Lecture28 OS Debug

1. 调试器的基本原理

为什么需要调试器?

> 定义和修复BUG，帮助程序员理解程序行为

- 基本功能
  - 中断程序运行读取内部状态
  - 获取程序异常退出原因
  - 动态修改软件状态
  - 控制流追踪

调试器-建立调试关系

- Linux的调试支持：ptrace系统调用
  - GDB建立调试控制关系
    1. 子进程通过PTRACE_TRACEME将调试权交给父进程
    2. 通过PTRACE_ATTACH调试指定pid的进程

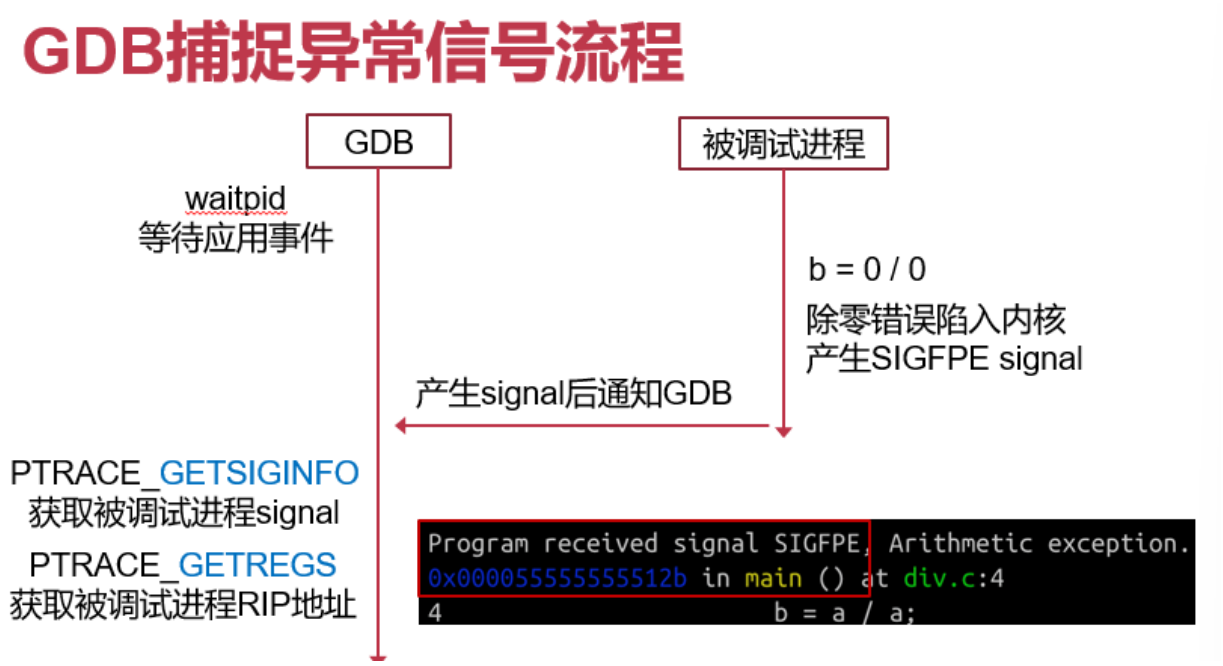
div.c

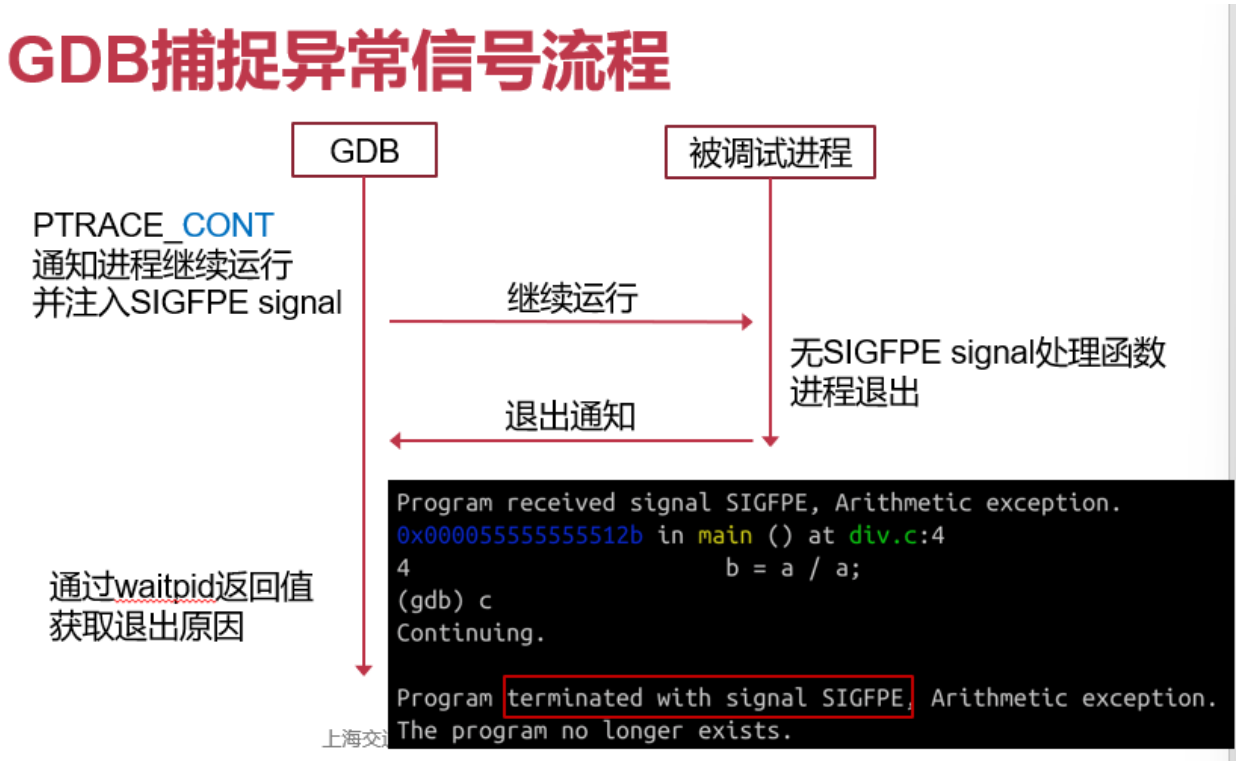
```c
int main() {
