ЛЕКЦИЯ 2
Отладка программного обеспечения

Мамойленко Сергей Николаевич
Кафедра вычислительных систем

ГОСТ 19.102-77 «Стадии разработки»*:
Этап «Рабочий проект»
Процессы: разработка программного обеспечения, его отладка, создание программной документации и проведение испытаний

*) http://www.rugost.com/index.php?option=com_content&view=article&id=49:19102-77&catid=19&Itemid=50

Тестирование — это последовательность действий, направленных на обнаружение несоответствий функционирования программного обеспечения требованиям технического задания на его разработку.

Конкретные причины несоответствий определяются в **процедуре локализации и исправления ошибок**.

Испытание – это процесс демонстрации программного обеспечения с целью фиксации факта выполнения им всех функций, определённых в техническом задании.

Термин баг (англ. bug —насекомое) применяется для обозначения ошибки в программном обеспечении. Считается, что этот **термин** в компьютерной индустрии **появился в 1945** году благодаря учёной из Гарвардского университета Грейс Хоппер, которая во время тестирования вычислительной машины Mark II Aiken Relay Calculator, нашла в ней мотылька, застрявшего между контактами электромеханического реле. Извлечённое насекомое было вклеено скотчем в технический дневник, с сопроводительной надписью: «First actual case of bug being found».

- механизм исключений;
- журналирование (вывод) текущего состояния программного обеспечения;
- применение отладчиков.

