探索调试之美

—— 调试器原理、实践和技巧

部　门：生态软件研发部

汇报人：刘汉旭

日　期：2022年3月4日

目录

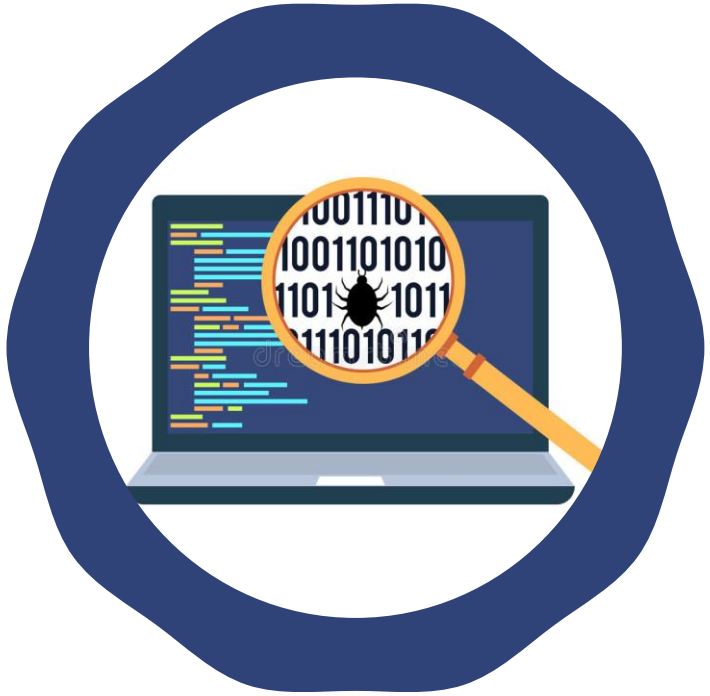
CONTENT

01

调试器介绍

What is a debugger ?

1. 为什么要调试？



Why Debugging ?

- 软件开发过程中，经常出现意料之外的结果，无从下手
- 从写代码、测试、后期维护，bug 无处不在
- 据统计，程序员20%时间写代码，80%时间调试

2. 程序调试的方法

How to fix bugs ?

- 硬看代码（人工检查每行代码）
- 添加打印信息（printf、error）
- 调试器（gdb、lldb、WinDbg）
- IDE 调试（VS、Eclipse、QtCreator、Clion）

Linux环境下常用调试方法：

① gdb命令行

② vscode + gdb

③ eclipse + gdb

3. 当前主流的调试器

名称	诞生	描述	语言支持	操作系统支持	最新版
GDB	1986	GNU Debugger	Assembly,C,C++,Fortran,Go,Objective-C, Rust, …	Windows, Linux, OS X	11.2, 2022
Visual Studio Debugger	1995	Debugger in Microsoft Visual Studio	C++,JavaScript, .net	Windows	Vs2022,2022
LLDB	2003	LLVM Debugger	C, Objective-C and C++	Windows, Linux, Mac OS, iOS	13.0.1, 2022
Valgrind	2007	tool suite for debugging and profiling Linux programs	C, C++, Java, Perl, Python, assembly code, Fortran, ...	Windows, Linux, Mac OS, Android	3.18.1, 2021
WinDbg	2007	Windows Debugger	Assembly,C,C++, C#, …	Windows	10.0, 2021
Pdb	2016	Python debugger	Python	Windows, Linux, Mac OS	3.10, 2021
Delve	2018	Debugger for the Go	Go	Windows, Linux, Mac OS	1.8.1, 2022

4. 三大平台常用调试器

Linux

GDB（各大Linux发行版默认调试器）、
LLDB ……

Windows

Visual Studio Debugger（Visual Studio自带的调试器）、
WinDbg ……

Mac OS

LLDB（Xcode默认调试器，Mac OS下开发必备调试器）、
GDB、 Visual Studio Debugger ……

5. 调试器主要功能

1. **控制程序执行过程：**启动、暂停、继续、跟踪 、attach、多线程、多进程
2. **读取程序状态信息：**变量、传参、CPU寄存器、内存数据、函数栈帧 ……
3. **修改程序运行数据：**修改变量值、寄存器值、内存地址 ……
4. **显示程序代码：**显示源代码、汇编代码、机器码……
5. **远程调试：**调试开发板、FPGA……
6. **GUI 图形界面：**各种窗口（源码/汇编/寄存器/变量/内存/线程 ……）

……

02

调试器原理

How debugger work ?

1. Linux系统调用 -- Ptrace

Ptrace 允许一个进程来获取其他进程的信息。

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

其中 request 参数指定了要使用 ptrace 的功能, 大致分为以下几类:

- 建立进程间的跟踪关系：

PTRACE_ATTACH 、PTRACE_TRACEME

- 读写被调试进程的内存（text 代码段，data 数据段）：

PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSR

PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSR

- 读写被调试进程的CPU寄存器：

PTRACE_GETREGSET、PTRACE_SETREGS

- 控制被调试进程的执行：

PTRACE_CONT、PTRACE_SINGLESTEP、PTRACE_KILL

1. Linux系统调用 -- Ptrace

```
asmlinkage int sys_ptrace(long request, long pid, long addr, long data)
{
child = find_task_by_pid(pid); // 获取 pid 对应的进程 task_struct 对象
if (request == PTRACE_ATTACH) {
        ret = ptrace_attach(child);
        goto out_tsk;
    }
    switch (request) {
    case PTRACE_PEEKTEXT:
    case PTRACE_PEEKDATA:
        ...
    case PTRACE_PEEKUSR:
        ...
    case PTRACE_POKETEXT:
    case PTRACE_POKEDATA:
        ...
```

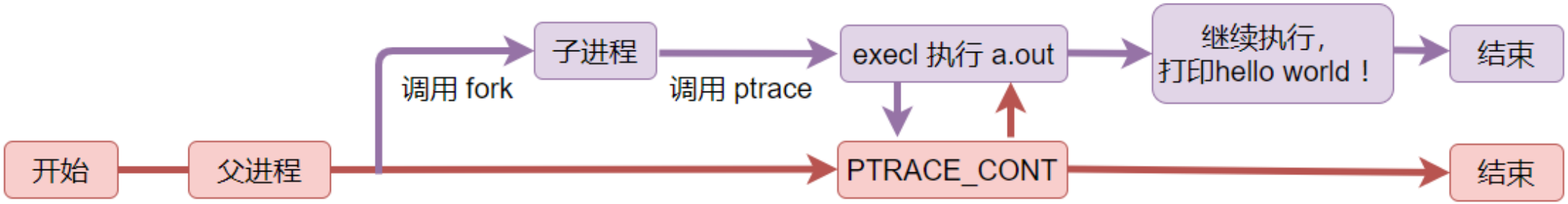
```
    case PTRACE_POKEUSR:
        ...
    case PTRACE_SYSCALL:
    case PTRACE_CONT:
        ...
    case PTRACE_KILL:
        ...
    case PTRACE_SINGLESTEP:
        ...
    case PTRACE_DETACH:
        ...
    }
}
```

```
#define PTRACE_TRACEME          0
#define PTRACE_PEEKTEXT         1
#define PTRACE_PEEKDATA         2
#define PTRACE_PEEKUSR          3
#define PTRACE_POKETEXT         4
#define PTRACE_POKEDATA         5
#define PTRACE_POKEUSR          6
#define PTRACE_CONT             7
#define PTRACE_KILL             8
#define PTRACE_SINGLESTEP       9
#define PTRACE_ATTACH           0x10
#define PTRACE_DETACH           0x11
#define PTRACE_SYSCALL          24
#define PTRACE_GETREGS          12
#define PTRACE_SETREGS          13
#define PTRACE_GETFPREGS        14
#define PTRACE_SETFPREGS        15
#define PTRACE_GETFPXREGS       18
#define PTRACE_SETFPXREGS       19
#define PTRACE_SETOPTIONS       21
```

