Кафедра вычислительных систем

Практические задания по дисциплине «Программирование»

семестр 2

2014/2015 учебный год

Подготовил:
Старший преподаватель Кафедры ВС – Пименов Евгений Сергеевич

Лабораторная работа №1

Отладка программ

ЦЕЛЬ РАБОТЫ

Получение навыков отладки программ на примере использования отладчика GDB.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Введение

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки.

Отладчик — компьютерная программа, предназначенная для поиска ошибок в других программах, ядрах операционных систем, SQL-запросах и других видах кода.

Компиляция программы для отладки

Для отладки программы, при компиляции необходимо сгенерировать отладочную информацию. Эта информация сохраняется в объектном файле; она описывает тип данных каждой переменной, сигнатуры функций, соответствие между номерами строк исходного текста и машинного кода и т.д.

Для определения уровня отладочной информации в файле используются опции компилятора, приведенные в таблице 1:

Таблица 1

Опция	Описание
-g	добавляет в выходной файл отладочную информацию в поддерживаемом операционной системой формате (stabs, COFF, XCOFF, DWARF2)
-g0	без отладочной информации
-g2	уровень по умолчанию, то же самое, что -g.
-g1	минимальный уровень информации, достаточный для просмотра backtrace
-g3	включение дополнительной информации, например, определения макросов

Также необходимо скомпилировать программу без оптимизации. Для управления уровнем оптимизации используются опции, представленные в таблице 2.

Таблица 2

Опция	Описание
-O0	Без оптимизации
-O1, -O2, -O3	Различные уровни оптимизации
-Os	Оптимизация по размеру кода
-Ofast	Все оптимизации уровня –O3, плюс дополнительные

Оптимизатор может перестроить код таким образом, что будет нарушено соответствие исходного кода фактически выполняющимся инструкциям. Рассмотрим следующий пример.

Код на языке C	gcc –O0 –S main.c
`int a = 10;`	`movl $10, -12(%rbp)`
`int b = 20;`	`movl $20, -8(%rbp)`

`int c = a + b;`	`movl -8(%rbp), %eax`
	`movl -12(%rbp), %edx`
	`addl %edx, %eax`
	`movl %eax, -4(%rbp)`

`printf("c = %d\n", c);`	`movl $.LC0, %eax`
	`movl -4(%rbp), %edx`
	`movl %edx, %esi`
	`movq %rax, %rdi`
	`movl $0, %eax`
	`call printf`

В приведенном примере был получен ассемблерный код без оптимизации. В нем можно установить однозначное соответствие между кодом на языке C и ассемблерными инструкциями.

Рассмотрим ассемблерный код, полученный с включенной оптимизацией первой степени:

Код на языке C	gcc –O1 –S main.c
`int a = 10;`	
`int b = 20;`	
`int c = a + b;`	`movl $30, %edx`

`printf("c = %d\n", c);`	`movl $.LC0, %esi`
	`movl $1, %edi`
	`movl $0, %eax`
	`call __printf_chk`

Компилятором были исключены определения переменных a и b. Результат операции сложения был вычислен на этапе компиляции.

Рассмотрим (возможный) ассемблерный код, полученный с включенной оптимизацией третьей степени:

Код на языке C	gcc –O3 –S main.c
`int a = 10;`	
`int b = 20;`	
`printf("c = %d\n", c);`	`movl $1, %edi` `movl $0, %eax`
`int c = a + b;`	`movl $30, %edx`
`printf("c = %d\n", c);`	`movl $.LC0, %esi` `call __printf_chk`

Помимо модификаций, введенных при оптимизации первого уровня, был изменен порядок выполнения элементарных операций процессора так, чтобы более интенсивно использовать его ресурсы. При этом оптимизация была выполнена так, чтобы результат выполнения операций не нарушен. При отладке такой программы последовательность выполнения операций языка Си не будет соответствовать программе.

Основные функции отладчика

Пошаговое выполнение программы

Основная цель использования отладчика – остановка программы до ее завершения и определение причины неисправности.

