Приложение А. Использование символьного отладчика GDB

Введение

Важным элементом процесса разработки программ является поиск и устранение ошибок (тестирование и отладка). В больших по размеру программах ошибки или «баги», практически неизбежны. Программы могут выдавать неверные результаты, зависать, попадая в бесконечные циклы или «падать» из-за некорректных операций с памятью. Задача поиска и устранения таких ошибок называется **отладкой** программы.

Многие «баги» не видны при простом просмотре кода программы. Дополнительный вывод, генерируемый тестовой версией программы – это один из полезных приемов диагностики ошибок. Мы в этом случае указываем дополнительные операторы печати значений переменных и другой информации во время выполнения. Однако, более общим (и более эффективным) методом считается использование программного обеспечения специального назначения, а именно *отладчика*.

Отладчик – это программа, которая запускает другую программу в полностью контролируемом окружении. Мы, например, можем пошагово выполнять нашу программу, смотреть содержимое переменных, ячеек памяти и регистров процессора после каждого оператора. Мы также можем проанализировать цепочку вызова функций до заданной точки программы.

В этом разделе мы сделаем обзор одного мощного и широко известного отладчика - *GNU debugger* или GDB. Мы опишем наиболее базовые опции и команды. Большая часть возможностей и принципов работы похожи на используемые в других отладочных средствах. За подробным описанием рекомендуется обратиться к официальной документации [1] или соответствующему разделу справочной системы ОС, если GDB уже в ней установлен.

Установка GDB

Если GNU C компилятор (GCC) доступен в работающей системе, то GDB вероятно так же установлен. Это можно проверить следующей командой в оболочке (заодно проверим версию):

```
$ gdb -version
```

Если GDB уже установлен в ОС, то должно появиться сообщение, наподобие следующего:

```
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
```

Версия 0.9pre-release от 03.05.2014. Возможны незначительные изменения.

```
This GDB was configured as "i586-debian-linux".
```

Если GDB еще не установлен, то его можно скачать и собрать из исходных текстов [2]. Хотя это редко оказывается необходимым. Кроме того, следует иметь в виду, что существуют «надстройки» над GDB, например, [3].

Пример отладочной сессии

Рассмотрим некоторые базовые операции при работе GDB на примере следующей программы, содержащей ошибку.

```
/* gdb_example.c:
 * Test the swap() function, which exchanges the contents of two int variables.
 */
#include <stdio.h>
void swap(int* p1, int* p2); // Exchange *p1 and *p2

int main()
{
  int a = 10, b = 20;

  /* ... */
  printf("The old values: a = %d; b = %d.\n", a, b);
  swap(&a, &b);

  printf("The new values: a = %d; b = %d.\n", a, b);

  /* ... */
  return 0;
}

void swap(int* p1, int* p2) /* Exchange *p1 and *p2. */
{
  int* p = p1;
  p1 = p2;
  p2 = p;
}
```

Многие проблемы в С-программах можно точно определить с помощью вводимых вручную команд. В нашей программе имеется логическая ошибка. Мы используем эту же программу в дальнейшем, чтобы показать, как GDB можно использовать для отслеживания таких ошибок.

GDB является символьным отладчиком. «Символьный» в данном случае означает, что мы ссылаемся на переменные и функции по тем именам, которые мы задаем в нашем С-коде. Чтобы показывать и интерпретировать эти имена, отладчику требуется информация о типах переменных и функций в программе и том, каким строкам в программе какие инструкции соответствуют в исполняемом файле. Обычно такая информация принимает форму *таблицы символов*, которую компилятор и компоновщик включает в исполняемый модуль, когда мы запускаем GCC с опцией *-g*:

```
$ gcc -g gdb_example.c
```

В проекте, состоящем из нескольких исходных файлов, следует каждый из них компилировать с этой опцией.

Следующая команда запустит нашу программу:

```
$ ./a.out
```

и программ выдаст следующее:

```
The old values: a = 10; b = 20.
The new values: a = 10; b = 20.
```

Хотя вызов функции *swap*() хорошо виден в исходном коде, содержимое переменных *a* и *b* не изменилось. Мы можем поискать причину, используя GDB. Чтобы начать отладочную сессию, мы запустим GDB из командной оболочки с указанием имени исполняемого файла в качестве аргумента командной строки для отладчика:

```
$gdb ./a.out
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
...
(gdb)
```

Отладчик загрузит программу, но будет ждать пользовательских команд до ее запуска. GDB выдает *(gdb*) в начале строки в качестве подсказки пользователю. Мы можем начать работу вводом команды *list* (для краткости достаточно даже ее первой литеры). По умолчанию, это выдаст нам листинг из 10 строк программы, которую мы собираемся отлаживать. В нашем случае программа только начала работу, поэтому следующей будет 8-ая строка, где находится функция *main*().

Если мы затем запишем еще одну команду *list*, то GDB напечатает следующие несколько строк исходного кода.

Перед инструктированием GDB, как ему запускать программу, мы должны сообщить, где он должен остановиться. Это можно сделать с помощью так называемой **точки останова** (*breakpoint*). Когда отладчик дойдет до этой точки, он прервет выполнение программы, предоставляя нам возможность просмотреть состояние программы в этой точке. Следовательно, прервав выполнение программы в точке останова, мы можем продолжить ее построчное выполнение, наблюдая за состоянием программных объектов по мере продвижения по программе.

