从基础到高级调试

—— 申威GDB&LLDB调试指南

LET'S LEARN DEBUGGER !

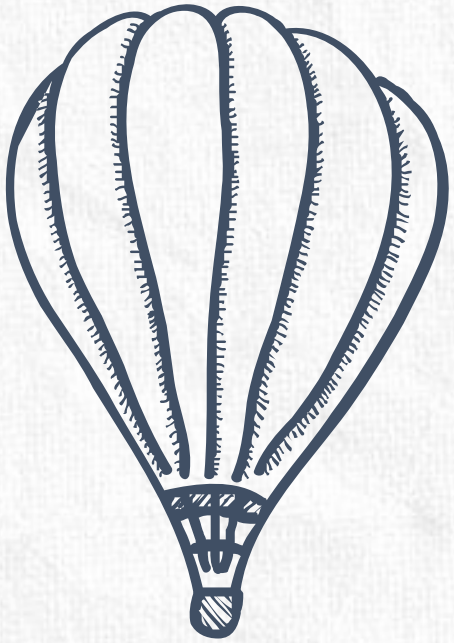
生态软件研发部

刘汉旭

开发生态组

2022.06.16

CONTENT



Part01 深入GDB调试

- 1.GDB常用命令
- 2.GDB高效配置
- 3.GDB高级技巧

01 GDB常用命令

七类命令

- 一. 启动
- 二. 断点
- 三. 运行
- 四. 查看源码、栈帧
- 五. 打印变量
- 六. 查看寄存器、汇编
- 七. 调试进程、线程

GDB常用命令

一、启动

```
gdb a.out                  # 调试 a.out
gdb a.out core             # 调试 a.out + core 文件
gdb -p 1234                # 调试正在运行的进程（pid：1234）
gdb -x cmd.txt             # 启用gdb并执行cmd.txt里的命令
gdb -nx                    # 启动gdb且不加载.gdbinit配置（默认加载）
gdb -tty /dev/pts/7        # 指定窗口作为程序的输入/输出
gdb -statistics            # 启动gdb并打印命令的执行时间和内存空间
```

GDB常用命令

二、断点

```
break {func|line}                 # 函数名或行号（b func|line）
break {file:func|file:line} #文件+函数名或文件+行号
break *0x400068                   # 内存地址
rbreak fun_*                      # 正则断点（所有fun_开头的函数）
info breakpoints                  # 查看断点（i b）
delete [n]                        # 删除所有断点[n号断点]（d）
disable/enable 2                  # 禁用/启用某断点
condition 2 i==10                 # 条件断点，当条件i==10为真断点2才生效
ignore 2 10                       # 忽略10次断点2
info functions                    # 查看所有函数符号（i fun）
save breakpoints bp.txt           # 将断点信息导出到bp.txt文件
```

GDB常用命令

三、运行

```
run                     # 运行程序（r）
set args {args}         # 设置运行的参数为args
show args               # 显示当前的运行参数
continue                # 继续运行（c）
step/stepi              # 源码级/指令级单步，碰到函数会进入(s/si)
next/nexti              # 源码级/指令级单步，碰到函数不进入(n/ni)
step/next {count}       # 单步count次
finish                  # 结束当前函数的运行
until [line]            # 跳出循环[持续执行到某行]
```

四、查看源码、栈帧

```
list                        # 显示源码 (set listsize N)
list 50                     # 显示第50行周围源码
list +/-                    # 向后/向前显示源码
list 1,20                   # 显示1到20行源码
dir {dirname}               # 增加源码搜索路径
show directories            # 显示源码路径

backtrace [-full]           # 查看函数调用栈[附带局部变量] (bt)
frame n                     # 显示第n层栈帧 (f n)
info locals                 # 查看帧内变量 (i locals)
info args                   # 查看函数参数 (i args)
```

GDB常用命令

五、打印变量

```
print a                          # 打印变量a（p a）
print /x a                       # 以16进制打印变量a
x /nfu {address}                 # 打印内存数据，n个数，f格式，u单位
x /10xb *0x11223344              # 按16进制打印地址0x11223344处的10个字节
ptype {var}                      # 查看变量类型或结构体定义
set var = {value}                # 变量赋值
display {expression}             # 监控查看某表达式的值
undisplay                        # 删除对某些值的监控
info display                     # 显示监视的表达式
```

GDB常用命令

六、查看寄存器、汇编

```
info registers          # 打印整数寄存器（i reg）
info all-registers      # 打印所有寄存器，整数+浮点+向量
info register $pc       # 打印单个寄存器（i reg $pc）
set $pc=0x4000a8        # 修改寄存器

display/i  $pc          # 打印下条指令汇编
disassemble fun         # 对函数fun进行反汇编
disassemble 0x2c4e      # 对地址0x2c4e进行反汇编
disassemble/m fun       # 反汇编fun，并匹配显示源码
```

GDB常用命令

七、调试进程、线程

```
info inferiors                      # 查看进程
inferior {id}                       # 切换进程
set follow-fork-mode child          # 调试子进程
set detach-on-fork off              # 同时调试父子进程
set schedule-multiple on            # 父子进程同时运行

info threads                        # 查看线程
thread {id}                         # 切换线程
break {line} thread all             # 所有线程设断点
thread apply all {cmd}              # 所有线程共同执行命令
set schedule-locking on             # 调试一个线程时，其他线程暂停
```

GDB高效配置

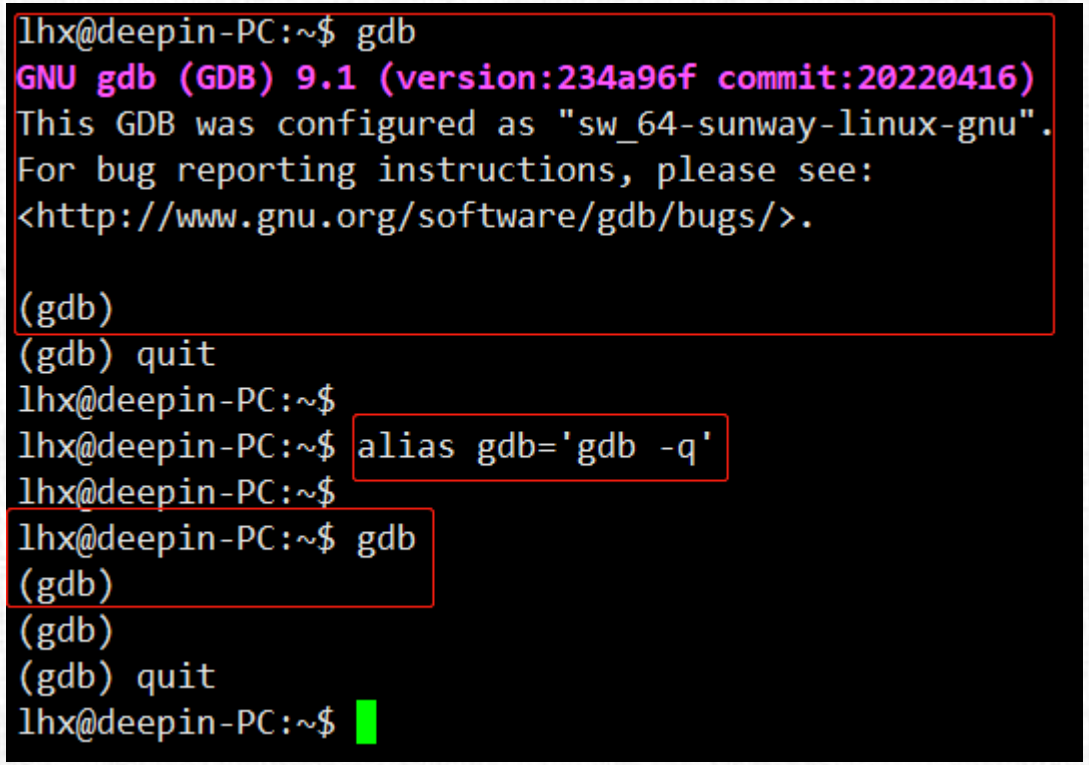
配置.gdbinit

```
set breakpoint pending on                  # 挂起断点，未来生效
set confirm off                            # 退出时不显示提示信息
set pagination off                         # 关闭分页显示模式
set history filename ~/.gdb_history
set history save on                        # 保存历史命令
set logging on                             # 开启日志
set disassemble-next-line auto             # 显示下一行反汇编
set print array-indexes on                 # 打印数组索引下标
set print pretty on                        # 打印缩进对齐的结构体
set print thread-events off                # 关闭线程产生和退出信息
```

02 GDB高效配置

配置.bashrc

```
alias gdb='gdb -q'          # 开启gdb安静模式
export EDITOR=vim           # 设置gdb里跳转的编辑器
```



GDB高级技巧

六大技巧

- 一. 自定义命令define
- 二. 钩子命令hook-cmd
- 三. 触发命令commands
- 四. 观察点watchpoint
- 五. 捕捉点catchpoint
- 六. 检查点checkpoint

03 GDB高级技巧

技巧 1 我的命令我做主

```
define xx          # 自定义命令xx
document xx        # 给自定义命令xx添加使用说明
```

```
(gdb) define bmr
Type commands for definition of "bmr".
End with a line saying just "end".
>b main
>run
>end
(gdb) bmr
Breakpoint 1 at 0x120000618: file test-macro.c, line 11.

Breakpoint 1, main () at test-macro.c:11
11          b = MACRO3(a);
(gdb) document bmr
Type documentation for "bmr".
End with a line saying just "end".
>break at main(), and run
>end
(gdb) help bmr
break at main(), and run
(gdb)
```

技巧 2 巧用hook，释放双手

hook（钩子）让执行某个命令前或命令后，先执行某些命令。
如想在print命令前后显示“----------”：

```
(gdb) define hook-print
Type commands for definition of "hook-print".
End with a line saying just "end".
>echo ----------\n
>end
(gdb) print b
----------
$1 = 0
(gdb) define hookpost-print
Type commands for definition of "hookpost-print".
End with a line saying just "end".
>echo ----------\n
>end
(gdb) print b
----------
$2 = 0
----------
(gdb)
```

```
(gdb) define hook-stop
Type commands for definition of "hook-stop".
End with a line saying just "end".
>i reg $pc
>set $pc=$pc+16
>i reg $pc
>x/i $pc
>end
(gdb) n
pc             0x12000064c          0x12000064c <main+76>
pc             0x12000065c          0x12000065c <main+92>
=> 0x12000065c <main+92>:       ldih    $r2,0($r29)
12          opcode = INSN_OPCODE(insn);
(gdb)
```

