第十七章　操作系统调测

调试与测试是软件开发流程中极其重要的步骤，调试用于分析、定位并修复软件实现中的错误，而测试则通常用于确定软件实现是否符合设计预期，或者评估软件实现的质量。作为计算机软件栈底层，操作系统的正确性关系到上层应用能否正常运行，同时操作系统的性能也会对应用的性能造成直接影响，因此调试和测试在操作系统中扮演了重要的角色。但是由于操作系统运行在软件栈底层，相比普通的应用而言，操作系统的调测手段较为有限，难度较大，这催生了各类基于软件或硬件的、帮助开发者对操作系统进行调测的方法。本章将简单介绍操作系统调试和测试环节中较为常用的手段和实现方法。

17.1　操作系统调试器

本节主要知识点

- ❑ 为什么需要调试器？
- ❑ 为了支持调试器，硬件需要提供什么功能？
- ❑ 有哪些常见的调试操作系统的方法？

调试器是一种用于调试其它程序的工具，能够追踪程序的运行及观察和修改程序的运行状态。在开发用户态应用时，使用如 GNU Debugger（gdb）之类的调试器能够有效地帮助程序员理解软件发生异常的原因，降低调试的难度。那我们能否使用调试器对操作系统进行调试呢？答案是肯定的，但我们无法完全复用调试用户态应用的方法，因为在调试用户态应用时，调试器通常需要操作系统提供监控其它进程运行的系统调用（如 Linux 中的 ptrace），通过该系统调用观察和修改待调试进程的运行状态。而内核本身并不以进程的抽象暴露给调试器，因此，我们需要其它手段支持对操作系统的调试。本小节将介

绍模拟器调试器以及内核内置调试器两种实现内核调试的方法，在介绍内核调试方法前，我们先简单了解内核调试中所需的硬件支持。

17.1.1 内核调试的硬件支持

无论是在用户态程序还是内核的调试中，设置**断点**（Breakpoint）都是非常有效的调试方法，断点可以打断调试程序的执行，并将控制权转移给调试器以对该程序进行观察和修改。根据断点触发的条件可以将断点分为**指令断点**（Instruction Breakpoint）和**内存断点**（Data Breakpoint）两类，指令断点会在执行到特定指令地址时触发，而内存断点则会在读或写某个内存地址时触发。

指令断点使得程序员能够在特定代码位置中断程序运行，它的实现通常需要硬件架构的支持，例如 x86 架构提供了 3 号中断作为断点异常（Breakpoint Exception）。调试器会将被调试程序中需要产生断点的地址修改为`INT 3`指令，被调试程序在执行到这条指令时会触发 3 号中断陷入内核，并由内核切换到调试器。为了能从断点继续运行，我们需要正常执行被修改的指令，一种实现方法是将`INT 3`指令换回原有的指令，然后通过设置状态标志寄存器（EFLAGS）中的 Trap Flag（TF）的方式将被调试程序切换为**单步运行**模式并切换到被调试程序运行，进入单步调试环境下运行的程序会在执行一条指令后再次触发异常进入内核，此后内核再次调度至调试器，调试器重新将`INT 3`指令填回断点位置，最后恢复正常运行。AARCH64 架构下的断点相关硬件支持和 x86 架构类似，AARCH64 架构定义了触发断点的指令`BRK`，配置断点时需要把原有的指令替换为`BRK`指令，在执行到该指令时触发断点异常，在需要单步运行时可以置上 MDSCR 寄存器中的 Software Step（SS）标志位。这种实现断点的方式在操作系统调试中面临着如下两个问题：1. 操作系统有时会动态地往内存中装载代码，如果使用修改指令的方式设置断点的话，该断点指令会在装载代码时被覆盖，无法触发断点；2. 操作系统有可能执行位于只读内存（ROM）的程序，无法通过修改指令的方式设置断点。

内存断点则提供了一种观测内存访问的方式。内存断点可以通过修改虚拟地址的读写权限实现，当需要监控某个地址的写操作时，将这个虚拟地址对应的页表项权限设置为只可读，此后在写这个地址时会触发虚拟地址翻译异常，操作系统收到该异常后暂停程序执行并将控制权转让给调试器做进一步处理（类似地，监控读操作则将虚拟地址所在页置为不可读写）。这种实现方式的缺点在于内存页的大小通常比内存断点大很多，在设置内存断点会导致对这个页其它位置的访问也出现地址翻译异常，调试器需要特别处理这种情况，导致性能下降，导致无法复现一些高并发环境下的异常。

上述两种断点的实现需要修改待调试程序内存内容或者改变虚拟地址权限，这类方式被称为**软件断点**，软件断点在功能和性能方面有一定局限性，而基于**断点寄存器**的**硬件断点**缓解了上述问题。以 x86 架构的调试寄存器为例，我们可以使用 DR0-DR3 四个调试地址寄存器（Debug Address Registers）存储产生指令断点或内存断点的地址，然后配置 DR7 调试控制寄存器（Debug Control Register）控制产生中断的条件，这些条件可以是当前指令的地址等于某调试地址寄存器的值，或者是读写操作的地址等于调试地址寄存器的值。配置好调试寄存器后，当程序执行满足产生中断条件时会触发断点异常。断点寄存器在功能和性能方面优于上述软件方法，但是缺点在于数量有限，只能够设置少数断点。

17.1.2 模拟器调试

一种较为方便的调试内核的方法是在模拟器中运行操作系统，并使用模拟器提供的调试功能。这些模拟环境普遍支持使用 gdb 调试器调试在模拟器中运行的操作系统，提供指令断点、内存断点等支持。除了能够使用调试器的功能外，使用模拟器进行操作系统调试还有其它一些优势：相对于在真实的硬件上进行调试，在模拟器环境下调试可以通过模拟器获取硬件信息，方便对硬件相关的功能进行调试；其次使用模拟器可以在没有实际硬件的情况下，在模拟器虚拟出的特定硬件执行环境进行操作系统调试。常见的模拟器有：

- Quick Emulator (QEMU)：一种广泛使用的多种架构的模拟器，支持大量模拟外设；
- FVP（Fixed Virtual Platform）：该模拟器能够模拟各大 ARM 架构处理器厂商的处理器及平台；
- User Mode Linux (UML)：将 um（user mode）作为一种特殊的架构编译 Linux，使之能在不借助虚拟化支持的情况下在用户态执行。UML 不会直接访问硬件资源，而是通过系统调用模拟硬件平台，因此无法用于调试硬件驱动。但是 UML 可以用于调试网络栈、文件系统等不直接访问硬件资源的模块。

UML 可以像调试普通用户态程序一样直接使用 gdb 调试，而其它部分模拟器提供了 gdbserver 的支持，可以使用 gdb 客户端连接模拟器调试在其内部运行的操作系统。以 QEMU/KVM 为例，QEMU 在内部实现了 gdbserver，可以借助 KVM 的支持设置硬件或软件断点，下以 x86 为例进行简单介绍。

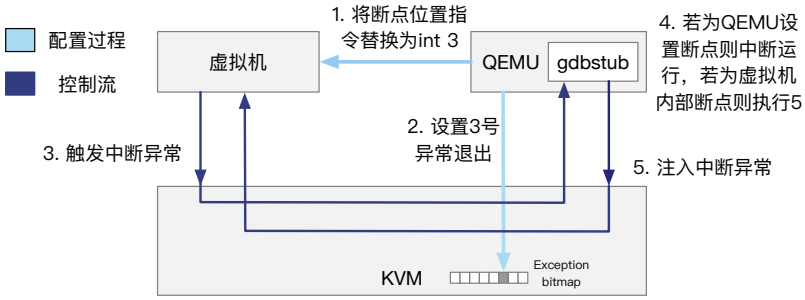

图 17.1: QEMU/KVM 对 gdb 软中断的支持（以 x86 为例）。

首先是软件断点方法，如图17.1所示，和普通的应用程序软件设置中断相同，在 QEMU/KVM 上运行的虚拟机操作系统也是通过插入`INT 3`指令触发断点异常的。KVM 通过配置 Exception Bitmap 的方式使得虚拟机在运行到`INT 3`指令时产生 VMExit，KVM 返回 QEMU 中处理断点异常。此时有两种情况，QEMU 的 gdbserver 会维护所有断点的位置信息用以判断是哪种情况并进行相应处理：