Чтобы указать точку останова, нужно ввести команду *break*, или для краткости просто литеру *b*. Точки останова обычно указываются в определенных местах программы или в начале отладки функции. Следующая команда указывает точку останова на 15-ой строке текущего исходного кода, в нашем случае это строка, содержащая вызов функции *swap*():

```
(gdb)b 15
Breakpoint 1 at 0x80483aa: file gdb_example.c, line 15.
```

Команда *run* (или просто *r*) стартует выполнение программы:

```
(gdb)r
Starting program: /home/user/12/a.out
Breakpoint 1, main () at gdb_example.c:15
15 swap( &a, &b );
```

По достижении этой точки отладчик прерывает выполнение программы и показывает строку, содержащую следующий оператор, который может быть выполнен. Поскольку мы предполагаем, что баг в нашем примере находится в функции *swap*(), мы будем выполнять ее пошагово. Для этого у GDB имеются команды *next* (или *n*) и *step* (или *s*). Команды *next* и *step* ведут себя по-разному, если следующая строка содержит вызов функции. Команда *next* выполняет следующую строку, включая вызовы функций, и опять прерывает программу на следующей строке. Команда *step*, со своей стороны, осуществляет переход в функцию, которая вызывается на следующей строке, предполагая доступность отладочных символов для этой функции, и приостанавливает программу на первом операторе в теле функции. В нашем случае команда *step* приводит нас к первой инструкции в функции *swap()*:

```
(gdb)s
swap (p1=0xbffff234, p2=0xbffff230) at gdb_example.c:24
24 int *p = p1;
```

Отладчик показывает значения аргументов функции (здесь это адреса переменных *a* и *b*) и следующую строку для выполнения. В этой точке мы можем проверить, корректны ли значения аргументов функции, передаваемых по ссылке. Для этого нам предлагается команда *print* (или *p* для краткости), которая выводит значение данного выражения:

```
(gdb)p *p1
$1 = 10
(gdb)p *p2
$2 = 20
```

Значение выражения **p1* равно 10, а у **p2* – 20. Отладчик GDB печатает результат в формате *$number = value*, где *$number* это временная переменная GDB, которую отладчик создает по мере необходимости, чтобы мы смогли сослаться на него впоследствии.

Если мы теперь трижды введем *n* (или *next*), то отладчик выполнит строки 24, 25 и 26:

```
(gdb)n
25 p1 = p2;
(gdb)n
26 p2 = p;
(gdb)n
27 }
(gdb)
```

Так как поток инструкций программы не вернулся из функции *swap*() в функцию *main()*, то можно командой *print* показать содержимое локальных переменных:

```
(gdb)p *p1
$3 = 20
(gdb)p *p2
$4 = 10
```

Теперь **p1* принимает значение 20, а **p2* – значение 10, что кажется правильным. Посмотрим состояние программы еще двумя командами *print*:

```
(gdb)p p1
$5 = (int *) 0xbffff230
(gdb)p p2
$6 = (int *) 0xbffff234
(gdb)
```

Эти команды *print* показывают, что произошел обмен значениями между указателями *p1* и *p2*, а вовсе не содержимым ячеек памяти, на который они указывают. Это и есть наш замечательный баг в функции *swap()*. Ее нужно изменить, чтобы она обменивала целочисленные значения, адресуемые как **p1* и **p2*, а не значения, хранимые в *p1* и *p2*. Исправленная версия функции показана далее.

```
/* fixed gdb_example.c:
 * Test the swap() function, which exchanges the contents of two int variables.
 */
#include <stdio.h>
void swap(int* p1, int* p2); // Exchange *p1 and *p2

int main()
{
  int a = 10, b = 20;

  /* ... */
  printf("The old values: a = %d; b = %d.\n", a, b);
  swap(&a, &b);

  printf("The new values: a = %d; b = %d.\n", a, b);

  /* ... */
  return 0;
```

```
}

void swap(int* p1, int* p2) /* Exchange *p1 and *p2. */
{
  int* p = *p1;
  *p1 = *p2;
  *p2 = *p;
}
```

Команда *continue*, или *c*, позволяет программе продолжать выполнение до следующей точки останова или до конца программы:

```
(gdb)c
Continuing.
The new values: a = 10; b = 20
Program exited normally.
(gdb)
```

Подсказка *(gdb)* указывает на то, что отладчик еще работает. Чтобы его остановить, нужно ввести команду *quit* (или *q*). Отладчик завершит работу, даже если отлаживаемая программа еще выполняется. Тем не менее, GDB запросит подтверждение завершения отладки:

```
(gdb)q
The program is running. Exit anyway? (y or n)y
$
```

Запуск GDB

Мы запускаем GDB путем ввода *gdb* в командной строке оболочки (СЛАЙД 7). GDB поддерживает большое количество опций и аргументов командной строки:

```
gdb [options] [executable_file[core_file|process_id]]
```

Например, следующая команда запускает отладчик без стартового баннера:

```
$gdb -silent
(gdb)
```

Здесь мы не указали имя исполняемого файла отлаживаемой программы. Ее можно загрузить в отладчике с помощью команды *file*.

Тем не менее, обычно следует указывать имя файла в командной строке. Например, так:

```
$gdb myprog
(gdb)
```

Как дополнительный аргумент после имени программы мы можем указать PID процесса программы или имя файла, куда будет копироваться дамп состояния программы. Например:

```
$gdb myprog 1001
(gdb)
```

Эта команда указывает GDB на присоединение к процессу, уже запущенному в системе, имя которого равно *myprog*, а PID – 1001. Если GDB найдет такой процесс, его выполнение может быть прервано сочетанием клавиш Ctrl+C, и далее мы сможем начать отладку. Обратим внимание на то, что если отладчик найдет в текущем каталоге файл с именем 1001, он будет интерпретирован как имя файла для дампа, а не как PID процесса.

Большую часть опций командной строки для GDB можно указывать в коротком или длинном формате. Длинный формат можно обрезать, если написанного достаточно для того, чтобы быть однозначно воспринятым. Для опций, которые принимают аргумент, например, *-tty device*, опцию от аргумента можно отделить пробелом либо одним знаком

равенства (*tty=/dev/tty6*). Опции можно вводить одним либо двумя дефисами:*-quiet* – это синоним для *--quiet*, например.

