

调试与测试

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

大纲

- 操作系统对调试器的相关支持
- 如何对操作系统进行调试
- 性能调试及代码追踪
- 测试的基本原则和方法

调试器的基本原理

为什么需要调试器

- **定位和修复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
- **以Linux和GDB为例介绍操作系统如何提供调试功能**

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

- 如何调试下述触发除零错误程序

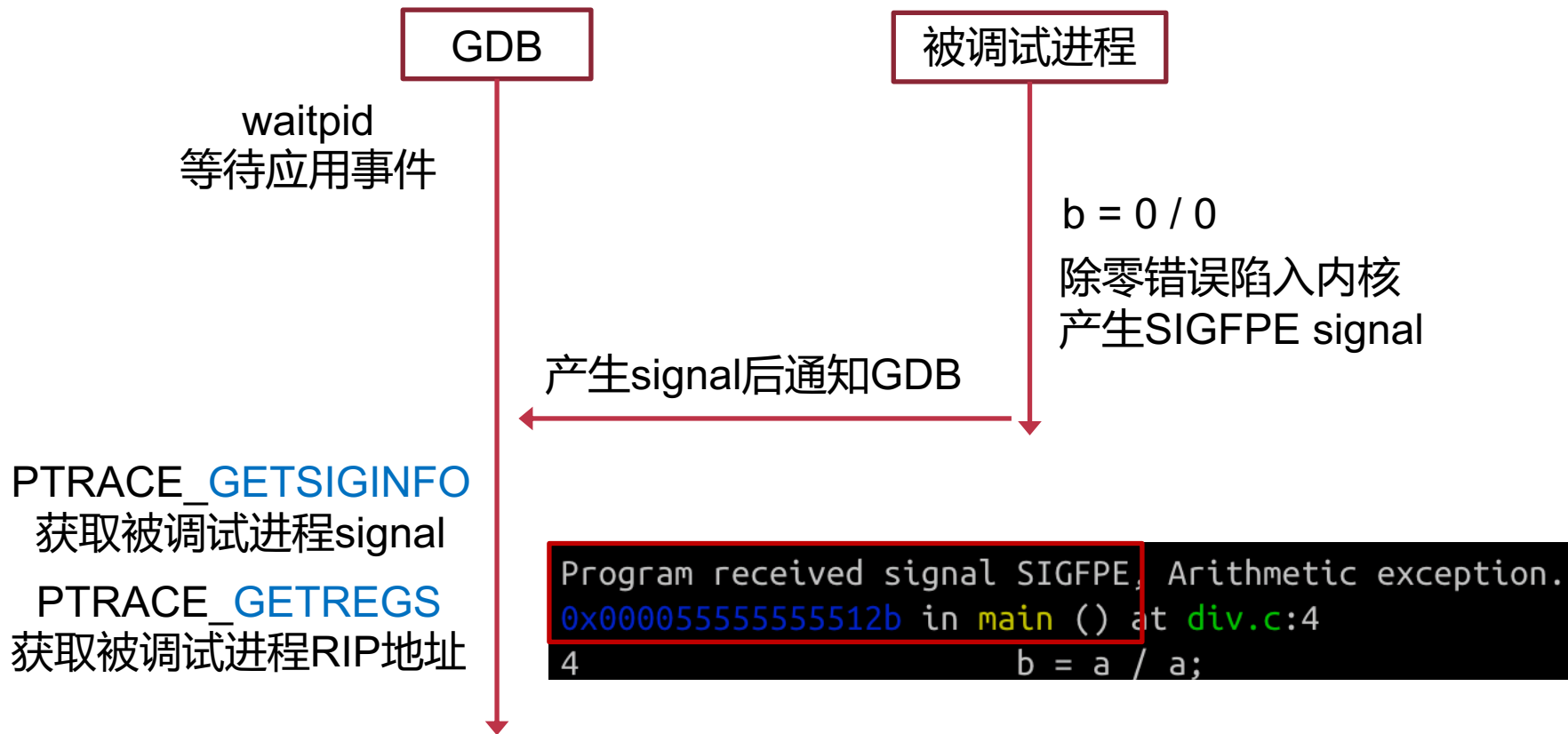
```
div.c
1 int main() {
2     volatile int a = 5, b;
3     while (1) {
4         b = a / a;
5         a = a / 2;
6     }
7     return 0;
8 }
```

需求1：捕捉到进程除零错误

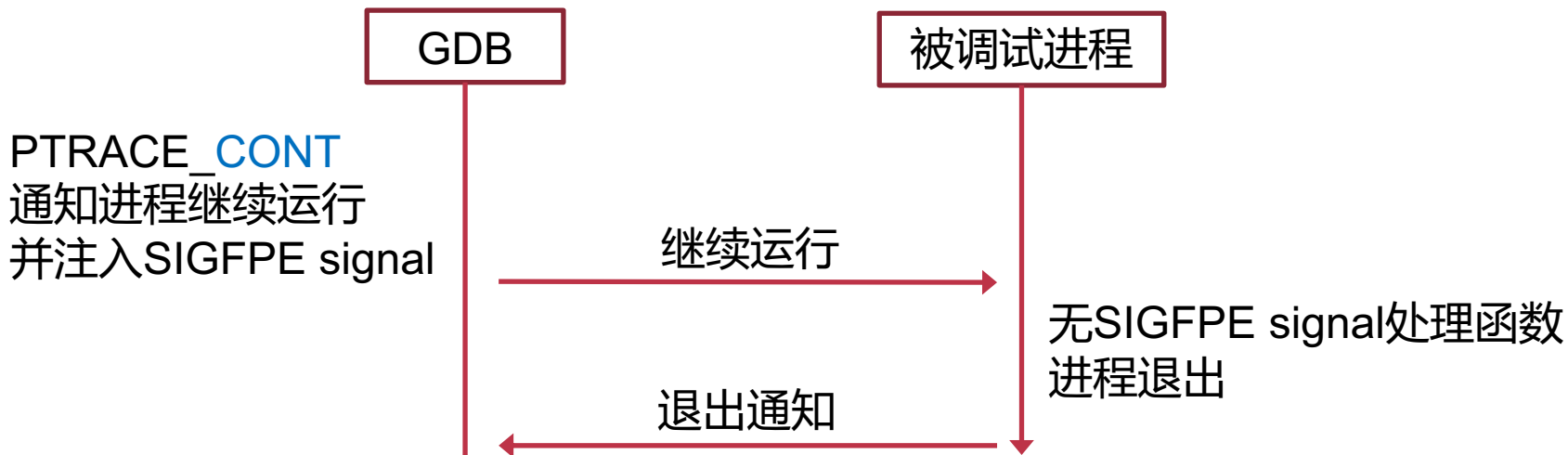
GDB捕捉除零错误产生的输出：

```
Program received signal SIGFPE, Arithmetic exception.
0x00005555555512b in main () at div.c:4
```

GDB捕捉异常信号流程



GDB捕捉异常信号流程



通过waitpid返回值
获取退出原因

```
Program received signal SIGFPE, Arithmetic exception.  
0x00005555555512b in main () at div.c:4  
4          b = a / a;  
(gdb) c  
Continuing.
```

```
Program terminated with signal SIGFPE, Arithmetic exception.  
The program no longer exists.
```