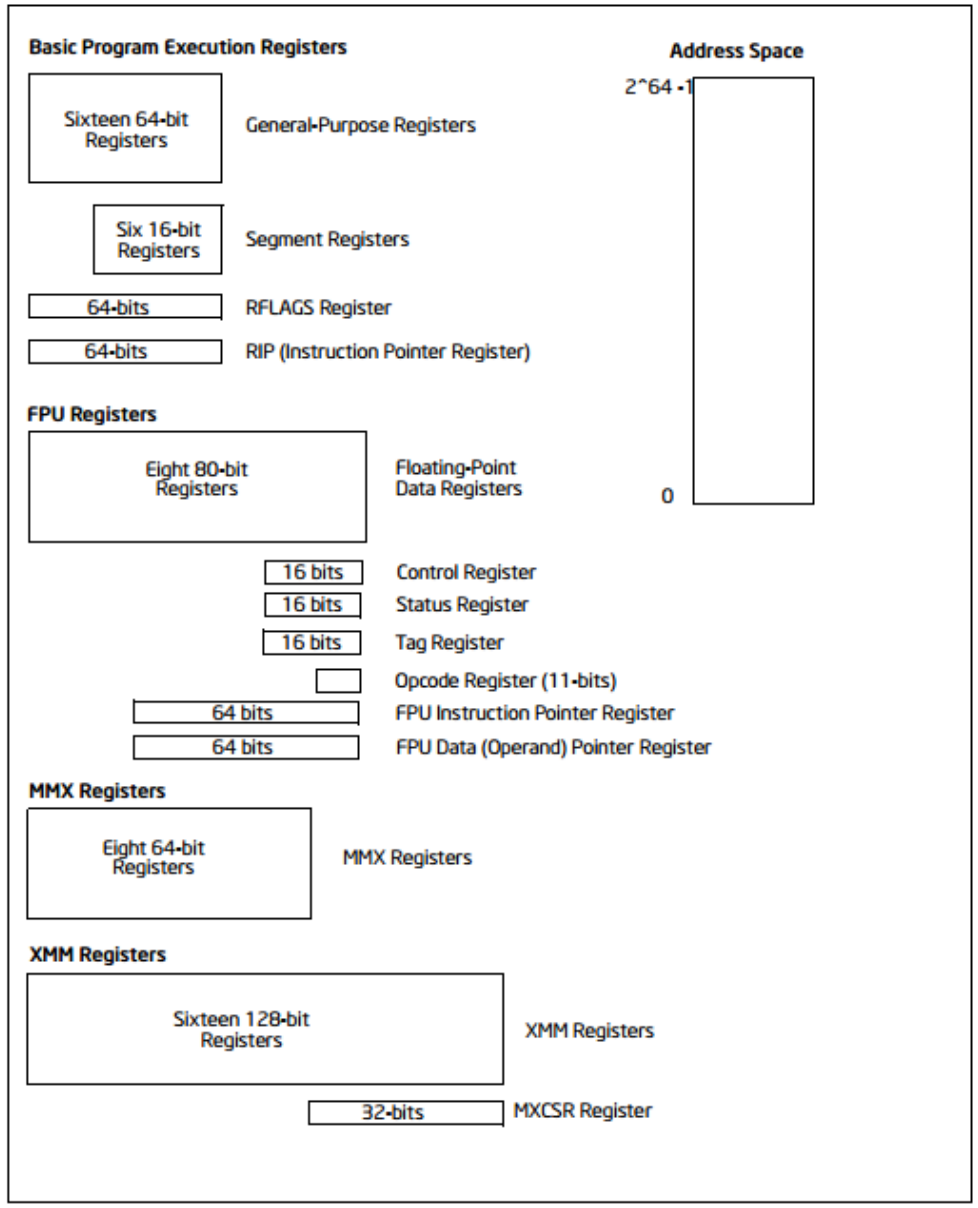


Figure 3-2. 64-Bit Mode Execution Environment

*) Intel 64 and IA-32 Architectures Software Developer's Manual

Во всех трех случаях разработчик программного обеспечения в процессе его отладки контролирует состояние вычислительных ресурсов:

- Регистров процессора (общего назначения, специальных, сегментных, флагов и т.п.);
- Оперативной памяти;
- Состояния внешних устройств.

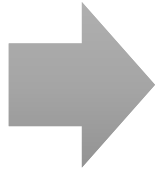
Может ли программное обеспечение вывести информацию о любых ресурсах системы?
Да, в случае однозадачной однопользовательской системы.
В других случаях не всегда.

Информацию о каких ресурсы, например, не может вывести процесс?
Состояние ядра операционной системы, состояние внешних устройств, состояние других процессов и т.п.

Почему нельзя предоставить процессам полный доступ к информации о ресурсах (и самим ресурсам) друг друга?
Вирусы, защита информации, защита от дурака и т.п.

Как же быть?
Неужели можно производить только «самоотладку» процесса?

Хочу управлять процессом 1234



GDB использует системные вызовы для управления процессами (нитями) и изменения их ресурсов.
В GNU\Linux используется вызов **ptrace**(), в Windows – **DebugActiveProcess**() и т.п.

#include <sys/ptrace.h>

long **ptrace** (request, pid, * addr, * data) – управление зависимым (трассируемым) процессом

Параметр	Тип	Назначение
request	enum __ptrace_request	Необходимое действие
pid	pid_t	Номер трассируемого процесса
addr	void *	Параметр 1 (зависит от действия)
data	void *	Параметр 2 (зависит от действия)

По сути, ptrace – это обычный системный вызов (прерывание), которые просит у операционной системы выполнить какие-то действия над другим процессом.

Действия, которые может выполнять вызов ptrace:

- **PTRACE_TRACEME** – включение трассировки родительским процессом;
- **PTRACE_ATTACH/PTRACE_DETACH** – подключение/отключение к другому процессу для трассировки.
- **PTRACE_CONT** – продолжить выполнение трассируемого процесса;
- **PTRACE_SINGLESTEP** – выполнить один шаг трассируемого процесса;
- **PTRACE_GETREGS/PTRACE_SETREGS** – получить/установить значения регистров процессора для трассируемого процесса;
- **PTRACE_PEEKTEXT/PTRACE_POKETEXT** – получить/установить значение памяти трассируемого значения;
- **PTRACE_GETSIGINFO/PTRACE_SETSIGINFO** – получить/установить информацию о системном вызове (прерывании), который вызвал событие трассировки.
- **PTRACE_SETOPTIONS** – установить значения параметров трассировки (обработки вызовом exec, clone, kill и т.п.)

```
child = fork ();
 if (child == 0) {
     ptrace (PTRACE_TRACEME, 0, NULL, NULL);
     execl ("/bin/ls", "ls", "-l", NULL);
 } else {
   while (1) {
     wait (&status);
     if (WIFEXITED (status)) { printf ("Exited\n"); break; }
     ptrace (PTRACE_GETREGS, child, NULL, &regs);
     printf ("Trace system call = %lld\n", regs.orig_rax);
     ptrace (PTRACE_SYSCALL, child, NULL, NULL);
   }
}
```