GDB高级技巧

```
# 断点或单步停下来的钩子
define hook-stop
....
end
```

```
# 执行run命令时的钩子
define hook-run
...
end
```

```
# 执行continue命令时的钩子
define hook-continue
...
end
```

常用的自定义钩子

03 GDB高级技巧

技巧 3 给断点找点事干

commands（简写为comm）：

可以在触发某个（或多个）断点时运行指定命令。

```c
#include <stdio.h>

int add()
{
  int a = 3, b = 4, sum;
  sum = a + b;
  return sum;
}

int main()
{
  printf("sum = %d\n", add());
  return 0;
}
```

```
(gdb) b main
Breakpoint 1 at 0x120000664: file test.c, line 12.
(gdb) b 6
Breakpoint 2 at 0x120000620: file test.c, line 6.
(gdb) commands 2
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>print a
>print b
>end
(gdb) r
Starting program: /home/lhx/demo/test

Breakpoint 1, main () at test.c:12
12          printf("sum = %d\n", add());
(gdb) c
Continuing.

Breakpoint 2, add () at test.c:6
6           sum = a + b;
$1 = 3
$2 = 4
(gdb)
```

03 GDB高级技巧

技巧 4 学会watchpoint，bug自动现身

watchpoint（观察点）：

观察点可以监控程序中变量或表达式的值，

只要发生改变，程序就会停止执行。

观察点适用于：

- 由于变量值异常变化导致的bug
- 多线程场景，不知道变量被谁改了

```
9       {
10        int a = 1;
11        for(int i=0; i<=100; i++)
12        {
13          printf("i = %d\n", i);
14          if (i == 66)
15            a = 0;
16        }
17        printf("a = %d\n", a);
18        return 0;
(gdb) watch a
Hardware watchpoint 2: a
(gdb) c
Continuing.
i = 1
i = 2
i = 3
```

```
i = 65
i = 66

Hardware watchpoint 2: a

Old value = 1
New value = 0
main () at test2.c:11
11        for(int i=0; i<=100; i++)
(gdb)
```

技巧 5 学会catchpoint，拦截程序小动作

catchpoint（捕捉点）：

捕捉点可以捕获程序运行中发生的事件（加载动态库、系统调用、fork ……）

捕捉点适用于：

- 想捕捉程序的一些运行状态
- 想查看程序发生的系统调用

```
lhx@deepin:~/demo$ gdb a.out
Reading symbols from a.out...
(gdb) catch load
Catchpoint 1 (load)
(gdb) catch syscall fork
Catchpoint 2 (syscall 'fork' [2])
(gdb) r
Starting program: /home/lhx/demo/a.out

Catchpoint 2 (returned from syscall fork), 0x0000020000020c10 in
(gdb) c
Continuing.

Catchpoint 1
  Inferior loaded /lib/libc.so.6.1
0x00000200000148d0 in _dl_debug_state () from /lib/ld-linux.so.2
(gdb) c
Continuing.
hello world
[Inferior 1 (process 12983) exited normally]
(gdb)
```

03 GDB高级技巧

技巧 6 学会checkpoint，避免重头再来

checkpoint（检查点）：

用来保存程序的运行快照，并且可以跳回，跳回时会撤销所有的后续操作。

检查点适用于：

- 重新运行耗时较长的程序
- 单步调试时bug发生地方很远

```
fun1 () at test-call.c:11
11          int a = 3, b = 4;
(gdb) checkpoint
checkpoint 1: fork returned pid 12642.
(gdb) n
12          printf("func1 called\n");
(gdb) n
func1 called
13          fun2();
(gdb) s
fun2 () at test-call.c:5
5           int c = 5, d = 6;
(gdb) info checkpoints
* 0 process 12639 (main process) at 0x0
  1 process 12642 at 0x120000644, file test-call.c, line 11
(gdb) restart 1
Switching to process 12642
#0  fun1 () at test-call.c:11
11          int a = 3, b = 4;
(gdb)
```

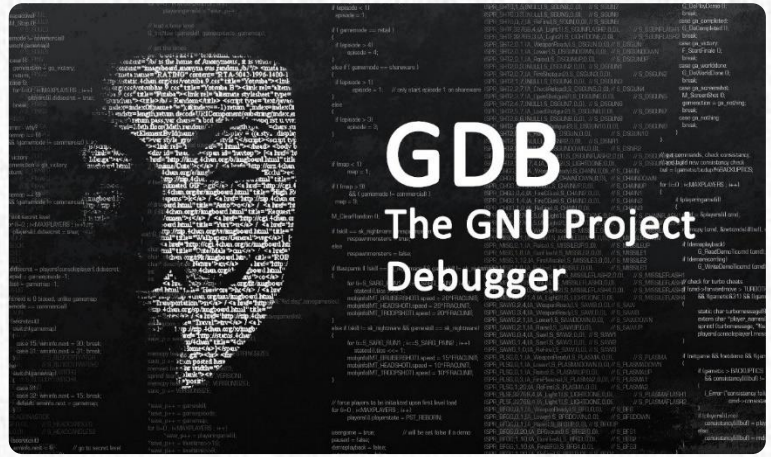
Part02 学会LLDB调试

- 1.LLDB与GDB区别
- 2.LLDB常用命令
- 3.LLDB特色

01 LLDB与GDB区别

LLDB（LLVM Debugger）

- 轻量级，插件化，更好集成LLVM
- C++支持更完善
- Clang编译，优先推荐
- 调试速度比GDB更快
- C/C++/Objective C



GDB（GNU Debugger）

- 重量级，更成熟，基于GCC
- 复杂的C++场景，调试不友好
- gcc编译，优先推荐
- C/C++/Objective C /Ada,/Pascal/FORTRAN/Go

02

LLDB常用命令

五类命令

- 一. 启动
- 二. 断点
- 三. 寄存器、反汇编
- 四. 查看变量
- 五. 查看线程、堆栈

LLDB常用命令

启动命令	GDB	LLDB
启动方式	gdb [program] gdb --args program arg1	lldb [program] lldb -- program arg1
加载程序	(gdb) file program	(lldb) file program
启动程序	(gdb) run [args]	(lldb) process launch [-- args]
设置参数	(gdb) set args 1	(lldb) settings set target.run-args 1
显示参数	(gdb) show args	(lldb) settings show target.run-args
单步进入-源码级	(gdb) step (s)	(lldb) thread step-in (step/s)
单步进入-指令级	(gdb) stepi (si)	(lldb) thread step-inst (si)
单步跳过-源码级	(gdb) next (next/n)	(lldb) thread step-over (next/n)
单步跳过-指令级	(gdb) nexti (ni)	(lldb) thread step-inst-over (ni)
跳出当前函数	(gdb) finish	(lldb) thread step-out (finish)

02 LLDB常用命令

断点命令	GDB	LLDB
函数断点	(gdb) break main (gdb) b main	(lldb) breakpoint set --name main (lldb) br s -n main (b main)
文件断点	(gdb) break test.c:7	(lldb) breakpoint set --file test.c –line 7
显示断点	(gdb) info break	(lldb) breakpoint list (br l)
删除断点	(gdb) delete 1	(lldb) breakpoint delete 1 (br del 1)
禁用/启用断点	(gdb) disable/enable 1	(lldb) breakpoint disable/enable 1

寄存器、反汇编	GDB	LLDB
读通用寄存器	(gdb) info registers	(lldb) register read
写寄存器	(gdb) set $r1 = 123	(lldb) register write r1 123
读所有寄存器	(gdb) info all-registers	(lldb) register read -all
反汇编当前函数	(gdb) disassemble	(lldb) disassable –frame (di -f)
反汇编main函数	(gdb) disassemble main	(lldb) disassable --name main

LLDB常用命令

查看变量	GDB	LLDB
查看参数和局部变量	(gdb) info args (gdb) info locals	(lldb) frame variable (lldb) fr v
打印变量	(gdb) print a (gdb) p a	(lldb) frame variable a (lldb) fr v a (p a)
打印变量-16进制	(gdb) p/x a	(lldb) frame variable --format x a
打印变量类型	(gdb) ptype a	(lldb) image lookup --type a

查看线程、堆栈	GDB	LLDB
显示线程	(gdb) info thread	(lldb) thread list
选择切换线程	(gdb) thread 1 (t 1)	(lldb) thread select 1 (t 1)
打印堆栈	(gdb) bt	(lldb) thread backtrace (bt)
打印5层堆栈	(gdb) bt 5	(lldb) thread backtrace -c 5 (bt 5)
选择第7层栈帧	(gdb) f 7	(lldb) frame select 7 (f 7)

03 LLDB特色-调试更精确

1

```
* thread #1, name = 'a.out', stop reason = breakpoint 1.1
    frame #0: 0x0000000120000664 a.out`main at test.c:13:8
   10   int main()
   11   {
   12     int i, sum;
-> 13     for(i=0; i<100; i++)
   14       sum += i;
   15     printf("sum = %d\n", add());
   16     return 0;
```

2

```
(lldb) s
Process 16983 stopped
* thread #1, name = 'a.out', stop reason = step in
    frame #0: 0x000000012000066c a.out`main at test.c:14:9
   11   {
   12     int i, sum;
   13     for(i=0; i<100; i++)
-> 14       sum += i;
   15     printf("sum = %d\n", add());
   16     return 0;
   17   }
```

3

```
(lldb) s
Process 16983 stopped
* thread #1, name = 'a.out', stop reason = step in
    frame #0: 0x000000012000067c a.out`main at test.c:13:20
   10   int main()
   11   {
   12     int i, sum;
-> 13     for(i=0; i<100; i++)
   14       sum += i;
   15     printf("sum = %d\n", add());
   16     return 0;
```