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



```
div.c
1 int main() {
2     volatile int a = 5, b;
3     while (1) {
4         b = a / a;
5         a = a / 2;
6     }
7     return 0;
8 }
```

断点的硬件支持

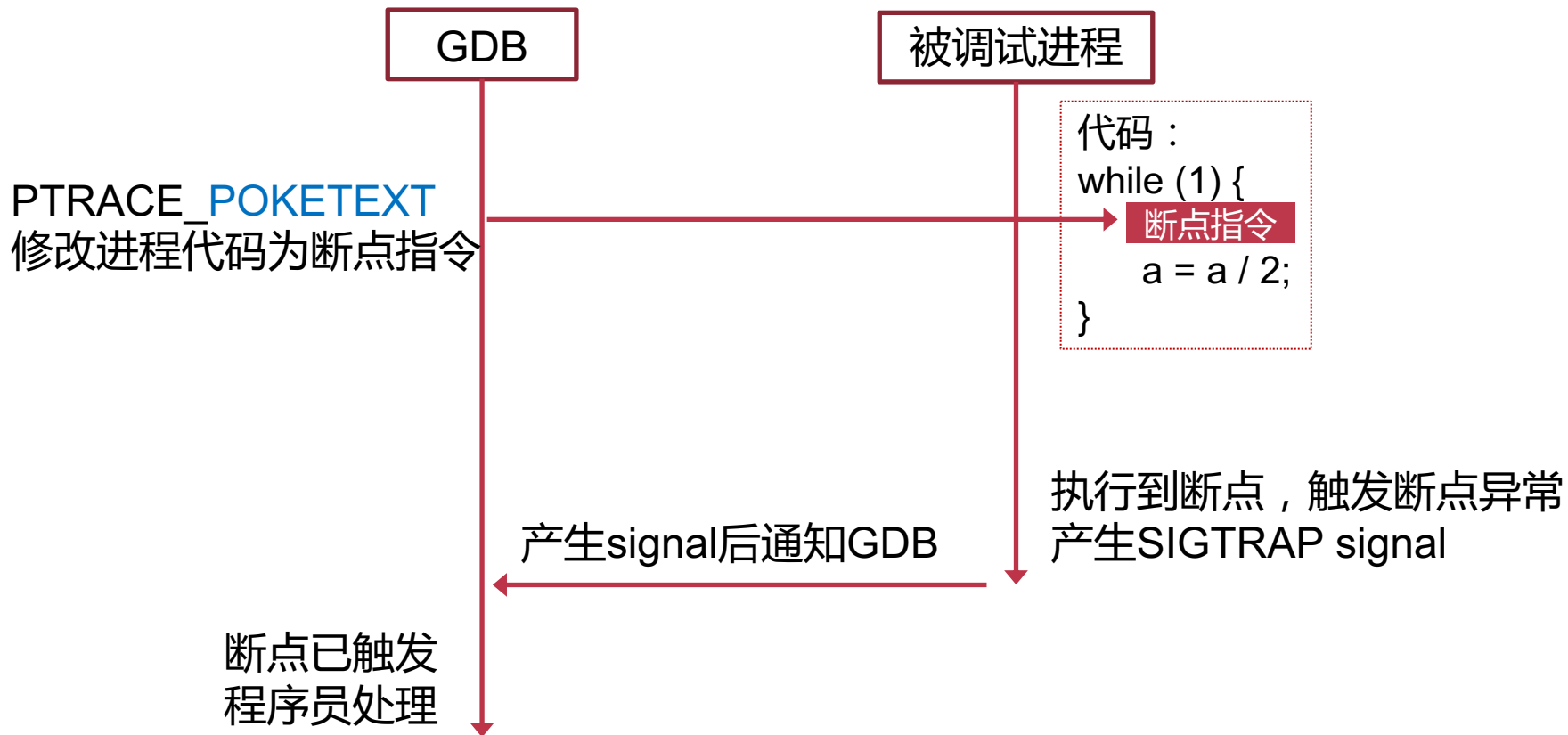
- **断点异常指令**

- 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

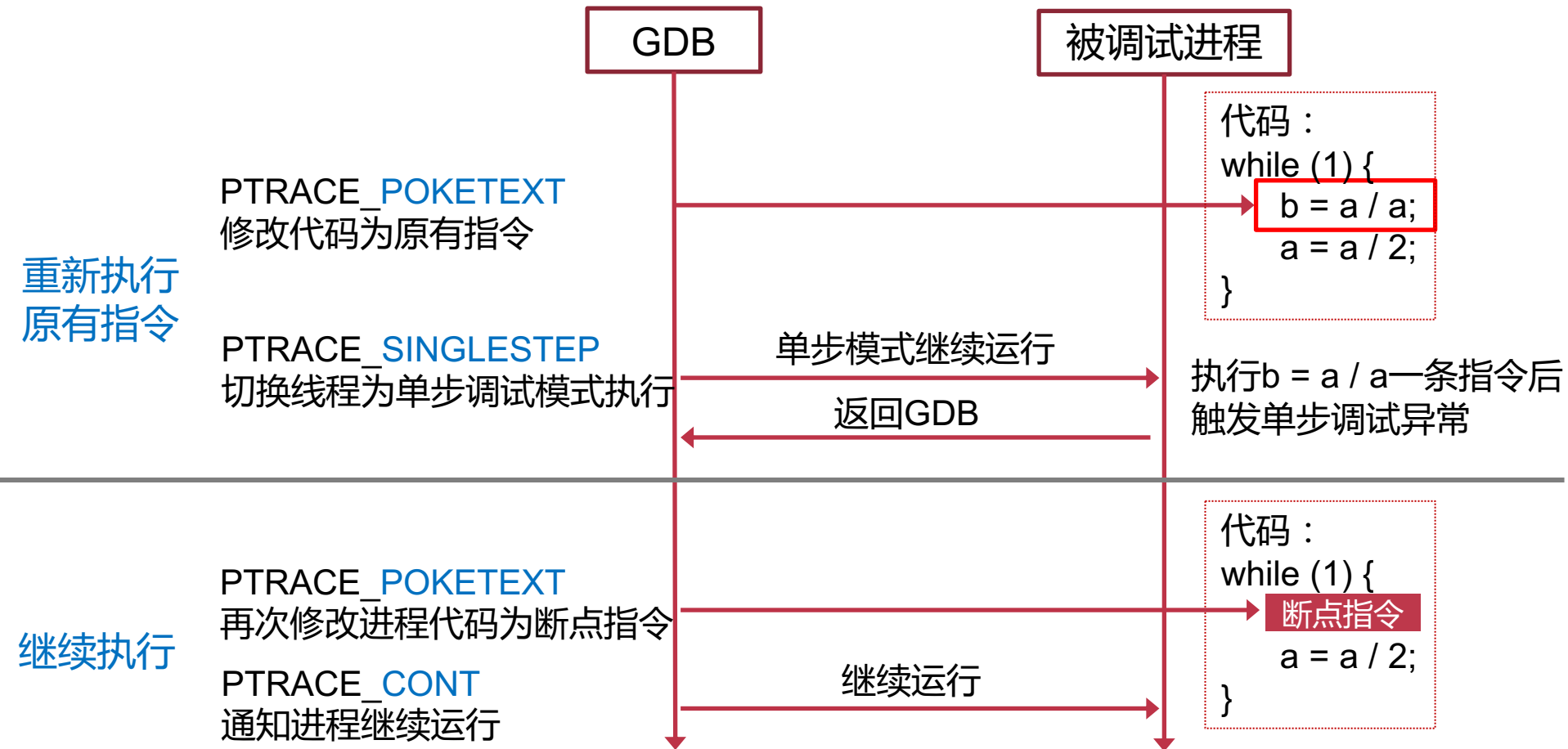
- **单步调试**

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

GDB配置断点及断点触发



GDB断点恢复运行



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受到修改时中断运行，观察是否为0

```
div.c
1 int main() {
2     volatile int a = 5, b;
3     while (1) {
4         b = a / a;
5         a = a / 2;
6     }
7     return 0;
8 }
```

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

- **Naïve实现**

- 把内存地址所在页设为只可读
- 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的值时，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）

