GDB调试原理与技巧探讨

柳少锋 2018.8

度秘事业部

提纲

- 预备知识
- 调试原理探秘
- 调试技巧探讨
- 安全编程
- 推荐书籍

预备知识

- **操作系统 ★ ★ ★ ★ ★**
  - 内存管理（页表）
  - 进程地址空间布局
  - Linux的进程与线程（LWP）
  - Linux signals
- **C++相关 ★ ★ ★ ★**
  - 对象内存结构
  - 虚函数实现原理
- **汇编相关**
  - 堆栈结构与函数调用约定 ★ ★ ★ ★ ★
  - AT&T 64位汇编 ★ ★

GDB调试原理探秘

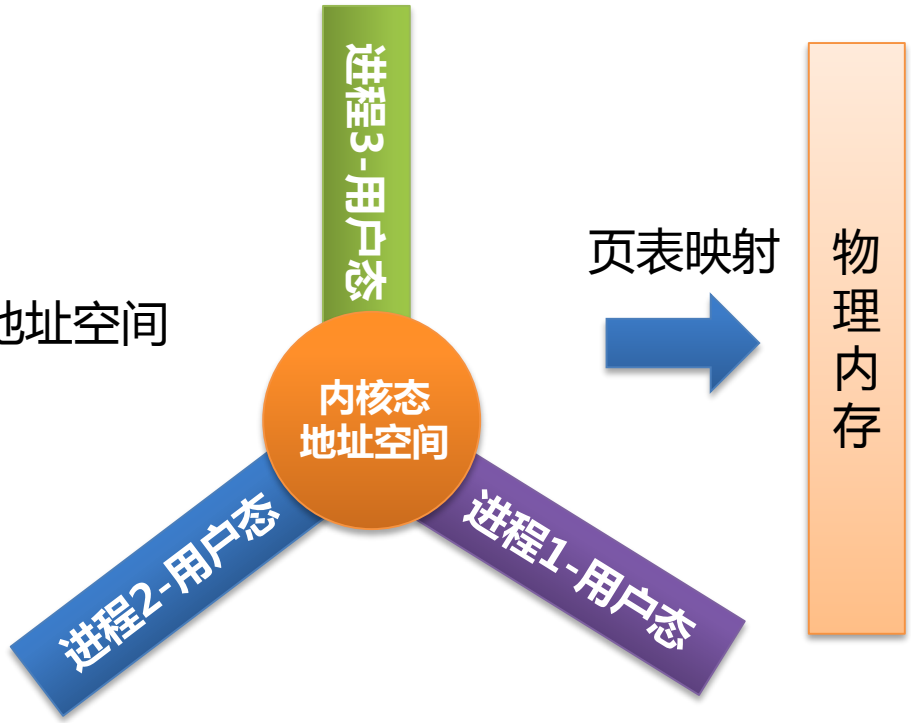
- 引言

进程的隔离性

- 进程虚拟地址空间相互隔离
- 每个进程只能看到2^{64}大小的虚拟地址空间

那GDB

- 是如何能够读取到被调试进程的内存数据的？