2. Ptrace 实现父子进程交互 -- fork

```c
#include <stdio.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

int main()
{
    pid_t child;
    child = fork();
    if(child == 0) // child process
    {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        printf("child to call execl, run a.out\n");
        execl("/home/lhx/test/a.out", "a.out", NULL); // run a.out and send signal
        printf("child exit\n");
    } else { // parent process
        wait(NULL); // wait for child process signal
        ptrace(PTRACE_CONT, child, NULL, NULL); // tell child to continue
        printf("after child continue, parent exit\n");
    }
    return 0;
}
```

```
lhx@ubuntu:~/test/demo1$ ./test-fork
child to call execl, run a.out
after child continue, parent exit
lhx@ubuntu:~/test/demo1$ Hell World !
```



2. Ptrace -- traceme

```
asmlinkage int sys_ptrace(long request, long pid, long addr, long
data)
{
    ...
    if (request == PTRACE_TRACEME) {
        if (current->ptrace & PT_PTRACED)
            goto out;
        current->ptrace |= PT_PTRACED; // 标志 PTRACE 状态
        ret = 0;
        goto out;
    }
}
static int load_elf_binary(struct linux_binprm * bprm, struct
pt_regs * regs)
{
    ...
    if (current->ptrace & PT_PTRACED)
        send_sig(SIGTRAP, current, 0);
    ...
}
```

2. Ptrace -- traceme

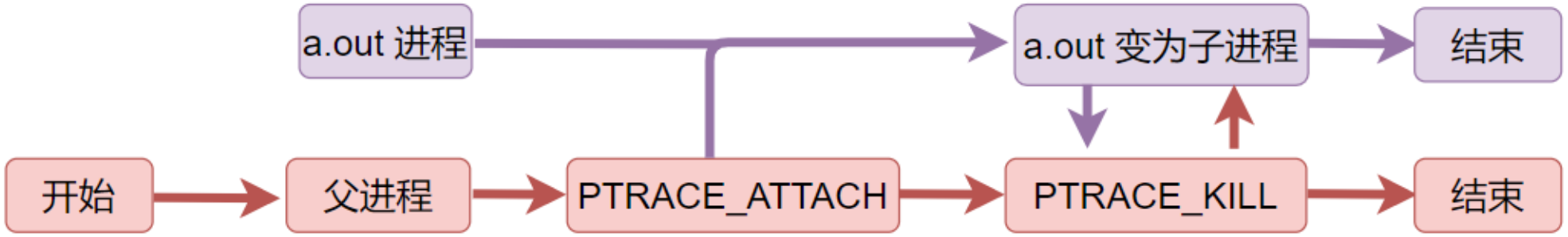
```
int do_signal(struct pt_regs *regs, sigset_t *oldset)
{
    for (;;) {
        unsigned long signr;

        spin_lock_irq(&current->sigmask_lock);
        signr = dequeue_signal(&current->blocked, &info);
        spin_unlock_irq(&current->sigmask_lock);

        // 如果进程被标记为 PTRACE 状态
        if ((current->ptrace & PT_PTRACED) && signr != SIGKILL) {
            /* 让调试器运行  */
            current->exit_code = signr;
            current->state = TASK_STOPPED;   // 让自己进入停止运行状态
            notify_parent(current, SIGCHLD); // 发送 SIGCHLD 信号给父进程
            schedule();                      // 让出CPU的执行权限
            ...
        }
    }
}
```

3. Ptrace 实现父子进程交互 -- attach

```c
#include <stdio.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

int main(int argc, char** argv)
{
    pid_t child;
    child = atoi (argv[1]);
    printf("try to attach child: %d\n", child);
    int ret = ptrace(PTRACE_ATTACH, child, NULL, NULL); // attch child process
    if (ret == 0)
      printf("attach child %d success !\n", child);
    else
      printf("attach error !\n");
    wait(NULL); // wait for child process signal
    ptrace(PTRACE_KILL, child, NULL, NULL); // kill child to process
    printf("kill child success !\n");
    return 0;
}
```

```
lhx@ubuntu:~/test/demo2$ ./a.out
[0] Hell World ! (pid = 4558)
[1] Hell World ! (pid = 4558)
[2] Hell World ! (pid = 4558)
[3] Hell World ! (pid = 4558)
已杀死
lhx@ubuntu:~/test/demo2$
```

```
lhx@ubuntu:~/test/demo2$ sudo ./test-attach 4546
try to attach child: 4546
attach child 4546 success !
kill child success !
lhx@ubuntu:~/test/demo2$
```



4. Ptrace 实现修改子进程内存 -- pokedata

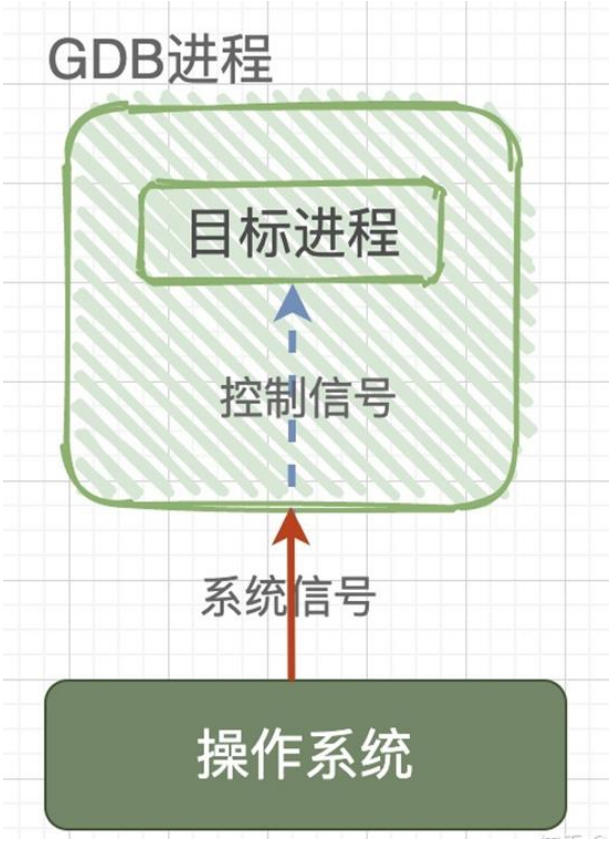
```c
#include <stdio.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    char c = 'B';  //把输出修改为'B'
    long addr = 0x0000000000404038;  // c.out的变量c的偏移地址
    pid_t apid = atoi(argv[1]);
    ptrace(PTRACE_ATTACH, apid, 0, 0); //attach进程号为apid的进程，进行进程内存访问
    wait(NULL);
    ptrace(PTRACE_POKEDATA, apid, addr, c); //对addr的偏移地址内容修改为c
    ptrace(PTRACE_CONT, apid, 0, 0);
    ptrace(PTRACE_DETACH, apid, NULL, NULL);
    return 0;
}
```

```
lhx@ubuntu:~/test/demo3$ ./a.out
A
A
A
A
A
A
A
A
A
B
B
```

```
lhx@ubuntu:~/test/demo3$ ps -aux | grep a.out
lhx         4930  0.0  0.0   2496   520 pts/0    S+   15:04
   0:00 ./a.out
lhx         4932  0.0  0.0  12132   728 pts/1    S+   15:04
   0:00 grep --color=auto a.out
lhx@ubuntu:~/test/demo3$ sudo ./test-memory 4930
lhx@ubuntu:~/test/demo3$
```