У GDB имеется специальная опция, которая предназначена для разделения собственных аргументов отладчика от аргументов отлаживаемой программы:

```
--args
```

Мы используем опцию *--args* в начале отладочной сессии для того, чтобы передавать аргументы отлаживаемой в GDB программе. Например: в следующем примере, *myprog* это программа, которую мы собираемся отладить.

```
$gdb --args myprog -d "$HOME"
(gdb)
```

Эта опция указывается непосредственно за командой, запускающей программу, и должна состоять из имени программы и ее аргументами в том порядке, в котором они должны появляться, как если бы мы запустили ее без GDB. У нас *-d* и *"$HOME"* – это аргументы для *myprog*, а не для *gdb*. Если в то же самое время мы захотим определить опции для GDB, нам придется поместить их перед*--args*. Иными словами, *--args* должно быть последней опцией GDB.

Мы можем также определить аргументы отлаживаемой программы уже после запуска GDB с помощью аргументов одной из интерактивных команд *run* или *set*.

За полным списком опций командной строки и их описанием рекомендуется обратиться к официальной документации.

Использование команд GDB

После запуска отладчик приглашает нас вводить команды, например для того, чтобы указать точку останова и запустить программу, которая указана в командной строке для отладки.

Каждая команда, которую мы пишем в GDB, это символьная строка, начинающаяся с ключевого слова, именующего команду. Остальная часть – аргументы. Мы можем сокращать любое ключевое слово, если это не делает имя команду неоднозначным. Например, мы можем ввести *q* (или *qu*, или *qui*) для того, чтобы выйти из отладчика по команде *quit*.

Когда мы вводим пустую команду, например, простым нажатием клавиши Return после приглашения GDB, тогда он повторяет последнюю команду, если ее действие допустимо. Например, таким способом GDB автоматически повторяет команды *step* и *next*, но не команду *run*.

Если мы введем неоднозначную или неизвестную аббревиатуру или совершим ошибку при определении аргументов команды, то GDB сообщит об этом, например:

```
(gdb)sh
Ambiguous command "sh": sharedlibrary, shell, show.
```

Отладчик GDB может уменьшить время на ввод путем автозаполнения имен команд, переменных, файлов и функций. Для этого надо написать несколько первых символов желаемого слова, а затем нажать клавишу табуляции. Например, программа *circle.c*, код которой приведен далее, содержит функцию *circularArea()*.

```
// circle.c: Calculate and print the areas of circles
#include <stdio.h>
double circularArea(double r);
int main()
{
  double radius = 1.0, area = 0.0;
  printf(" Areas of Circles\n\n");
  printf(" Radius Area\n""-------------------------\n");
  area = circularArea(radius);
```

```
  printf("%10.1f %10.2f\n", radius, area);
  radius = 5.0;
  area = circularArea(radius);
  printf("%10.1f %10.2f\n", radius, area);
  return 0;
}

// The function circularArea() calculates the area of a circle
// Parameter: The radius of the circle
// Return value: The area of the circle
double circularArea(double r)
{
  const double pi = 3.1415926536;
  return pi * r * r;
}
```

Чтобы показать эту функцию в сессии работы с GDB, нужно ввести следующее:

```
(gdb)list ci
```

затем нажать клавишу табуляции. Автозаполнение выдаст следующую командную строку:

```
(gdb) list circularArea
```

Чтобы команду выполнить, нажмем клавишу *Return*. Если же существует несколько способов заполнения некоего слова, GDB вставит следующие литеры, которые являются общими для всех возможных вариантов, потом попросит нас ввести еще что-нибудь. Мы можем написать еще один или два знака, чтобы сделать наш ввод более точным, а затем снова нажать клавишу табуляции. Если мы нажмем ее дважды в одной строке, GDB напечатает все возможные варианты автозаполнения. Вот пример автозаполнения команды за несколько шагов:

```
(gdb)break ci<tab>
```

GDB добавит два знака, затем приостановится в ожидании дальнейшего ввода, который может разрешить неоднозначность:

```
(gdb) break circ
```

Если мы два раза нажмем клавишу табуляции, GDB покажет все варианты, а затем повторит приглашение на ввод.

```
(gdb)break circ<tab><tab>
circle.c circularArea
(gdb) break circ
```

У GDB есть встроенная справочная подсистема, которая классифицирует многие команды, чтобы помочь нам отыскать подходящую команду. Когда мы вводим *help* (или *h*) без аргументов, GDB выдает список классов всех возможных команд.

Чтобы прочитать подробнее о командах из заданного класса, нужно написать *help*, а затем имя класса. Чтобы прочитать о том, как работает конкретная команда, вслед за вводом *help* нужно указать имя команды.

Чтобы вывести информацию о статусе отладчика или отлаживаемой программы, у GDB имеются команды *info* and *show*:

```
(gdb)help status
Status inquiries.
List of commands:
info -- Generic command for showing things about the program being debugged
show -- Generic command for showing things about the debugger
```

Команда *info* без аргументов перечисляет все, что мы можем запросить об отлаживаемой программе.

Когда мы указываем один из аргументов, GDB печатает соответствующую информацию. Как и команды, эти аргументы могут быть сокращены:

```
(gdb)info all-reg
```

Здесь выдается содержимое всех регистров, включая расширенные. Команда *info register*, со своей стороны, покажет только регистру ЦП.

Информацию о текущем исходном файле (тот, что содержит отлаживаемую функцию) печатает следующая команда. Если программа еще не была запущена, то текущим исходным файлом будет считаться тот, который содержит функцию *main*():

```
(gdb)info source
Current source file is circle.c
Compilation directory is /home/user/12/
...
(gdb)
```

Некоторые из этих команд принимают аргументы. Например, команда *info address* в следующем примере принимает имя объекта или функции в качестве аргумента:

```
(gdb)info address radius
Symbol "radius" is a local variable at frame offset -8.
(gdb)
```

Важным элементом отладки является работа с точками останова, по достижении которых отладчик прерывает выполнение программы и выводит строку, на которой установлена точка. Если быть более точным, то печатаемая строка содержит тот оператор, который будет выполнен после возобновления программы.