LLDB特色-寄存器别名

```
(lldb) register read
General Purpose Registers:
        r0 = 0x000040000002fd10
        r1 = 0x0000400000249330  libc.so.6.1`_environ
        r2 = 0x0000000000000000
        r3 = 0x0000000000000000
        r4 = 0x000000011fffdfa0
        r5 = 0x0000400000249db0
        r6 = 0x0000400000249db0
        r7 = 0x000000000000007f
        r8 = 0x00000000000000ff
        r9 = 0x0000000000000000
       r10 = 0x0000000120000740  a.out`__libc_csu_init
       r11 = 0x000000011fad91a8
       r12 = 0x000000011fad91b0
       r13 = 0x000000011fad91a8
       r14 = 0x0000000120ac41e0
        fp = 0x000000011fffdf50
       r16 = 0x0000000000000001
       r17 = 0x000000011fffe088
       r18 = 0x000000011fffe098
       r19 = 0x0000400000248928
       r20 = 0x0000000120000a10  a.out`__libc_csu_fini
       r21 = 0x00004000000141c0  ld-2.28.so`___lldb_unnamed_symbol56$$ld-2.28.so
       r22 = 0x0000000000000001
       r23 = 0x0000000000000000
       r24 = 0x00000000f63d4e2e
       r25 = 0x000000011fffdde0
        ra = 0x000040000008db9c  libc.so.6.1`__libc_start_main + 284
       r27 = 0x000000012000066c  a.out`main at test.c:10:1
       r28 = 0x00000000f63d4a75
        gp = 0x000000012001a010
        sp = 0x000000011fffdf50
      zero = 0x000000011fda8b38
        pc = 0x000000012000068c  a.out`main + 32 at test.c:12:6
```

```
(lldb) register read -A
General Purpose Registers:
        v0 = 0x000040000002fd10
        t0 = 0x0000400000249330  libc.so.6.1`_environ
        t1 = 0x0000000000000000
        t2 = 0x0000000000000000
        t3 = 0x000000011fffdfa0
        t4 = 0x0000400000249db0
        t5 = 0x0000400000249db0
        t6 = 0x000000000000007f
        t7 = 0x00000000000000ff
        s0 = 0x0000000000000000
        s1 = 0x0000000120000740  a.out`__libc_csu_init
        s2 = 0x000000011fad91a8
        s3 = 0x000000011fad91b0
        s4 = 0x000000011fad91a8
        s5 = 0x0000000120ac41e0
        fp = 0x000000011fffdf50
        a0 = 0x0000000000000001
        a1 = 0x000000011fffe088
        a2 = 0x000000011fffe098
        a3 = 0x0000400000248928
        a4 = 0x0000000120000a10  a.out`__libc_csu_fini
        a5 = 0x00004000000141c0  ld-2.28.so`___lldb_unnamed_symbol56$$ld-2.28.so
        t8 = 0x0000000000000001
        t9 = 0x0000000000000000
       t10 = 0x00000000f63d4e2e
       t11 = 0x000000011fffdde0
        ra = 0x000040000008db9c  libc.so.6.1`__libc_start_main + 284
        pv = 0x000000012000066c  a.out`main at test.c:10:1
        at = 0x00000000f63d4a75
        gp = 0x000000012001a010
        sp = 0x000000011fffdf50
      zero = 0x000000011fda8b38
        pc = 0x000000012000068c  a.out`main + 32 at test.c:12:6
```