5. GDB 调试原理

- ✓ 程序正常运行时，操作系统与目标进程之间是直接交互的；当使用gdb调试，产生追踪者（debugger）和被追踪者（debugee）
- ✓ gdb进程通过ptrace接管了目标进程a.out，操作系统向a.out发送的所有信号，都被gdb接收到，从而达到调试的目的。
- ✓ gdb进程通过ptrace读写a.out进程的指令空间、数据空间、堆栈和寄存器的值。
- ✓ 当追踪时， a.out每发一个信号就会停一次，即使这个signal被会忽略掉。 gdb将通过waitpid捕捉到，并通过ptrace来监控修改a.out，然后gdb会告诉a.out继续运行



5. GDB 调试原理

GDB 命令	Ptrace 底层支持
run	PTRACE TRACEME
attach	PTRACE_ATTACH
continue	PTRACE_CONT
step	PTRACE SINGLESTEP
info registers	PTRACE_GET(FP)REGS / and PTRACE_SET(FP)REGS
x	PTRACE PEEKTEXT / PTRACE POKETEXT PTRACE PEEKDATA / PTRACE POKEDATA
……	……

6. GDB 断点原理

假设想让a.out在addr处停下来，那么GDB会做如下事情。

① 读取addr处的指令，存入GDB维护的断点链表中。

② 将中断指令int 3 （0xCC）打入原本的addr处，指令掉换成int 3。

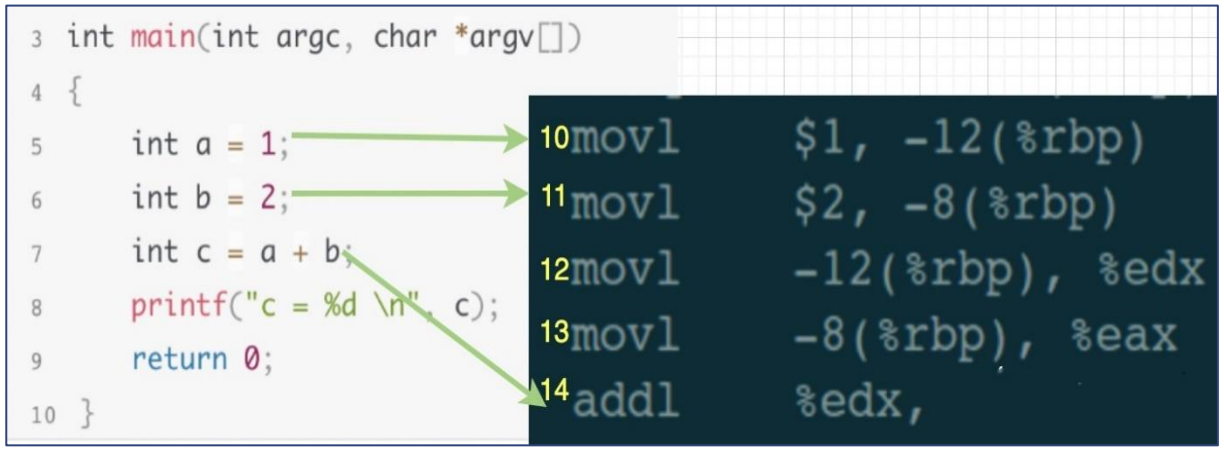
③ 当执行到addr处时，CPU执行该指令时会发生断点异常（breakpoint exception），a.out产生一个SIGTRAP。