    volatile int a = 5, b;
    while (1) {
        b = a / a;
        a = a / 2;
    }
    return 0;
}
```

> 以该程序为例，调试触发除0错误

- **需求1：捕捉到进程除0错误**

```
Program received signal SIGFPE, Arithmetic exception.
0x000055555555512b in main () at div.c:4
```





调试器-配置断点

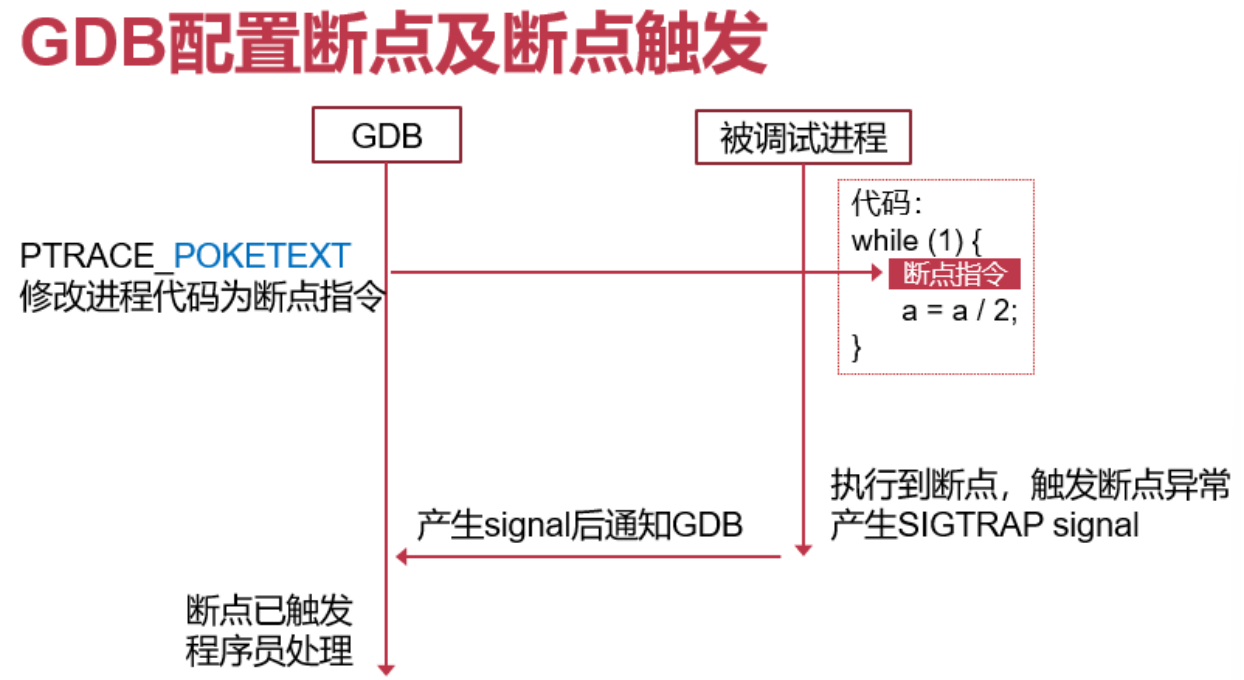
- **需求2：停止进程运行，用以观察进程状态**
  - 发送SIGINT至进程
  - 或断点：在执行到特定指令地址时停止运行
- 使用断点调试