Part03 图形界面&调试插件

- 1.GDB TUI界面
- 2.LLDB GUI界面
- 3.两大IDE界面
- 4.三大调试插件

01 GDB TUI界面-命令

TUI进入、退出、分割和切换命令：

01
`gdb -tui`
进入TUI

02
`tui enable`
进入TUI

03
`tui disable`
退出TUI

04
`layout src`
源码窗口

05
`layout asm`
汇编窗口

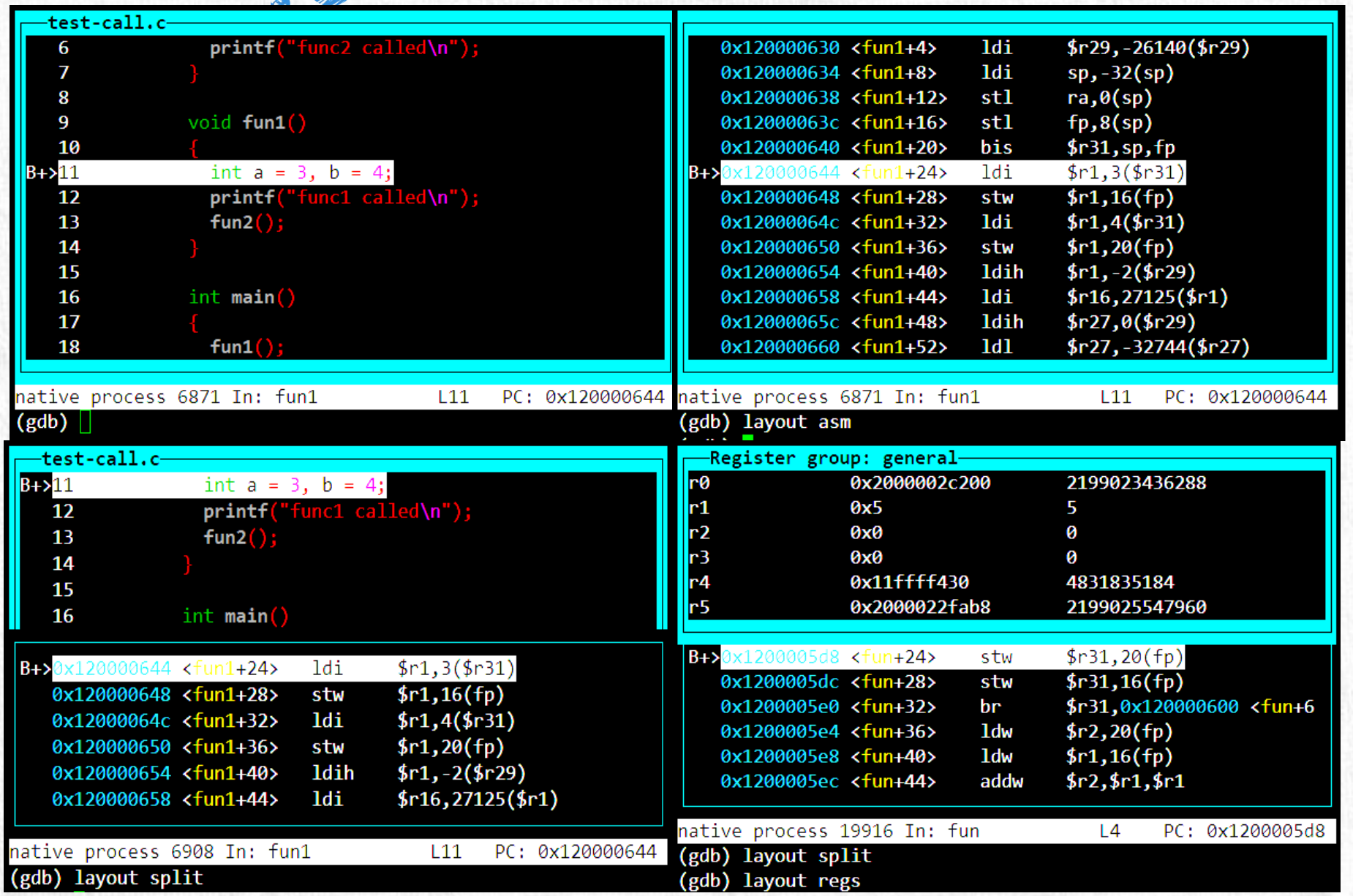
06
`layout regs`
寄存器窗口

07
`layout split`
分割窗口

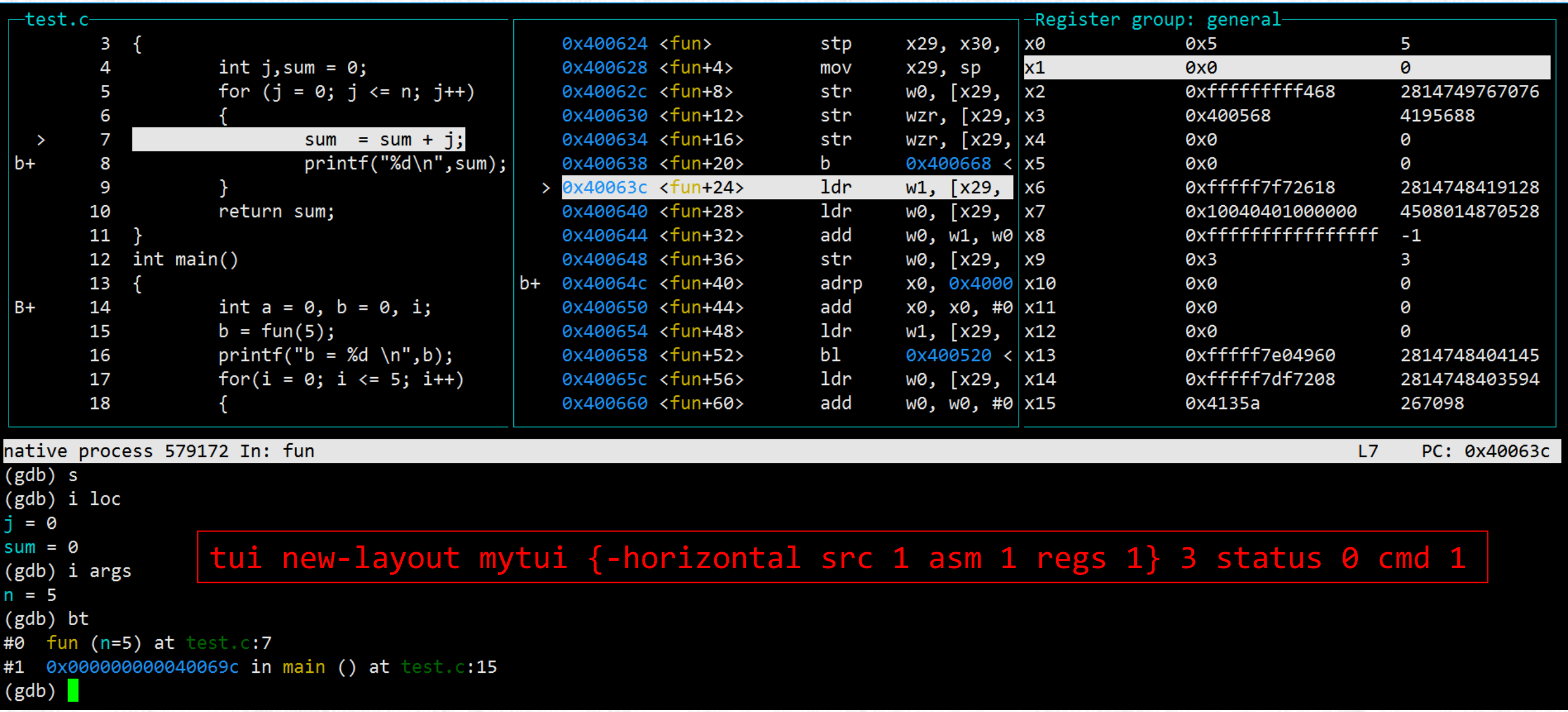
08
`layout next`
下个视图

09
`layout prev`
上个视图

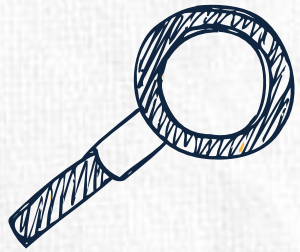
01 GDB TUI界面-效果展示



01 GDB TUI界面-自定义布局



01 GDB TUI界面-操作技巧

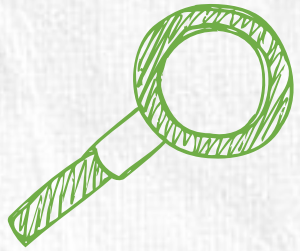


断点状态显示

B 表示该断点已执行　　b 表示该断点未执行
\+ 表示断点使能　　\- 表示断点失能

快捷键操作

方向键、page up/down	翻阅源码
ctrl + p/n	浏览上条/下条命令
ctrl + b/f	光标前移/后移
Ctrl + x + a	进入TUI界面
Ctrl + x + 2	切换TUI界面分割窗口的布局

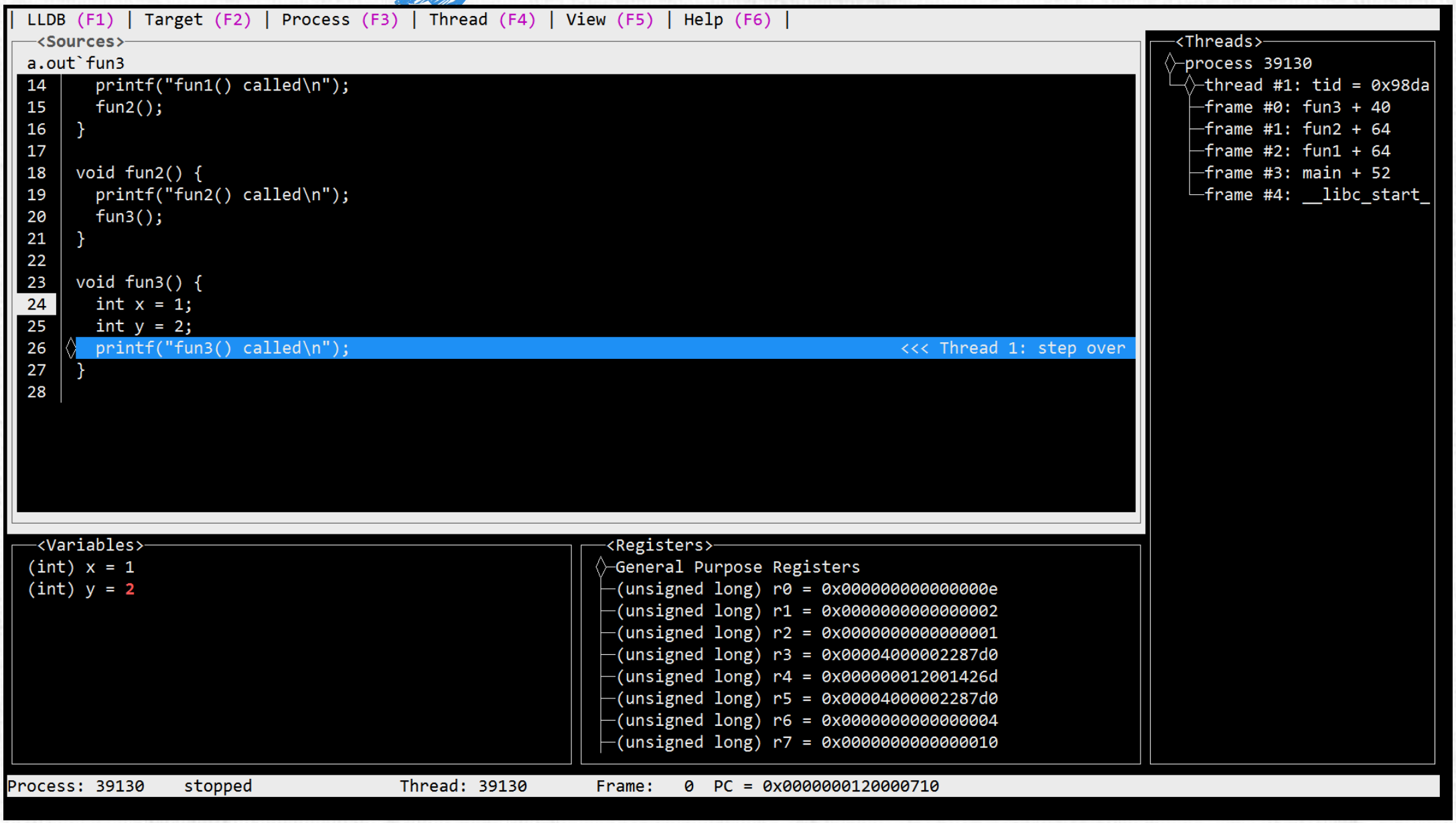


窗口大小调整

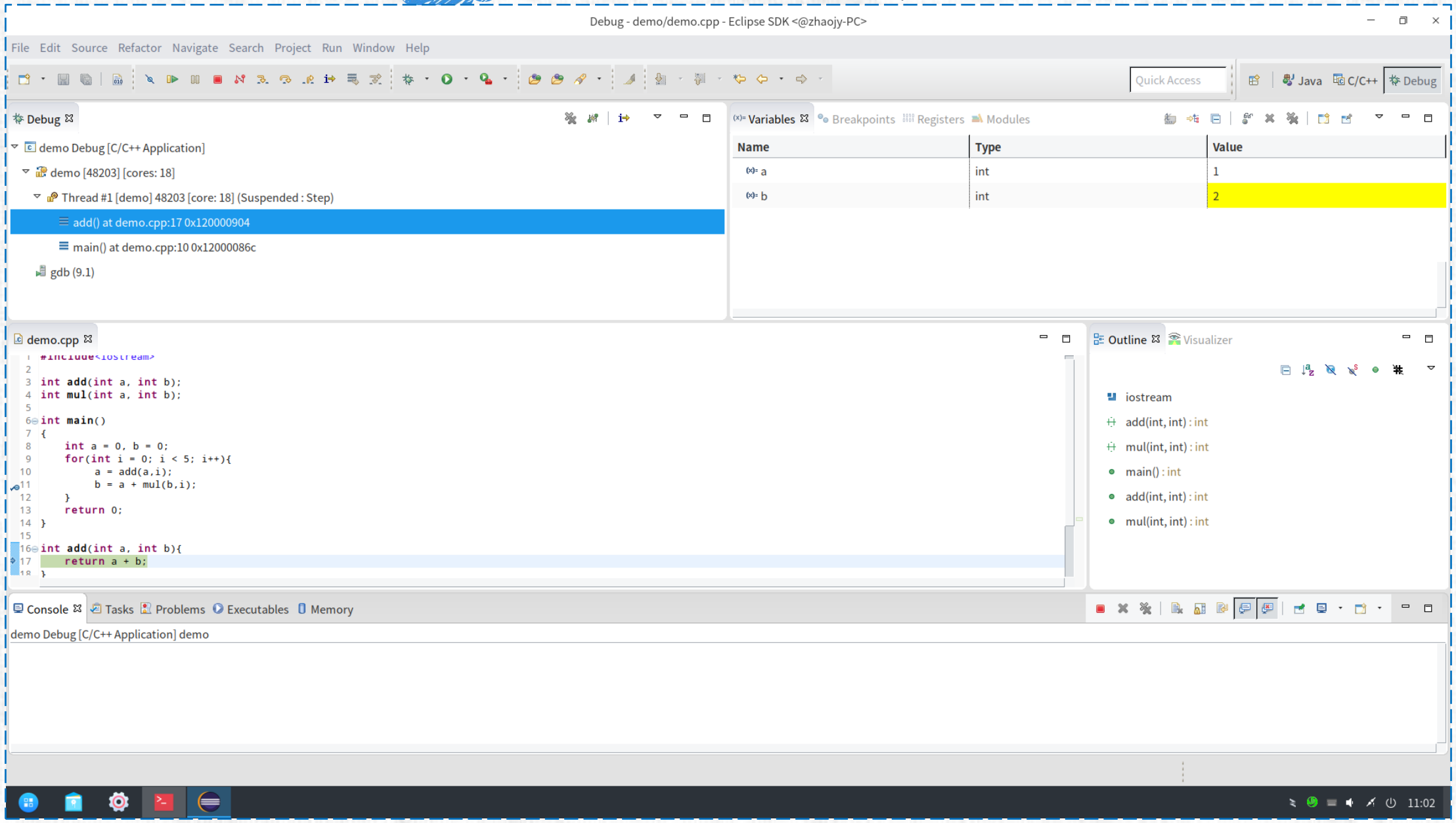
winheight src/asm/cmd +N　　winheight src/asm/cmd -N
单位：行数

```
test.c
   8                    }
   9                    return sum;
   10           }
   11           int main()
   12           {
B+>13                   int a = 0, b = 0, i;
   14                   b = fun(5);
b+ 15                   printf("b = %d \n",b);
   16                   for(i = 0; i <= 5; i++)
   17                   {
b- 18                           a  = a + i;
   19                   }
   20                   printf("a = %d \n",a);
native process 19803 In: main          L13   PC: 0x120000644
(gdb) b 15
Breakpoint 2 at 0x12000066c: file test.c, line 15.
(gdb) b 18
Breakpoint 3 at 0x120000698: file test.c, line 18.
(gdb) disable 3
```