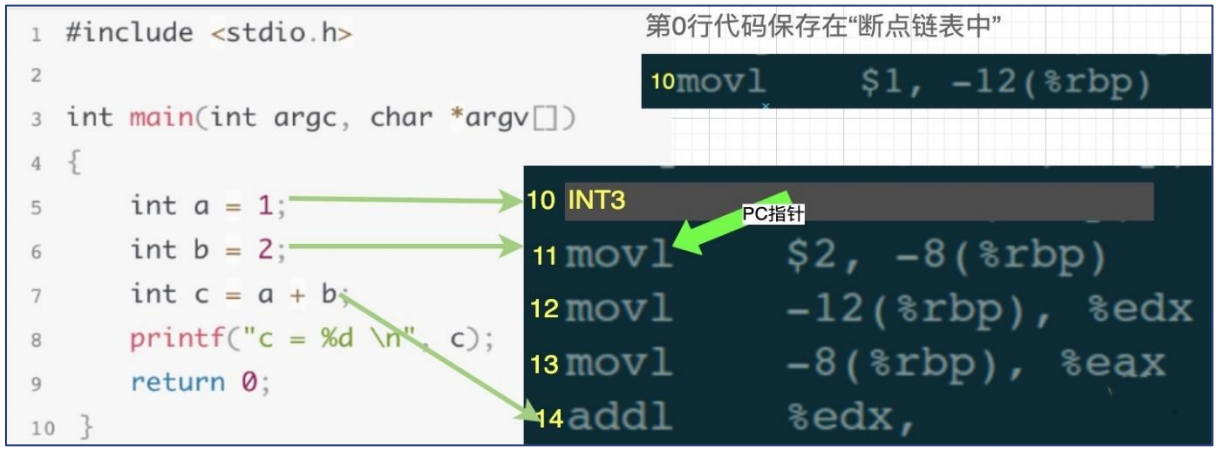
④ 由于处于被跟踪模式，a.out的SIGTRAP会被GDB捕捉。

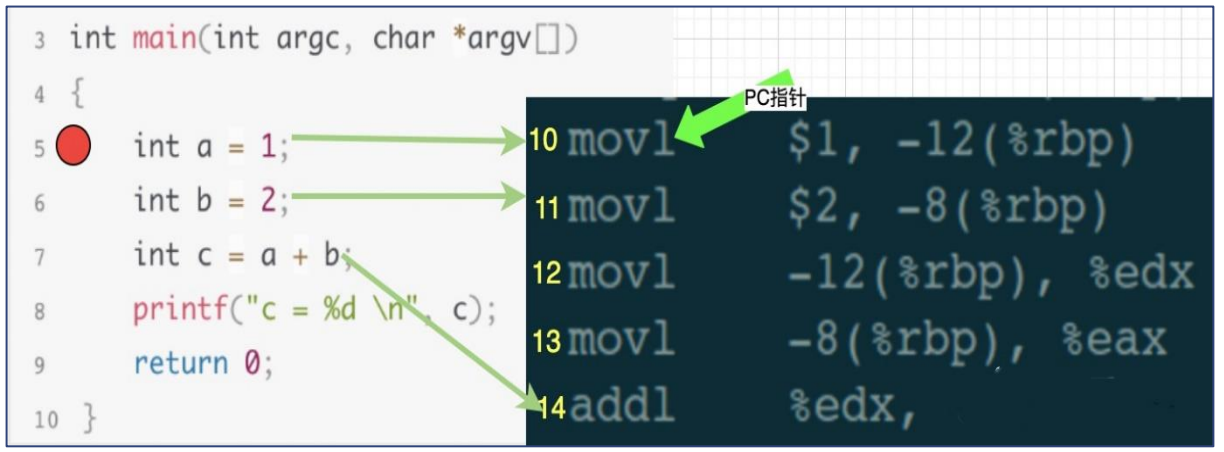
⑤ GDB去断点链表中查找对应位置，如果找到了，说明hit到了breakpoint。