- 又是如何能够设置断点的呢？



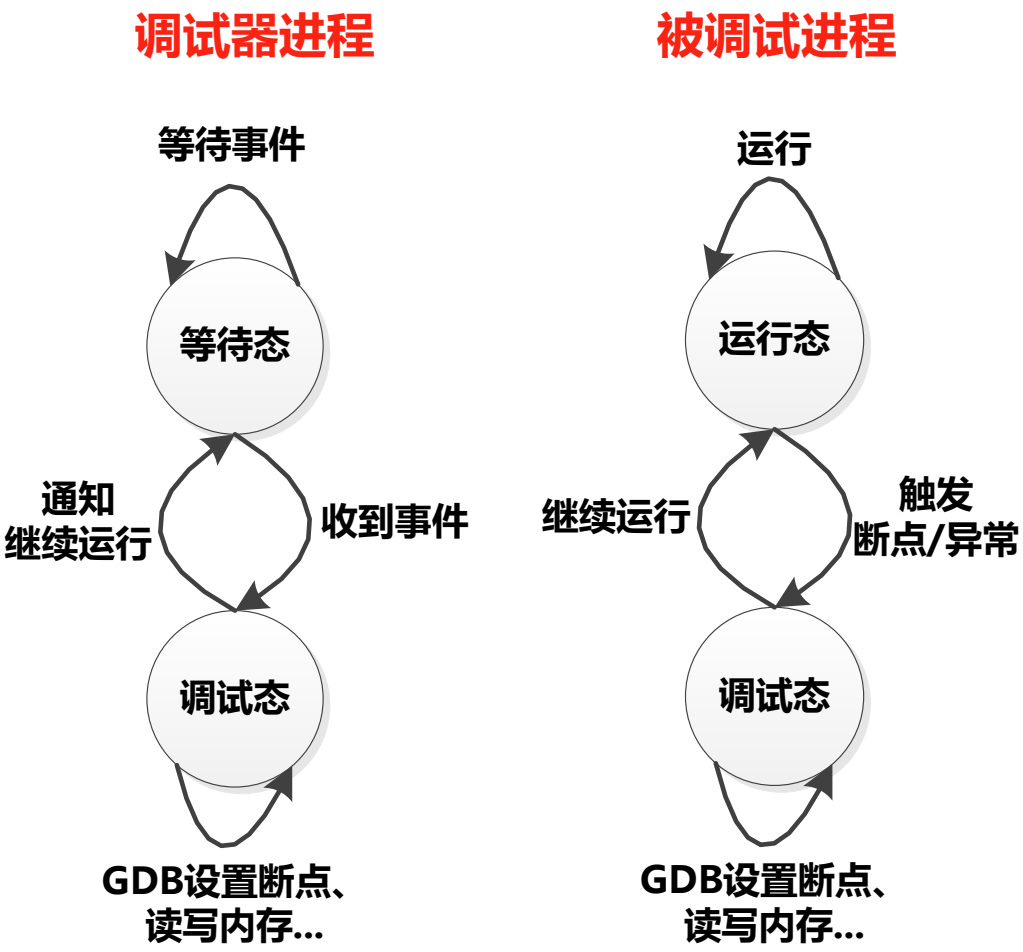
Linux进程地址空间

GDB调试原理探秘

- Linux内核的调试支持

ptrace

- 提供一个进程对另一个进程的上帝视角控制权
  - 事件处理控制
  - 设置断点
  - 读写内存
  - 读写寄存器



调试过程

GDB调试原理探秘

- **断点探秘**

软中断指令：int 3（操作码0xCC）

- **设置断点**
  - 保存断点位置原指令操作码
  - 写入软中断指令操作码（int 3指令，操作码0xCC）
- **触发断点**
  - 执行int 3 => SIGTRAP信号 => gdb捕获
- **继续执行**
  - 恢复操作码并修改$rip（$rip减1）
  - 通知继续执行（借助ptrace）