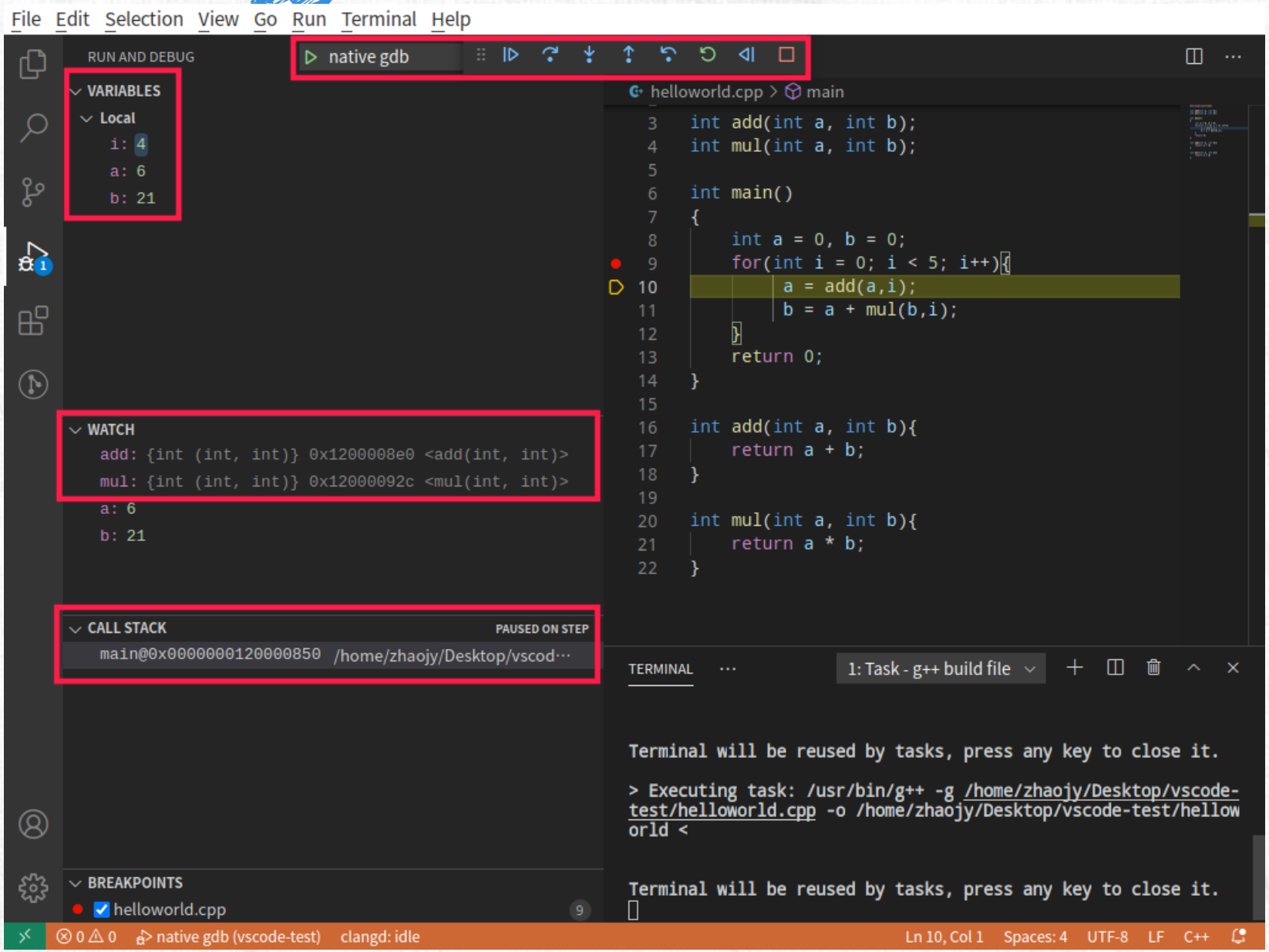
02 LLDB GUI界面



03 IDE-eclipse界面



03 IDE-vscode界面



04 三大调试插件

GEF

(GDB Enhanced Features)

通过Python脚本扩充GDB功能，更方便进行动态分析和逆向分析

完成初步移植，近期社区开源

Pwndbg

包含一系列工具集，可分析got/plt表，颜色和功能丰富

移植中，待功能完善后社区开源

dashboard

界面美观，组件丰富，架构无关

https://github.com/cyrus-and/gdb-dashboard

04 调试插件-GEF

```
Breakpoint 1, fun (n=0x5) at test.c:4
4               int j,sum = 0;
[ Legend: Modified register | Code | Heap | Stack | String ]
registers
$ra   : 0x0000012000065c  →  <main+48> ldih $r29, 2(ra)
$sp   : 0x0000011ffff390  →  0x0000012000065c  →  <main+48> ldih $r29, 2(ra)
$pc   : 0x000001200005d8  →  <fun+24> stw $r31, 20(fp)
stack
0x0000011ffff390|+0x0000: 0x0000012000065c  →  <main+48> ldih $r29, 2(ra)        ← $sp
0x0000011ffff398|+0x0008: 0x0000011ffff3c0  →  0x0002000007a564  →  <__libc_start_main+292> ldih $r29, 28(ra)
0x0000011ffff3a0|+0x0010: 0x00000120000700  →  <__libc_csu_init+0> ldih $r29, 2($r27)
0x0000011ffff3a8|+0x0018: 0x00000001201b84a0
0x0000011ffff3b0|+0x0020: 0x0000000000000005
0x0000011ffff3b8|+0x0028: 0x00000001201b8460
0x0000011ffff3c0|+0x0030: 0x0002000007a564  →  <__libc_start_main+292> ldih $r29, 28(ra)
0x0000011ffff3c8|+0x0038: 0x0000000000000000
code:sw_64:
   0x1200005cc <fun+12>          bis    $r31, sp, fp
   0x1200005d0 <fun+16>          bis    $r31, $r16, $r1
   0x1200005d4 <fun+20>          stw    $r1, 32(fp)
 → 0x1200005d8 <fun+24>          stw    $r31, 20(fp)
   0x1200005dc <fun+28>          stw    $r31, 16(fp)
   0x1200005e0 <fun+32>          br     $r31, 0x120000600 <fun+64>
   0x1200005e4 <fun+36>          ldw    $r2, 20(fp)
```

04 调试插件-GEF

查看16进制反汇编视图

```
gef➤  hexdump byte $sp
0x00007fffffffe290     00 00 00 00 00 00 00 00 c8 e3 ff ff ff 7f 00 00    ................
0x00007fffffffe2a0     b8 e3 ff ff ff 7f 00 00 50 50 55 55 01 00 00 00    ........PPUU....
0x00007fffffffe2b0     b0 e3 ff ff ff 7f 00 00 00 00 00 00 00 00 00 00    ................
0x00007fffffffe2c0     00 00 00 00 00 00 00 00 25 1b df f7 ff 7f 00 00    ........%.......
gef➤
```

分析堆信息

```
[#0] RetAddr: 0x7ffff7acf328, Name: __strcpy_sse2_unaligned()
[#1] RetAddr: 0x400692, Name: main()

[*] Heap-Analysis
Possible Use-after-Free:
Pointer 0x602010 was freed, but is attempt to be used at 0x7ffff7acf328

gef➤
```

展示详细的ELF信息

```
gef➤  elf
Magic                 : 7f 45 4c 46
Class                 : 0x2 - 64-bit
Endianness            : 0x1 - Little-Endian
Version               : 0x1
OS ABI                : 0x0 - System V
ABI Version           : 0x0
Type                  : 0x3 - Shared
Machine               : 0x3e - x86-64
Program Header Table  : 0x0000000000000040
Section Header Table  : 0x0000000000003a88
Header Table          : 0x0000000000000040
ELF Version           : 0x1
Header size           : 64 (0x40)
Entry point           : 0x0000000000001050

Program Header
  [ #] Type           Offset   Virtaddr   Physaddr   FileSiz   MemSiz Flags     Align
  [ 0] PHDR             0x40       0x40       0x40     0x2d8    0x2d8 r--         0x8
  [ 1] INTERP          0x318      0x318      0x318     0x1c     0x1c r--         0x1
  [ 2] LOAD              0x0        0x0        0x0     0x670    0x670 r--      0x1000
  [ 3] LOAD           0x1000     0x1000     0x1000     0x225    0x225 r-x      0x1000
  [ 4] LOAD           0x2000     0x2000     0x2000     0x120    0x120 r--      0x1000
  [ 5] LOAD           0x2de8     0x3de8     0x3de8     0x250    0x258 rw-      0x1000
  [ 6] DYNAMIC        0x2df8     0x3df8     0x3df8     0x1e0    0x1e0 rw-         0x8
  [ 7] NOTE            0x338      0x338      0x338      0x40     0x40 r--         0x8
  [ 8] NOTE            0x378      0x378      0x378      0x44     0x44 r--         0x4
  [ 9] UNKNOWN         0x338      0x338      0x338      0x40     0x40 r--         0x8
  [10] GNU_EH_FLAME   0x2014     0x2014     0x2014      0x34     0x34 r--         0x4
  [11] GNU_STACK         0x0        0x0        0x0       0x0      0x0 rw-        0x10
  [12] GNU_RELRO      0x2de8     0x3de8     0x3de8     0x218    0x218 r--         0x1

Section Header
  [ #] Name                      Type     Address   Offset     Size   EntSiz Flags Link Info     Align
  [ 0]                           NULL         0x0      0x0      0x0      0x0         0x0  0x0      0x0
  [ 1] .interp                   PROGBITS   0x318    0x318    0x1c     0x0 A       0x0  0x0      0x1
  [ 2] .note.gnu.property        NOTE       0x338    0x338    0x40     0x0 A       0x0  0x0      0x8
  [ 3] .note.gnu.build-id        NOTE       0x378    0x378    0x24     0x0 A       0x0  0x0      0x4
  [ 4] .note.ABI-tag             NOTE       0x39c    0x39c    0x20     0x0 A       0x0  0x0      0x4
  [ 5] .gnu.hash                 GNU_HASH   0x3c0    0x3c0    0x1c     0x0 A       0x6  0x0      0x8
  [ 6] .dynsym                   DYNSYM     0x3e0    0x3e0    0xc0     0x18 A      0x7  0x1      0x8
  [ 7] .dynstr                   STRTAB     0x4a0    0x4a0    0x9d     0x0 A       0x0  0x0      0x1
  [ 8] .gnu.version              HIOS       0x53e    0x53e    0x10     0x2 A       0x6  0x0      0x2
  [ 9] .gnu.version_r            GNU_verneed 0x550   0x550    0x30     0x0 A       0x7  0x1      0x8
  [10] .rela.dyn                 RELA       0x580    0x580    0xc0     0x18 A      0x6  0x0      0x8
  [11] .rela.plt                 RELA       0x640    0x640    0x30     0x18 AI     0x6  0x17     0x8
```