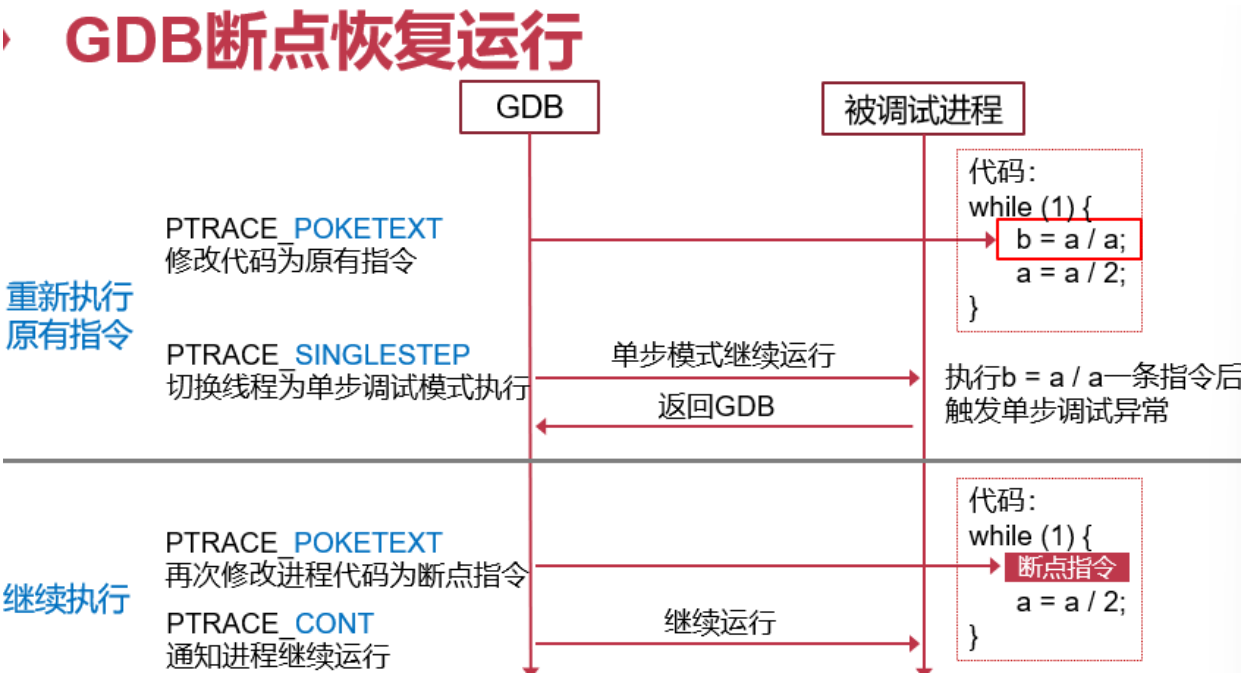
> 在第4行插入断点，观察变量a的值是否为0

- 断点的硬件支持
  - 断点异常指令

- 在执行到特定指令时，触发断点异常陷入内核
- x86的int 3指令，AArch64的BKP指令

- 单步调试
  - 程序在用户态执行一条指令后立刻陷入内核
  - 通过特殊寄存器配置：x86的Trap Flag，AArch64的Software Step





调试器-配置内存断点

- **需求3：变量遭到异常修改时中断运行**
- 内存断点

> 在变量a被修改时中断运行，观察是否为0

```c
// div.c
int main() {
    volatile int a = 5, b;
    while (1) {
        b = a / a;
        a = a / 2;
    }
    return 0;
}
```