- 该断点是由 QEMU 的 gdbserver 插入的：虚拟机确实运行到了 QEMU 插入断点的位置，此时 QEMU 调用 KVM 提供的接口将所有虚拟处理器全部暂停运行，等待 gdb 调试；
- 该断点是虚拟机内部调试时插入的：此时 QEMU 的 gdbserver 不应当中断虚拟机运行，因此 QEMU 向虚拟机注入一次 3 号异常，直接返回虚拟机执行，虚拟机返回后开始按正常操作系统处理断点异常的方式处理该异常。

其次是硬件断点方法，x86 架构下，QEMU gdbserver 可以使用 4 个硬件断点寄存器，根据断点情况配置虚拟的 DR0-3、DR6、DR7 寄存器，最后进入 KVM 将这些值载入真实寄存器中。客户机操作系统通过断点寄存器触发断点异常时，同样按照软件断点方法中处理断点异常的方式返回 QEMU 进一步处理。另外也有虚拟化手段提供调试寄存器给虚拟机使用，此处不再展开。

17.1.3 内核调试器

虽然在模拟环境下可以较为方便地使用调试器调试内核，但是模拟器不能完全模拟出真实硬件环境，因此存在使用调试器调试在真实硬件上运行的操作系统的需求。例如 Linux 内核加入了 kgdb 用于调试在真实硬件上运行的内核，

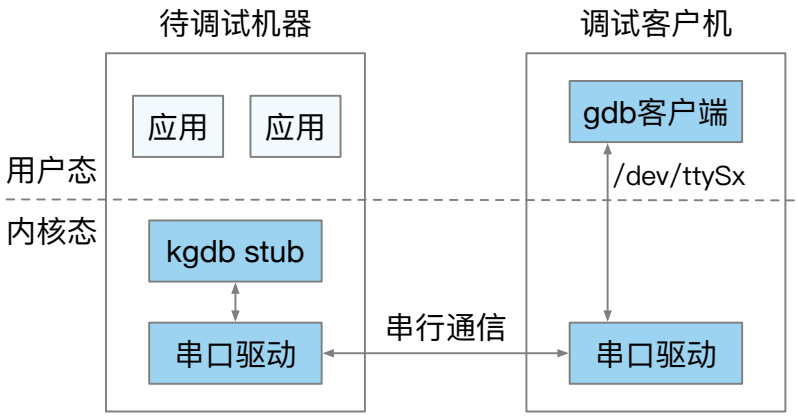

图 17.2: Linux kgdb 架构示意图。

如图17.2所示，内核中加入了调试模块 kgdb stub 作为 gdbserver，调试时需要另一台机器运行 gdb 客户端远程连接到被调试的机器上，两台机器通过串口进行通信。

和 QEMU 中提供的 gdbserver 支持类似，kgdb 可以在内核中插入软件或者硬件断点。两者比较大的一点区别在于一个处理器核心触发断点后，如何让其它核心停止运行等待调试。在虚拟化环境中，虚拟机监控器通常使用虚拟机管理的接口暂停所有虚拟处理器运行。而在 kgdb 中，一个核心触发断点异常被 kgdb 判定为内核调试中断后，kgdb 会向所有核心发送 NMI（不可屏蔽中断，Non-maskable Interrupt），其它核心收到 NMI 后会中止正常运行等待调试。使用 NMI 是因为操作系统可能由于处理中断等情形暂时关闭了中断，若使用普通的核间中断可能导致某些核心长时间不能收到中断请求，进而无法停止运行等待调试。

除了支持调试直接在硬件上运行的操作系统外，内核调试器还能够获取操作系统语义。模拟器所支持的调试功能对于操作系统往往是透明的，因此比较难以获得操作系统语义相关的信息。而内核调试器直接实现在内核内部，能够比较方便地获取操作系统一些内部信息，例如 kgdb 内置了 ps 命令，可以获取内核中有哪些正在运行的线程。

17.2 内核追踪机制

本节主要知识点

- 什么是追踪机制，为什么操作系统需要此类机制？
- 常见的硬件和软件追踪机制是如何实现和使用的？它们的局限性是什么？
- 用户态如何配置内核的追踪方法，如何获取追踪信息？

追踪机制（Tracing）是指在程序执行的过程中，记录程序发生的某些事件。为了调试内核、了解内核运行时的行为等，程序员经常有追踪内核运行时信息的需求。例如为了解某系统调用执行情况，需要每次调用该系统调用时记录各个参数。理论上讲，许多追踪的需求可以通过增加日志输出的方式解决，但是每当有这样的需求就在源代码基础上增加打印的操作再重新部署内核显得费时费力。而内核追踪机制则大大缓解了重新编写和部署内核的负担，它提供了在不修改内核的情况下动态获取运行信息的方法。内核追踪机制大致可以分为硬件追踪和软件追踪两类：硬件追踪机制通常关注于获取较为底层的信息，如统计控制流转换、页表切换等，而软件追踪机制则通常包含较为丰富的语义。

17.2.1 硬件追踪方法

硬件追踪方法主要用于追踪程序执行流的跳转。在操作系统运行时，主要有两种情况会导致执行流跳转：一种是 jmp、call 等有关跳转的指令，另一种是有关中断、异常等处理流程。从纯软件的角度很难对所有执行流跳转进行完整的分析，因此往往需要借助硬件的支持，下面以 Intel 处理器为例，简单介绍三种用于跟踪执行流的硬件支持。

Last Branch Record

Last Branch Record (LBR)是一种使用多个寄存器记录跳转信息的硬件支持。如图17.3所示，LBR 支持定义了多个成对的Model Specic Register (MSR)寄存器 MSR_LASTBRANCH_X_FROM_IP 以及 MSR_LASTB_RANCH_X_TO_IP，分别用于记录跳转的来源及目的地址，其中 X 可以为从 0 到最大值 N。这两组（N+1）个寄存器分别构成了两个栈，每次有跳转发生时，新的来源和目的地址被放在编号为 0 的寄存器中，原来寄

图 17.3: Last Branch Record 寄存器。

存器中的地址被依次放到编号加一的寄存器中，最后一组寄存器原有地址被丢弃。使用时，只需按顺序读出两组寄存器中储存的地址，即可还原出最近（N+1）次发生跳转的情况。另外，MSR_LASTBRANCH_X_FROM_IP 寄存器的高位还包含了此次跳转的一些额外信息，例如其中一个 bit 表示此次跳转是否预测正确。

Branch Trace Store

LBR 机制虽然能在几乎对性能无影响的情况下获取最近发生的跳转信息，但是由于寄存器的数量是有限的，在跳转次数较多时无法恢复完整的控制流跳转。而同样用于记录跳转信息的Branch Trace Store (BTS)机制则将信息记录到内存中，大大扩展了能够被记录的跳转信息数量。BTS 允许将跳转相关信息存储到由 IA32_DS_AREA 指定的内存区域，如图17.4所示，BTS 基地址指向了一块用于存储 BTS 记录的内存区域，当有跳转指令发生时，处理器会生成一条包含了跳转来源地址及目的地址的记录，并将其写入 BTS 当前索引的位置，然后把当前索引位置移到下一条空的 BTS 记录。当 BTS 记录累积到超过中断触发索引时，处理器产生中断通知操作系统 BTS 存储区域即将用尽，此时操作系统可以批量处理之前收集的数据，并清空 BTS 记录以继续收集跳转信息。

BTS机制通过往一段内存区域中写入跳转来源地址和目的地址的方式来跟踪执行流跳转，但这样的方法有一些局限性：首先与LBR相比，BTS 通过虚拟地址指定内存区域记录跟踪信息，这样会占用比较宝贵的 TLB 资源，另外 BTS 在记录过程中产生大量的写操作还会占用缓存，因此使用 BTS 会造成一定性能下降；其次 BTS 记录的信息非常有限，只包含了跳转来源和目的虚拟地址等少量信息，这在某些情况下不足以构建完整的程序执行流，例如在切换页表后，存储虚拟地址可能仅在原来的页表中有效。