断点寄存器



配置中断条件

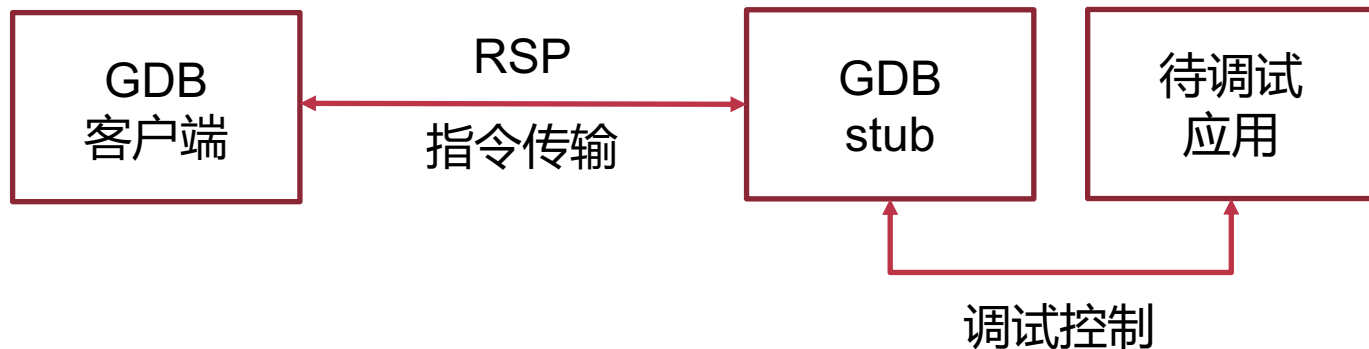
调试控制寄存器



- **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**

操作系统的调试器支持

操作系统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和系统调用跟踪）
- ptrace相关接口替换为虚拟机管理接口
 - 如内存读写由PTRACE_POKE TEXT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**

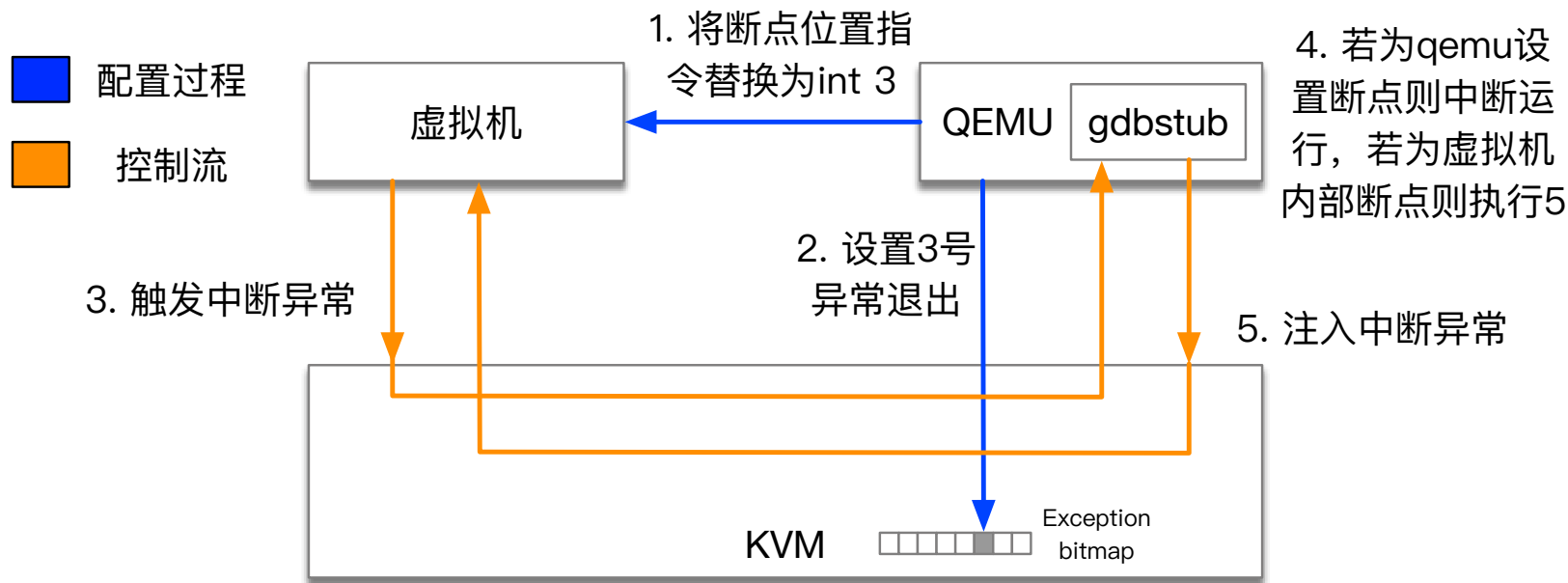
- 不能干扰客户机操作系统内部使用调试功能
- 断点指令失效
 - 动态代码装载覆写断点指令使断点失效

案例：QEMU GDB的断点支持

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

- **使用断点指令在操作系统调试中的困难**
 - 动态代码装载复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

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

- **分析 hackbench 测试用例执行时的性能瓶颈**
 - Hackbench: Linux Test Project 中的调度测试之一
 - 多个进程相互使用socket读写进行同步
- **步骤一：确定内核执行中耗时较长的函数**
 - Naïve：在可能代码路径上插桩获取时间，统计时间占比较长的部分
 - 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