Для остановки выполнения программы используются точки останова (breakpoints), точки наблюдения (watchpoints) и точки перехвата (catchpoints). В рамках данной лабораторной работы будет рассмотрен только первый класс (breakpoints). *Точка останова (breakpoint)* останавливает выполнение программы каждый раз, когда ее выполнение доходит до определенной точки. Для установки точки останова служит команда **break**, которая может использоваться в одной из нескольких форм:

```
break <имя_функции>
(gdb) break main

break <номер_строки>
(gdb) break 35

break <имя_файла>:<номер_строки>
(gdb) break source/file.c:35

break <имя_файла>:<имя_функции>
(gdb) break source/file.c:parse_arguments
```

Получить список всех точек останова можно командой **info breakpoints**.

После останова программы часто бывает необходимо произвести ее пошаговое исполнение. Для этого применяются команды **next** (**n**) и **step** (**s**).

1. Команда **next** осуществляет переход к следующему оператору текущей функции или (если в данной функции больше нет операторов) к следующему оператору вызывающей функции. Заход в вызываемые функции данная программа не предусматривает.
2. Команда **step** работает аналогично **next**, однако, если исполняемый оператор является вызовом функции, то будет осуществлен заход внутрь нее.

Рассмотрим работу этих команд на примере:

Отлаживаемая программа:

```
#include <stdio.h>
int f(int x){
 int y = x * 10;
 return y;
}
int main(){
 int k = 5, l;
 l = f(k);
 printf("l = %d\n",l);
 return 0;
}
```

Использование команды **next**

```
(gdb) break main
(gdb) run
9   int k = 5, l;
(gdb) n
10  l = f(k);
(gdb) n
11  printf("l = %d\n",l);
(gdb) n
l = 50
12  return 0;
```

Использование команды **step**

```
(gdb) b main
(gdb) r
9  int k = 5, l;
(gdb) step
10 l = f(k);
(gdb) s
f (x=5) at step_in_over.cpp:4
4  int y = x * 10;
(gdb) s
5  return y;
(gdb) s
6  }
(gdb) s
main () at step_in_over.cpp:11
11  printf("l = %d\n",l);
(gdb) s
l = 50
12  return 0;
(gdb) s
13     }
```

Просмотр исходного кода

Для просмотра листинга исходного кода применяется команда **list** (или сокращенно **l**) Она может быть вызвана одним из следующих способов:

list – отобразить 5 строк исходного кода перед и 5 строк после текущей позиции:

```
(gdb) list
1   #include <stdio.h>
2   int f(int x)
3   {
4       int y = x * 10;
5       return y;
6   }
7
8   int main(){
9       int k = 5, l;
10      l = f(k);
```

list <номер_строки> отобразить исходный код вокруг заданной строки:

```
(gdb) list 9
4       int y = x * 10;
5       return y;
6   }
7
8   int main(){
9       int k = 5, l;
10      l = f(k);
11      printf("l = %d\n",l);
12      return 0;
13  }
```

list <имя_функции> отобразить исходный код вокруг заданной функции:

```
(gdb) l main
3   {
4       int y = x * 10;
5       return y;
6   }
7
8   int main(){
9       int k = 5, l;
10      l = f(k);
11      printf("l = %d\n",l);
12      return 0;
```

Для многофайловых программ можно указывать файл, в котором находится строка или функция:

```
list <имя_файла>:<номер_строки>
list <имя_файла>:<имя_функции>
```

Просмотр значений переменных

Для просмотра значения, хранящегося в некоторой переменной можно использовать команды **print** и **display**.

Команда **print** в качестве аргумента принимает значение выражение, написанного на том же языке, что и программа, вычисляет и выводит его значение.

Команда **display** может быть использована в случаях, когда необходимо часто выводить значение некоторого выражения. С помощью этой команды выражение добавляется в список автоматического отображения, и значение выражения выводится при каждой остановке программы.