Точки останова так же можно сделать условными, т.е. по достижении такой точки отладчик прерывает программу только при выполнении определенных условий.

Указание точки останова делается командой *break* или ее краткой формой. Определить местоположение точки останова можно несколькими способами. Наиболее общими из них являются:

```
break [filename:]line_number
```

Точка останова указывается на заданной строке текущего исходного файла или файла с именем *filefilename*.

```
break function
```

Точка ставится на первой строке указанной функции.

```
break
```

Точка ставится на следующем операторе, который будет выполнен. Иначе говоря, программа автоматически прервется в следующий раз, когда заданная точка будет достигнута.

Возвращаясь к нашему примеру (*gdb_example.c*), следующая команда задаст точку останова в начале функции *swap()*:

```
(gdb)b swap
Breakpoint 1 at 0x4010e7: file gdb_example.c, line 27.
```

GDB сообщит нам, что точка останова действительно зафиксирована на строке 27. При желании можно убедиться в этом с помощью команды *list 27*.

Укажем вторую точку останова на строке 30, т.е. перед завершением функции:

```
(gdb)b 30
```

```
Breakpoint 2 at 0x4010f9: file gdb_example.c, line 30.
```

Можно посмотреть все заданные точки останова:

```
(gdb)info breakpoints
Num Type       Disp Enb Address    What
1   breakpoint keep y   0x004010e7 in swap at gdb_example.c:27
2   breakpoint keep y   0x004010f9 in swap at gdb_example.c:30
```

Если нужна временная точка останова, то GDB автоматически удаляет ее после того, как дойдет до нее в следующий раз. Для указания вместо *break* используется команда *tbreak*:

```
(gdb)tbreak 16
Breakpoint 3 at 0x4010c0: file gdb_example.c, line 16.
```

Когда используется обычная точка останова, то отладчик приостанавливает выполнение программы всякий раз после ее достижения. Когда используется условная точка останова, то GDB останавливает программу, только если истинно заданное условие. Мы можем определить это условие путем добавления ключевого слова *if* в привычную команду *break*:

```
break [position] ifexpression
```

Здесь *position* может быть именем функции или номером строки с указанием или без указания имени файла как и с безусловными точками останова. Условием в нашем случае может быть любое С-выражение скалярного типа и может включать вызовы функций:

```
(gdb)s
27 for ( i = 1; i <= limit ; ++i )
(gdb)break 28 if i == limit - 1
Breakpoint 1 at 0x4010e7: file gdb_test.c, line 28.
```

Смысл этой команды очевиден.

Если у нас уже указана точка останова в заданной позиции, чтобы добавить или изменить условие останова, мы можем использовать команду *condition*:

```
condition bp_number [expression]
```

Аргумент *expression* становится новым условием для точки останова с номером *numberbp_number*. Вывод команды *info breakpoints* включает все точки останова, в том числе условные:

```
(gdb)condition 2 *p1 != *p2
(gdb)info b
Num Type Disp Enb Address What
1 breakpoint keep y 0x004010ae in main at gdb_example.c:12
2 breakpoint keep y 0x004010e7 in swap at gdb_example.c:21
stop only if *p1 != *p2
3 breakpoint del y 0x004010f9 in swap at gdb_example.c:24
(gdb)
```

Чтобы удалить условие прерывания, используется команда *condition* без аргумента *expression*:

```
(gdb)condition 2
```

Точка останова с номером 2 теперь безусловная.

Когда мы заканчиваем анализ состояния приостановленной программы, существует несколько способов возобновления. Мы можем пошагово выполнять нашу программу (*step* или *next*), можем позволить программе выполниться до следующей точки останова (*continue*) или до заданной позиции (*finish*)

Очевидно, что строки в программе не всегда выполняются в том порядке, в котором

они записаны в исходном коде. Например, представим, что следующая функция вычисляет факториал:

```
(gdb)list factorial
21
22 // factorial() calculates n!, the factorial of a nonnegative number n.
23 // For n > 0, n! is the product of all integers from 1 to n inclusive.
24 // 0! equals 1.
25
26 long double factorial( register unsigned int n )
27 {
28 long double f = 1;
29 while ( n > 1 )
30 f *= n--;
31 return f;
32 }
```

Следующий фрагмент сессии GDB показывает, что строки выполняются не в линейном порядке. Команда *frame* показывает, что выполнение программы приостановлено в 30-ой строке. Команда *step* выполняет 30-ую строку, а следующей будет выполняться 29-ая строка:

```
(gdb)frame
#0 factorial (n=10) at factorial.c:30
30 f *= n--;
(gdb)s
29 while ( n > 1 )
(gdb)
```

Причины такого порядка выполнения понятны: условие останова цикла проверяется всякий раз после тела цикла.

Если какая-то строка выполняется командой *step*, и она содержит вызов функции, а у GDB есть необходимая информация о символах и строках, то выполнение снова будет приостановлено на первой строке внутри вызванной функции. В следующем примере команда *step* заходит в функцию *factorial*(), а не в *printf*():

```
(gdb)frame
#0 main () at factorial.c:14
14 printf( "%u factorial is %.0Lf.\n", n, factorial(n) );
(gdb)s
factorial (n=10) at factorial.c:28
28 long double f = 1;
(gdb)
```

Функция *printf()* в отличие от *factorial()*, включалась в программу из стандартной библиотеки, которая компилировалась без отладочной информации. В результате GDB может отобразить содержимое *factorial()*, но не *printf()*.