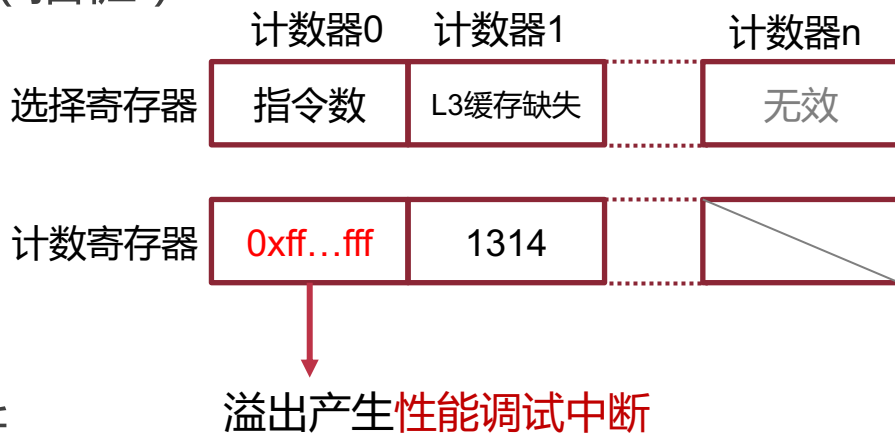
硬件计数器 – 采样

- 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



Linux性能计数器采样支持

- **性能相关事件 perf events**

- 以event的抽象暴露性能计数器（以及一些其它性能调试方法）
- perf_event_open通知内核需要使用哪些计数器
- 采样过程由内核完成
- 采样结果放入内核与用户态共享内存中，减少读取大量采样信息时的开销

- **前端工具perf**

- 直接使用perf events相关系统调用仍然较复杂
- perf工具包装常见的性能分析方法

采样分析hackbench

- 针对cycle数进行采样

- perf record -e cycles hackbench
- perf report

确定两个影响性能因素

1. socket读写用户态数据
2. 系统调用产生的上下文切换

收集到的采样次数

预估的event次数

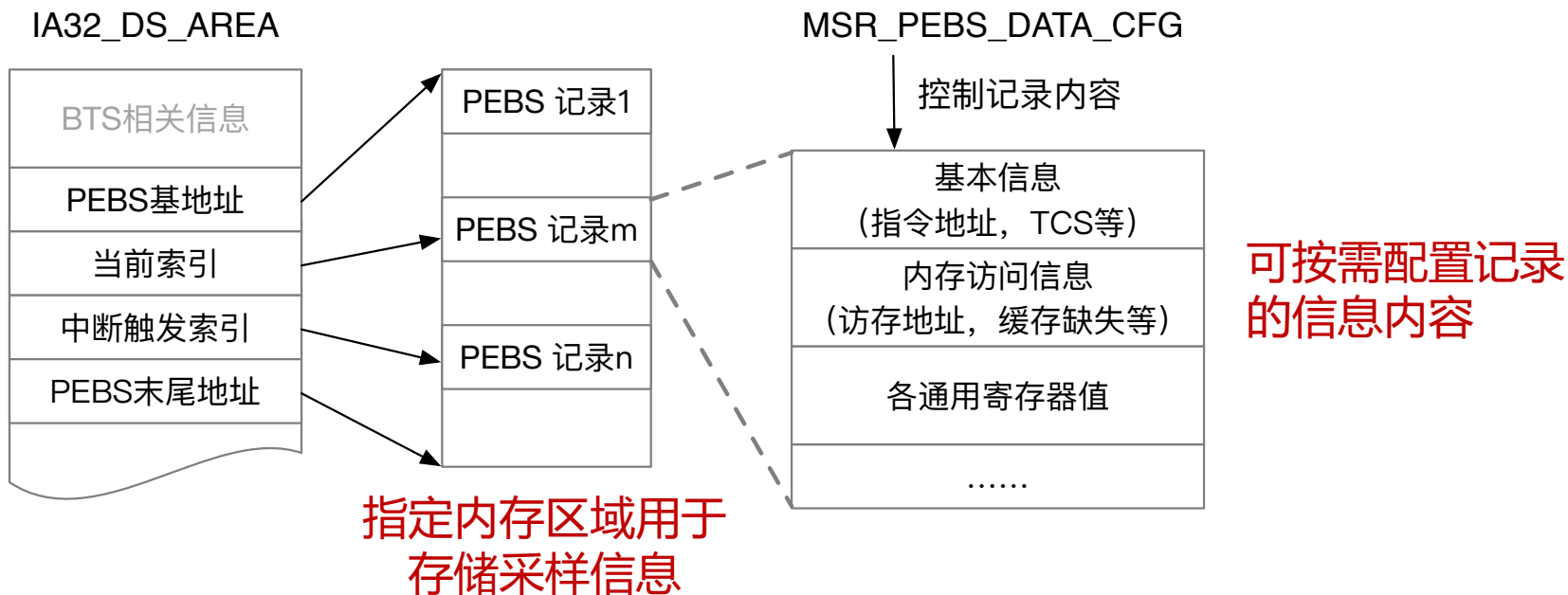
Samples: 53K of event 'cycles', Event count (approx.): 39862780526			
Overhead	Command	Shared Object	Symbol
7.09%	hackbench	[kernel.vmlinux]	[k] syscall_return_via_sysret
6.36%	hackbench	[kernel.vmlinux]	[k] entry_SYSCALL_64
6.16%	hackbench	[kernel.vmlinux]	[k] copy_user_enhanced_fast_string
4.68%	hackbench	[kernel.vmlinux]	[k] unix_stream_read_generic
2.18%	hackbench	[kernel.vmlinux]	[k] __check_object_size
2.04%	hackbench	[kernel.vmlinux]	[k] _raw_spin_lock_irqsave

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

精确采样硬件支持

- 例如x86的PEBS (Processor/Precise Event Based Sampling)




精确采样硬件支持

- **有无必要启用精确采样分析 hackbench**
 - `perf record -e cycles:ppp hackbench`
 - 不具备必要性：即使指令地址有偏移，针对cycle数采样情况下，各函数收到中断概率大致不变
- **何时需要启用精确采样**
 - 需要确定发生特定事件（缓存缺失、跳转预测失败）时指令地址
 - 需要除了指令地址外的其他采样信息
 - e.g., 获取缓存缺失地址：`perf mem record`

控制流追踪

- 步骤一：确定哪些函数占用了较长执行时间 - 采样 ✓
- 步骤二：确定是如何执行到该函数的？

谁调用了该函数？



Samples: 53K of event 'cycles', Event count (approx.): 39862780526			
Overhead	Command	Shared Object	Symbol
7.09%	hackbench	[kernel.vmlinux]	[k] syscall_return_via_sysret
6.36%	hackbench	[kernel.vmlinux]	[k] entry_SYSCALL_64
6.16%	hackbench	[kernel.vmlinux]	[k] copy_user_enhanced_fast_string
4.68%	hackbench	[kernel.vmlinux]	[k] unix_stream_read_generic
2.18%	hackbench	[kernel.vmlinux]	[k] __check_object_size
2.04%	hackbench	[kernel.vmlinux]	[k] _raw_spin_lock_irqsave