04 调试插件-Pwndbg

04 调试插件-Pwndbg

- 查看虚拟内存信息
- 打印当前上下文信息
- 显示寄存器、堆栈
- 反汇编
- 取消指针引用
- 对内存地址进行颜色区分
- ……

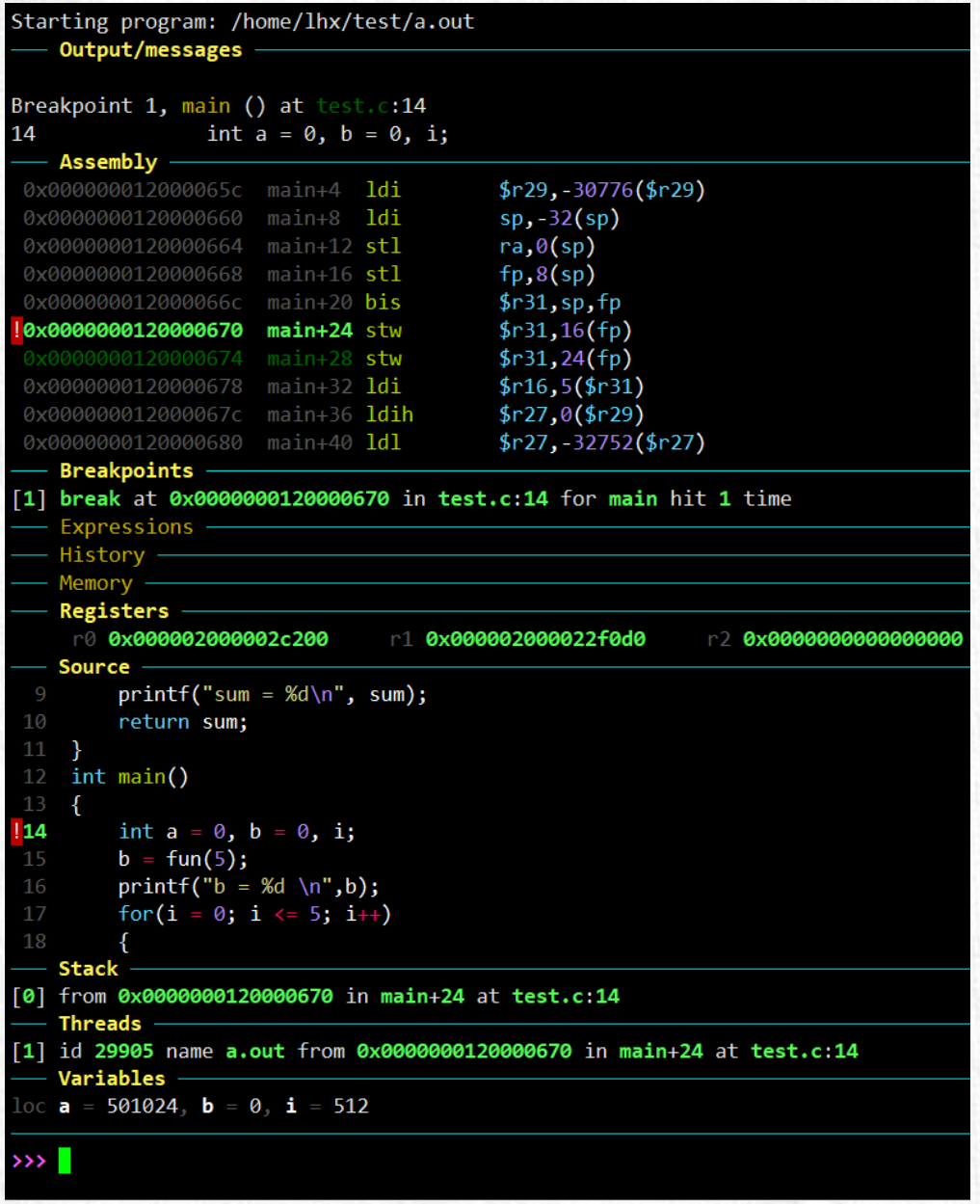
```
  2 int fun(int n)
  3 {
► 4     int j,sum = 0;
  5     for (j = 0; j <= n; j++)
  6     {
  7             sum  = sum + j;
  8     }
  9     return sum;
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x11ffde000          0x120000000 rw-p    22000 0      [stack]
          0x120000000          0x120002000 r-xp     2000 0      /home/lhx/test/a.out
          0x120010000          0x120012000 rw-p     2000 0      /home/lhx/test/a.out
      0x20000000000        0x2000002a000 r-xp    2a000 0      /usr/lib/ld-2.23.so
      0x2000002a000        0x2000002e000 rw-p     4000 0      [anon_2000002a]
      0x20000038000        0x2000003a000 r--p     2000 28000  /usr/lib/ld-2.23.so
      0x2000003a000        0x2000003c000 rw-p     2000 2a000  /usr/lib/ld-2.23.so
      0x2000004e000        0x20000216000 r-xp   1c8000 0      /usr/lib/libc-2.23.so
      0x20000216000        0x20000226000 ---p    10000 1c8000 /usr/lib/libc-2.23.so
      0x20000226000        0x2000022a000 r--p     4000 1c8000 /usr/lib/libc-2.23.so
      0x2000022a000        0x20000230000 rw-p     6000 1cc000 /usr/lib/libc-2.23.so
      0x20000230000        0x20000234000 rw-p     4000 0      [anon_20000230]
```

04 调试插件-dashboard

界面炫酷，组件众多，自定义配置，多终端显示：

- assembly 汇编信息
- breakpoints 断点信息
- expressions 表达式
- history 历史信息
- memory 内存信息
- registers 寄存器信息
- source 源代码
- stack 栈帧信息
- threads 多线程信息
- variables 变量信息



04 调试插件-dashboard

```
Source
 9      printf("sum = %d\n", sum);
10      return sum;
11  }
12  int main()
13  {
!14     int a = 0, b = 0, i;
15      b = fun(5);
16      printf("b = %d \n",b);
17      for(i = 0; i <= 5; i++)
18      {
```

```
Assembly
0x0000000120000658  main+0   ldi     $r29,-30776($r29)
0x000000012000065c  main+4   ldi     $r29,-30776($r29)
0x0000000120000660  main+8   ldi     sp,-32(sp)
0x0000000120000664  main+12  stl     ra,0(sp)
0x0000000120000668  main+16  stl     fp,8(sp)
0x000000012000066c  main+20  bis     $r31,sp,fp
!0x0000000120000670  main+24  stw     $r31,16(fp)
 0x0000000120000674  main+28  stw     $r31,24(fp)
0x0000000120000678  main+32  ldi     $r16,5($r31)
0x000000012000067c  main+36  ldih    $r27,0($r29)
0x0000000120000680  main+40  ldl     $r27,-32752($r27)
```

1 153.35.179.26

```
      r24 0x00000000f63d4e2e      r25 0x000000011ffff228
       ra 0x000002000007a564      r27 0x0000000120000658
      r28 0x00000000f63d4a75      r29 0x0000000120018e20
       sp 0x000000011ffff3c0
Stack
[0] from 0x0000000120000670 in main+24 at test.c:14
Threads
[1] id 29747 name a.out from 0x0000000120000670 in main+24 at tes
t.c:14
>>>
```

1 153.35.179.26

```
Variables
loc a = 501024, b = 0, i = 512
```

04 调试插件-dashboard

```
lhx@deepin-PC:~$ tty
/dev/pts/2
lhx@deepin-PC:~$
── Source ──
  9       printf("sum = %d\n", sum);
 10       return sum;
 11   }
 12   int main()
 13   {
!14       int a = 0, b = 0, i;
 15       b = fun(5);
```

```
lhx@deepin-PC:~$ tty
/dev/pts/5
lhx@deepin-PC:~$
── Assembly ──
 0x000000012000065c  main+4   ldi      $r29,-30776($r29)
 0x0000000120000660  main+8   ldi      sp,-32(sp)
 0x0000000120000664  main+12  stl      ra,0(sp)
 0x0000000120000668  main+16  stl      fp,8(sp)
 0x000000012000066c  main+20  bis      $r31,sp,fp
!0x0000000120000670  main+24  stw      $r31,16(fp)
 0x0000000120000674  main+28  stw      $r31,24(fp)
```

```
1 153.35.179.26
lhx@deepin-PC:~/test$ gdb -ex dash a.out
Reading symbols from a.out...
>>> dashboard source -output /dev/pts/2
>>> dashboard assembly -output /dev/pts/5
>>> dashboard variables -output /dev/pts/4
```

```
1 153.35.179.26
lhx@deepin-PC:~$ tty
/dev/pts/4
lhx@deepin-PC:~$
── Variables ──
loc a = 501024, b = 0, i = 512
```

Part04 不同调试场景案例分析

- 场景1：宏定义
- 场景2：段错误+core文件
- 场景3：release版
- 场景4：动态库
- 场景5：多进程
- 场景6：多线程+死锁

04 不同调试场景案例分析

场景1：调试宏定义

对于大量复杂宏定义，常规方法：

- 肉眼查看源码，自己展开
- 通过gcc -E预编译展开
- 通过gdb调试展开（要使用-g3选项）

```c
#include <stdio.h>
#define MACRO1(x) (++(x))
#define MACRO2(x) (MACRO1(x)+100)
#define MACRO3(x) (MACRO2(x)+200)
#define SHIFT 26
#define INSN_OPCODE(insn) ((insn & 0xfc000000) >> SHIFT)
int a = 5, b;
long insn = 0x075b0000, opcode;
int main(void)
{
    b = MACRO3(a);
    opcode = INSN_OPCODE(insn);
    printf("b = %d, opcode = %#04x\n", b, opcode);
    return 0;
}
```

```
(gdb) p MACRO3(a)
No symbol "MACRO3" in current context.
(gdb) p SHIFT
No symbol "SHIFT" in current context.
(gdb) p INSN_OPCODE(insn)
No symbol "INSN_OPCODE" in current context.
(gdb)
No symbol "INSN_OPCODE" in current context.
(gdb)
No symbol "INSN_OPCODE" in current context.
(gdb)
```

```
(gdb) p MACRO3(a)
$1 = 306
(gdb) p SHIFT
$2 = 26
(gdb) p INSN_OPCODE(insn)
$3 = 1
(gdb) macro expand MACRO3(a)
expands to: (((++(a))+100)+200)
(gdb) macro expand INSN_OPCODE(insn)
expands to: ((insn & 0xfc000000) >> 26)
(gdb)
```