Чтобы пропустить вызовы функций, можно воспользоваться командой *next*. Продемонстрируем отличие этих команд на том же примере:

```
(gdb)frame
#0 main () at factorial.c:14
14 printf( "%u factorial is %.0Lf.\n", n, factorial(n) );
(gdb)n
10 factorial is 3628800.
16 return 0;
(gdb)
```

На той же функции мы проиллюстрируем команду *finish*:

```
(gdb)s
14 printf( "%u factorial is %.0Lf.\n", n, factorial(n) );
(gdb)s
factorial (n=10) at factorial.c:28
28 long double f = 1;
```

```
(gdb)finish
Run till exit from #0 factorial (n=10) at factorial.c:28
0x0040112b in main () at factorial.c:14
14 printf( "%u factorial is %.0Lf.\n", n, factorial(n) );
Value returned is $2 = 3628800
(gdb)
```

Выполнение будет идти до завершения функции *factorial()*, если, конечно, до этого момента не встретится точка останова. Вызов *printf()*, тем не менее, к этому моменту не выполнялся.

Далее рассматриваются вопросы анализа стека и областей данных, использования так называемых точек просмотра и дампов ядра.

Анализ стека

Стек **вызовов**, который обычно называют просто **стеком**, это область памяти, организованная по принципу LIFO (*last in, first out*; последний пришел – первый ушел). Всякий раз, когда программа выполняет вызов функции, она создает в стеке структуру данных, которую принято называть **стековым фреймом**. Он содержит не только адрес вызывающей стороны и значения регистров, что позволяет программе передать управление вызывающей стороне после завершения функции, но также параметры функции и локальные переменные. Когда функция выполняет оператор *return*, память, занятая ее стековым фреймом, освобождается.

Когда отладчик приостанавливает программу, часто бывает полезно узнать, какая последовательность вызовов функций привела поток выполнения к текущей позиции. GDB предоставляет пользователям эту информацию в форме **трассировки вызовов**, которая показывает вызовы всех активных в текущий момент функций с их аргументами. Чтобы отобразить трассировку вызовов, мы можем использовать команду *backtrace* (короткая форма *bt*). У этой команды есть два синонима: *where* и *info stack* (или *info s*).

Следующий пример показывает трассировку, когда приведенная выше программа *circle.c* приостанавливается внутри функции *circularArea*():

```
(gdb)bt
#0 circularArea (r=5) at circle.c:30
#1 0x0040114c in main () at circle.c:18
```

Видно, что функция *circularArea*() вызвана из *main*() на 18-ой строке со значением аргумента, равным 5. Отладчик нумерует стековые фреймы от последнего к первому, так что фрейм текущей функции всегда равен 0. Очевидно, что наибольший номер у фрейма, соответствующего функции *main*().

Адрес, указанный после номера фрейма, это **адрес возврата**. Иными словами, это адрес инструкции, которая будет выполнена после выхода из функции, чей стековый фрейм был сгенерирован. Тем не менее, его не пишут, если он соответствует той же строке исходного кода, на которой была приостановлена программа.

Проиллюстрируем обратные трассировки с использованием рекурсивной функции *factorial()*:

```
long double factorial(register unsigned int n);

int main()
{
  unsigned int n = 10;

  /* ... */
  printf("n?: ");
  scanf("%i", &n);

  printf("n! = %d.\n", (unsigned int)factorial(n));

  /* ... */
  return 0;
```

```
}


// factorial() calculates n!, the factorial of a non-negative number n.
// For n > 0, n! is the product of all integers from 1 to n inclusive.
// 0! equals 1.
// Argument: A whole number, with type unsigned int.
// Return value: The factorial of the argument, with type long double.
long double factorial(register unsigned int n)
{
  long double f = 1;
  while ( n > 1 )
    f *= n--;
  return f;
}
```

Реализация кому-то может показаться своеобразной, но для демонстрации возможностей GDB этого должно хватить.

```
34 long double factorial( register unsigned int n)
35 {
36   if (n <= 1)
37     return 1.0L;
38   else
39     return n * factorial(n-1);
40 }
```

Следующий вывод GDB показывает стек вызовов во время последнего рекурсивного вызова функции *factorial*(). По определению факториала финальный вызов произойдет, когда *n* равен 1. Для приостановки программы на последнем шаге рекурсии, мы укажем условную точку останова, где условие – это выражение сравнения *n* == 1:

```
(gdb)b factorial if n == 1
...
(gdb)r
...
(gdb)bt
#0 factorial (n=1) at factorial.c:36
#1 0x0040117c in factorial (n=2) at factorial.c:39
#2 0x0040117c in factorial (n=3) at factorial.c:39
#3 0x0040117c in factorial (n=4) at factorial.c:39
#4 0x0040117c in factorial (n=5) at factorial.c:39
#5 0x0040112b in main () at factorial.c:14
(gdb)
```

Обратная трассировка в нашем примере показывает, что функция *main*() производит вычисления путем запроса значения 5! (факториал 5). Функция *factorial*() затем рекурсивно вызывается для того, чтобы вычислить 4!, потом 3!, далее 2!, и наконец 1!.

Большая часть команд манипуляций со стеком работают с **текущим стековым фреймом**. Например, мы адресуем локальные переменные в текущем фрейме по их именам. Когда доступно несколько фреймов, то GDB позволяет выбирать из них.

Когда отладчик приостанавливается в точке останова, текущим является фрейм, который соответствует исполняемой в данный момент функции, т.е. это фрейм с номером 0 в списке обратной трассировки. Команда *frame* позволяет отображать текущий стековый фрейм или выбрать другой:

```
frame [number]
```

Команда *frame* (сокращенно *f*) выбирает и отображает фрейм с заданным номером. Этот фрейм становится текущим. Понятно, что команда *frame* без аргументов просто показывает информацию о текущем стековом фрейме. Команда выдает две строки текста: имя вызванной функции с аргументами и номер текущей строки исходного текста, соответствующего этой функции.

В следующем примере программа *circle* приостановлена в функции *circularArea()*:

```
(gdb)bt
#0 circularArea (r=5) at circle.c:27
#1 0x0040114c in main() at circle.c:18
(gdb)f 1
#1 0x0040114c in main() at circle.c:18
18 area = circularArea(radius);
(gdb)p radius
$1 = 5
(gdb)
```

Команда *f 1* выбирает фрейм, который содержит вызов текущей функции. У нас это фрейм, соответствующий функции *main*(). Если стековый фрейм выбран, то можно получить доступ по именам к локальным переменным в *main*(), что и демонстрирует команда *p radius*.