⑥ 接下来若继续正常运行，GDB将int 3指令换回原来正常的指令，ip减1后接着运行。

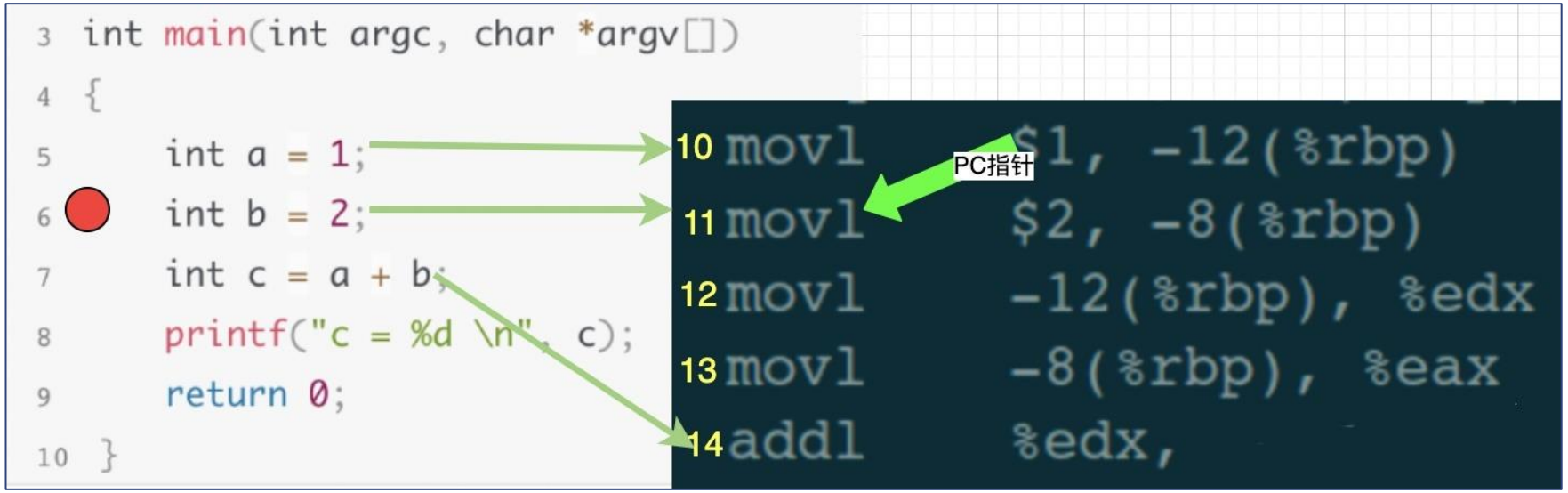
7. 偷窥GDB如何实现断点

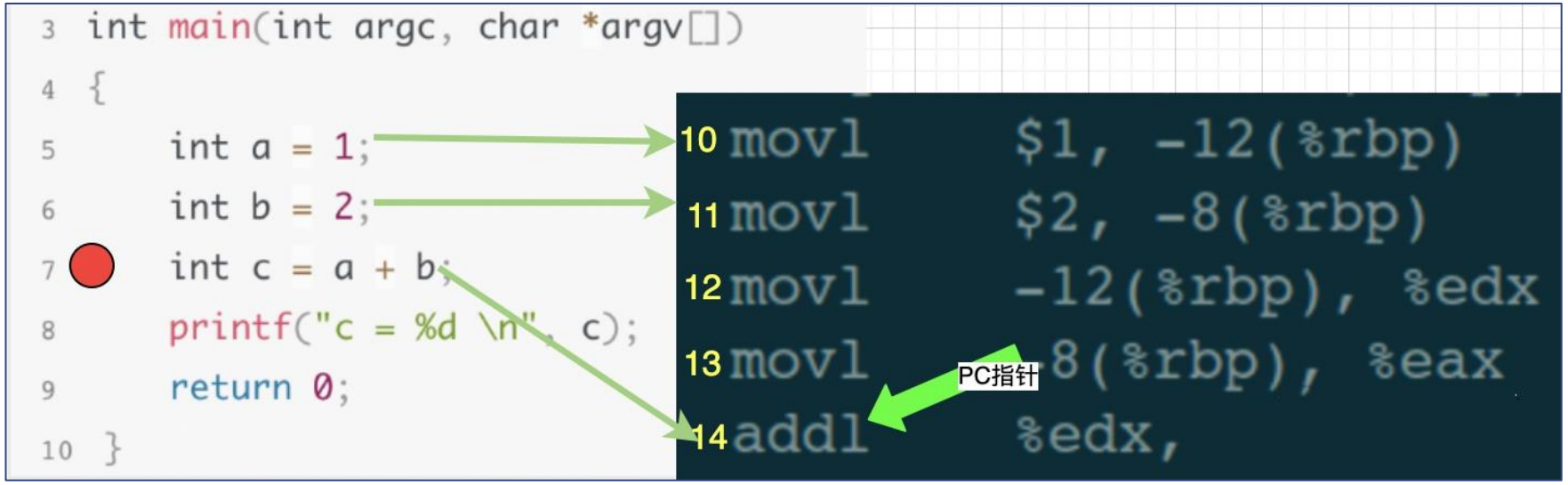




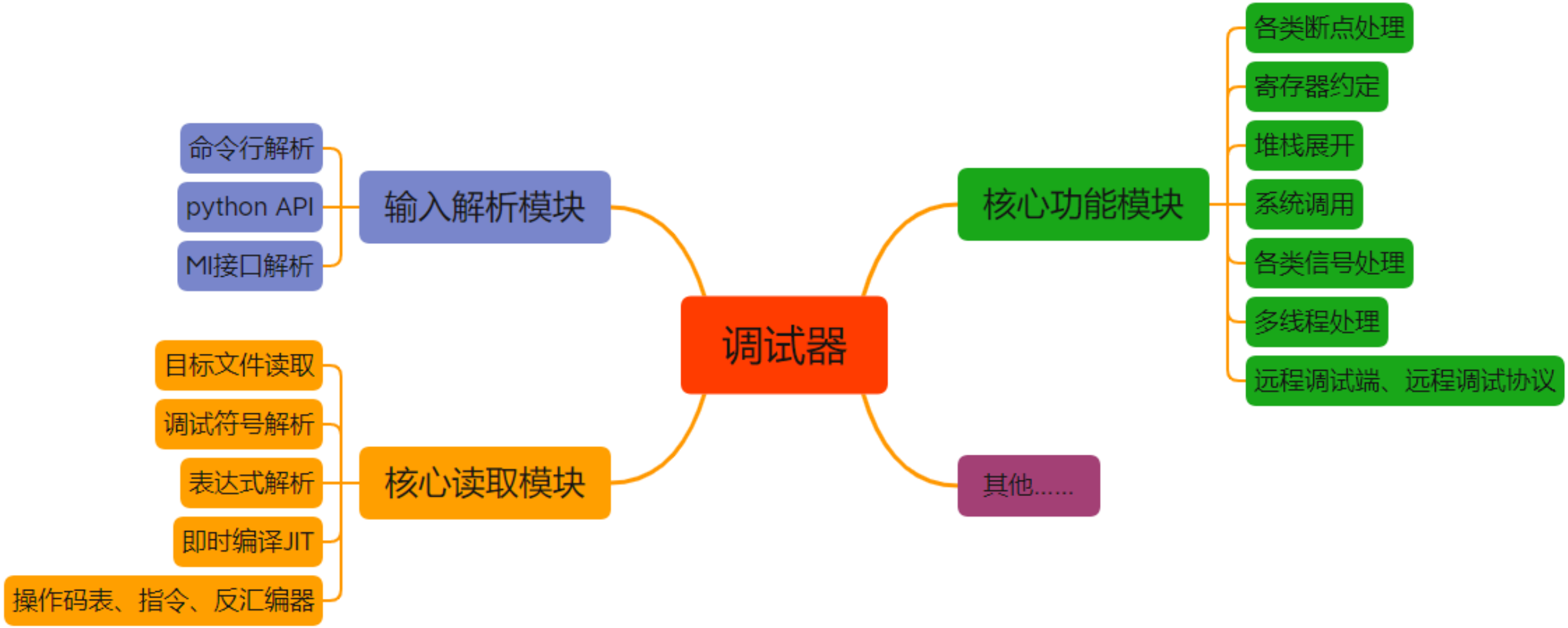


8. 偷窥GDB如何实现单步





9. GDB 模块结构图（代码约335万行）



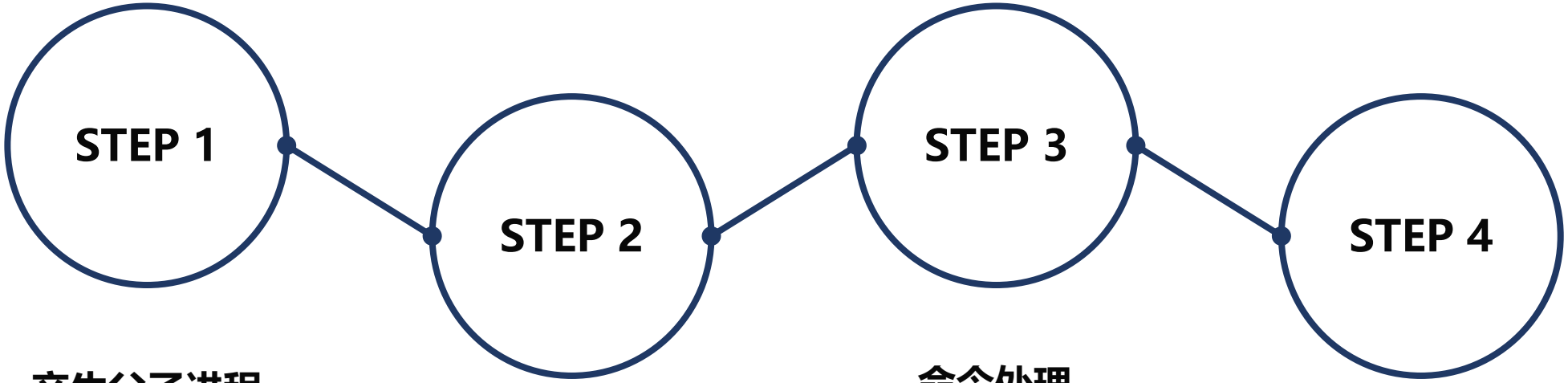
03

手撸一个mini版调试器

Let’s write a mini debugger !