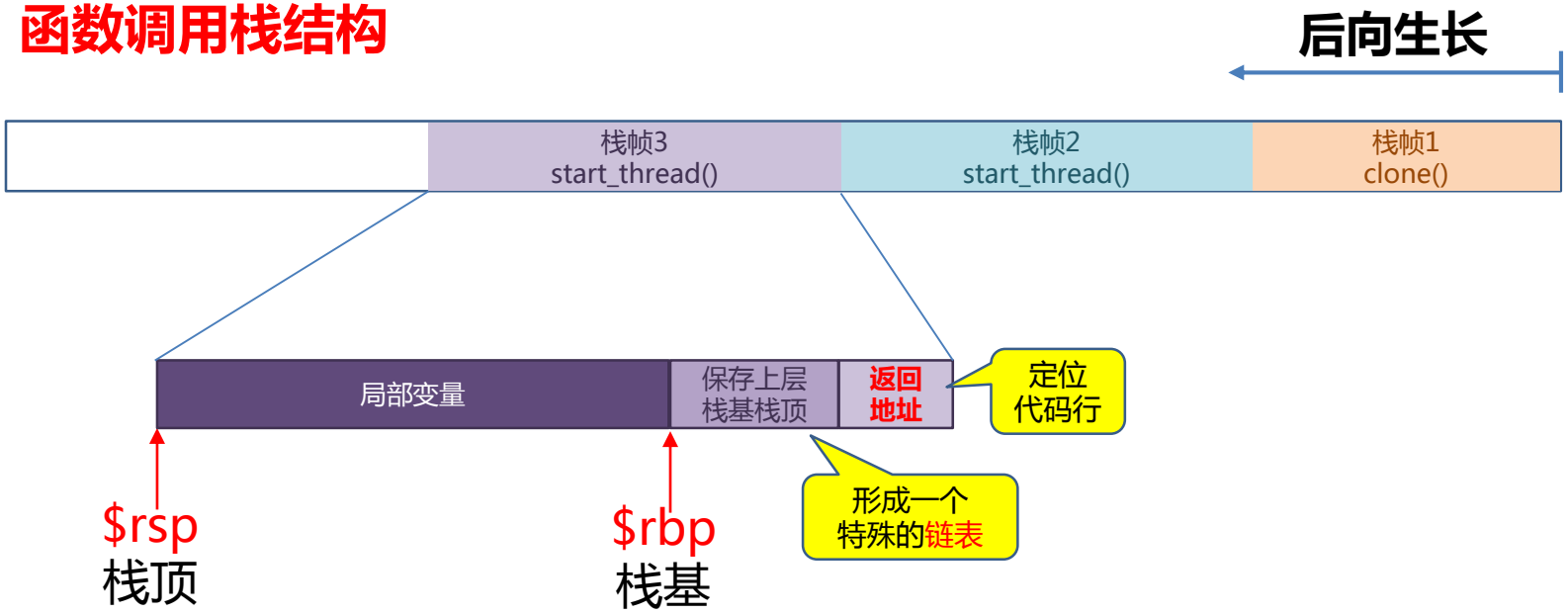
【注】为了写回断点，其实中间还有两步，即在下一条指令设置临时断点，断下来后写回上一个断点

题外：int 3指令还常用于二进制执行文件的间隙填充

GDB调试原理探秘

- **栈回溯（backtrace）探秘**

函数调用栈结构




```
(gdb) disassemble
Dump of assembler code for function DaService::run_nlu_da(mc_pack_t*):   注：函数返回地址由callq指令压栈
   0x00007fd84eb2392a <+0>:     push   %rbp      ——→ 保存上层栈基
   0x00007fd84eb2392b <+1>:     mov    %rsp,%rbp ——→ 保存上层栈顶并设置本层栈基
   0x00007fd84eb2392e <+4>:     push   %rbx
   0x00007fd84eb2392f <+5>:     sub    $0x198,%rsp ——→ 为局部变量预留出空间
   0x00007fd84eb23936 <+12>:    mov    %rdi,-0x198(%rbp)
   0x00007fd84eb2393d <+19>:    mov    %rsi,-0x1a0(%rbp)
   0x00007fd84eb23944 <+26>:    lea    -0x101(%rbp),%rax
   0x00007fd84eb2394b <+33>:    mov    %rax,%rdi
   0x00007fd84eb2394e <+36>:    callq  0x7fd84e840900 <_ZNSaIcEC1Ev@plt>
```

GDB调试技巧探讨

- **动态调试**
  - **启动gdb（** alias gdb=/opt/compiler/gcc-4.8.2/sbin/gdb **）**
    - `$ gdb ./rac_jemalloc`
    - `(gdb) set sysroot /`
    - `(gdb) run -d ../conf -f rac.conf`
  - **常规调试流程**
    - 以gdb启动被调试进程或启动gdb后attach到已启动进程上
    - 在可疑位置设置断点（若进程处于运行态，可Ctrl-C中断之）
    - 触发断点后，观察上下文状态（变量值等），推断问题所在
    - 若问题定位则结束，否则从步骤2重复