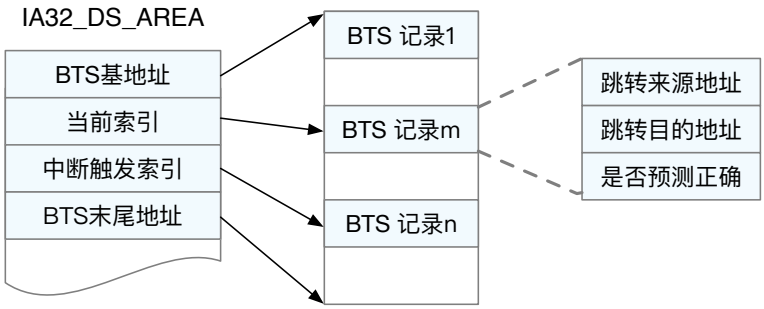

图 17.4: Branch Trace Store 存储结构。

Intel Processor Tracing

为了解决上述 BTS 的缺陷，Intel 提出了Intel Processor Tracing (IPT)用于性能损耗更小、记录信息更广泛的程序执行分析。和 BTS 记录跟踪信息的方式类似，IPT 也会将运行过程中产生的跟踪信息不断地写入内存中。为了减少占用 TLB 和缓存带来的性能损失，IPT 的记录内存区域直接用物理地址表示，并且跟踪信息会绕过缓存被直接写入内存中。为了更完整地构建程序执行流，IPT 支持记录更广泛的信息，例如页表切换、进入或退出 non-root 模式等。IPT 为这些不同类型的信息定义了统一格式的包（packet），运行时将产生的包写入内存中，这些包中的内容在后续分析时会被解析。

那么三种硬件追踪机制各自有哪些优点和缺点，我们该如何选择呢？首先我们对比 LBR 与另外两种机制（BTS 和 IPT），LBR 的配置最简单，只需读取寄存器即可获取最近的跳转地址，而 BTS 和 IPT 机制需要将跳转地址存储在内存中，引入了额外的配置和解析内存存储的步骤，相对较复杂。由于 LBR 在跟踪时不会像 BTS 和 IPT 一样将跳转信息存储到内存中，而是保存在寄存器上，因此对性能的影响是最小的，但是存储跳转信息的寄存器数量有限，所以只能够获取较近期的跳转信息。因此，在对性能要求严苛并且不需要过长的跳转信息回溯时，使用 LBR 较为合适。其次对比 BTS 和 IPT，IPT 作为一种通用性极强的跟踪手段，它涵盖了绝大部分 BTS 所记录的信息，仅有“跳转预测是否正确”等少数信息只出现在 BTS 记录中，另外由于占用 TLB 和缓存的原因，BTS 会引入较大的额外开销，因此一般而言优先使用 IPT。

在 Linux 中，我们可以在 perf 工具的帮助下使用硬件追踪机制获取指令跳转信息（perf 的详细介绍请参考第17.3.2章），注意需要使用 root 权限运行：

```
# 使用LBR对函数调用栈进行跟踪：
[root@osbook ~] $ perf record --call-graph lbr -a sleep 1
```

```
[root@osbook ~] $ perf report

# 使用IPT统计1秒内所有跳转指令的来源和目的地址:
[root@osbook ~] $ perf record -e Intel_pt/branch/ -a sleep 1
[root@osbook ~] $ perf script
```

AARCH64 架构同样提供了执行流跟踪硬件支持，该机制被称为Statistical Profiling Extension (SPE)。和 IPT 机制类似，SPE 机制会指定一块内存区域用于存储采样的信息，并且支持多种不同的事件，可以记录跳转指令执行、数据存取、缓存缺失发生等事件发生时相关的信息。

17.2.2 软件追踪方法

上述硬件追踪机制通常用于指令级别的执行流分析，但我们有时需要追踪机制包含更多语义信息，如某函数执行的参数是什么。软件追踪方法允许程序员在编写内核或者运行时灵活地配置需要追踪的信息，因此能够较好地满足上述需求。软件追踪机制通常需要支持动态的开关功能，并且无论是在打开还是关闭时都应该尽可能降低性能开销。下面简单介绍 Linux 内核中静态和动态追踪自身运行状况的两种方法。

静态追踪方法 tracepoint

静态追踪方法要求程序员编写代码时在需要追踪的位置插桩，调用记录追踪信息的函数。在操作系统中，对于关键的函数可以使用此类方法进行运行状态跟踪，例如为了分析内核内存使用情况，Linux 对`kmalloc`和`kfree`都插入了静态的 tracepoint。

简化的 tracepoint 使用方法如代码片段17.1所示，首先定义一个 trace 函数，该定义包含了待记录的参数、如何打印信息等。与调用一般函数的方法类似，插桩时只需调用该函数，并填入相关参数即可。tracepoint 可以在运行时选择打开或者关闭，该功能实现的原理是一个分支判断，通过改变判断条件选择是否进行信息记录。在追踪功能关闭时，每次运行到 tracepoint 仅仅增加了一次分支判断的额外开销；在打开时，每当运行到 tracepoint 处，trace 函数中参数所指定的信息都会按照预定的方法被写入到一段环形缓存中，使用者后续可查看这些信息。

```
// 定义一个名为 kmalloc 的 tracepoint
DEFINE_EVENT(kmem_alloc, kmalloc,
  // 指定需要收集哪些参数
  TP_PROTO(unsigned long call_site, const void *ptr,
           size_t bytes_req, size_t bytes_alloc,
           gfp_t gfp_flags),
  // 其他定义内容已省略
  ...
);

// 在 kmalloc 函数内部，调用追踪函数记录此次调用相关信息
void *__kmalloc(size_t size, gfp_t flags)
{
  ...
  ret = slab_alloc(s, flags, _RET_IP_);
  trace_kmalloc(caller, ret, size,
                cachep->size, flags);
  ...
}
```

代码片段 17.1: 使用 tracepoint 记录 kmalloc 调用信息。

下面是一个简单的样例，通过 debugfs 打开或关闭 Linux 内核中内存分配函数kmalloc的 tracepoint，并获取kmalloc的调用信息，注意需要 root 权限：

```
[root@osbook ~] $ cd /sys/kernel/debug/tracing
# 打开kmalloc的tracepoint
[root@osbook ~] $ echo 1 > events/kmem/kmalloc/enable
# 等待一段时间后，从buffer中读取追踪信息
# 追踪信息包含了trace_kmalloc指定获取的参数，包括分配内存大小、flag
    等
[root@osbook ~] $ cat trace_pipe
kmalloc: bytes_req=64 bytes_alloc=64 gfp_flags=GFP_KERNEL
kmalloc: bytes_req=1088 bytes_alloc=2048 gfp_flags=GFP_KERNEL|
    __GFP_ZERO
...
# 关闭tracepoint
[root@osbook ~] $ echo 0 > events/kmem/kmalloc/enable
```

由于 perf（见第17.3.2章）对上述操作进行了包装，在实际使用中可使用如下等价的命令：

```
[root@osbook ~] $ perf record -e kmem:kmalloc
```

动态追踪方法 kprobe

动态的追踪方法不需要在程序编写时预先嵌入相关代码。该方法能够在不影响原有的代码功能性的前提下，动态地获取任意地址处的追踪信息，同时在该方法中，获取追踪信息的方法也是动态确定而不是预先设定好的。常见的动态追踪方式是 gdb 的 trace 功能，它将原有指令替换为一个跳转指令，跳转到动态定义的收集运行信息的代码段，最后跳回原来地址执行，该方法可以用于在程序执行中收集寄存器值、变量值等信息。kprobe 方法在 Linux 中提供了类似的功能，它可以动态地在 Linux 内核正在运行的代码中嵌入一段处理函数。

kprobe 的实现原理如图17.5所示：在注册处理函数时，将需要触发处理函数的指令替换为产生断点异常的指令（在 x86 中即`INT 3`），同时保存好原来的指令。当 CPU 执行到此处触发断点异常后，所有寄存器在异常处理流程中保存下来。然后 kprobe 模块调用注册好的 pre_handler 处理函数，如果该函数需要获取触发 kprobe 时某个寄存器的值，可以读取之前保存好的寄存器状态。接下来 kprobe 模块需要执行原来被覆盖的指令——将原有的指令填回到触发中断异常的地址，以单步调试的方式返回该地址执行，然后立即触发调试异常返回 kprobe 并执行 post_handler，最后 kprobe 将`INT 3`指令填到该地址，返回该地址的下一条指令继续运行，至此完成一次完整的 kprobe 处理流程。除了上述两种 handler 外，kprobe 还定义了更复杂的触发条件和对应的处理函数，此处不再展开。