1. 调试器的基本框架

- 编写一个简单的调试器，主要分为以下四步：



产生父子进程

通过 fork、execl

ptrace_traceme实现

进入命令循环

获取用户输入命令，识

别退出命令和调试命令

命令处理

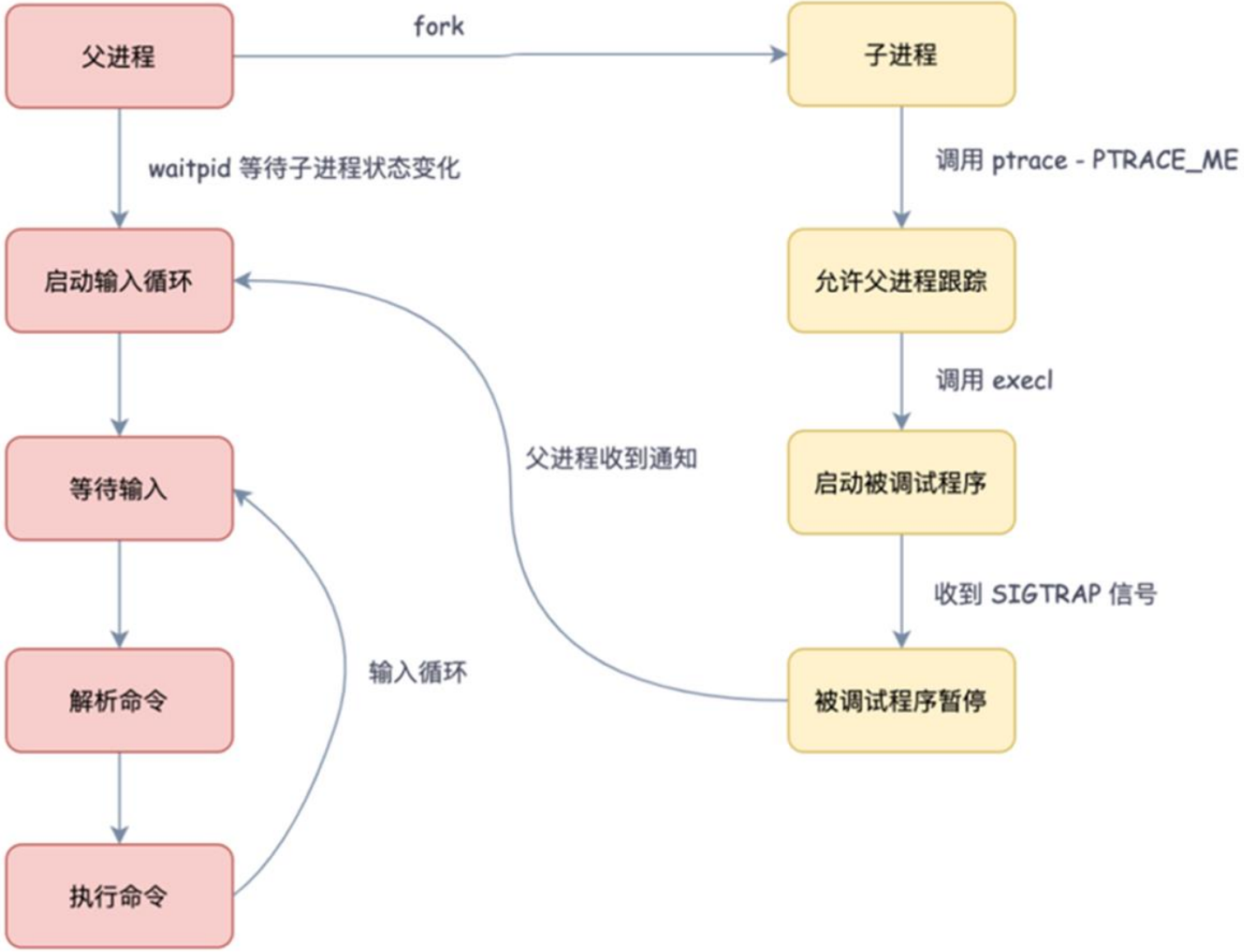
命令匹配、解析，

进入对应处理模块

实现命令功能

继续运行、设置断点

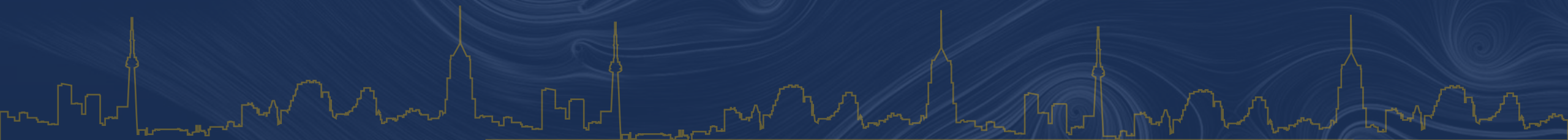
2. 调试器的运行流程


父进程
fork
子进程
waitpid 等待子进程状态变化
调用 ptrace - PTRACE_ME
启动输入循环
允许父进程跟踪
调用 execl
等待输入
父进程收到通知
启动被调试程序
收到 SIGTRAP 信号
解析命令
输入循环
被调试程序暂停
执行命令

04

高级调试技巧

Advanced Debugging Tips



Tips 1. 反向调试

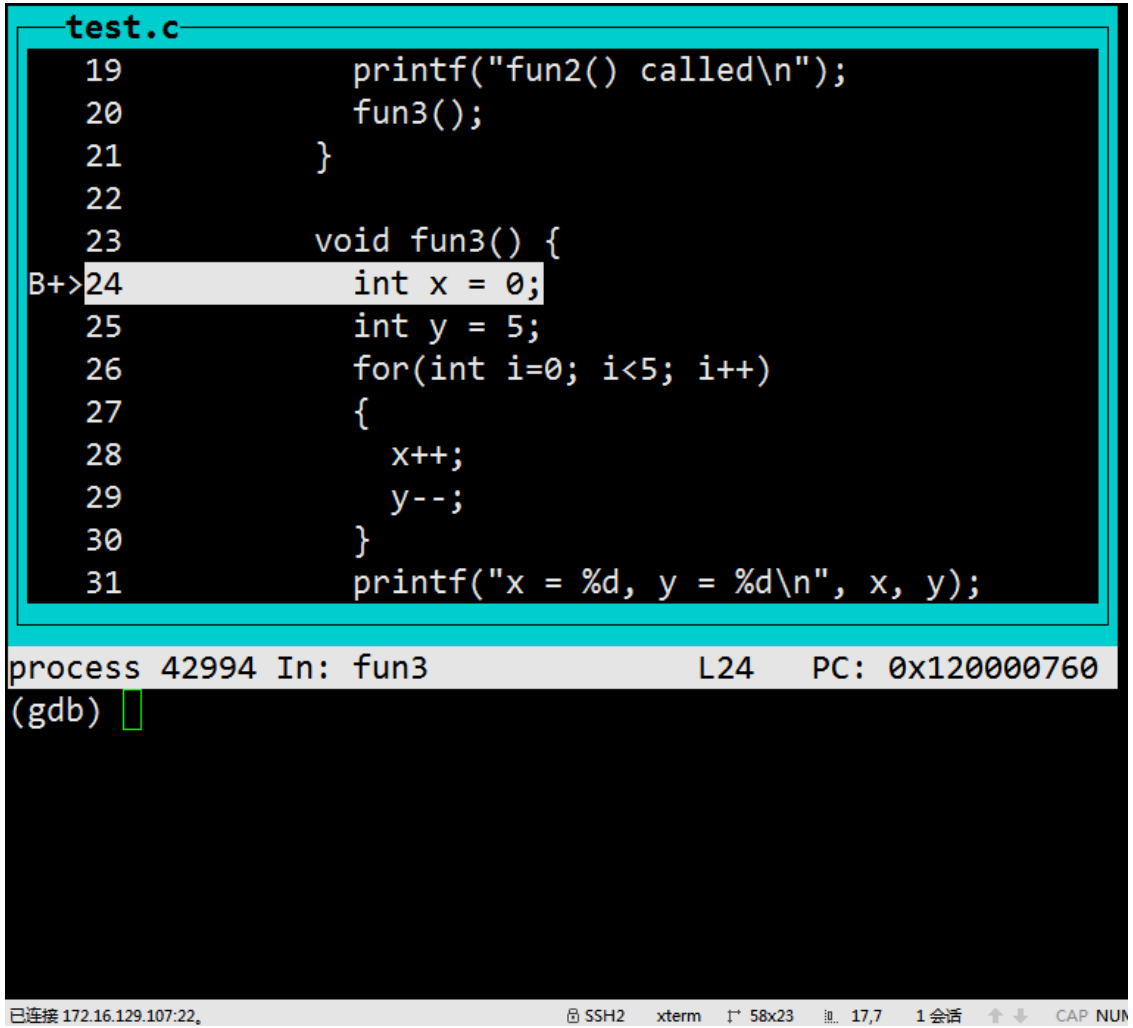
- ✓ 调试查错过程中，因为多跳过几步而错过关键代码，从而重来一遍，伤神费力！！
- ✓ 有没有后悔药可以吃，用来逆转时间？？