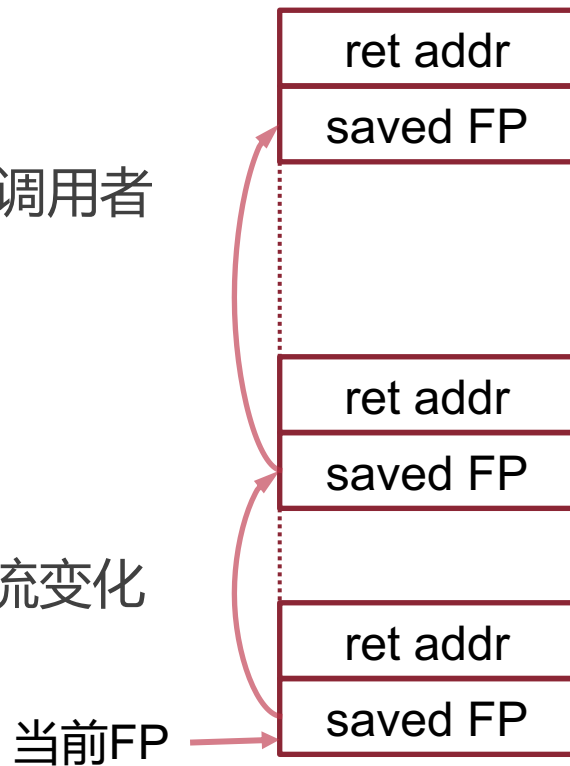
控制流追踪

- **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



控制流追踪

• 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



追踪hackbench控制流

- 指示perf采样时记录控制流变化

- perf record -e cycles -g hackbench
- perf report

```
- 32.74% vfs_read
  - 30.53% new_sync_read
    - 29.41% sock_read_iter
      - 28.14% unix_stream_recvmsg
        - 27.58% unix_stream_read_generic
          - 9.90% unix_stream_read_actor
            - skb_copy_datagram_iter
              - __skb_datagram_iter
                - 5.50% _copy_to_iter
                  - 5.15% copyout
                    copy_user_enhanced_fast_string
```

确定了一条访问
copy_user_xxx
的代码路径

由vfs_read导致

程序执行追踪

- **步骤一：确定哪些函数占用了较长执行时间 - 采样 ✓**
- **步骤二：确定是如何执行到该函数的 – 控制流跟踪 ✓**
- **步骤三：理解程序行为，为什么会产生这种调用关系**
 - hackbench 中大量时间处理socket读写，读写数据规模有多大
 - 作为调度测试，hackbench 是否对调度器产生了足够压力
- **需要具有更多程序语义的跟踪机制**

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

静态追踪方法

- 以 hackbench 为例，如何了解线程切换情况

- 分析调度是一种常见需求，因此Linux内核在__schedule函数中内置了Tracepoint

```
static void __sched notrace __schedule(bool preempt)
{
    ...
    if (likely(prev != next)) {
        ...
        trace_sched_switch(preempt, prev, next);
        ...
    }
}
```

用户态应用通过接口修改
trace_sched_switch_enabled
控制Tracepoint开关

编译生成

```
if (unlikely(trace_sched_switch_enabled))
    调用打印方法
}
```

分析hackbench中线程切换

- Tracepoint作为perf event事件
 - perf record -e sched:sched_switch hackbench
 - 作为调度测试，hackbench确实触发了大量调度
 - 获取了调度前后进程名、pid、优先级等信息

```
Samples: 132K of event 'sched:sched_switch', Event count (approx.): 132159
```

```
Overhead  Trace output
```

```
0.10% prev_comm=hackbench prev_pid=3157715 prev_prio=120 prev_state=S ==> next_comm=hackbench next_pid=3157718 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157725 prev_prio=120 prev_state=S ==> next_comm=hackbench next_pid=3157728 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157718 prev_prio=120 prev_state=S ==> next_comm=hackbench next_pid=3157720 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157726 prev_prio=120 prev_state=S ==> next_comm=sh next_pid=3157764 next_prio=120
0.09% prev_comm=hackbench prev_pid=3157731 prev_prio=120 prev_state=S ==> next_comm=swapper/5 next_pid=0 next_prio=120
```

动态追踪方法

• 静态追踪方法缺陷

- 修改静态定义的跟踪点需要重新编写、部署内核
- Hackbench中，已知socket读操作耗时较长
 - 是否是数据量较大导致的
- vfs_read没有预置静态Tracepoint

```
- 32.74% vfs_read
- 30.53% new_sync_read
- 29.41% sock_read_iter
- 28.14% unix_stream_recvm
- 27.58% unix_stream_rea
- 9.90% unix_stream_r
```

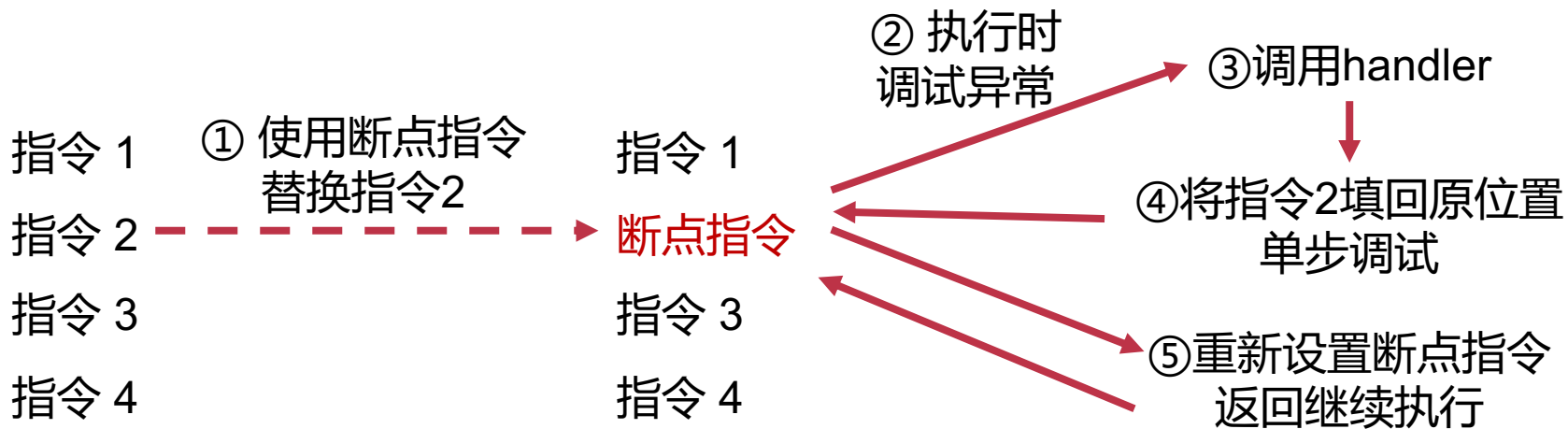
• 动态方法

- 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

Linux动态追踪方法kprobe

- 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



分析hackbench数据读写大小

- 使用 kprobe 探究 hackbench 中数据读写大小
 - 目标：获取vfs_read每次调用时count的大小

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
```

- 方法：在vfs_read执行前插入打印count值的函数
- 定义probe点

- perf probe --add 'vfs_read count=%dx:u64'



定义probe位置



x86上第三个参数位于rdx中，大小为64bit

分析hackbench数据读写大小

- 使用 kprobe 探究 hackbench 中数据读写大小
 - 新定义的kprobe点作为perf event采样
 - perf record -e probe:vfs_read hackbench