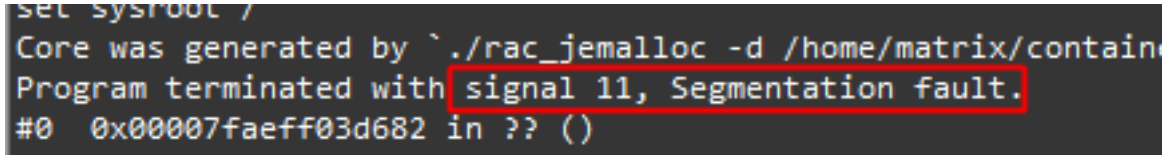
GDB调试技巧探讨

- **静态调试**
  - **启动gdb（** alias gdb=/opt/compiler/gcc-4.8.2/sbin/gdb **）**
    - $ gdb ./rac_jemalloc <CORE_FILE_PATH>
    - (gdb) set sysroot / #设置动态链接库搜索根目录
    - (gdb) bt #查看栈回溯，等价于backtrace
  - **常规调试流程**
    - 按上述方式启动
    - 观察触发crash的signal是什么（如下图），不同触发signal分析方式不尽相同

```
set sysroot /
Core was generated by `./rac_jemalloc -d /home/matrix/contain
Program terminated with signal 11, Segmentation fault.
#0  0x00007faeff03d682 in ?? ()
```

GDB调试技巧探讨

- 静态调试
  - signal 11，Segmentation Fault：访存错误（段错误）



    - 野指针
      - 空指针
      - 内存已释放
    - 写越界
      - 堆写越界
        - => 内存分配器被破坏
      - 栈写越界
        - => 栈无法回溯（bt命令显示异常）
      - => （1）非法指针（2）数据异常（NaN浮点溢出错误）

GDB调试技巧探讨

- **静态调试**
  - **signal 6, Aborted：代码抛异常**

```
warning: Unable to find libthread_db matching inferior's thread library, thread d
Core was generated by `/home/disk1/workspace/du-da_presubmit_322/duer-ci/du_da_de
Program terminated with signal 6, Aborted.
#0  0x00007f7288a993f7 in raise () from /opt/compiler/gcc-4.8.2/lib/libc.so.6
(gdb)
```

  - **signal 8, Arithmetic exception：算术异常**

```
warning: Unable to find libthread_db matching inferior's thread library, thread d
Core was generated by `/home/disk1/workspace/du-da_presubmit_322/duer-ci/du_da_de
Program terminated with signal 6, Aborted.
#0  0x00007f7288a993f7 in raise () from /opt/compiler/gcc-4.8.2/lib/libc.so.6
(gdb)
```

    - **除零异常**
    - **浮点溢出**
      - 常见：数据异常，非浮点数当作浮点数处理

GDB调试技巧探讨

- **栈破坏的处理方法**
  - **方法一：暴力扫描法**
    - 扫描栈，寻找特征数据
  - **方法二：栈恢复**
    - 根据栈上的函数返回地址链表尝试手工恢复栈
  - **方法三：栈打桩法**
    - 每个工作线程在栈上打桩并在邻近位置存储关键数据

```
289 // 用于调用栈打桩，方便出core时的debug
290 struct StackFlagForDebugTrace {
291     // echo -n 'DA_STACK_FOR_DEBUG_TRACE' | md5sum
292     uint64_t __flag1 = 0xfcfebd7439715f9d; // 用于查找定位
293     uint64_t __flag2 = 0x2b00c4bc1bcefc3c; // 用于查找定位
294     const char* request = nullptr;
295
296     StackFlagForDebugTrace(const char* req_) : request(req_) {}
297 };
```

  - **方法四：？**

GDB调试技巧探讨

- 案例分析

安全编程

- **良好的编程习惯**
  - 遵守C++编码规范
- **做好安全检查与错误检查**
  - buffer要检查长度，防止写越界
    - 避免默认buffer长度，用带有安全检查的内存/字符串拷贝函数
  - 检查函数返回值，异常处理
  - 指针要判空

  指导思想：所有的操作（函数调用、内存拷贝）都是不可靠的
- **并发场景选用线程安全的库函数**（函数名通常以_r结尾）
  - 如：gethostbyname_r()
- **其他。。。**

我们的最高理想：永不犯错，丢掉GDB！

推荐书籍

- **操作系统原理、Linux内核解析**
  - 特别是进程管理与内存管理
- **学一点汇编**
- **《gdb internals》**

扎实的系统知识与良好的编程习惯

才是最重要的！

Questions ?

THANKS

谢 谢！