Рисунок - Пример работы с функцией ptrace. Вывод информации обо всех системных вызовах трассируемого процесса.

```
#include <sys/user.h>
#include <sys/reg.h>
…
struct user_regs_struct regs;
long long ins;
…
ptrace(PTRACE_ATTACH, traced_process, NULL, NULL);
wait(NULL);
ptrace(PTRACE_GETREGS, traced_process, NULL, &regs);
ins = ptrace(PTRACE_PEEKTEXT, traced_process, regs.rip, NULL);
printf("RIP: %llx Instruction executed: %llx\n", regs.rip, ins);
ptrace(PTRACE_DETACH, traced_process, NULL, NULL);
```

Рисунок - Пример работы с функцией ptrace. Подключение к уже существующему процессу и вывод адреса текущей команды (значение регистра RIP) и её значения (кода команды в бинарном виде)

```
.data
hello:
    .string "hello world\n"

.globl  main
main:
    movl    $4, %eax
    movl    $2, %ebx
    movl    $hello, %ecx
    movl    $12, %edx
    int     $0x80
    movl    $1, %eax
    xorl    %ebx, %ebx
    int     $0x80
    ret
```

Пример использования системного вызова для вывода сообщения на экран.
Архитектура – intel x32

В **архитектуре intel x64** аргументы для системного вызова ядром GNU\Linux передаются в следующих регистрах:
RAX – номер системного вызова
RDI, RSI, RDX, R10, R8, R9

```
while (1) {
  wait (&status);  if (WIFEXITED (status)) break;
  orig_eax = ptrace (PTRACE_PEEKUSER, child, 8 * ORIG_RAX, NULL);
  if (orig_eax == SYS_write) {
    if (toggle == 0) { toggle = 1;
      params[0] = ptrace (PTRACE_PEEKUSER, child, 8 * RDI, NULL);
      params[1] = ptrace (PTRACE_PEEKUSER, child, 8 * RSI, NULL);
      params[2] = ptrace (PTRACE_PEEKUSER, child, 8 * RDX, NULL);
      str = (char *) malloc ((params[2] + 1) * sizeof (char));
      getdata (child, params[1], str, params[2]);
      reverse (str);
      putdata (child, params[1], str, params[2]);
    } else { toggle = 0; }
  }
  ptrace (PTRACE_SYSCALL, child, NULL, NULL);
}
```

Рисунок – пример использования ptrace для изменения ресурсов трассируемого процесса («переворачивается» вывод системного вызова write)

```
void getdata (pid_t child, long addr, char *str, int len) {
  char *laddr;  int i, j;
  union u { long long val; char chars[long_long_size]; } data;
  i = 0;  j = len / long_long_size;   laddr = str;

  while (i < j)     {
    data.val = ptrace (PTRACE_PEEKDATA, child, addr + i * 8, NULL);
    memcpy (laddr, data.chars, long_size);
    ++i;
    laddr += long_size;
  }

  j = len % long_size;
  if (j != 0) {
    data.val = ptrace (PTRACE_PEEKDATA, child, addr + i * 8, NULL);
    memcpy (laddr, data.chars, j);
  }
  str[len] = '\0';
}
```

Рисунок – функция получения данных из памяти трассируемого процесса (**архитектура x64**)

```
void putdata (pid_t child, long addr, char *str, int len) {
  char *laddr; int i, j;
  union u { long long val; char chars[long_long_size]; } data;
  i = 0; j = len / long_long_size; laddr = str;

  while (i < j) {
    memcpy (data.chars, laddr, long_long_size);
    ptrace (PTRACE_POKEDATA, child, addr + i * 8, data.val);
    ++i;
    laddr += long_long_size;
  }
  j = len % long_long_size;
  if (j != 0) {
    memcpy (data.chars, laddr, j);
    ptrace (PTRACE_POKEDATA, child, addr + i * 8, data.val);
  }
}
```