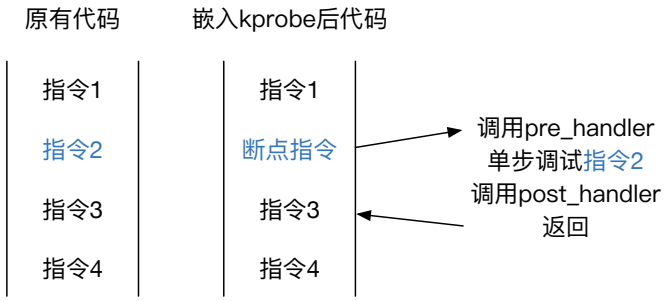

图 17.5: kprobe 实现示意图。

与静态的 tracepoint 比较，kprobe 机制功能更加强大，允许在大部分内核代码中嵌入提取信息的代码。但是 kprobe 在注册时会更加繁琐，并且由于涉及多次异常处理，运行时的开销较 tracepoint 更大。kprobe 通常可以通过两种方式进行配置和调试信息获取：第一种方式是编写内核模块，在内核模块中定义 kprobe 的触发位置和调用的函数等信息，该方法泛用性最广，但是要

求用户手动编写、编译并装载内核模块；第二种方式是使用 debugfs 提供的接口，此方法可以按照预定格式定义需要获取的信息以及进行代码嵌入的地址，该方式定义的 kprobe 方法只能按指定的方式获取和提取特定信息，但是配置较为简单。下面是一个简单的示例，该示例通过 debugfs 接口使用 kprobe 对`kmalloc`调用进行分析，注意需要 root 权限运行：

```
[root@osbook ~] $ cd /sys/kernel/debug/tracing
# 定义一个probe点
[root@osbook ~] $ echo 'p:myprobe __kmalloc size=%di' >
    kprobe_events
```

probe 点的定义中 p 表示在函数开始时调用，myprobe 为自定义 probe 点的名字，添加 trace 的函数为`__kmalloc`。size 的定义需要特别说明，在 x86_64 的 calling convention 里，函数调用的第一个参数放在`%rdi`寄存器中，而 size 为`__kmalloc`的第一个参数，因此此处使用`size=%di`将`%rdi`的值放到名为 size 的变量中，接下来：

```
# 打开myprobe
[root@osbook ~] $ echo 1 > events/kprobes/myprobe/enable
# 等待一段时间后，从buffer中读取追踪信息
# 追踪信息包含了跟踪点名称、地址、size大小等信息
[root@osbook ~] $ cat trace_pipe
170776.162515: myprobe: (__kmalloc+0x0/0x210) size=0x80
170776.162537: myprobe: (__kmalloc+0x0/0x210) size=0x40
...
# 关闭所有probe
[root@osbook ~] $ echo 0 > events/kprobes/enable
# 销毁myprobe
[root@osbook ~] $ echo > kprobe_events
```

在上述过程中，使用者需要手动确定函数各个参数位于寄存器还是栈上，在本样例中需要确定 size 的值在`%rdi`寄存器中。各个架构不同的 calling convention 进一步使之变得复杂。因此 perf 对参数处理提供了更为自动化的方法：

```
# 使用perf添加probe点
[root@osbook ~] $ perf probe --add '__kmalloc size'
```

perf 会尝试从 Linux 的源代码中获取__kmalloc的参数定义，这要求内核在编译时配置额外的调试信息，用于确认源代码目录地址、二进制代码所对应的文件和行数等，这类调试信息被称为 debuginfo （如果没有 debuginfo 则无法提供参数配置支持，请去掉此命令中的 “size”）。在新增 probe 点完成后，可以对该 probe 点进行跟踪：

```
# 新定义的probe名称为probe:__kmalloc
[root@osbook ~] $ perf record -e probe:__kmalloc -a sleep 1
[root@osbook ~] $ perf report
```

在需要使用软件跟踪机制时，一般优先观察预置的静态追踪点是否能满足使用需求，如果有则直接使用，否则使用动态追踪方法自行定义需要的追踪功能。另外如果该追踪方法经常被使用，并且使用者能够重新编写和配置内核，可以考虑新增一个静态追踪点避免使用复杂的动态追踪方法，方便后续使用。

17.2.3 内核与用户态追踪交互机制

现在大家已经了解一些基本的基于软硬件的内核追踪机制，但还有一个问题亟待解决：内核的追踪机制应该如何暴露给用户使用呢？首先，内核需要从用户态获取控制追踪的信息，例如哪些追踪方法需要被开启，追踪的配置是什么；其次，在追踪完成后，内核需要将追踪到的信息暴露给用户态。我们回忆下在 tracepoint 和 kprobe 的样例中，尽管我们使用了不同的内核追踪方法，但是都使用了位于/sys/kernel/debug/tracing 下的多个文件来定义追踪方法并获取追踪信息，这是为什么呢？实际上这是因为它们都使用了 ftrace——一套基于文件系统的内核与用户态交互调试信息的方法。除了使用文件系统接口外，我们还可以使用系统调用、共享内存的方式实现内核与用户态的信息交互，例如 ebfp 就采用了这样的方法。接下来本小节将简单介绍 ftrace 和 eBPF 这两种用于内核态与用户态交互追踪信息的手段。

ftrace 机制

ftrace 最早仅仅是一套跟踪内核函数调用流程的机制（function tracer），但在演变过程中，ftrace 成为了一套更为通用的、用于用户态和内核交互的机制。该机制向下可以支持许多 tracer，包括原有的 function tracer、用于记录线程唤醒的 wakeup tracer 和用于记录开关中断的 irqsoff tracer 等，以及后来

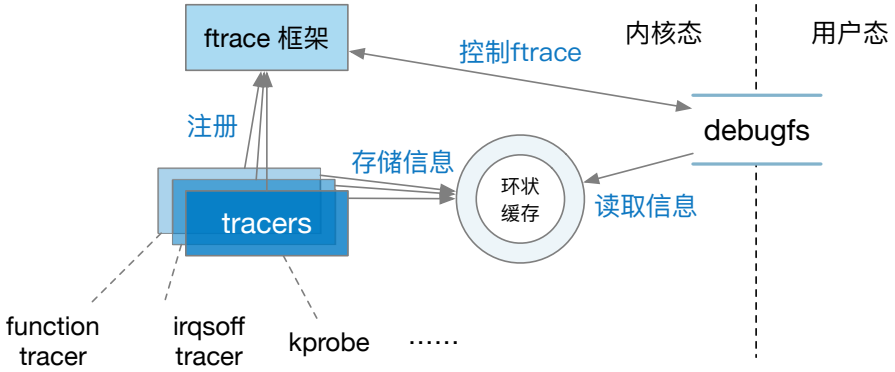

图 17.6: ftrace 框架。

加入的 tracepoint、kprobe 等。同时 ftrace 向上给用户态提供文件系统的接口操作。

ftrace 的架构大致如图17.6所示，ftrace 向下为不同的 tracer 提供了统一的接口，用于打开和关闭 tracer、传输用户定义的控制信息等。注册过的 tracer 可以将输出信息缓存到一段统一的环状缓存当中，等待用户的读取。用户态通过 debugfs 中文件的接口与 ftrace 交互，一方面可以向 ftrace 框架输入控制信息或者读取控制信息，另一方面可以读取环状缓存中各个 tracer 产生的输出。

eBPF 机制（extended Berkeley Packet Filter）

eBPF 由 BPF（Berkeley Packet Filter）发展而来，早期 BPF 机制主要用于在内核中插入用于过滤网络包的代码，将符合条件的网络包发送到用户态。为了定义过滤网络包的规则，BPF 提出了一套大约包含 20 种指令的伪指令集（伪指令集是指相关指令无法在真实的硬件上运行，最终需要软件翻译为真实的指令运行），同时 BPF 在内核中实现了一套解释器用于执行该伪指令集。由于解释器的执行效率远低于在真实硬件上运行的效率，因此 Linux 在主流的架构上都提供了 JIT（just-in-time compile）的功能，用于将 BPF 伪指令集动态翻译成原生指令，并在处理器上直接执行。用户可使用该指令集定义相应的网络包过滤规则，通过 socket 的接口将规则传入到内核以及读取过滤出的信息。

与 ftrace 的发展类似，开发者发现 BPF 机制可以成为更广泛的在内核中插入用户书写代码的框架，因此对原有的机制作出扩展，命名为 eBPF 机制（如图17.7所示），而原有的 BPF 机制被称为 cBPF（classical BPF）。相对于 BPF，eBPF 作出了如下改进：