Передача аргументов командной строки

При отладке программы аргументы командной строки могут быть переданы несколькими способами.

1. При запуске отладчика
   $ gdb --args ./prog arg1 arg2 arg3
2. При запуске программы в отладчике:
   (gdb) run arg1 arg2 arg3
3. Командой **set args**:
   (gdb) set args “arg 1 arg2 arg3”

Для просмотра аргументов, переданных в программу, используется команда **show args**

Примеры отладки программ

Рассмотрим применение приведенных команд а примерах.

Пример 1

```
#include <stdio.h>
#define N 10
int main()
{
    int arr[N];
    unsigned i;
    for (i = N – 1; i >= 0; --i) {
        arr[i] = i;
        printf(“%d ”, arr[i]);
    }
    return 0;
}
```

Предполагается, что программа должна выполнить некоторую операцию с каждым элементом массива arr, обходя массив в порядке от последнего элемента к первому. Для примера выбрано заполнение элементов массива их номерами и вывод на экран. Скомпилируем и запустим программу:

```
$ gcc –Wall –o example main.c
$ ./example
Bus error (core dumped)
```

Программа компилируется без ошибок и предупреждений, но ее выполнение завершается аварийно. Для определения причины неисправности перекомпилируем программу без оптимизации и с отладочной информацией.

```
$ gcc –Wall –g –O0 –o example main.c
```

Запустим ее в отладчике.

```
$ gdb ./example
(gdb) run
Program received signal SIGBUS, Bus error.
0x0000000000400509 in main () at main.c:11
11              arr[i] = i;
```

Отладчик остановил выполнение программы в момент получения сигнала SIGBUS и указал, что выполнение остановлено на 11 строке при попытке присваивания i-тому элементу массива значения i. В этот момент мы можем воспользоваться командой **print**:

```
(gdb) print i
$1 = 4294967295
```

Очевидно, что обращение к элементу с индексом 4294967295 в массиве из 10 элементов некорректно и приводит к «падению» программы. Для того, чтобы точнее определить в какой момент переменная i принимает некорректное значение, выполним следующие действия:

1) Установим точку останова на функции main
2) Добавим в список автоматического отображения переменную i
3) Выполним программу пошагово, отслеживая значение переменной i

Продолжение сессии отладчика представлено ниже:

```
(gdb) break main
(gdb) run
(gdb) Breakpoint 1, main () at main.c:10
  10 for (i = N - 1; i >= 0; --i) {
(gdb) display i
(gdb) next
. . .
(gdb) next
10          for (i = N - 1; i >= 0; --i) {
1: i = 0
(gdb) next
1: i = 4294967295
```

Становится очевидно, что цикл некорректен, поскольку за итерацией, когда i = 0 декремент приводит к переполнению значения i, условие продолжения цикла снова становится истинно. Для решения этой проблемы следует сменить тип счетчика с unsigned на int.

Пример 2

```
#include <stdio.h>
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int a = 0;
    int b = 1;
    printf("a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

Предполагается, что функция swap должна произвести обмен значениями переменных a и b. Скомпилировав и запустив программу, легко убедиться, что этого не происходит:

```
$ gcc -Wall swap.c -o swap_example
$ ./swap_example
a = 0, b = 1
```

Установим точку останова: на функции main.

```
(gdb) b main
Breakpoint 2 at 0x400553:
```

Запустим программу и начнем пошаговое выполнеие (команда next):

```
(gdb) r
Breakpoint 2, main () at main.c:12
    int a = 0;
(gdb) n
    int b = 1;
(gdb) n
    swap(a, b);
```

В точке вызова функции swap выведем адреса переменных a и b

```
(gdb) p &a
$3 = (int *) 0x7fffffffdd18
(gdb) p &b
$4 = (int *) 0x7fffffffdd1c
```

Выполним вход в функцию swap (команда step)

```
(gdb) s
```

```
swap (a=0, b=1) at main.c:5
5    int tmp = a;
```

Выведем адреса переменных a и b в функции swap