04 不同调试场景案例分析

场景2：调试段错误+core文件

记得开启：

```
ulimit -c unlimited
```

仍未出现core文件，需配置：

```
/proc/sys/kernel/core_pattern
```

```
lhx@deepin:~/demo/test_core$ ls
test_core  test_core.c
lhx@deepin:~/demo/test_core$ ulimit -c unlimited
lhx@deepin:~/demo/test_core$ ./test_core
段错误 (核心已转储)
lhx@deepin:~/demo/test_core$ ls
core  test_core  test_core.c
lhx@deepin:~/demo/test_core$ gdb test_core core
Reading symbols from test_core...
[New LWP 27135]

warning: Unexpected size of section `.reg/27135' in core file.
Core was generated by `./test_core'.
Program terminated with signal SIGSEGV, Segmentation fault.

warning: Unexpected size of section `.reg/27135' in core file.
#0  0x0000000120000594 in fun () at test_core.c:6
6           *ptr = 0x0;
(gdb) bt
#0  0x0000000120000594 in fun () at test_core.c:6
#1  0x00000001200005d4 in main () at test_core.c:11
```

04 不同调试场景案例分析

场景3：调试release版

➢ 方法一

没有调试符号的函数，默认不进入，需开启set step-mode on，对照汇编调试

```
Breakpoint 2, main () at hello1.c:6
6           printf("a = %d\n", a);
(gdb) set step-mode on
(gdb) s
0x00000200001d9830 in printf () from /lib/libc.so.6.1
=> 0x00000200001d9830 <printf+0>:       07 00 bb ff     ldih    $r29,7($r27)
(gdb) bt
#0  0x00000200001d9830 in printf () from /lib/libc.so.6.1
#1  0x000000012000063c in main () at hello1.c:6
```

```
(gdb)
0x000002000010a978 in strchrnul () from /lib/libc.so.6.1
=> 0x000002000010a978 <strchrnul+136>:  88 07 e5 40     xor     $r7,$r5,$r8
(gdb) bt
#0  0x000002000010a978 in strchrnul () from /lib/libc.so.6.1
#1  0x00000200000bf82c in vfprintf () from /lib/libc.so.6.1
#2  0x00000200001d98bc in printf () from /lib/libc.so.6.1
#3  0x000000012000063c in main () at hello1.c:6
```

04 不同调试场景案例分析

场景3：调试release版

- 方法二

通过objcopy从debug版分离出调试信息，在gdb里加载

```
lhx@deepin:~/demo/test-release$ gdb test_release
Reading symbols from test_release...
(No debugging symbols found in test_release)
(gdb) b main
Function "main" not defined.
Breakpoint 1 (main) pending.
(gdb) r
Starting program: /home/lhx/demo/test-release/test_release
sum = 8
[Inferior 1 (process 28271) exited normally]
```

```
lhx@deepin:~/demo/test-release$ objcopy --only-keep-debug test_debug test_symbol
lhx@deepin:~/demo/test-release$ ls
test_debug  test_release  test_release.c  test_symbol
lhx@deepin:~/demo/test-release$ gdb -exec=test_release -symbol=test_symbol
Reading symbols from test_symbol...
(gdb) b main
Breakpoint 1 at 0x120000678: file test_release.c, line 11.
(gdb) r
Starting program: /home/lhx/demo/test-release/test_release

Breakpoint 1, main () at test_release.c:11
11          int a = 3, b = 4;
```

04 不同调试场景案例分析

场景4：调试动态库

- 编译glibc时添加：

--enable-debug=yes

- 编译a.out时添加：

-Wl,-rpath选项指定glibc

```
13          printf("i = %d\n", i);
(gdb) s
__printf (format=0x555555556004 "i = %d\n") at printf.c:28
28      {
(gdb) bt
#0  __printf (format=0x555555556004 "i = %d\n") at printf.c:28
#1  0x000055555555517b in main () at test2.c:13
(gdb) l
23
24      /* Write formatted output to stdout from the format string FORMAT.  */
25      /* VARARGS1 */
26      int
27      __printf (const char *format, ...)
28      {
29        va_list arg;
30        int done;
31
32        va_start (arg, format);
(gdb)
33        done = __vfprintf_internal (stdout, format, arg, 0);
34        va_end (arg);
35
36        return done;
```

04 不同调试场景案例分析

场景5：调试多进程

（1）gdb默认只跟踪父进程

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

int main(void) {
    pid_t pid;

    pid = fork();
    if (pid > 0)
    {
        printf("Parent\n");
        exit(0);
    }
    printf("Child\n");
    return 0;
}
```

```
Temporary breakpoint 1, main () at test_thread2.c:7
7           pid = fork();
(gdb) n
[Detaching after fork from child process 14571]
Child
8           if (pid > 0)
(gdb) n
10              printf("Parent\n");
(gdb) n
Parent
11              exit(0);
(gdb) n
[Inferior 1 (process 14568) exited normally]
(gdb)
```

04 不同调试场景案例分析

（2）同时调试父子进程，调试一个进程时，另一个进程处于挂起状态

```
(gdb) set detach-on-fork off
(gdb) start
Temporary breakpoint 1 at 0x120000678: file test_thread2.c, line 7.
Starting program: /home/lhx/demo/test-thread/test_thread2

Temporary breakpoint 1, main () at test_thread2.c:7
7           pid = fork();
(gdb) n
[New inferior 2 (process 14625)]
8           if (pid > 0)
(gdb) i inferiors
  Num  Description       Executable
* 1    process 14622     /home/lhx/demo/test-thread/test_thread2
  2    process 14625     /home/lhx/demo/test-thread/test_thread2
(gdb) n
10              printf("Parent\n");
(gdb) c
Continuing.
Parent
[Inferior 1 (process 14622) exited normally]
```

```
(gdb) i inferiors
  Num  Description       Executable
* 1    <null>            /home/lhx/demo/test-thread/test_thread2
  2    process 14625     /home/lhx/demo/test-thread/test_thread2
(gdb) inferior 2
[Switching to inferior 2 [process 14625] (/home/lhx/demo/test-thread/test_thread2)]
[Switching to thread 2.1 (process 14625)]
Reading symbols from /home/lhx/demo/test-thread/test_thread2...
#0  0x000002000013c794 in fork () from /lib/libc.so.6.1
(gdb) n
Single stepping until exit from function fork,
which has no line number information.
main () at test_thread2.c:8
8           if (pid > 0)
(gdb) n
13          printf("Child\n");
(gdb) c
Continuing.
Child
[Inferior 2 (process 14625) exited normally]
(gdb) i inferiors
  Num  Description       Executable
  1    <null>            /home/lhx/demo/test-thread/test_thread2
* 2    <null>            /home/lhx/demo/test-thread/test_thread2
```

04 不同调试场景案例分析

（3）父子进程都同时运行

```
(gdb) set detach-on-fork off
(gdb) set schedule-multiple on
(gdb) start
Temporary breakpoint 1 at 0x120000678: file test_thread2.c, line 7.
Starting program: /home/lhx/demo/test-thread/test_thread2

Temporary breakpoint 1, main () at test_thread2.c:7
7           pid = fork();
(gdb) n
[New inferior 2 (process 14750)]
Child
[Inferior 2 (process 14750) exited normally]
Reading symbols from /home/lhx/demo/test-thread/test_thread2...
(gdb) c
Continuing.
Parent
[Inferior 1 (process 14747) exited normally]
(gdb)
```

04 不同调试场景案例分析

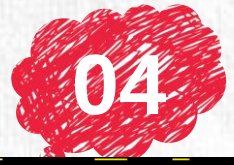
场景5：调试多线程死锁

```cpp
#include <iostream>
#include <mutex>
#include <thread>
#include <unistd.h>
using namespace std;
mutex m1, m2;
void thread1() {
  lock_guard< mutex > guard( m1 );
  printf("--> thread1 locked mutex1\n");
  printf("--> then sleep 1 seconds and try to lock mutex2\n");
  sleep( 1 );
  lock_guard< mutex > guard2( m2 );
}

void thread2() {
  lock_guard< mutex > guard( m2 );
  printf("--> thread2 locked mutex2\n");
  printf("--> then sleep 1 seconds and try to lock mutex1\n");
  sleep( 1 );
  lock_guard< mutex > guard2( m1 );
}

int main( int argc, char const* argv[] ) {
  thread t1( thread1 );
  thread t2( thread2 );

  t1.join();
  t2.join();
  return 0;
}
```

```
lhx@107:~/test-lock$ ./test_lock
--> thread1 locked mutex1
--> then sleep 1 seconds and try to lock mutex2
--> thread2 locked mutex2
--> then sleep 1 seconds and try to lock mutex1
^C
lhx@107:~/test-lock$ gdb test_lock
Reading symbols from test_lock...
(gdb) r
Starting program: /mnt/home/lhx/test-lock/test_lock
Traceback (most recent call last):
  File "/usr/lib/libstdc++.so.6.0.25-gdb.py", line 60, in <module>
    from libstdcxx.v6 import register_libstdcxx_printers
ModuleNotFoundError: No module named 'libstdcxx'
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/home/lhx/gdb/install/lib/libthread_db.so.1
[New Thread 0x400000daf1b0 (LWP 30466)]
--> thread1 locked mutex1
--> then sleep 1 seconds and try to lock mutex2
[New Thread 0x4000015b11b0 (LWP 30467)]
--> thread2 locked mutex2
--> then sleep 1 seconds and try to lock mutex1
^C
Thread 1 "test_lock" received signal SIGINT, Interrupt.
0x00004000003926e4 in __GI___pthread_timedjoin_ex (threadid=70368758526384,
89              lll_wait_tid (pd->tid);
(gdb)
```

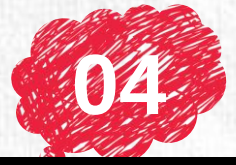
不同调试场景案例分析