◘ GDB 反向调试（Reverse Debugging）帮你解决，其原理为：

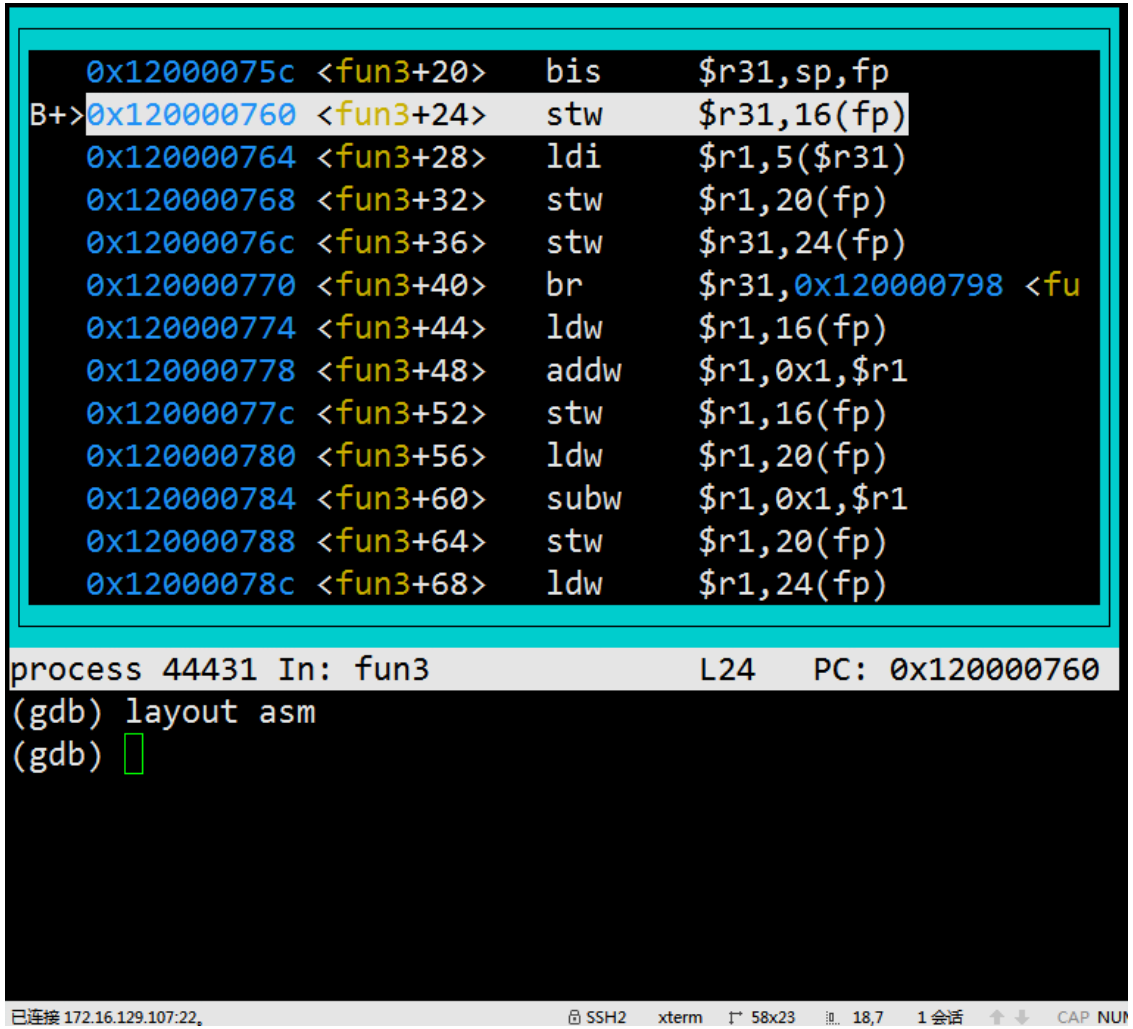
- 通过record功能来记录CPU执行过的指令流、使用过的寄存器和操作过的内存地址等信息，从而将程序运行的详细状态轨迹进行保存。
- 想要恢复还原时，只需要通过replay功能，撤消每条机器指令修改的寄存器或内存地址，还原出原始指令信息，从而可依次移除每条指令的执行效果，将被调试进程恢复到历史状态。
- rn / rs / rc / rf / set exec-direction [forward | reverse]

Tips 1. 反向调试

源码级反向



指令级反向



Tips 2. 动态打印

- ✓ 添加打印到代码后，突然发现打印位置不对，或其他地方也需要添加打印
- ✓ 重新改源码、加打印、编译、运行，结果不幸发现了新的问题，继续重复……
- ✓ 定位出问题后，添加的调试信息还要删除掉

◘ GDB + log 打印 帮你解决，随时随地printf，不需修改代码，不需重新编译，其原理为：

- GDB 动态打印，本质是一种特殊断点。动态打印断点被触发后，程序暂时中断执行，无需等待用户响应，直接执行格式化打印，并自动恢复程序执行。

- dprintf 5, "Hello, World!\n"
- dprintf 8, "a = %d, b = %d\n", a, b

Tips 2. 动态打印

```
(gdb) l
1       #include<stdio.h>
2       int main()
3       {
4         int sum = 0;
5         for(int i=0; i<=10; i++)
6         {
7           sum = sum + i;
8         }
9         return 0;
10      }
(gdb) dprintf 7, "sum=%d\n", sum
Dprintf 1 at 0x1141: file a.c, line 7.
(gdb) i b
Num     Type           Disp Enb Address            What
1       dprintf        keep y   0x0000000000001141 in main at a.c:7
        printf "sum=%d\n", sum
```

```
(gdb) r
Starting program: /home/lhx/test/tips/a.out
sum=0
sum=0
sum=1
sum=3
sum=6
sum=10
sum=15
sum=21
sum=28
sum=36
sum=45
[Inferior 1 (process 6278) exited normally]
(gdb)
```