```
Samples: 262K of event 'probe:vfs_read'
Overhead  Trace output
98.47%    (ffffffff902a0700) count=100
1.53%     (ffffffff902a0700) count=1
0.00%     (ffffffff902a0700) count=68
0.00%     (ffffffff902a0700) count=32
0.00%     (ffffffff902a0700) count=784
0.00%     (ffffffff902a0700) count=832
```

单次读数据量不大为100byte，
但是读次数较多，
因此vfs_read时间占比较大

测试的基本原则和方法

测试的目的

- **验证程序功能正确性**
 - 程序是否会崩溃
 - 功能是否与设计一致
- **基准测试**
 - 确定程序在特定运行环境下的性能指标
- **操作系统测试的必要性**
 - 作为基础设施，操作系统的正确性和性能直接影响上层应用

操作系统测试的基本方法与原则

- 问题1: 如何快速使错误暴露？
- 测试规模由小至大：小规模测试中暴露的错误更方便定位
 - 先对各功能模块做独立测试：单元测试
 - 以细粒度方式进行测试：函数或功能模块粒度

如测试顺序遍历
链表元素的方法

验证其中每个元素
都为预期的值

```
for_each_in_hlist(iter, hnode, &head) {  
    i--;  
    printf("traverse: %d %p->%p\n", iter->value,  
        iter, iter->hnode.next);  
    mu_check(iter->value == data[i].value);  
}
```

操作系统测试的基本方法与原则

- 测试规模由小至大

- 单元测试：以函数或功能模块为粒度测试
 - ChCore：单独编译内存管理、调度器等模块，在无需运行内核条件下测试
 - Linux：KUNIT，仅编译部分代码，在UML模式下运行
- 单元测试完成后进行集成测试
- 集成测试：整合各个功能模块统一测试
 - ChCore：完整部署内核并运行用户态应用，对网络、文件系统、同步原语等测试

操作系统测试的基本方法与原则

- **代码迭代中，及早确认新修改是否引入BUG**
 - 产生BUG的原因
 - 新的修改本身异常
 - 代码修改触发原有隐藏的异常
 - 回归测试
 - 小规模代码修改后马上运行测试，即使该测试与修改部分无直接关系
 - ChCore：每次代码被push到远端，以及代码合并进主线前，都会运行完整的测试

兼容性测试

- **问题2: 如何确保操作系统能够运行在不同硬件平台，支持各类不同应用？**
- **测试不同硬件环境下兼容性**
 - ChCore：各类测试在虚拟化x86，虚拟化AArch64，真实AArch64硬件均部署运行
 - Linux：kernelci验证Linux在各类不同硬件上能否完成基本测试

兼容性测试

<https://kernelci.org>

最近主线代码在3
个平台出现错误

Tree	Branch	Latest Build Status	Latest Test Results	Date	Status
mainline	master	159 153 6 0	948 912 3 33	2020-06-07	✓
rt-stable	v4.9-rt	160 158 2 0	366 277 0 89	2020-06-06	✓
android	android-4.14-stable	105 105 0 0	431 327 0 104	2020-06-06	✓
android	android-4.19-q-release	104 104 0 0	541 524 1 16	2020-06-06	✓
android	android-4.4-q-release	100 99 1 0	126 95 0 31	2020-06-06	✓

不同分支

不同平台

不同配置

Lab «lab-baylibre» (475 — 463 / 3 / 9)

at91-sama5d4_xplained sama5_defconfig - arm - gcc-8

bcm2837-rpi-3-b defconfig - arm64 - gcc-8

meson-gxm-khadas-vim2 defconfig+CONFIG_RANDOMIZE_BASE=y - arm64 - gcc-8

出现异常的3个平台及对应配置

兼容性测试

- **测试向上能否兼容应用**

- 向后兼容性，操作系统开发迭代后仍能运行较老的应用
- 不同操作系统间如何提供统一的接口
 - 如针对Linux开发的应用能够部署在众多Linux发行版中
- 遵循各类标准
 - 如Linux目录树标准 FHS (Filesystem Hierarchy Standard)
 - 如API 标准 POSIX (Portable Operating System Interface)
- 针对标准进行测试
 - 使用 POSIX Test Suite 验证 POSIX 接口符合标准

兼容性测试

- 测试向上能否兼容应用

- 案例：POSIX Test Suite 验证 fork 接口是否符合标准

fork测试目录部分文件，罗列多项测试

messcat_src.txt	9-1.c	7-1.c	4-1.c	22-1.c	21-1.c	18-1.c	17-1.c	14-1.c	12-1.c
assertions.xml	8-1.c	6-1.c	3-1.c	2-1.c	19-1.c	17-2.c	16-1.c	13-1.c	1-1.c

fork的各项接口标准

```
<assertion id="3" tag="ref:XSH6TC2:12994:12995">
  The new process' ID does not match any existing
  process or group ID.
</assertion>
<assertion id="4" tag="ref:XSH6TC2:12996:12997">
  The parent process ID (ppid) of the child process
  is the process ID (pid) of the parent process
  (caller of fork()).
</assertion>
```

具体测试

4-1.c 测试方法

```
* The steps are:
* -> create a child
* -> check its parent process ID is
*     the PID of its parent.