Рисунок – функция изменения данных в памяти трассируемого процесса (**архитектура x64**)

```
example> ls
attach     dummy1.s   part1     ptrace00     ptrace1     registers
attach.c   dummy2     ptrace    ptrace00.c   ptrace1.c   registers.c
dummy1     dummy2.c   ptrace0   ptrace0.c    ptrace.c
example> ./ptrace1
Write called with 1, 140619352846336, 59
sretsiger     1ecartp     00ecartp     1trap   s.1ymmud      hcatta
Write called with 1, 140619352846336, 64
c.sretsiger   c.1ecartp   c.00ecartp    ecartp     2ymmud   c.hcatta
Write called with 1, 140619352846336, 48
c.ecartp         c.0ecartp   0ecartp   c.2ymmud     1ymmud
example>
```

Рисунок – пример работы рассмотренной программы преобразования параметров системного вызова write

```
char code[] = {0xcd,0x80,0xcc,0,0,0,0};
…
ptrace(PTRACE_ATTACH, traced_process, NULL, NULL);
wait(NULL);
ptrace(PTRACE_GETREGS, traced_process, NULL, &regs);
getdata(traced_process, regs.rip, backup, 3);
putdata(traced_process, regs.rip, code, 3);
ptrace(PTRACE_CONT, traced_process, NULL, NULL);
wait(NULL);
….
putdata(traced_process, regs.rip, backup, 3);
ptrace(PTRACE_SETREGS, traced_process, NULL, &regs);
ptrace(PTRACE_DETACH, traced_process, NULL, NULL);
```

Рисунок – пример установки точки останова
в трассируемом процессе

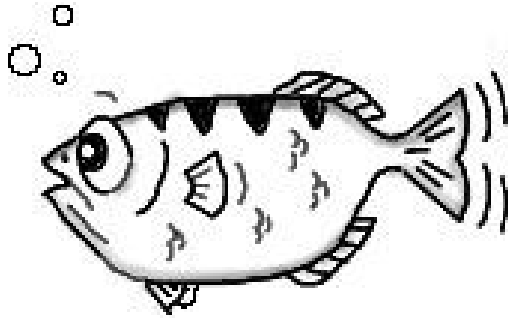
GDB = GNU project debugger

GNU – рекурсивный акроним

GNU's is Not Unix

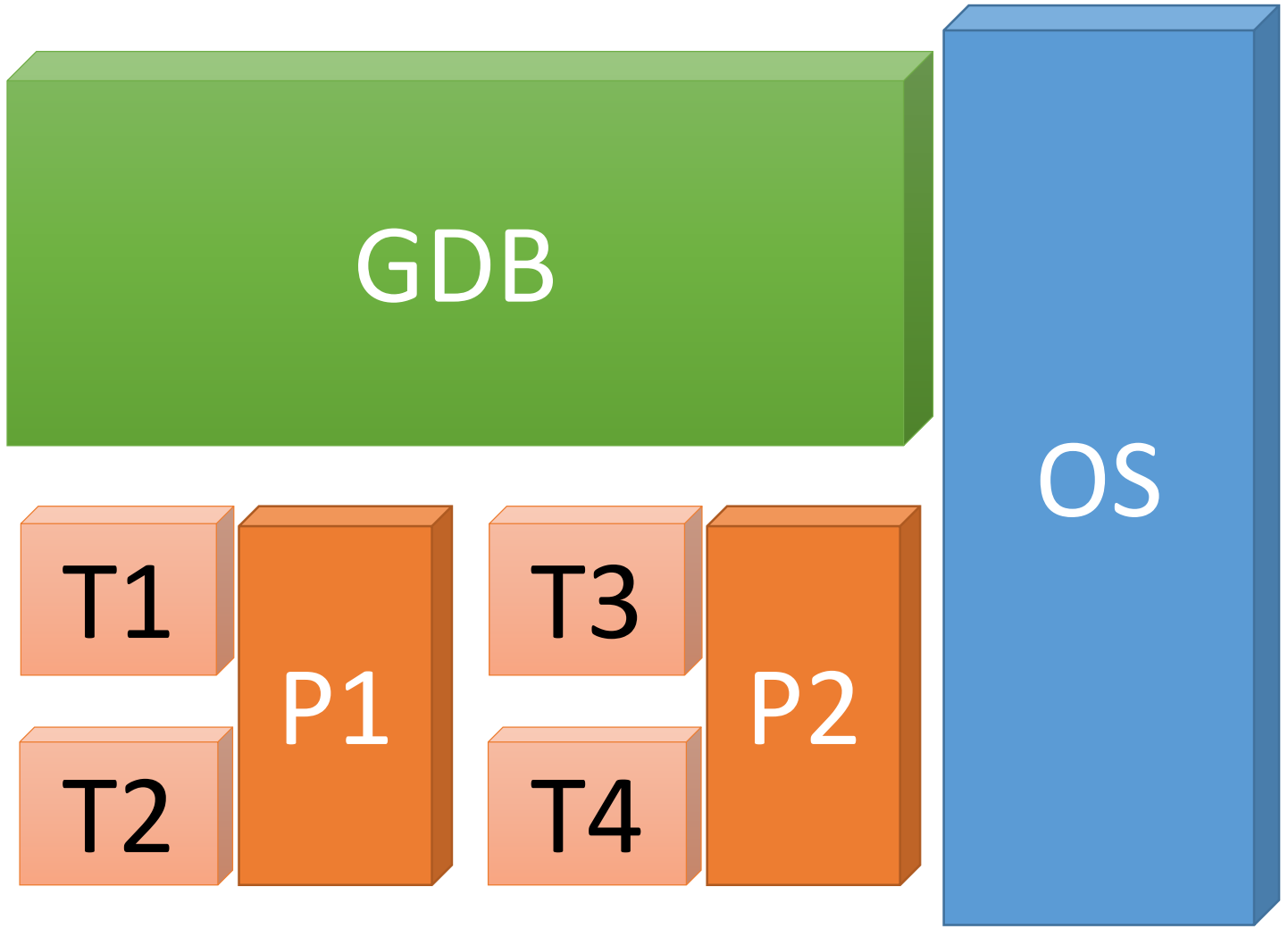
Был разработан Ричардом Столмененом в 1988 году для проекта GNU