У команды *info* есть три подкоманды, которые полезны для отображения содержимого стекового фрейма:

```
info frame
```

Показывает информацию о текущем фрейме, в том числе адрес возврата и сохраненные значения регистров.

```
info locals
```

Перечисляет локальные переменные с их текущими значениями в функции, соответствующей стековому фрейму.

```
info args
```

Перечисляет значения аргументов функции, соответствующей фрейму.

Как работают эти команды, покажем на примере функции *swap*().

```
(gdb)info frame
Stack level 0, frame at 0x22f010:
eip = 0x4010e7 in swap (gdb_example.c:27); saved eip 0x4010c0
called by frame at 0x22f030
source language c.
Arglist at 0x22f008, args: p1=0x22f024, p2=0x22f020
Locals at 0x22f008, Previous frame's sp is 0x22f010
Saved registers:
ebp at 0x22f008, eip at 0x22f00c
```

В регистре *%eip* содержится адрес следующей машинной инструкции, соответствующей строке 27. Регистр *%ebp* указывает на текущий стековый фрейм.

Команда *info args* выдает примерно следующее:

```
(gdb)info args
p1 = (int *) 0x22f024
p2 = (int *) 0x22f020
```

GDB помимо собственно значений аргументов так же отображает их тип (*int* *). Команда *info locals* отображает следующую информацию:

```
(gdb)info locals
tmp = 0
```

Отображение данных

Как правило, нам достаточно использовать команду *print* для отображения переменных и выражений. Дополнительно мы можем использовать команду *x* для проверки неименованных блоков памяти.

Команда *print* (или *p*) принимает в качестве аргумента любое выражение на языке реализации программы:

```
p [/format] [expression]
```

Эта команда вычисляет выражение *expression* и отображает результат. Команда *print* без аргумента *expression* выводит ранее вычисленное значение. При желании можно указать другой формат вывода.

Необязательный аргумент */format* позволяет выбрать подходящий формат выражения. Без этого аргумента *print* форматирует вывод, наиболее подходящий для заданного типа данных.

Выражения в командах *print* могут иметь побочные эффекты, как показано в следующем примере. Текущий стековый фрейм соответствует функции *circularArea()* в программе *circle*:

```
(gdb)p r
$1 = 1
(gdb)p r=7
$2 = 7
(gdb)p r*r
$3 = 49
```

Здесь выражение *r=7* во второй команде *print* присваивает значение 7 переменной *r*. Можно изменить значение этой переменной с использованием команды *set*:

```
(gdb)set variable r=1.5
```

Команда *print* выдаст значение выражение в виде переменной *$i*, где *i* – это положительное целое число. Мы можем сослаться на такие переменные в последующих командах, например:

```
(gdb)p 2*circularArea($2)
$4 = 307.87608005280003
```

Эта команда вызывает функцию *circularArea()* с аргументом $2 (а это 7 согласно предыдущему примеру), потом умножает возвращенное значение на 2 и сохраняет результат в новой переменной ($4).

Мы можем использовать команды *p* и *set* для определения новых переменных в GDB с именами, начинающимися со знака доллара. Например, команда *set $var = *ptr* создает переменную с именем *$var* и присваивает ей значение, на которое указывает *ptr*. The Переменные отладчика отделены от переменных отлаживаемой программы. GDB также хранит значения регистров ЦП в переменных с именами, начинающимися с префикса в виде знака доллара.

Для доступа к переменным из других стековых фреймов без смены текущего фрейма используется префикс в виде имени функции и двойного двоеточия (::):

```
(gdb)p main::radius
$8 = 1
```

Необязательный аргумент */format* команды *print* состоит из косой черты и последующей литеры, специфицирующей формат вывода. Допустимые литеры в основном похожи на спецификаторы строки формата функции *printf*. Например, команда *print /x* отображает значение в 16-ричной нотации.

Команда *print* при необходимости преобразует значение к подходящему типу. Для целочисленных значений могут использоваться следующие форматы:

d	Десятичная нотация. Используется по умолчанию
u	Десятичная нотация, но значение интерпретируется как беззнаковое
x	Шестнадцатеричная нотация.
o	Восьмеричная нотация.
t	Двоичная нотация

c	Символьная нотация, вместе с десятичным кодом символа

Следующий пример иллюстрирует, что опция формата может идти непосредственно за командой *p* без дополнительных пробелов:

```
(gdb)p/x 65
$10 = 0x41
(gdb)p/t
$11 = 1000001
(gdb)p/c
$12 = 65 'A'
(gdb)p/u -1
$13 = 4294967295
```

Все команды *print* без аргумента-выражения в этом примере показывают то же значение, что и предыдущая команда, но с другим форматированием.

Команда *print* принимает еще две опции для нецелочисленных выражений:

a	Отображает адрес и 16-ой нотации, вместе со смещение относительно ближайшего именованного адреса, ниже в памяти
f	Интерпретирует выражение к число с плавающей точкой и отображает его

Вот примеры:

```
(gdb)p/a 0x401100
$14 = 0x401100 <swap+31>
(gdb)p/f 123.0
$15 = 123
(gdb)p/f 123
$16 = 1.72359711e-43
```

Команда *x* позволяет проверять неименованные блоки памяти. Аргументы команды включают начальный адрес блок и размер, а также формат вывода:

```
x [/nfu] [address]
```

Команда отображает содержимое блока памяти, начинающегося с заданного адреса, указанного размера. Формат отображения задается опцией */nfu*. Аргумент *address* может быть любым выражением, дающим корректный адрес. Если его не указать, то команда *x* отображает блок памяти, следующий за последней ячейкой памяти, отображенной командами *x* или *print*.