```
(gdb) bt
#0  0x00004000003926e4 in __GI___pthread_timedjoin_ex (threadid=70368758526384, thread_return=0x0, abstime=<optimized out>, block=<optimized out>) at pthread_join_common.c:89
#1  0x000040000011f910 in std::thread::join() () from /lib/libstdc++.so.6
#2  0x0000000120001380 in main (argc=1, argv=0x11fffdf78) at test_lock.c:27
(gdb) f 2
#2  0x0000000120001380 in main (argc=1, argv=0x11fffdf78) at test_lock.c:27
27        t1.join();
(gdb) l
22
23      int main( int argc, char const* argv[] ) {
24        thread t1( thread1 );
25        thread t2( thread2 );
26
27        t1.join();
28        t2.join();
29        return 0;
30      }
(gdb) i threads
  Id   Target Id                                    Frame
* 1    Thread 0x400000031750 (LWP 30452) "test_lock" 0x0000000120001380 in main (argc=1, argv=0x11fffdf78) at test_lock.c:27
  2    Thread 0x400000daf1b0 (LWP 30466) "test_lock" __lll_lock_wait (futex=0x120014330 <m2>, private=<optimized out>) at lowlevellock.c:46
  3    Thread 0x4000015b11b0 (LWP 30467) "test_lock" __lll_lock_wait (futex=0x120014308 <m1>, private=<optimized out>) at lowlevellock.c:46
(gdb)
```



不同调试场景案例分析

```
(gdb) t 2
[Switching to thread 2 (Thread 0x400000daf1b0 (LWP 30466))]
#0  __lll_lock_wait (futex=0x120014330 <m2>, private=<optimized out>) at lowlevellock.c:46
46          lll_futex_wait (futex, 2, private); /* Wait if *futex == 2.  */
(gdb) bt
#0  __lll_lock_wait (futex=0x120014330 <m2>, private=<optimized out>) at lowlevellock.c:46
#1  0x00004000003943a8 in __GI___pthread_mutex_lock (mutex=0x120014330 <m2>) at pthread_mutex_lock.c:78
#2  0x00000001200015ac in __gthread_mutex_lock (__mutex=0x120014330 <m2>) at /usr/include/c++/8.3.0/sw_64-sunway-linux-gnu/bits/gthr-default.h:748
#3  0x0000000120001684 in std::mutex::lock (this=0x120014330 <m2>) at /usr/include/c++/8.3.0/bits/std_mutex.h:103
#4  0x00000001200018a8 in std::lock_guard<std::mutex>::lock_guard (this=0x400000dae8c8, __m=...) at /usr/include/c++/8.3.0/bits/std_mutex.h:162
#5  0x000000012000113c in thread1 () at test_lock.c:12
#6  0x0000000120001d48 in std::__invoke_impl<void, void (*)()> (__f=@0x120027e78: 0x120001094 <thread1()>) at /usr/include/c++/8.3.0/bits/invoke.h:60
#7  0x000000012000196c in std::__invoke<void (*)()> (__fn=@0x120027e78: 0x120001094 <thread1()>) at /usr/include/c++/8.3.0/bits/invoke.h:95
#8  0x0000000120002980 in std::thread::_Invoker<std::tuple<void (*)()> >::_M_invoke<0ul> (this=0x120027e78) at /usr/include/c++/8.3.0/thread:244
#9  0x00000001200028f8 in std::thread::_Invoker<std::tuple<void (*)()> >::operator() (this=0x120027e78) at /usr/include/c++/8.3.0/thread:253
#10 0x00000001200028a8 in std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)()> > >::_M_run (this=0x120027e70) at /usr/include/c++/8.3.0/thread:196
#11 0x000040000011f4e8 in ?? () from /lib/libstdc++.so.6
#12 0x0000400000390bdc in start_thread (arg=0x400000daf1b0) at pthread_create.c:486
#13 0x00004000004e1fb4 in thread_start () at ../sysdeps/unix/sysv/linux/sw_64/clone.S:100
(gdb) f 5
#5  0x000000012000113c in thread1 () at test_lock.c:12
12        lock_guard< mutex > guard2( m2 );
(gdb) l
7       void thread1() {
8         lock_guard< mutex > guard( m1 );
9         printf("--> thread1 locked mutex1\n");
10        printf("--> then sleep 1 seconds and try to lock mutex2\n");
11        sleep( 1 );
12        lock_guard< mutex > guard2( m2 );
13      }
14
15      void thread2() {
16        lock_guard< mutex > guard( m2 );
(gdb)
```



不同调试场景案例分析

```
(gdb) t 3
[Switching to thread 3 (Thread 0x4000015b11b0 (LWP 30467))]
#0  __lll_lock_wait (futex=0x120014308 <m1>, private=<optimized out>) at lowlevellock.c:46
46          lll_futex_wait (futex, 2, private); /* Wait if *futex == 2.  */
(gdb) bt
#0  __lll_lock_wait (futex=0x120014308 <m1>, private=<optimized out>) at lowlevellock.c:46
#1  0x00004000003943a8 in __GI___pthread_mutex_lock (mutex=0x120014308 <m1>) at pthread_mutex_lock.c:78
#2  0x00000001200015ac in __gthread_mutex_lock (__mutex=0x120014308 <m1>) at /usr/include/c++/8.3.0/sw_64-sunway-linux-gnu/bits/gthr-default.h:748
#3  0x0000000120001684 in std::mutex::lock (this=0x120014308 <m1>) at /usr/include/c++/8.3.0/bits/std_mutex.h:103
#4  0x00000001200018a8 in std::lock_guard<std::mutex>::lock_guard (this=0x4000015b08c8, __m=...) at /usr/include/c++/8.3.0/bits/std_mutex.h:162
#5  0x0000000120001274 in thread2 () at test_lock.c:20
#6  0x0000000120001d48 in std::__invoke_impl<void, void (*)()> (__f=@0x120027fc8: 0x1200011cc <thread2()>) at /usr/include/c++/8.3.0/bits/invoke.h:60
#7  0x000000012000196c in std::__invoke<void (*)()> (__fn=@0x120027fc8: 0x1200011cc <thread2()>) at /usr/include/c++/8.3.0/bits/invoke.h:95
#8  0x0000000120002980 in std::thread::_Invoker<std::tuple<void (*)()> >::_M_invoke<0ul> (this=0x120027fc8) at /usr/include/c++/8.3.0/thread:244
#9  0x00000001200028f8 in std::thread::_Invoker<std::tuple<void (*)()> >::operator() (this=0x120027fc8) at /usr/include/c++/8.3.0/thread:253
#10 0x00000001200028a8 in std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)()> > >::_M_run (this=0x120027fc0) at /usr/include/c++/8.3.0/thread:196
#11 0x000040000011f4e8 in ?? () from /lib/libstdc++.so.6
#12 0x0000400000390bdc in start_thread (arg=0x4000015b11b0) at pthread_create.c:486
#13 0x00004000004e1fb4 in thread_start () at ../sysdeps/unix/sysv/linux/sw_64/clone.S:100
(gdb) f 5
#5  0x0000000120001274 in thread2 () at test_lock.c:20
20        lock_guard< mutex > guard2( m1 );
(gdb) l
15      void thread2() {
16        lock_guard< mutex > guard( m2 );
17        printf("--> thread2 locked mutex2\n");
18        printf("--> then sleep 1 seconds and try to lock mutex1\n");
19        sleep( 1 );
20        lock_guard< mutex > guard2( m1 );
21      }
22
23      int main( int argc, char const* argv[] ) {
24        thread t1( thread1 );
(gdb)
```

04 不同调试场景案例分析

场景6：远程调试

适用场景：

- 开发板环境下程序报错，无法直接运行gdb调试
- 服务器端环境下程序报错，而开发环境不能复现bug

sw 70节点

```
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$ ./gdbserver 172.16.130.63:6
666 ~/test/a.out
Process /home/lhx/test/a.out created; pid = 23284
Listening on port 6666
Remote debugging from host 172.16.130.63
b = 6
a = 6
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
lhx@deepin-PC:/mnt/sdb/lhx/gdb712/install-gdbserver/bin$
```

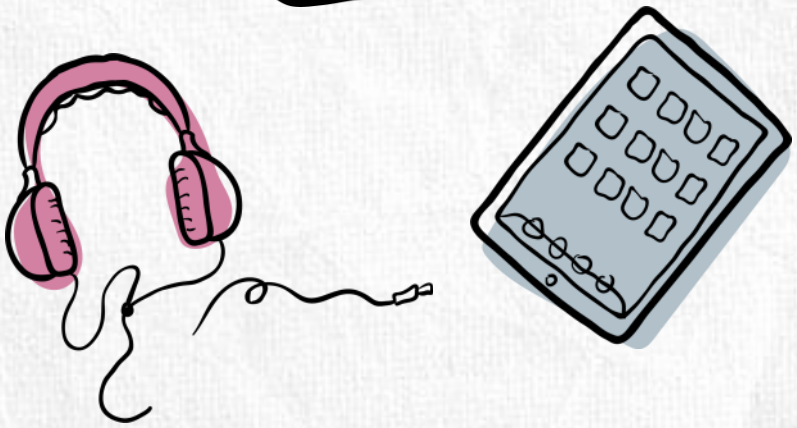
sw 63节点

```
(gdb) target remote 172.16.130.70:6666
Remote debugging using 172.16.130.70:6666
warning: while parsing target description (at line 11): Could not load XML document
"sw64-cpu.xml"
warning: Could not load XML target description; ignoring
Reading /home/lhx/test/a.out from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access
 files locally instead.
Reading /home/lhx/test/a.out from remote target...
Reading symbols from target:/home/lhx/test/a.out...done.
Reading /lib/ld-linux.so.2 from remote target...
Reading /lib/ld-linux.so.2 from remote target...
Reading symbols from target:/lib/ld-linux.so.2...Reading /lib/7d79705c14d7d2eba40cd6
4bff931e8a403190.debug from remote target...
Reading /lib/.debug/7d79705c14d7d2eba40cd64bff931e8a403190.debug from remote target.
..
(no debugging symbols found)...done.
0x000002000790e2e0 in ?? () from target:/lib/ld-linux.so.2
(gdb) b main
```




谢 谢 ！

Q&A



■ 主讲人：刘汉旭　　■ 时间：2022.06.16