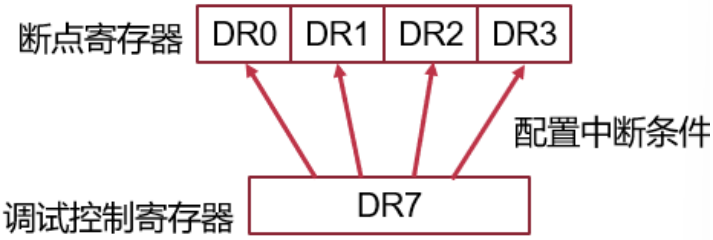
在调用watch a命令后
GDB捕捉到a的值由5变为2

```
Hardware watchpoint 2: a

Old value = 5
New value = 2
main () at div.c:4
```

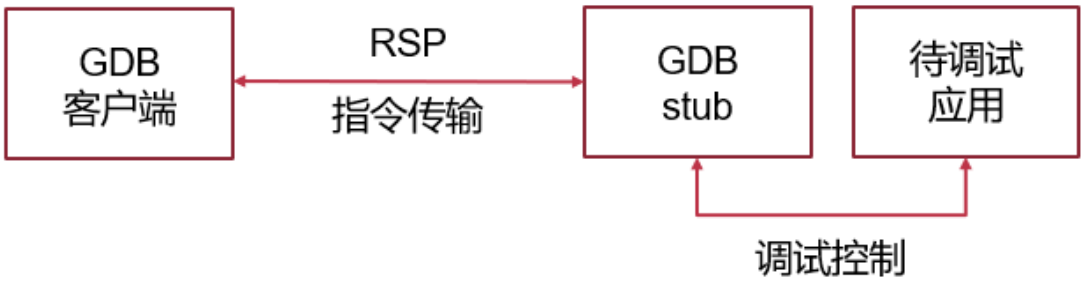
- 内存断点的硬件支持
  - Naive实现
    - 把内存地址所在页设置为只可读
    - 访问时触发page fault
    - 缺点：对该页所有写操作均导致page fault，性能较差
  - 断点寄存器
    - 当访存地址为寄存器中值时，触发断点异常
- 断点寄存器

- **x86断点寄存器**
  - 访存地址等于断点寄存器触发中断
  - 访存条件可配置
    - 数据写 （内存断点）
    - 数据读和写
    - 指令地址（断点）



- **GDB配置被调试应用的断点寄存器**
  - 通过PTRACE_POKEUSER设置

远程调试



- **GDB客户端负责指令发送**
  - GDB远端串行协议（GDB Remote Serial Protocol，RSP）
  - 通过串口线、网络等连接传输控制指令
- **GDB stub 负责实际调试**

2. 操作系统的调试器支持

OS调试器常见实现方法

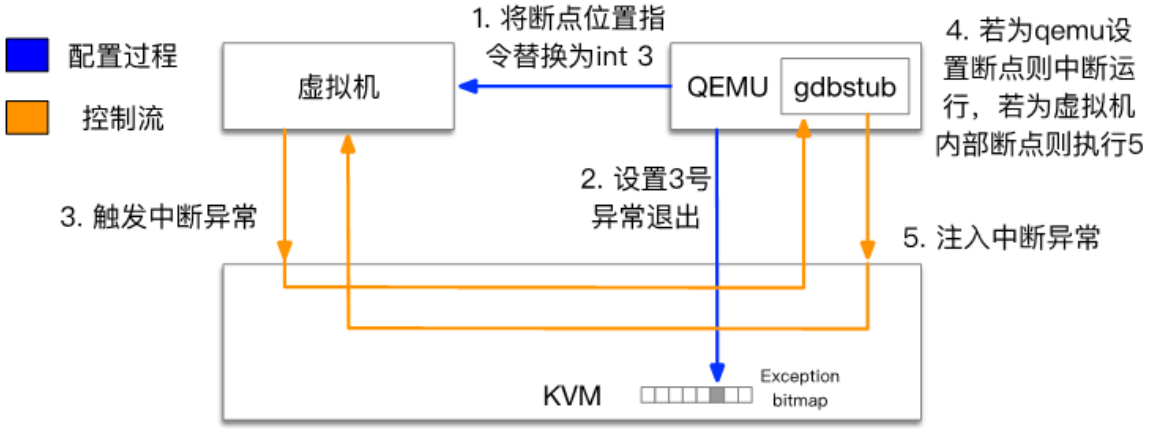
- 调试操作系统调试支持的难点
  - 缺乏操作系统提供给用户态的调试功能支持
  - 硬件相关问题，如外部设备、页表等
- 模拟器
  - 虚拟机：完整模拟底层硬件，在模拟器中提供GDB stub
  - 用户态模拟：例如User-mode Linux，忽略硬件相关的实现，使Linux内核以普通进程的方式运行
- 内核自身实现的调试器