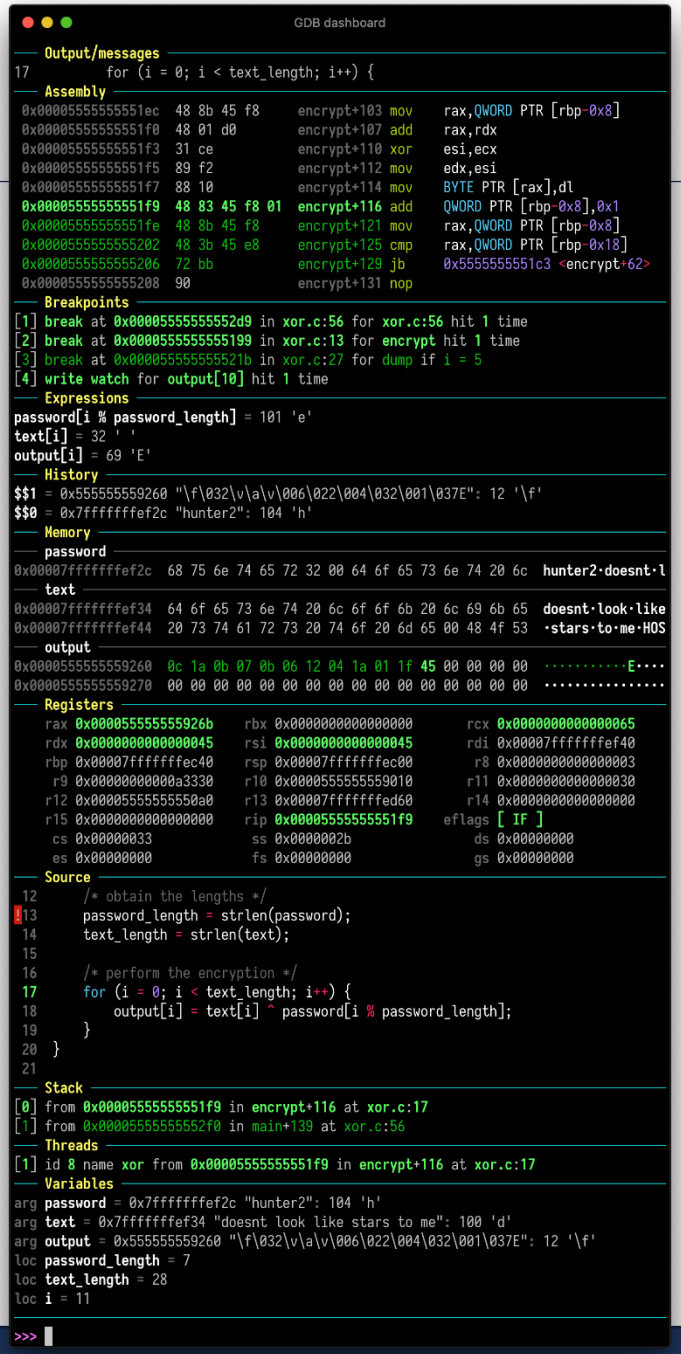
Tips 3. 图形界面高效调试

✓ 命令行调试，操作不友好，调试界面显示内容较少，必须通过敲命令查看？？

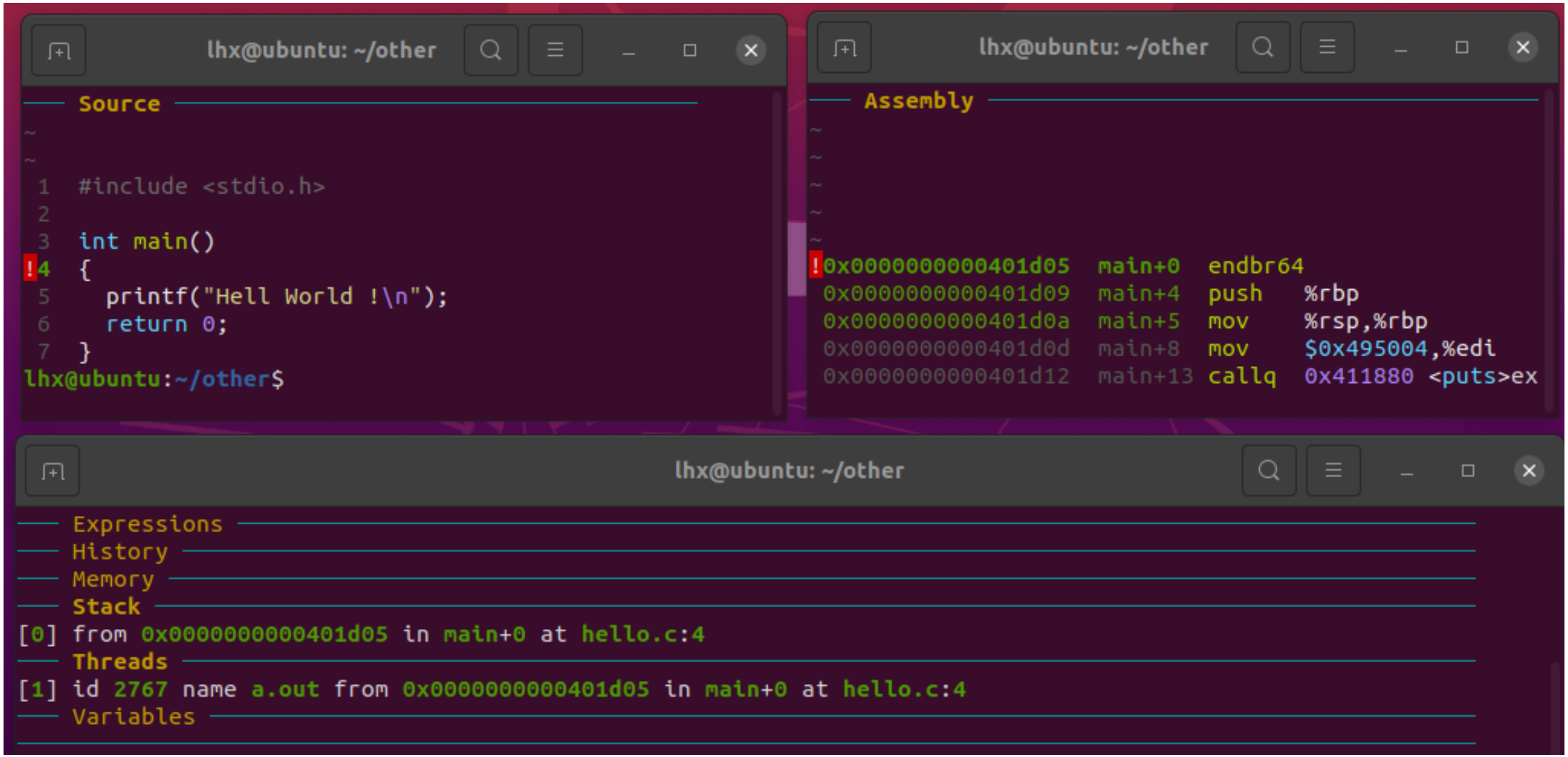
GDB python脚本扩展功能帮你解决：

- 自定义调试界面，堪比IDE，显示美观、启动快，完全基于终端。
- 界面是使用 Python API 编写，通过调用GDB提供的接口来操作。

Tips 3. 图形界面高效调试



Tips 3. 图形界面高效调试



Tips 4. 高效命令+快捷键

- ✓ Ctrl + d ： 快速退出调试
- ✓ Ctrl + x + a：快速切换图形调试界面
- ✓ Ctrl + x + 2：使图形界面显示两个窗口，可切换
- ✓ Ctrl + p/n：在图形界面上翻/下翻命令
- ✓ Ctrl+ l：清屏，图形界面可能会造成控制台花屏
- ✓ i args：打印所有传参
- ✓ i loc：打印当前栈帧所有变量

针对日常开发中各类报错的调试思路、方法和技巧，

后期会有专题培训，敬请关注~~

T

H

谢 谢 ！

THANK YOU FOR LISTENING !