* The test fails if the IDs differ.
```

操作系统稳定性

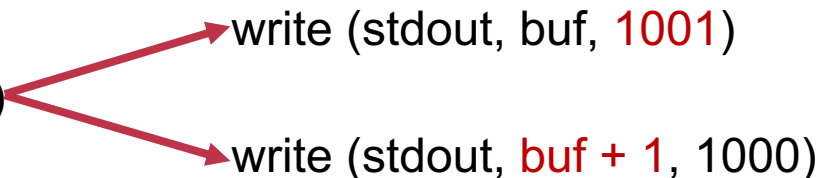
- **问题3: 如何验证操作系统的可靠性？**
 - 基本功能正常前提下，需要确保极端状况下操作系统的正常运行
- **压力测试**
 - 压榨处理器、内存、I/O等资源至极限
 - 频繁进行系统调用
 - 长时间测试 (长稳测试)
- **提高测试时的代码覆盖率**
 - 未测试代码出现异常概率更高

压力测试案例：syzkaller模糊测试

- 模糊测试 (fuzzing)

- 为操作系统构造大量随机系统调用并执行
- 系统调用参数随机变化，期望能覆盖更多代码
 - 但是纯随机变化参数效果不佳
 - 大量输入可能属于同一等价类，代码执行路径相同
- 基于变异 (mutation) 的参数生成

- 在已有参数基础上随机变化

write (stdout, buf, 1000) 

write (stdout, buf, 1001)

write (stdout, buf + 1, 1000)

压力测试案例：syzkaller模糊测试

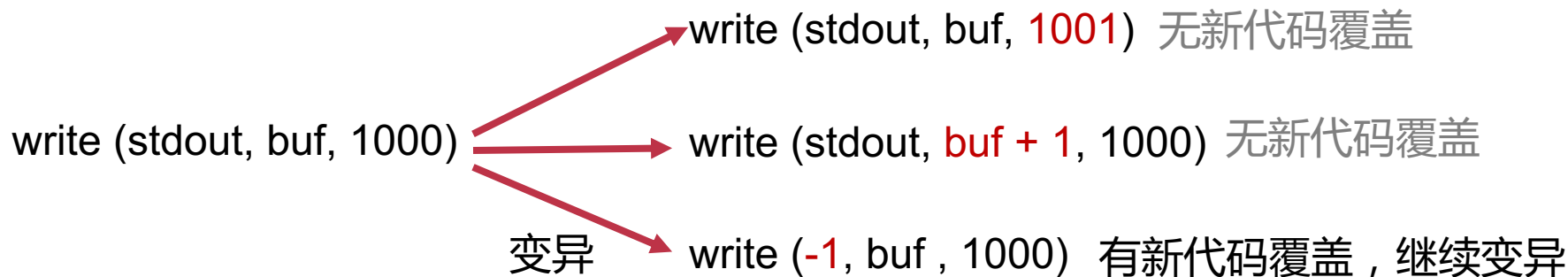
- 模糊测试 (fuzzing)

- 基于变异 (mutation) 的参数生成

- 部分随机变化会引入新的代码覆盖

- 收集产生新代码覆盖的输入，作为新的等价类

- 在新输入基础上继续变异



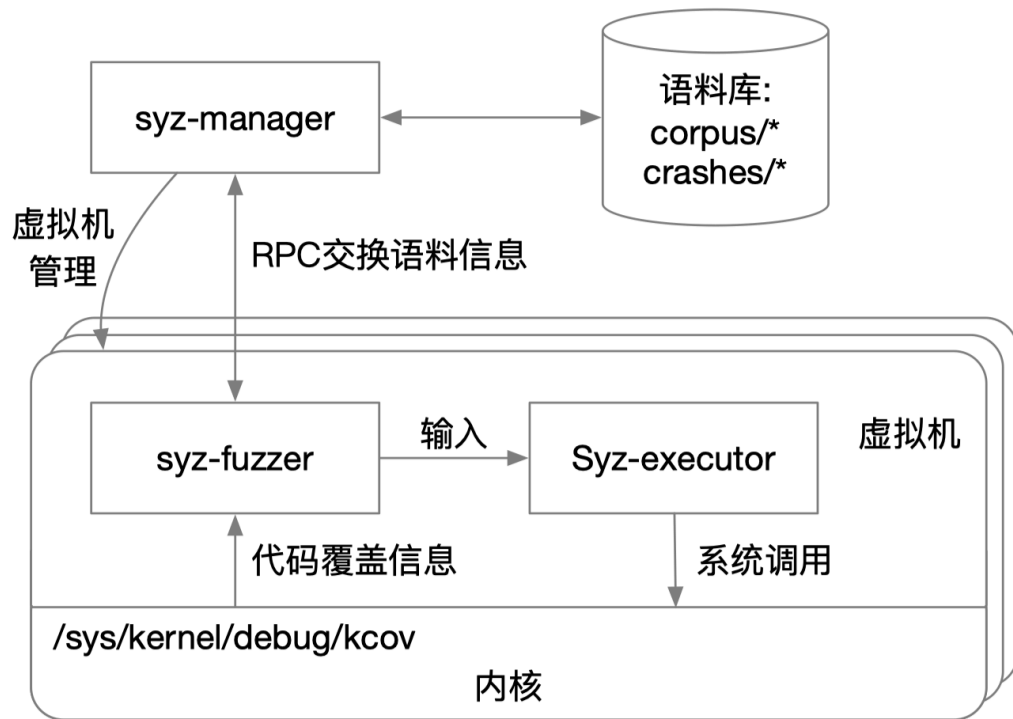
压力测试案例：syzkaller模糊测试

- 多个虚拟机运行操作系统

- 提升测试效率
- 虚拟机内部不断进行：
 - 系统调用
 - 系统调用参数变异
 - 收集代码覆盖信息

- 共享语料库

- 提升变异的效率



性能测试

- **问题4: 如何定量比较不同软硬件配置下性能表现**
 - 性能测试
- **选择合适的测试程序**
 - 明确测试的性能指标（如吞吐量，延迟等）
 - 明确测试的场景（如文件系统读写测试使用顺序还是随机）
 - 已有测试不满足需求时可以自己实现测试程序
 - 但必须确保测试符合真实场景，有代表性

性能测试

- **控制无关变量**

- 以文件系统测试为例
- 软件配置：文件系统类型，配置（日志级别，缓存）
- 硬件配置：硬盘种类和型号
- 其他无关因素：内核版本，时钟中断频率，无其它占用大量资源的应用

- **减少随机不稳定因素**

- 考虑预热一段时间再测试
- 绑核心运行
- 避免跨NUMA节点的内存访问

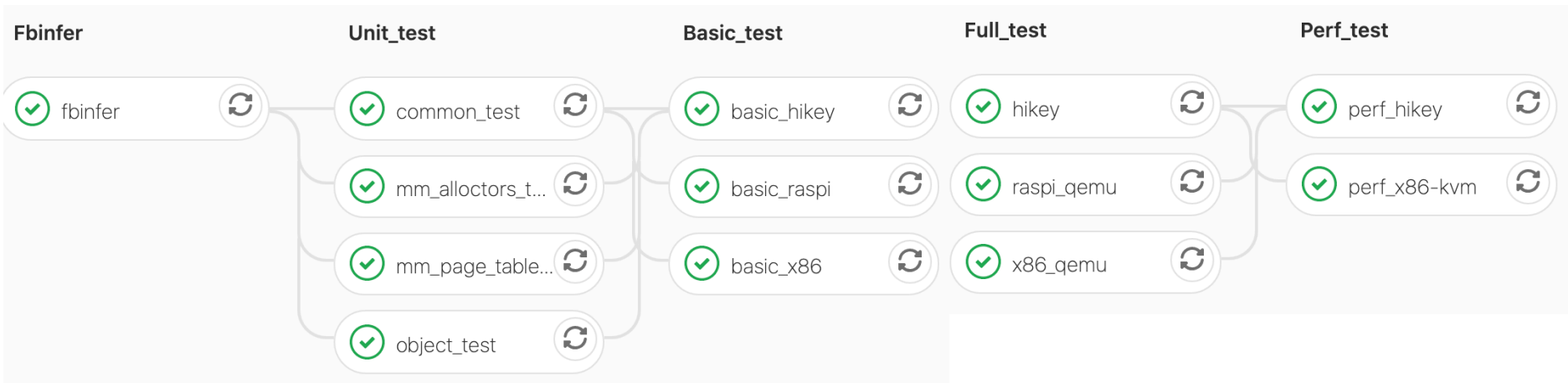
持续集成

- **问题5: 如何有效地进行和管理测试？**
 - 测试考量因素众多，流程复杂
- **持续集成 (CI, Continuous Integration)**
 - 开发者较频繁地将代码合并入主线，使用自动化测试保障代码正确性
 - 自动化部署和回归测试
 - ChCore：代码在push到远端分支时，自动开始进行如下流程
 - 各个平台版本内核的**编译**
 - **静态检查**工具分析
 - 各模块**单元测试**和整体**集成测试**
 - 针对IPC和系统调用的**性能测试**

持续集成

- 门禁系统：确保主线代码的可靠性
 - 通过自动化测试才能合并进入主线
 - ChCore的门禁设置：按顺序通过如下测试

静态分析 → 单元测试 → 跨平台的集成测试 → 性能测试



总结

- **操作系统的调试器支持**
 - 操作系统提供的调试支持
 - 如何对操作系统进行调试
- **性能调试**
 - 使用采样的方式分析性能
 - 软件跟踪机制
- **测试环节的基本方法与原则**

Linux安全漏洞修复流程

1 发现

2 汇报

3 处理

4 公开

- 基于真实的漏洞修复 (Credit: Fan Yang)

- CVE-2020-10757
- Linux commit 5c7fb56e5e3f

- 阶段1：发现bug

- 调研是否已被发现、是否已有解决方案
- 简化复现过程
- 严重性评估，若属于安全漏洞，可以提供exploit
- 若有解决方案，可以提供patch

知情范围：自己

漏洞修复

1 发现

2 汇报

3 处理

4 公开

• 阶段2：汇报

- 非安全漏洞：kernel邮件列表，bugzilla 公开渠道
- 安全漏洞：security@kernel.org 非公开渠道
- 申请CVE id (Common Vulnerability and Exposures List)
 - 意味着公开？此时CVE处于RESERVED状态，信息不公开

知情范围：邮件列表订阅者+自己

漏洞修复

1 发现

2 汇报

3 处理

4 公开

• 阶段3：处理漏洞

— 相关子系统开发、维护人员加入讨论

Will Deacon <will@kernel.org>

回复: ***UNCHECKED*** [vs-plain] User can control PTE value to read/write anywhere, when "mre
DAX file to a mmaped anonymous memory region

收件人: 杨帆 <Fan_Yang@sjtu.edu.cn>,

抄送: Marcus Meissner <meissner@suse.de>, Andrew Morton <akpm@linux-foundation.org>,

Security Officers <security@kernel.org>, +Williams, Dan J <dan.j.williams@intel.com>, **DAX, nvdim**相关开发者

+Kirill A. Shutemov <kirill.shutemov@linux.intel.com>, +Mel Gorman <mgorman@suse.de>

huge page相关开发者

memory management相关开发者

知情范围：相关开发者+邮件列表订阅者+自己

漏洞修复

1 发现

2 汇报

3 处理

4 公开

• 阶段3：处理漏洞

– Patch提议与讨论

Should this be using `pmd_devmap()` instead, i.e. along the lines of 5c7fb56e5e3f ("mm, dax: dax-pmd vs thp-pmd vs hugetlbfs-pmd")?

I too thought about "`|| pmd_devmap()`". I am wondering if a new helper which does