Официальный логотип GNU



Логотип GDB – рыба стрелец
(она убивает жуков, стреляя в них водой)

Ричард Мэтью Столлман
(Richard Matthew Stallman)



GDB использует системные вызовы для управления процессами (нитями) и изменения их ресурсов.
В GNU\Linux используется вызов **ptrace**(), в Windows – **DebugActiveProcess**() и т.п.

```
$ gdb ./f -silent
Reading symbols from /home/user/f...(no debugging symbols found)...done.
(gdb) break *0x400538
Breakpoint 1 at 0x400538
(gdb) r
Starting program: /home/user/f
Breakpoint 1, 0x0000000000400538 in ?? ()
Missing separate debuginfos, use: debuginfo-install glibc-2.17-4.fc19.x86_64
(gdb) x/20i $pc-16
   0x400528:    jmpq   0x4004a0
   0x40052d:    nopl   (%rax)
   0x400530:    push   %rbp
   0x400531:    mov    %rsp,%rbp
   0x400534:    sub    $0x10,%rsp
=> 0x400538:    addl   $0xc7,-0x4(%rbp)
   0x40053f:    mov    -0x4(%rbp),%eax
   0x400542:    mov    %eax,%esi
   0x400544:    mov    $0x4005f0,%edi
   0x400549:    mov    $0x0,%eax
   0x40054e:    callq  0x400410 <printf@plt>
   0x400553:    mov    $0x0,%eax
   0x400558:    leaveq
   0x400559:    retq
   0x40055a:    nopw   0x0(%rax,%rax,1)
   0x400560:    push   %r15
   0x400562:    mov    %edi,%r15d
   0x400565:    push   %r14
   0x400567:    mov    %rsi,%r14
   0x40056a:    push   %r13
(gdb)
```

Рисунок **1** – Пример отладки программы, не имеющей символьной (отладочной) информации.

- C
- C++
- D
- Go
- Objective-C
- Fortran
- Java
- OpenCL C
- Pascal
- Assembly
- Modula-2
- Ada

```
$ gcc -Wall -ansi -pedantic -ggdb3 -o file file.c
```

Рисунок **1** – Командная строка компиляции программы с включением отладочной информации.

```
$ objcopy --only-keep-debug file file.debug
$ objcopy --add-gnu-debuglink=foo.debug foo
$ strip -g file
```

Рисунок **1** –Выделение отладочной информации

из исполняемого файла `file` в отдельный файл `file.debug`.