- 为了提供更用户友好接口，允许使用 C 语言编写代码片段，并通过 LLVM 编译成 eBPF 伪指令；
- 定义了多种 BPF 程序类型，原有的 cBPF 是其中一种类型 SOCKET_FILTER，除此外还有用于定义 kprobe 点的 KPROBE 类型、定义 perf event 发生时回调的 PERF_EVENT 类型；
- 原有使用 socket 实现用户态与内核交互的方法接口较为局限，eBPF 做出了两点改进：1. 定义了一个专用于 eBPF 的新的系统调用，用于装载 BPF 代码段、创建和读取 BPF map（稍后介绍）内容等操作；2. 提出了 BPF map 机制，用于在内核中以 key-value 的方式临时存储 BPF 代码产生的数据。例如装载进 BPF 内核的代码在每次判断有 TCP 或者 UDP 包时，往 BPF map 中 key 为 TCP 或 UDP 的项中的计数加一，用户态程序可以选择在任意时刻通过系统调用读取 BPF map 中 TCP 和 UDP 包的数量。
- 在内核中提供检查机制防止装载威胁内核安全的 BPF 代码。注意这种检查并不能完全避免代码中出现危害内核安全或者功能实现错误的情况，只能尽可能控制隐患。具体来讲该验证机制分为两个步骤：1. First Pass：使用深度优先搜索确认 BPF 代码是有向无环的，避免在 BPF 代码中永久循环；限制 BPF 代码的大小；避免存在通过 jmp 的方式跳出此段 BPF 代码；以及其它一些基本检查。2. Second Pass：此处检查过程会从第一条指令开始尝试运行，运行时会访问的内存、会调用的函数等信息都会被检查。

直接使用 BPF 指令编写插入内核的代码难度较高，既需要手动编写 BPF 代码用于获取内核信息，还需要处理注册、读取 BPF map 等与内核的交互，因此许多用户希望能有工具简化 eBPF 使用，甚至预先配置好一些常见的 eBPF 调用方法。bpftrace 和 bcc 正是在这样的需求下诞生的工具。bpftrace 和 bcc 都是 eBPF 机制的包装，但是两者使用方式略有不同。bcc 全称 BPF Compiler Collection，名字中的 compiler 和 collection 对应了它两种主要的使用方式。从 compiler 方面讲，bcc 提供了一套使用 Python 的 eBPF 代码编写框架。比起原有使用 C 语言或以字节码方式编写 eBPF 代码的方式，使用 bcc 编写会相对更简洁。但即使如此，使用 bcc 编写 eBPF 程序仍然需要对内核有比较深的了解，为了方便大部分用户使用，bcc 提供了一组工具的 collection。它收集了数十种通用性较强的已经编译好的工具，用户简单地调用一个可执行文件即可获得期望的数据。

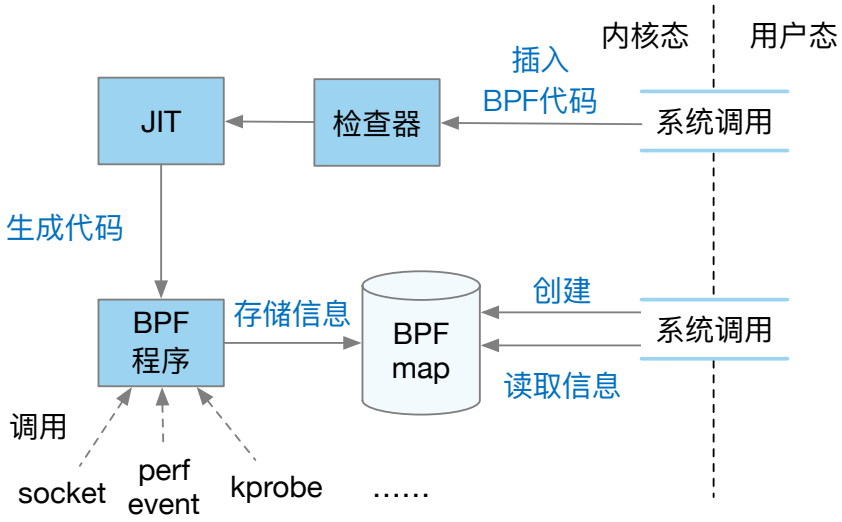

图 17.7: eBPF 框架。

bcc 中的工具需要通过比较复杂的编译流程生成，每次编译需要耗费较长时间，并且学习 bcc 代码编写难度也不低，因此若仅仅只是临时使用 eBPF，这种方法会显得非常繁琐。bpftrace 一定程度上缓解了这个问题，它使用了比较简单的语法，以一句简单的命令即可完成装载 eBPF 代码并输出的工作。例如，如果想获取一段时间内`kmalloc`相关的调用信息，可以使用：

```
# __kmalloc函数定义：void *__kmalloc(size_t size, gfp_t flags)
[root@osbook ~] $ bpftrace -e 'kprobe:__kmalloc {  printf("pid:%d
    size:%ld flags:0x%x\n", pid, arg0, arg1); }'
Attaching 1 probe...
pid:319 size:48 flags:0xd40
pid:633 size:4 flags:0xcc0
pid:633 size:8 flags:0xcc0
```

该 eBPF 程序使用 kprobe 的方式，在`__kmalloc`函数中嵌入了打印进程号及函数调用参数的方法，bpftrace 根据 calling convention 推断存储函数参数`arg0`和`arg1`的寄存器，嵌入的打印方式会读取这些寄存器的值作为函数参数。由于这些寄存器在函数执行过程可能被修改，因此在函数入口位置使用该方式获取参数值更为可靠。总体来说从编写 eBPF 程序的角度讲，bcc 的使用难度较 bpftrace 高，但是具有更加丰富的语义。

17.2.4 内核日志管理

在本小节末，我们简单讨论内核的日志机制。日志机制可以看作是一种基础的静态追踪手段，日志会记录程序在执行过程中打印的信息，打印调试信息是一种非常基础却可靠的调试方法，能够有效地帮助程序员理解程序的执行状况。下面我们以 Linux 的 printk 为例，分析该日志机制提供的功能以及解决了哪些挑战。

从日志的基本功能角度考虑，Linux 的日志机制有如下需求：第一，日志带来的性能开销应当尽可能低，尽量不影响内核运行性能。因此 printk 会先将日志记录在环状缓存中，与日志读取操作保持异步关系，尽量减少日志读取对内核正常运行带来的性能影响。第二，需要能够支持使用多种方式访问内核日志。除了能够在本地获取日志外，为了处理被调试机器崩溃后无法从本地访问日志等情况，还需要支持跨机器日志获取。在 Linux 中可以通过文件系统的接口访问/dev/kmsg 或者/proc/kmsg 获取环状缓存中的日志信息，还可以通过串口传输日志到其他机器以支持远程读取日志。第三，用户能够灵活地配置选择记录哪些日志。printk 提供了日志级别控制，用户可以根据自身需求配置日志级别，例如开发者希望尽可能获取详细的调试信息，而普通用户可能只需要警告或者错误相关的日志用于确保系统的正常运行。Linux 将日志分为 DEBUG、INFO、WARN、EMESG 等 7 个级别，用户通过文件系统的接口访问/proc/sys/kernel/printk 动态定义日志级别。

除了上述基本的日志功能外，由于操作系统的特殊性，日志的实现还需要考虑到其它一些因素，例如：

- 支持在大部分执行环境下都能正常使用日志，例如中断处理函数中、中断打开或者关闭条件下以及抢占打开或者关闭条件下；
- 尽量保证日志的先后顺序不变，这便于用户根据日志推断程序执行情况；
- 在多核情况下，不同核产生的日志之间不应当产生冲突；
- 尽可能可靠，当操作系统其它部分崩溃时，日志模块还能够尽量输出调试信息用于调试；

这些因素相互组合使得 printk 的实现相当困难，例如由于需要在中断处理（包括 NMI 处理）中使用 printk，内核日志的并发控制不能完全依赖于锁，否则会使得日志的实现不可重入（即在调用日志相关函数并上锁后，如果触发中断，然后在中断处理函数中再次调用日志相关函数，此时无法成功上锁）。因此 printk 的并发控制较多地使用了无锁的以及 per-cpu 的数据结构设计，并且

对于 NMI 中断预留了独立的日志缓存区域。另外，可靠性和功能性的矛盾也体现在 printk 的设计中，在最初的日志设计（Linux 0.01 版本）中，日志会被同步地打印到终端，这种设计虽然在性能和功能上都有局限性，却能较好地保障日志不被丢失。而当 printk 的设计为了支持各种功能变得越发复杂后，可靠性越发难以得到保障。如何平衡 printk 的可靠性和功能性仍然是值得研究的问题，例如有的设计希望将部分重要的日志同步输出到特定读者 [12]。