Аргумент */nfu* может состоять из 3-х необязательных частей.

n	Десятичное число, показывающее, как много блоков памяти отображать. Размер каждого такого блока определяется третьей частью опции */nfu*,т.е. *u*. По умолчанию, *n* равно 1.
f	Спецификатор формата, аналогичный используемым с командой *print,* дополненных еще двумя спецификаторами: - *s* Данные с заданным адресом трактуются как завершаемая нулем строка - *i* Машинные инструкции отображаются в виде ассемблерных команд. По умолчанию изначально используется формат *x*, но чаще тот, который был задан последней выполненной командой *x* или *print*
u	Третья часть аргумента */nfu*, определяет размер блока памяти и может принимать одно из следующих значений: - *b* Один байт - *h* Два байта («полуслово») - *w* Четыре байта («слово») - *g* Восемь байт («гигантское слово») По умолчанию изначально используется *w*, но далее может изменен. Это делает

	бессмысленным определения размера блока с опциями формата *s* или *i*. Если же мы их укажем, то GDB будет игнорировать опцию размера блока.

Следующий пример показывает использование команды *x*. Предполагается, что переменные объявлены в текущей области видимости:

```
char msg[100] = "Hello world!\n";
char* cPtr = msg + 6;
```

Каждая строка вывода команды *x* начинается с базового адреса ячейки памяти и соответствующего имени из символьной таблицы, если таковые имеются. Первая команда *x* показывает строку *msg*:

```
(gdb)x/s msg
0x402000 <msg>: "Hello world!\n"
```

Следующая команда отображает первые 15 байт строки *msg* в 16-ной системе:

```
(gdb)x/15xb msg
0x402000 <msg>: 0x48 0x65 0x6c 0x6c 0x6f 0x20 0x77 0x6f
0x402008 <msg+8>: 0x72 0x6c 0x64 0x21 0x0a 0x00 0x00
```

Два 32-битных слова в 16-ной нотации по адресу *msg*:

```
(gdb)x/2xw msg
0x402000 <msg>: 0x6c6c6548 0x6f77206f
```

Строка, которая начинается с указателя *cPtr*:

```
(gdb)x/s cPtr
0x402006 <msg+6>: "world!\n"
```

Начиная с того же адреса, восемь десятичных кодов и самих символов:

```
(gdb)x/8cb cPtr
0x402006 <msg+6>: 119 'w' 111 'o' 114 'r' 108 'l' 100 'd' 33 '!' 10 '\n' 0 '\0'
```

Значение указателя *cPtr* в 16-ной и 2-ной нотациях:

```
(gdb)x/a &cPtr
0x22f00c: 0x402006 <msg+6>
(gdb)x/tw &cPtr
0x22f00c: 00000000010000000010000000000110
```

Точки просмотра: наблюдение за операциями над переменными

Отладчик GDB позволяет получать уведомления о чтении или изменении переменных путем установки **точки просмотра**. Она похожа на точку останова, за исключением того, что это не ограничивается одной строкой исходного кода. Если мы устанавливаем точку просмотра для переменной, то GDB приостановит программу, когда значение переменной изменяется. Фактически GDB позволяет наблюдать не только за отдельными переменными, но и выражениями. Можно указывать различные виды точек просмотра с использованием команд *watch*, *rwatch* и *awatch*.

У всех одинаковый синтаксис:

```
watch expression
```

Отладчик приостанавливает программу, когда значение выражения изменяется.

```
rwatch expression
```

Отладчик приостанавливает программу, когда она читает значение любого объекта вовлеченного в вычисление выражения.

```
awatch expression
```

Отладчик приостанавливает программу, когда она читает или модифицирует значение любого объекта вовлеченного в вычисление выражения.

Обычное использование точек просмотра – это наблюдение за тем, как программа модифицирует переменную. Когда наблюдаемая переменная меняется, GDB показывает ее старое и новое значения, а также строку, содержащую следующий исполняемый оператор. Для иллюстрации мы используем программу со следующим кодом.

```
#include <stdio.h>

int main()
{
  int a = 10;
  int b = 20;
  int* iPtr = &a;

  ++*iPtr;
  puts("This is the statement following ++*iPtr.");

  printf("a = %d; b = %d.\n", a, b);
  return 0;
}
```

Перед тем, как установить точку просмотра локальной переменной, мы должны начать выполнение программы до тех пор пока поток инструкций не попадет в области видимости желаемой переменной. По этой причине мы стартуем программу до обычной точки останова на строке 9.

```
(gdb)b 9
Breakpoint 1 at 0x4010ba: file ptr.c, line 9.
(gdb)r
Starting program: ...
Breakpoint 1, main () at ptr.c:9
9 ++*iPtr;
```

Теперь можно установить точку просмотра переменной *a* и продолжить выполнение:

```
(gdb)watch a
Hardware watchpoint 2: a
(gdb)c
Continuing.
Hardware watchpoint 2: a
Old value = 10
New value = 11
main () at ptr.c:10
10 puts( "This is the statement following ++*iPtr." );
```

Поскольку *iPtr* указывает на *a*, то выражение *++*iPtr* изменяет переменную *a*. В результате отладчик приостанавливает программу после этой операции и выводит следующий исполняемый оператор.

Чтобы продолжить наш пример, мы установим «точку чтения» переменной *b*. Точки просмотра включаются в список точек останова, которые можно просматривать с помощью команды *info breakpoints*(или *info b*):

```
(gdb)rwatch b
Hardware read watchpoint 3: b
(gdb)info b
Num Type Disp Enb Address What
1 breakpoint keep y 0x004010ba in main at ptr.c:9
breakpoint already hit 1 time
2 hw watchpoint keep y a
breakpoint already hit 1 time
3 read watchpoint keep y b
(gdb)c
Continuing.
This is the statement following ++*iPtr.
```

```
Hardware read watchpoint 3: b
Value = 20
0x004010ce in main () at ptr.c:12
12 printf( "a = %d; b = %d.\n", a, b );
(gdb)c
Continuing.
a = 11; b = 20.
...
```

Когда программа выходит из блока (т.е. поток управления прошел закрывающую фигурную скобку), отладчик автоматически удаляет все точки просмотра выражений, использующих локальные переменные, которые теперь вне области видимости.