- **Stabs**
- COFF
- Mips debug (Third Eye)
- DWARF1
- DWARF2
- SOM

```
$ gdb [program] [core | process_id] [опции]
```

Рисунок **1** – Команда запуска отладчика GDB

Таблица 1 – Некоторые опции командной строки отладчика GDB	
Опция	Назначение
-help	Вывод подсказки по параметрам командной строки
-s file | --symbols file	Указание файла, содержащего отладочную информацию исследуемой программы
-x file | --command file	Требование исполнения последовательности команд, расположенных в файле.
-batch	Запуск отладчика в пакетном режиме. Используется для автоматизированного тестирования программного обеспечения.
-t device | -tty device	Устройство для перенаправления ввода/вывода исследуемой программы
--args arg1 agr2 …	Аргументы командной строки для исследуемой программы. Эта опция всегда является последней.

```
$ gdb -silent ./file
Reading symbols from /home/user/file...done.
(gdb)r
Starting program: /home/user/file
Program exited normally.
(gdb)q
```

Рисунок **1** – Запуск программы для отладки.

```
$ gdb -silent ./file
Reading symbols from /home/user/file...done.
(gdb)attach 1234
Attaching to program: /home/user/file, process 1234
0x00000036b80d89b0 in __read_nocancel () from /lib64/libc.so.6
(gdb)continue
Continuing.
…
Program exited normally.
(gdb)q
```

Рисунок **1** – Подключение к существующему процессу и его отладка
(процесс в момент подключения ожидал ввод информации с терминала).

set args – задать параметры командной строки

set tty – определить перенаправление потоков ввода вывода

set environment – установить переменную среды окружения

cd – изменить рабочий каталог

```
$ gdb -silent ./fork
Reading symbols from /home/user/fork...done.
(gdb) set detach-on-fork off
(gdb) r
Starting program: /home/user/fork
[New process 23260]
parent finishes
Program exited with code 020.
(gdb) info inferiors
  Num  Description       Executable
  2    process 23260     /home/user/fork
* 1    <null>            /home/user/fork
(gdb) inferior 2
[Switching to inferior 2 [process 23260] (/home/user/fork)]
[Switching to thread 2 (process 23260)]
(gdb) c
Continuing.
child finishes
Program exited with code 017.
(gdb)
```

Рисунок **1** – Отладка многопроцессного приложения в «неотключаемом» режиме (программа создает родительский и дочерний процессы. Оба процесса выводят информацию о своем завершении).

set follow-fork-mode – режим работы с новыми процессами

set detach-on-fork – начинать отладку нового процесса

Аналогичная работа с нитями (thread, info threads)

```
$ gdb -silent ./file
Reading symbols from /home/user/file...done.
(gdb) list
1       #include <stdio.h>
2
3       int main(void){
4         int i = 0, j;
5         for (i = 0; i < 10; i ++){
6           j = i;
7         }
8         printf ("1234\n");
9       }
(gdb) break 6 if i > 6
Breakpoint 1 at 0x4004dc: file file.c, line 6.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004dc in main at file.c:6
        stop only if i > 6
(gdb) r
Starting program: /home/user/file
Breakpoint 1, main () at file.c:6
6           j = i;
(gdb) disable breakpoints 1
(gdb) c
Continuing.
1234
Program exited with code 05.
(gdb)
```

Рисунок **1** – Отладка приложения с применением точки останова.

break – установить точку останова

info breakpoints – показать все точки останова

disable / enable – включение отключение точек останова

save breakpoints / source – сохранение и восстановление точек останова

```
$ gdb -silent ./file
Reading symbols from /home/user/file...done.
(gdb) display/t i
(gdb) display i
(gdb) display
2: i = 7
1: /t i = 111
(gdb)
```

Рисунок **1** – Пример вывода информации о вычислительных ресурсах.

display – вывод результата расчета выражения
watch – настройка вывода в точке останова
list – вывод исходного кода
examine – вывод ассемблерного кода

Форматы вывода…