```
(gdb) p &a
$5 = (int *) 0x7fffffffdcec
(gdb) p &b
$6 = (int *) 0x7fffffffdce8
```

Становится очевидно, что функция swap меняет значениями локальные переменные a и b, не затрагивая переменных в вызывающем коде. Для исправления этой ошибки следует передавать не значения, а их адреса, при этом в функции swap принимать указатели – int*.

Пример 3

Отладчик может быть полезен в изучении поведения сложных макросов. Для этого необходимо скомпилировать программу с опцией –g3 и воспользоваться командой отладчика macro expand.

```
#include <stdio.h>
#define QUOTE_(WHAT) #WHAT
#define QUOTE(WHAT) QUOTE_(WHAT)
#define CSC_TRACE(format, ...) \
    printf("%s: "format, __FILE__":"QUOTE(__LINE__), \
    ## __VA_ARGS__)
void f()
{
    int answer = 42;
    CSC_TRACE("the answer is %d\n", answer);
}
int main()
{
    f();
    return 0;
}
```

Gdb позволяет отобразить во что фактически будет развернут макрос CSC_TRACE:

```
(gdb) break main
Breakpoint 1 at 0x400523
(gdb) run
Breakpoint 1, main () at csc_trace.c:16
16          f();
(gdb) macro expand CSC_TRACE("Hello")
expands to: printf("%s: ""Hello",__FILE__":"__LINE__")
```

Пример 4

```
#include <stdio.h>
#include <stdlib.h>
int char_cmp(const void* a, const void* b)
{
    return *(const char*)a - *(const char*)b;
}
int main()
{
    char greeting[] = "Hello!";
    qsort(greeting, sizeof(greeting),
        sizeof(*greeting), char_cmp);
    printf("%s\n", greeting);
    return 0;
}
```

Ожидается, что будет выведена строка, символы в которой отсортированы. Фактически не будет выведено ничего. Ниже приведена сессия в отладчике, в которой демонстрируется состояние строки после сортировки:

```
(gdb) b 14
```

```
Breakpoint 1 at 0x40061f: file sort_string.c, line 14.
(gdb) run
Breakpoint 1, main () at sort_string.c:14
14          printf("%s\n", greeting);
(gdb) print greeting
$1 = "\000!Hello"
```

ЗАДАНИЯ

Общая информация

В приведенных программах содержатся ошибки. Необходимо с помощью отладчика локализовать и исправить их.

Задание 1

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

void init(int* arr, int n)
{
    arr = malloc(n * sizeof(int));
    int i;
    for (i = 0; i < n; ++i)
    {
        arr[i] = i;
    }
}

int main()
{
    int* arr = NULL;
    int n = 10;
    init(arr, n);

    int i;
    for (i = 0; i < n; ++i)
    {
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

Задание 2

```
#include <stdio.h>

typedef struct
{
    char str[3];
    int num;
} NumberRepr;

void format(NumberRepr* number)
{
    sprintf(number->str, "%3d", number->num);
}

int main()
{
    NumberRepr number = { .num = 1025 };

    format(&number);

    printf("str: %s\n", number.str);
    printf("num: %d\n", number.num);

    return 0;
}
```

Задание 3

```
#include <stdio.h>

#define SQR(x) x * x

int main()
{
    int y = 5;
    int z = SQR(y + 1);
    printf("z = %d\n", z);
    return 0;
}
```

Задание 4

```
#include <stdio.h>

void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void bubble_sort(int* array, int size)
{
    int i, j;
    for (i = 0; i < size - 1; ++i) {
        for (j = 0; j < size - i; ++j) {
            if (array[j] > array[j + 1]) {
                swap(&array[j], &array[j + 1]);
            }
        }
    }
}


int main()
{
    int array[100] = {10, 15, 5, 4, 21, 7};

    bubble_sort(array, 6);

    int i;
    for (i = 0; i < 6 ; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```