

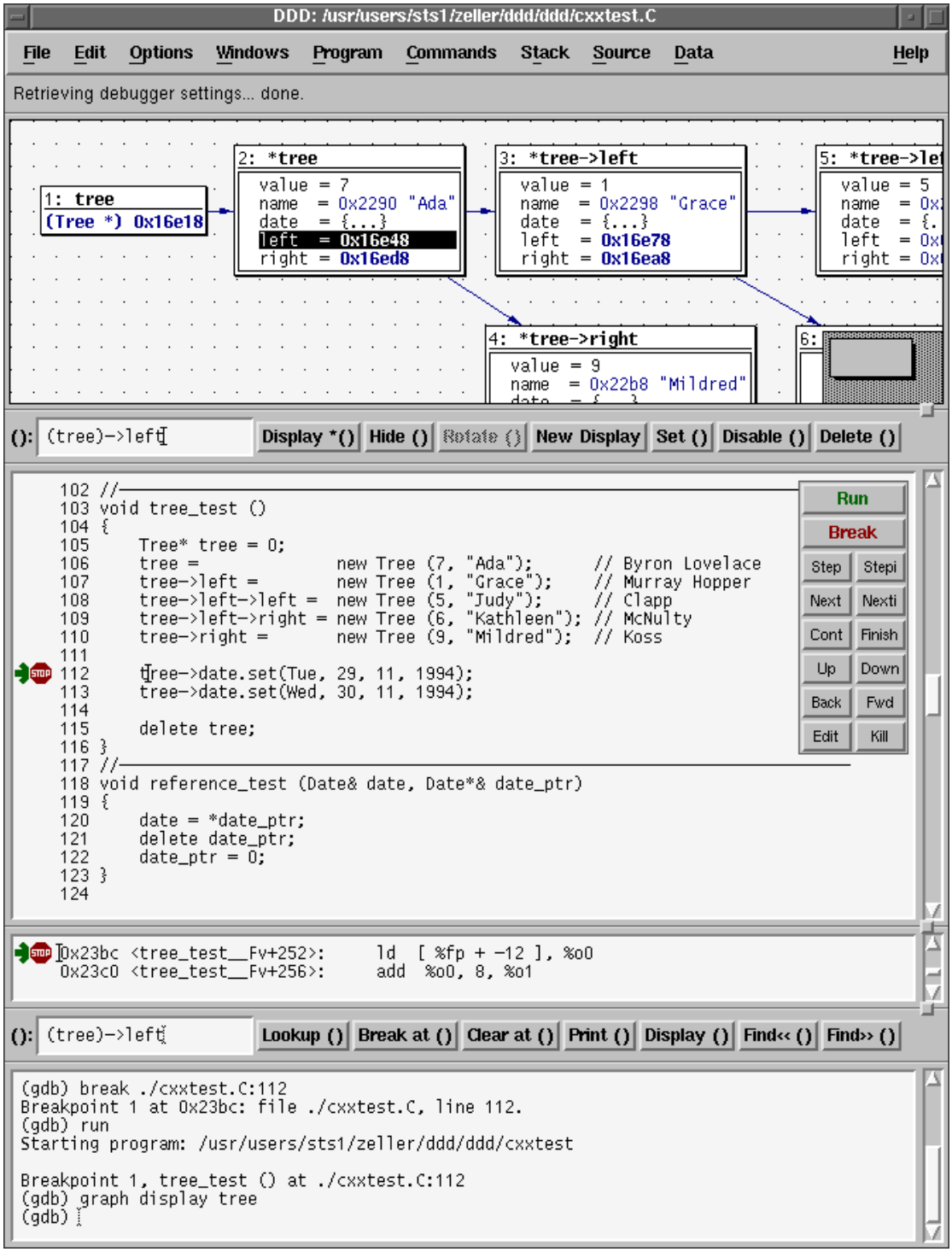

Рисунок 4 отладчик DDD – графический интерфейс к GDB

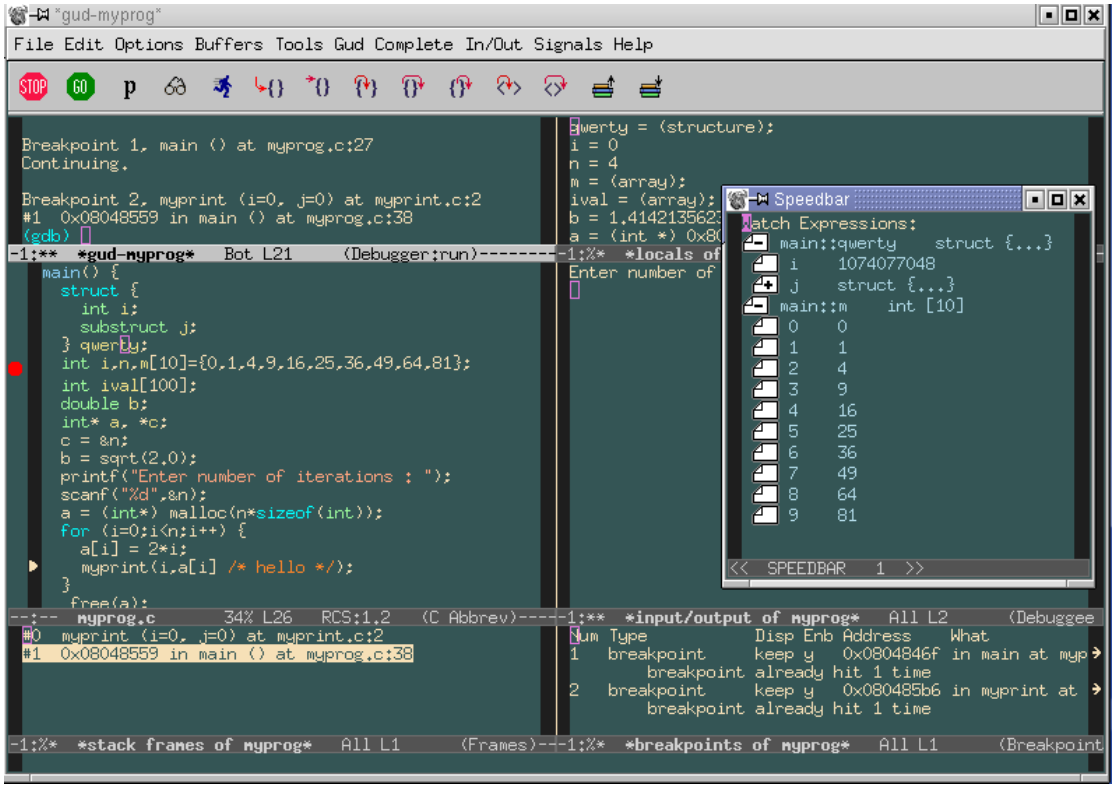

Рисунок 5 еще один графический интерфейс к GDB

>>>> врезка трассировка системных функций

Перехват системных функций – это настоящее окно во внутренний мир подопытной программы, показывающее имена вызываемых функций, их аргументы и коды возврата. Отсутствие "лишних" проверок на ошибки – болезнь всех начинающих программистов и отладчик – не самое лучшее средство для их поиска. Воспользуйтесь одной из штатных утилит `truss/ktrace` или возьмите любой бесплатный/коммерческий анализатор.