17.3 操作系统性能监控

本节主要知识点

- ❑ 性能监控的基本功能是什么？
- ❑ 硬件的性能监控计数和采样是如何实现的？
- ❑ 操作系统如何将性能监控硬件暴露给用户态程序？

在对内核进行性能优化时存在这样的需求：找到操作系统哪些函数执行时间占比较大、在哪些地址经常发生缓存缺失。对此，我们可以使用性能监控寄存器了解内核性能相关的状况。本小节将会首先介绍性能监控的硬件基础——性能监控寄存器，然后介绍配套的 Linux 前端工具 perf。

17.3.1 性能监控计数器

性能监控计数可以用来监控程序执行过程中处理器发生某些事件的次数，例如执行了多少条指令、缓存未命中次数等。通过这些监控统计数据可以对程序的执行进行调优。通常使用性能计数器的步骤是：首先设置需要计数的事件类型，然后将相关事件计数器复位，接下来开始执行待测试程序，在测试程序执行完后读取计数寄存器获得某些事件发生次数。

以 Intel 的 Architectural Performance Monitoring 为例，处理器提供了一组性能监控寄存器（performance-monitoring counter MSR），其中 IA32_PERFEVTSELx 为事件选择寄存器，IA32_PMCx 为事件计数寄存器。两种寄存器成对存在，且每个核心的性能监控寄存器是独立的。但是不同型号处理器拥有的两种寄存器数量可能不同（x 表示数量不定）。IA32_PERFEVTSELx 的定义如图17.8所示，比较关键的几个区域为：

- Event Select (bit 0-7) 和 Unit Mask (bit 8-15)：两者共同选择待记录的

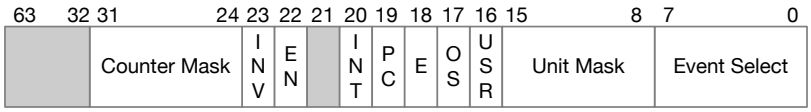

图 17.8: IA32_PERFEVTSELx 寄存器定义。

特定事件，例如当 Event Select 为 0xC4，Unit Mask 为 0x41 时，将对 Last Level Cache 的缓存缺失数量进行统计，计数结果会被放到对应的 IA32_PMCx 寄存器中；

- INT (bit 20)：当该 bit 为 1 时，IA32_PMCx 寄存器计数溢出时会发送中断至处理器。

除了这两组寄存器之外，该机制还提供了 IA32_PERF_GLOBAL_CTRL 寄存器用于控制各个事件寄存器的开关，以及 IA32_PERF_GLOBAL_STATUS 寄存器用于在处理溢出产生的中断时判断是哪些事件寄存器溢出导致的中断。经过配置用户态程序可以在不处于 ring 0 时使用 rdpmc 指令读取计数寄存器的值。

AARCH64 架构也有性能监控计数器相关支持，和 Intel x86 类似，同样有寄存器用于选择事件（PMSELR）、读取事件计数（PMEVCNTR）。对于溢出的情况，既可以使用中断处理计数器溢出，也可以读取溢出寄存器判断是否发生过溢出。

计数器溢出产生的中断可以作为采样点使用。例如为了找出内核中发生缓存缺失较多的函数，可以使用处理器缓存缺失次数做采样。当缓存缺失计数触发中断时，记录发生中断的函数地址，最后统计这些地址大部分处于哪些函数，即可能分析出哪些函数是当前内核性能的瓶颈。但是该方法面临着两点问题：首先，中断的产生和内核处理中断之间存在时间差，因此统计出的指令地址等信息极有可能与真实触发计数器中断的位置产生偏差；其次，每次都依赖中断产生采样数据的方法效率较低下，很有可能对待测程序本身的执行造成影响。为了解决这两个问题，Intel 在硬件层面为各个计数器提供了采样功能的支持Processor Event Based Sampling (PEBS)：当一个计数器产生溢出时，处理器会收集当前相关状态（各寄存器值、访存地址等），并把该状态保存至一段指定的内存中。如图17.9所示，和BTS相同，PEBS 也使用 IA32_DS_AREA 指定的内存区域用于存储采样信息，存储格式也非常类似，同样是维护了一块用于存储 PEBS 记录的数组。区别主要在于 PEBS 记录的信息更加丰富，例如产生某条 PEBS 时的访存地址、寄存器值等，并且可以通

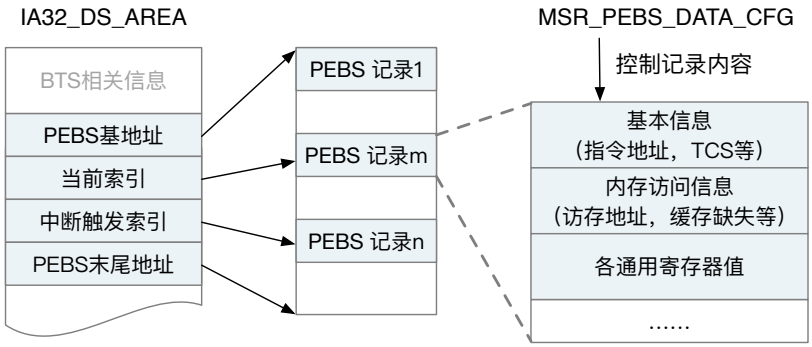

图 17.9: Processor Event Based Sampling 存储结构。

过设置 MSR_PEBS_DATA_CFG 寄存器指定需要记录哪些信息，避免记录所有种类信息带来较大的性能开销。

我们可以在 Linux 中使用 perf 获取性能监控计数器信息（perf 的详细介绍请参考章节17.3.2），注意需要 root 权限：

```
# 1.获取1秒内LLC的缓存缺失次数
[root@osbook ~] $ perf stat -e LLC-misses -a sleep 1
[root@osbook ~] $ perf report

# 2.使用针对cycle数采样的方式统计内核函数时间占比情况
[root@osbook ~] $ perf record -e cycles -a sleep 1
[root@osbook ~] $ perf report

# 3.使用硬件采样支持的方式针对cycle数统计函数时间占比，比较2和3的结
    果有何差异
[root@osbook ~] $ perf record -e cycles:pp -a sleep 1
[root@osbook ~] $ perf report
```

17.3.2 性能相关事件（perf events）

在前文中，我们使用 perf record/report 间接地访问了性能监控寄存器，避免了手动读写寄存器的繁琐工作，下面我们简单了解下 perf 是如何工作的。perf 工具是 Linux 内核 perf events 机制的包装，perf events 机制将性能监控寄存器和其它许多内核跟踪机制（见第17.2章）抽象为 event，即发生了某种事件（如性能监控寄存器值改变、某个函数被调用、Branch Trace Store 产生了新的条目等），并可以根据配置附带记录下事件发生时的某些运行信息（如指令地址、位于哪个处理器核心等）。perf events 主要维护了两类事件相关的

信息：

- 事件计数统计：统计特定的 event 发生的次数。例如若需要统计运行一个应用运行时 LLC 的读缓存缺失数量，可以将 perf 的 event 设置为 LLC-load-misses。在运行期间，该事件的数量会不断累加。应用运行结束后，读出 LLC-load-misses 事件发生的次数即可；
- 事件 profile：发生某个事件时记录指定的运行信息。例如我们如果需要了解一个应用运行时在哪些位置发生了 LLC 读缓存缺失，则可以对 LLC-load-misses 事件 profile，从而统计产生 LLC 读缓存缺失的指令。另外用户很可能只是关心在程序哪些位置产生了比较多的缓存缺失，用于指导程序性能调试，而并不需要详细地了解每个发生缓存缺失的地址，并且如果每次发生缓存缺失事件时都统计指令地址会产生较大的性能开销。针对上述问题，perf events 提供了采样的支持，在这个例子中，可以在 LLC-load-misses 事件发生了一定次数后，才收集一次产生缓存缺失的地址。

perf events 事件主要分为如下几类：

- Hardware Events：通常是本小节硬件计数器相关的事件；
- Software Events：内核实现的软件计数器记录的事件，例如每次发生缺页异常时，都会增加名为 page-faults 的计数器的值；
- Tracepoint Events：第17.2.2章中描述的 tracepoint 产生的事件；
- Dynamic Tracing: 第17.2.2章中描述的 kprobe 所动态定义的事件；
- Timed Profiling：每经过一段固定时长产生的采样事件。