```
pmd_trans_huge() || pmd_devmap()
```

Agreed. Looks like a right fix.

So I take that back. I think the fix for this case right now is to just add the `pmd_devmap()` case, and the cleanup is to try to change these all to be "for large pages, do this.."

Ugh. So is that one-liner sufficient?

Linus



patch

Yes, it looks sufficient for the bug at hand.

漏洞修复

1 发现

2 汇报

3 处理

4 公开

• 阶段3：处理漏洞

— 测试与patch review

fsdax:

	remap 4K size	remap 2M size
normal page mapped[1]	observed move_ptes, OK	observed move_normal_pmd, OK
huge pmd mapped	observed split_huge_pmd, OK	observed move_huge_pmd, OK

[1] DAX file system use huge page by default,

I'd probably drop this paragraph, as it's likely just to provoke a reaction from people who like making noise about this stuff.

devdax:

	remap 4K size
normal page mapped[2]	move_ptes, as expected
huge pmd mapped	not supported [3]

I would propose trimming this comment to just:

/* TODO: kernel-wide: replace "pmd_trans_huge() || pmd_devmap()" pattern with pm

...or just deleting it since it won't live long.

[2] by setting --align 4K when ndctl create-name

[3] dmesg shows "__dev_dax_pte_fault: fail, unalign

Reviewed-by: Dan Williams <dan.j.williams@intel.com>

Fixes: 5c7fb56e5e3f ("mm, dax: dax-pmd vs thp-pmd vs hugetlbfs-pmd")

Cc: <stable@vger.kernel.org>

Acked-by: Kirill A. Shutemov <kirill.shutemov@linux.intel.com>

漏洞修复



• 阶段3.5 : Embargo Period (可选)

- 各大发行版加入讨论，准备修复
- 为什么不能直接修复？
 - 在公开的邮件列表发patch、往主线push都属于公开
 - 预防各发行版未及时修复已公开的漏洞
- 决定是否需要该阶段：评估危害

this vulnerability as compared to the original report? Is it limited to scenarios with a DAX filesystem backed by nvdimmem/pmem or is there a wider attack surface?

We need to figure out when (in which commit, kernel version) the problem was introduced. Have you researched this? Also, if it's fairly recent, what stable and/or distros kernels it's possibly been backported to.

漏洞修复

1 发现

2 汇报

3 处理

4 公开

3.5 Embargo

- 阶段3.5 : Embargo Period

- 协商Embargo Period长短

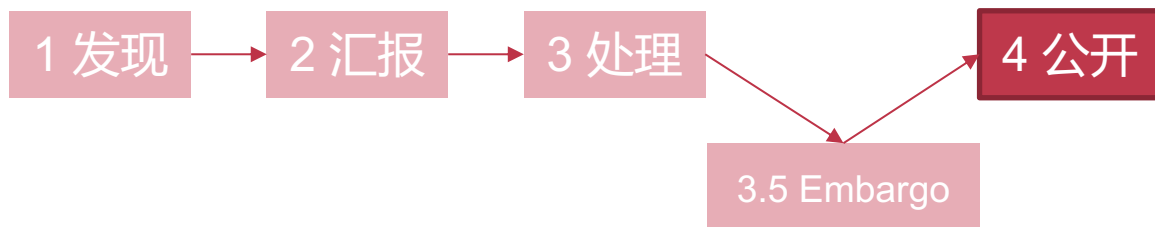
At SUSE we would be fine with either an immediate release or shorter term embargo.
Ciao, Marcus

We (Amazon Linux) are also fine with pushing out immediately.
Regards,
Anthony Liguori

We (VMware Photon OS) are also okay with pushing out the fix immediately.
Thank you!
Regards,
Srivatsa

知情范围：各大发行版+相关开发者+邮件列表订阅者+自己

漏洞修复



• 阶段4：公开

– 并入主线

```
Just FYI, this is now in my tree as commit 5bfea2d9b17f.  
Linus
```

- Backport受影响的stable版本
- 公布至公开的邮件列表、安全话题社区，比如oss-security@lists.openwall.com
- 更新CVE状态至公开

知情范围：所有人！