Вот протокол, полученный `truss`. Смотрите, перед тем как умереть, программа открывает файл "my_good_file", не находит его и, как следствие, сбрасывает кору впадая в нирванну. Разумеется, это простейший случай, но правило "десяти" гласит, что девяносто процентов времени отладки уходит на поиск ошибок, которые вообще недостойны того, чтобы их искать!

```
__sysctl(0xbfbffb28,0x2,0x2805bce8,0xbfbffb24,0x0,0x0) = 0 (0x0)
mmap(0x0,32768,0x3,0x1002,-1,0x0)              = 671469568 (0x2805d000)
geteuid()                                      = 0 (0x0)
getuid()                                       = 0 (0x0)
getegid()                                      = 0 (0x0)
getgid()                                       = 0 (0x0)
open("/var/run/ld-elf.so.hints",0,00)          = 3 (0x3)
read(0x3,0xbfbffb08,0x80)                      = 128 (0x80)
lseek(3,0x80,0)                                = 128 (0x80)
read(0x3,0x28061000,0x4b)                      = 75 (0x4b)
close(3)                                       = 0 (0x0)
access("/usr/lib/libc.so.4",0)                 = 0 (0x0)
open("/usr/lib/libc.so.4",0,027757775600)      = 3 (0x3)
fstat(3,0xbfbffb50)                            = 0 (0x0)
read(0x3,0xbfbfeb20,0x1000)                    = 4096 (0x1000)
mmap(0x0,626688,0x5,0x2,3,0x0)                 = 671502336 (0x28065000)
mmap(0x280e5000,20480,0x3,0x12,3,0x7f000)      = 672026624 (0x280e5000)
mmap(0x280ea000,81920,0x3,0x1012,-1,0x0)       = 672047104 (0x280ea000)
close(3)                                       = 0 (0x0)
sigaction(SIGILL,0xbfbffba8,0xbfbffb90)        = 0 (0x0)
sigprocmask(0x1,0x0,0x2805bc1c)                = 0 (0x0)
sigaction(SIGILL,0xbfbffb90,0x0)               = 0 (0x0)
sigprocmask(0x1,0x2805bbe0,0xbfbffbd0)         = 0 (0x0)
sigprocmask(0x3,0x2805bbf0,0x0)                = 0 (0x0)
```

```
open("my_good_file",0,0666)                 ERR#2 'No such file or directory'
SIGNAL 11
SIGNAL 11
Process stopped because of:  16
process exit, rval = 139
```

Листинг 6 поиски бага с помощью truss

>>>> врезка интересные ссылки

- Отладка с помощью GDB
  - добротно сверстанная документация на GDB (на русском языке) http://www.linux.org.ru/books/GNU/gdb/gdb-ru.pdf;
- GDB Internals
  - отличное руководство по внутреннему миру GDB (на английском языке). Очень помогает при доработке исходников: http://gnuarm.org/pdf/gdbint.pdf;
- трассировка процессов с помощью Ptrace
  - статья про трассировку в LINUX с примерами простейших трассировщиков (во Free BSD все не так) http://gazette.linux.ru.net/lg81/sandeep.html;
- Отладчик GNU GDB
  - введение в отладку программ, написанных на Free Pascal: http://linuxshop.ru/linuxbegin/article496.html;
- Squashing Bugs at the Source
  - использование библиотеки CTrace для отладки многопоточных программ (на английском языке): http://www.linux-mag.com/2004-04/code_01.html.
- Kernel- und UserSpace Debugging Techniken
  - тезисы доклада, посвященного отладке и раскрывающего малоизвестные детали строения GDB (на немецком языке): http://www.unfug.org/files/debugging.pdf
- Reverse engineering des syst`emes ELF/INTEL
  - исследование и отладка ELF файлов на i386-платформе без исходных текстов: www.sstic.org/SSTIC03/articles/SSTIC03-Vanegue_Roy-Reverse_Intel_ELF.pdf

заключение

Сравнение UNIX-отладчиков с Windows, показывает значительное отставание последних и их непрофессиональную направленность. Трехмерные кнопки, масштабируемые иконки, всплывающие менюшки – все это конечно, очень красиво, но жать <F10> до потери пульса лениво, да и не фиг. В GDB проще макрос написать или использовать уже готовый (благо все, что только было можно запрограммировать здесь запрограммировали задолго до нас, пользуйся – не хочу).

Отладочные средства в UNIX мощены и разнообразны (свет клином не сошелся на GDB!), и единственное чего ей недостает – нормального ядерного отладчика системного уровня, ориентированно на работу с двоичными файлами без символьной информации и исходных тестов. Тяжелое детство и скитание по множеству платформ наложило на UNIX мрачный отпечаток и неприкаянное стремление к переносимости и кросс-платформенности. Какое там хакерство в таких условиях! Впрочем, доступность исходных текстов делает эту проблему неактуальной.