下面简单介绍用户态和内核是如何交互 perf events 信息的。在控制方面，用户态程序通过`perf_event_open`系统调用通知内核哪些 event 需要触发跟踪机制，以及需要记录 event 的哪些信息。操作系统将返回一个 fd，用户通过文件的接口访问产生的事件。为了获取上文所述的 perf events 信息，应用可以使用读文件的操作获取计数统计，由于事件 profile 的数据量可能较大，应用使用`mmap`建立与内核的共享内存，内核会在该共享内存中维护一个环状缓存用以存储 profile 信息，可以随时在用户态读取。

但是对于大部分场景而言，通过 perf events 相关的系统调用和共享内存读取跟踪信息还是比较繁琐，此时使用 Linux 提供的用户态前端工具 perf 即

可。我们可以列举 perf 具体提供了哪些事件（tracepoint、kprobe、性能计数器等）：

```
[root@osbook ~] $ perf list
cache-misses       [Hardware event]
context-switches   [Software event]
kmem:kmalloc       [Tracepoint event]
...
```

我们可以使用 perf 工具对于某个事件进行计数统计或者 profile：

```
# 事件计数统计：统计1秒内LLC读缓存缺失的数量
[root@osbook ~] $ perf stat -e LLC-load-misses -a sleep 1

# 事件profile：统计1秒内发生LLC读缓存缺失的地址
# 每1000次事件收集1次记录
# 结束后会产生一个记录了事件信息的临时文件
[root@osbook ~] $ perf record -e LLC-load-misses -c 1000 -a sleep 1

# 根据临时文件获取缓存缺失发生时的函数地址
[root@osbook ~] $ perf report
# 或者直接读取临时文件中的原始数据
[root@osbook ~] $ perf script
```

另外除了可以按照事件发生次数进行采样，还可以按照固定的时间间隔采样：

```
# 以每秒99次的频率收集采样时指令所处的地址
[root@osbook ~] $ perf record -F 99 -a sleep 1
```

17.4 操作系统测试方法

本节主要知识点

- ❑ 为什么操作系统需要测试，根据不同目的有哪些针对性的测试？
- ❑ 各类测试需要遵循哪些基本的原则以保障测试效率和效果？
- ❑ 有哪些辅助手段提升测试的效率？

软件测试是一种验证程序功能是否满足预期的方法，通常需要在特定的条件下运行软件，收集软件的各种指标确认是否满足某些条件。例如软件执行过程中是否产生崩溃、软件执行行为是否符合设计、在特定时间内是否能完成运行等。测试环节对于绝大部分的软件开发都是不可或缺的，并且由于许多操作系统具有在多平台运行、代码规模较大、可配置性较强等特点，操作系统特别需要完备的测试验证其正确性。

在常见的操作系统测试流程中，一般需要先通过功能测试验证执行过程中没有崩溃等异常现象，并且基本的功能实现符合设计。由于在小规模测试中暴露的错误更方便定位与修复，因此功能测试通常会遵循测试规模从小到大的原则，常见的测试顺序如下：

- **单元测试**：对软件的基本模块进行测试，粒度最低可能细化到函数的层次；
- **集成测试**：将通过单元测试的多个基本单元组装，验证通过接口组合的多个模块能否协同工作；
- **系统测试**：系统测试是在整个系统完整运行的条件下运行的测试。

除了功能测试以外，操作系统还可能出于其它目的进行对应的测试：

- **性能测试**：验证系统在稳定运行时的性能指标满足规格要求，且性能指标波动在可控范围内；
- **压力测试**：验证系统在超负荷运行状态下是否仍然能够保证正确性；
- **回归测试**：在对代码作出修改后，重新进行完整的测试，目的是验证新的代码没有引入新的 bug 或者引发原来未触发的 bug；
- **兼容性测试**：验证系统在多种环境、不同配置下是否能够正确运行；

本节将以 Linux 系统为例，从上述角度介绍操作系统是如何进行完整的测试的。

17.4.1 操作系统测试流程

规模从小至大的功能测试

在操作系统的开发中，保障功能正确性是最基本的需求，这要求操作系统在运行时不会产生崩溃等异常行为，且系统调用等功能行为符合预期。因此在

操作系统测试流程中，通常都会先进行功能测试，并且由于方便定位错误和及早暴露错误，功能测试的规模通常由小到大。

规模最小的测试为在函数的粒度上进行的**单元测试**，通常由操作系统的内部直接调用测试函数。测试函数应当尽量只涉及单个模块的功能。例如 Linux 内核为了测试锁的正确性，提供了 lock torture 测试，该测试对各类内核锁的接口进行调用并验证正确性。代码片段17.2展示了 lock torture 如何验证读写锁中写锁的正确性，该方法执行拿锁放锁操作并在临界区检查变量lock_is_write_held值是否符合预期。测试时会有多个内核线程同时调用lock_torture_writer函数验证并发时锁的正确性。lock torture 测试会被编译为一个内核模块，需要进行该测试时把此内核模块装载进内核开始对各类锁进行测试，测试结果记录在内核日志中。

```
// 对读写锁的写锁进行测试
int lock_torture_writer()
{
  ...
  // 拿写锁
  cxt.cur_ops->writelock();
  // 解锁前 lock_is_write_held 被设为 0，因此检查为 1 则发生错误
  if (WARN_ON_ONCE(lock_is_write_held))
    // 增加错误计数
    lwsp->n_lock_fail++;
  lock_is_write_held = 1;
  ...
  // 放写锁前将 lock_is_write_held 重设为 0
  lock_is_write_held = 0;
  cxt.cur_ops->writeunlock();
  ...
}
```

代码片段 17.2: lock torture 测试中对读写锁的测试。

但是这种为每个模块单独添加内核模块进行单元测试的方法有一些弊端：首先运行这样的单元测试需要编译并运行一个完整的内核，并且需要一定用户态的支持用于加载内核模块。这会使得单元测试部署效率降低，并且无法排除内核其他部分对该模块的影响。其次，这种即兴地为每个模块独立地编写单元测试的方法缺乏统一的单元测试管理，当测试规模变得庞大时，很难统一地进行测试、收集测试结果等。因此 Linux 引入了统一的 kunit 单元测试框架，kunit 框架默认在UML模式下运行（见第17.1.2章），避免由于架构相关因素影响测试结果。当启动 kunit 后，内核会尽可能只编译与启用测试相关的代码，因此

拥有比完整内核更快的编译和部署速度，并且能尽量避免无关模块运行的间接影响。当然由于 kunit 测试运行在 UML 环境下，该方法无法对驱动等涉及硬件底层的代码进行测试。对于用户来说，编写 kunit 测试主要需要实现一系列函数调用及判断返回值是否符合预期，如代码片段17.3所示，该测试通过多个判断语句验证`list_add`接口的正确性。

```
// 依次将 a 和 b 加入链表 list 头部
list_add(&a, &list);
list_add(&b, &list);

// KUNIT_EXPECT_PTR_EQ 判断第二和第三个参数值是否相等且为指针
// list 的后继节点为 b
KUNIT_EXPECT_PTR_EQ(test, list.next, &b);
// b 的前驱节点为 list
KUNIT_EXPECT_PTR_EQ(test, b.prev, &list);
// b 的后节点为 a
KUNIT_EXPECT_PTR_EQ(test, b.next, &a);
```

代码片段 17.3: 使用 kunit 测试 list_add

当使用单元测试验证各模块基本的正确性后，我们可以开始组合各模块进行**集成测试**。由于大部分操作系统提供给用户态的服务是以系统调用方式的方式体现的，因此一种比较常见的测试方式是通过系统调用较为独立地测试某几个模块，例如对文件读写系统调用会涉及上下文切换、VFS、文件系统、硬盘驱动等。

Linux 内核源代码中包含了一个基于系统调用的功能性测试框架 kselftest。该框架测试了大量系统调用的基本使用方式，这些测试相对比较简单，可以以较快的速度完成测试，但缺点在于测试完整度较低，例如测试一个系统调用时只有较少参数配置。而独立于 Linux 内核源码的测试框架Linux Test Project (LTP)提供了基于系统调用的详尽的基本功能测试，它通过改变系统调用的参数、组合多个系统调用等方式，较为完整地完成了各类系统调用的测试。

下面我们以`clone`系统调用为例，分析其在两个测试框架中是如何测试的。`clone`系统调用可配置性极强，可以控制创建进程还是线程、新进程是否属于另一用户、是否和新进程保持父子关系等，因此对其做正确性验证比较有挑战性。kselftest 中`clone`的测试较为简单，在 5.2 内核版本的 kselftest 中，`clone`仅有两处针对性的测试，一处是创建新的线程，另一处是创建属于其它