> 操作系统内部实现GDB stub，如Linux的KGDB

案例：QEMU的GDB支持

- 与调试普通进程对比
  - 不再有进程抽象相关的支持(如signal和系统调用追踪)
  - ptrace相关接口替换为虚拟机接口
    - 如内存读写由PTRACE_POKETEXT替换为直接读写虚拟机内存(假设hypervisor能直接访问虚拟机内存)
- 挑战
  - 不能干扰客户机OS内部使用调试功能
  - 断点指令失效
    - 动态代码装载覆写断点指令使断点失效

断点调试
步骤2: 配置虚拟机内部产生断点异常时，退出虚拟机
步骤4和5: QEMU判断断点是否是虚拟机内部断点



断点指令相关问题

- 使用断点指令在OS调试中的困难
  - 动态代码装载覆写断点指令使断点失效
- 解决方法：硬件断点
  - 指令地址等于断点寄存器即触发中断
  - 缺点：影响虚拟机内部使用硬件断点

3. 性能调试

为什么需要性能调试

- 程序功能性正确，但性能未达到理想情况
- 分析程序性能瓶颈
  - 程序运行时哪部分代码耗时较长
  - 哪部分内存发生较多缓存缺失
  - 跳转指令是否发生大量错误预测

实际性能调试样例

1. **确定哪些函数占用了较长的执行时间-采样**
2. **确定是如何执行到该函数的-控制流跟踪**
3. **理解程序行为，为什么会产生这种调用关系**

步骤一：确定内核执行中耗时较长的函数

- 在可能的代码路径上插桩获取时间，统计时间占比最长的部分
- 缺点：大量修改内核代码，统计复杂，通用性极低
- 硬件计数器

- **监控程序执行过程中处理器发生某些事件的次数**
  - E.g., 执行指令数量，各级缓存缺失（cache miss）次数

- **使用方法1：获取事件发生次数**
  - 设置事件类型，打开计数器
  - 一段时间后读取计数器
    - 用户态通过特定指令或系统调用读取
  - 使用该方法分析 hackbench 性能瓶颈仍需大量插桩，意义不大

- **直接读取计数缺点**
  - 缺点：可能涉及对原有代码修改（插桩）

- **使用方法2: 采样**
  - 设置事件类型，打开计数器
  - 当计数器溢出时，产生中断
    - 在中断处理中获取地址信息
    - 清空计数器，等待下一次中断
  - 分析 hackbench 性能瓶颈：每经过一定cycle数触发一次中断，统计中断时指令地址，观察这些地址属于哪些函数

- 基于中断采样的缺点

- **中断时收集信息的缺陷**
  - 采样获取的指令地址不准确
    - 中断发送需要时间，CPU收到中断时的指令地址，与产生采样点指令地址可能存在偏移 （skid）
    - 乱序执行
  - 中断时无法收集完整的采样信息
    - E.g., 缓存缺失时，对应的内存地址未知
- **更精确的采样支持需要：**
  - 计数器溢出时马上收集信息
  - 能够收集更广泛的信息

步骤二：确定是如何执行到该函数的？

- 控制流追踪

- **基于软件的控制流追踪**
  - backtrace：根据调用栈递归获取上层调用者
- **缺点**
  - 编译器优化可能去除栈指针存储
  - 只能处理函数调用
    - 无法应对jmp、中断等导致的控制流变化

- **基于硬件的控制流追踪**
  - 记录jmp、call、中断等导致跳转的前后位置，构建完整控制流
  - e.g., Last Branch Record (Intel)
    - 两组寄存器分别构成栈，记录最近N次跳转的信息

步骤三：理解程序行为，为什么会产生这种调用关系

- 静态追踪方法
  - 在代码编写时静态插桩获取信息的方法
    - 简单可靠的方法：打印
  - 在常用的函数中预置静态的跟踪函数
    - 打印可能造成性能开销
    - 提供打开或关闭选项，关闭时应几乎不产生性能开销
      - e.g. Linux的Tracepoint
  - 缺陷
    - 修改静态定义的跟踪点需要重新编写、部署内核
- 动态追踪方法

- 程序运行时，在不确定的代码位置插入一段动态指定的追踪函数
- e.g. Linux kprobe，实现方式类似于断点调试

- **使用和调试器类似的原理动态插入代码**
  - e.g., 配置handler函数在执行 指令2 之前执行