Теперь попробуем посмотреть, как отладчик поступает, когда мы ставим точку просмотра выражения с несколькими переменными. GDB приостанавливает программу всякий раз, когда она получает доступ к любой переменной в заданном выражении. В следующей отладочной сессии, мы перезапустим программу из предыдущих примеров. Когда она остановится в точке на 9-ой строке, мы установим «точку чтения» выражения *a* + *b*:

```
(gdb) b 9
Breakpoint 1 at 0x4010ba: file myprog2.c, line 9.
(gdb) r
Starting program: ...
Breakpoint 1, main () at myprog2.c:9
9 ++*iPtr;
(gdb) rwatch a+b
Hardware read watchpoint 2: a + b
```

Если мы теперь дадим программе возобновить выполнение, то отладчик остановит ее на следующем операторе, который читает переменные *a* или *b*. У нас это оператор вызова функции *printf* на строке 12. Поскольку там считываются и *a*, и *b*, то отладчик остановит программу дважды, выдавая вычисленное значение выражения *a* + *b*.

```
(gdb) c
Continuing.
This is the statement following ++*iPtr.
Hardware read watchpoint 2: a + b
Value = 31
0x004010ce in main () at myprog2.c:12
12 printf( "a = %d; b = %d.\n", a, b );
(gdb) c
Continuing.
Hardware read watchpoint 2: a + b
Value = 31
0x004010d5 in main () at myprog2.c:12
12 printf( "a = %d; b = %d.\n", a, b );
(gdb) c
Continuing.
a = 11; b = 20.
...
```

Анализ дампов ядра в GDB

Файл ядра (или **дамп ядра**) – это файл, содержащий образ памяти, используемой процессом. Unix-подобные системы обычно пишут дамп ядра в рабочем каталоге, когда процесс завершается с ошибкой (В Unix-системах можно прочитать, какие сигналы соответствуют дампу ядра, в *man signal*). Передавая имя файла дампа отладчику GDB через командную строку, мы можем узнать состояние процесса в тот момент, когда он был прерван.

Сессия работы с GDB для анализа дампов ядра похожа на обычную отладочную сессию, за исключением того, что наша программа уже приостановлена в некоторой позиции, и мы не можем использовать команды *run*, *step*, *next* или *continue*, чтобы сделать это заново. Прогуляемся по примеру такой «посмертной» сессии, чтобы показать, как

отлаживать программы с использованием других команд GDB. Представим, что программа *myprog*, расположенная в текущем каталоге, была скомпилирована с опцией *-g*. Следующая команда запускает программу:

```
$./myprog
Segmentation fault (core dumped)
```

Сообщение об ошибке подсказывает нам, что *myprog* прервана из-за некорректного обращения к памяти. Система сгенерировала дамп ядра в текущем каталоге в файле с именем *core*. Для анализа ошибки запустим GDB, передав ему имя дампа и исполняемого файла в командной строке. В начале отладчик сразу же отображает адрес и функцию, в которой программа была прервана:

```
$gdb myprog core
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
...
Core was generated by `./myprog'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x4008ff06 in strcpy () from /lib/tls/libc.so.6
(gdb)
```

Последняя выведенная до подсказки отладчика строка показывает, что ошибка имела место в функции *strcpy*(). Если мы предположим, что *strcpy*() сама по себе свободна от «багов», то *myprog* совершила ошибку при ее вызове. Команда *backtrace* покажет вызовы функций, которые привели к текущей точке программы:

```
(gdb)bt
#0 0x4008ff06 in strcpy () from /lib/tls/libc.so.6
#1 0x080483f3 in main () at myprog.c:13
(gdb)
```

Вывод показывает, что вызов strcpy() произошел на строке 13 файла *myprog.c*, в функции *main()*. Выше мы рассмотрели тонкости команды *backtrace*. Выберем стековый фрейм с номером 1, т.к. предположительно мы совершили ошибку в функции *main()*:

```
(gdb)f 1
#1 0x080483f3 in main () at myprog.c:13
13 strcpy( name, "Alejandro" );
```

Поскольку вызов функции показал, что второй аргумент для *strcpy()* – это строковый литерал, то мы вправе предположить, что другой аргумент – *name* – это неверный указатель. Чтобы проверить это, используем команду *print*:

```
(gdb)p name
$1 = 0x0
(gdb)
```

Значение *name* – нуль: *myprog* «падает», пытаясь совершить запись по нулевому указателю. Для исправления этого «бага» мы должны гарантировать, что *name* указывает на символьный массив достаточного размера, чтобы хранить строку, копируемую в него.

Для того чтобы дамп ядра генерировался во время выполнения, необходимо сначала проверить системные настройки размера дампа ядра, выполнив следующую команду:

```
$ ulimit -c
$0
```

Если ответ именно такой (нулевой размер файла), то его можно изменить, сняв

ограничения:

```
$ ulimit -c unlimited
$
```

Литература и дополнительные источники

1. “Debugging with GDB” by the Free Software Foundation - http://www.gnu.org/software/gdb/documentation/
2. Download GDB - http://www.gnu.org/software/gdb/download/
3. Data Display Debugger - http://www.gnu.org/software/ddd/
4. Знакомство с отладчиком gdb - http://www.linuxcenter.ru/lib/books/linuxdev/linuxdev9.phtml
5. Отладчик GDB - http://docstore.mik.ua/manuals/ru/linux_base/node199.html
6. Linux / Unix Command: ulimit - http://linux.about.com/library/cmd/blcmdl1_ulimit.htm