用户的进程。相对而言，LTP 的测试比较完全，在 20200120 版本中，有 9 组针对 clone 的测试，通过如下方式验证 clone 的正确性：

- 改变参数，覆盖大部分的配置，并且对各种配置进行组合；
- 功能性检查，在多种配置的情况下，验证这些配置是否生效，例如如果配置了父子进程共享内存空间（CLONE_VM），则子进程对内存的修改应该能够在父进程中读到；
- 对于边界条件有对应的检查，例如故意为子进程设置栈为空，验证 clone 是否能返回正确的错误代码；
- 验证多次调用或嵌套调用 clone 后功能的正确性；

验证可靠性

在操作系统完成各项基本的功能测试后，需要考虑这些功能在极端情形下是否仍然能保持正常工作。因此接下来我们考虑使用**压力测试**，即把一部分系统资源压榨到极致，并且保持长时间运行，观察操作系统是否仍然正常工作。这些产生压力的系统资源包括处理器资源、内存资源、硬盘 I/O 资源、网络资源等。极其频繁地调用某些系统功能（如改变线程优先级、建立文件映射等）也能构成压力，另外压力测试还需要验证高并发环境下操作系统的正确性。

以LTP为例，该测试集内置了多种针对各类资源的压力测试方法，这些测试通常具有较高的可配置性。例如用于尽可能造成磁盘碎片化的 growfile 测试，它会不断 create、truncate 文件，并且按照一定的步长执行 write 和 seek 操作，这里文件大小、步长、写操作大小等都是可配置的。一次比较完整的 growfile 测试会初始化大量不同配置的测试进程并发执行，保证测试的随机化程度。多个对某一资源的压力测试可以进行组合，例如希望对文件系统做完整的测试，除了运行上述的 growfile 外，还可以运行造成大量读写操作的测试、大量对目录操作的测试、对特殊目录（/dev，/sys等）访问的测试等，保证涵盖大部分用户可能使用到该资源的方式。最后，一个完整的压力测试会涉及多种资源，用户可以根据自身场景的需求自行设计需要测试哪些资源的使用，例如一个用户自行配置编译了主要用于单机高性能运算的系统，则压力测试中应该尽量增加处理器和内存负载，并且应当加入有关处理器和内存管理的大部分测试，而对网络方面则可以只对少部分基本功能进行压力测试。但是对于操作系统发行者来说，对所有系统资源均进行压力测试是必须的，因为无法确定使用者会以何种方式使用该系统。

确保性能指标

在通过功能测试和压力测试保障操作系统的基本正确性后，我们需要预防错误或者不合理的设计及实现对操作系统的性能造成影响，因此需要**性能测试**定量地分析操作系统的性能指标是否在预期范围内。另外，性能测试还可以用于横向比较操作系统在不同的配置下、在不同硬件上运行、甚至是不同操作系统间的性能表现。

当需要针对某种系统资源进行性能测试时，测试大致可以分为如下几个步骤：

1. **选择和配置测试程序**：在测试前需要明确测试的性能指标（吞吐量、延迟等）是什么，根据不同的测试目的考虑需要调整哪些变量，例如为了测试可扩展性有可能需要改变测试程序线程数量。同时需要确定测试针对的场景是怎样的，例如测试文件系统吞吐量时是针对随机还是顺序读写的情况。在明确这些测试需求后，选择符合该要求的测试程序，在现有测试程序不能满足要求前提下也可以自行设计实现。另外测试输入的选择需要尽量符合真实的场景，并且具有代表性。

2. **进行实验**：首先，在多次实验中我们需要保证无关变量尽可能相同，这些变量既包含待测试资源相关的配置，也包含其它系统配置。例如对文件系统测试时，如果文件系统类型（如 ext4、btrfs）、文件系统配置（如日志级别、缓存）、硬盘的型号和种类（如 SSD、RAID）不属于测试自变量，则应当在不同测试中保持相同配置。另外还需要保证内核版本、调度策略、时钟中断频率等系统配置尽可能一致，确保后台没有其他占用大量系统资源的进程影响测试。其次，我们需要尽可能减少测试环境中随机和不稳定因素对测试结果的影响，例如将测试程序绑定核心运行，避免线程的跨核心调度影响测试结果，以及在单个 NUMA 节点进行测试，以避免跨 NUMA 的内存访问、I/O 资源访问等。最后，为了确保实验结果的参考价值，进行实验的环境需要与实际运行时相匹配，例如实验开始时需要保持一段时间预热，达到稳定状态后才开始收集测试结果。

3. **结果收集及处理**：实验需要进行多次，当测试结果出现波动时，去除明显偏离正常范围的数据。如果测试结果仍然不稳定，可以使用方差表示误差范围。另外为了对性能测试进行详细分析，除了收集性能指标数据外，还有可能需要收集额外的数据，例如周期性统计 CPU、内存占用率确定两者是否成为性能瓶颈，以及为了分析程序性能瓶颈时，可以使用第17.3.2章中 perf 的采样机制，确定哪些函数执行时间占总时间比例

较大。最后，为了方便分析，可以考虑将结果以可视化的方式呈现，例如在使用 perf 采样各函数时间占比并记录函数调用关系后，可以使用 FlameGraph [3] 直观地观察哪些函数可能导致性能瓶颈。

上述方法适用于针对特定的运行场景对某种系统资源进行测试，但是在内核开发和发行过程中往往无法确定使用者的具体场景，因此需要广泛地测试众多性能指标，保证没有出现预期之外的性能下降问题。在这样的需求下，我们需要一种集成了大量测试、从多个维度进行性能评估的测试框架，并且还能够便捷地整合各类自定义的测试。例如 Phoronix Test Suite [8] 是一种比较成熟的综合性能测试框架，该框架提供了统一的途径配置测试程序下载地址、安装脚本、参数输入、输出处理等，集成了众多第三方的测试程序，用户也可以按照框架提供的统一的方式配置其它测试程序。

确保兼容性

由于操作系统很有可能需要在不同硬件环境下运行，并且需要支持在其上运行的各类应用，因此需要**兼容性测试**用于确定操作系统能否在不同的软硬件环境下正常运行。

首先我们需要对**硬件兼容性**进行测试，由于许多操作系统支持在不同的平台上运行，并且需要直接控制各类不同的硬件资源，因此硬件兼容性的保障显得尤为重要并且有挑战性。为了验证操作系统的硬件兼容性，我们需要保证操作系统在大量的硬件平台上能够正常启动，并且能够完成一些基本的功能测试。kernelci [5] 正是基于这样的目的所实现的工具，它是一个分布式的自动测试框架，会跟踪 Linux 系统代码变化，自动编译各个内核分支，然后将其部署在大量不同的设备上，最后运行针对少数几个子系统的测试程序，以此判断 Linux 系统在不同的硬件下是否均能正常工作。

其次我们需要对**软件兼容性**进行测试。由于不同操作系统的存在，以及操作系统随着时间的推进不断更迭，应用程序很有可能需要在各种不同的操作系统下工作。为了不同版本的操作系统能够对上层应用提供一致的服务，避免应用需要针对不同版本系统做针对性开发，操作系统的实现可能需要遵循各类标准，例如对编程接口（API）进行规范的Portable Operating System Interface (POSIX)标准，以及对 Linux 的目录树、初始化流程等进行定义的Linux Standard Base (LSB)标准。操作系统可以运行与这类标准相关的测试，依次验证自身的各个功能是否符合标准的定义，例如通过测试 POSIX Test Suite [9] 验证接口是否符合 POSIX 标准。

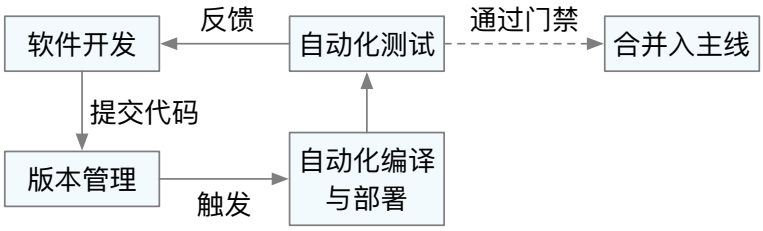

图 17.10: 持续集成的基本步骤。

持续集成

在操作系统不断的开发和部署过程中，往往涉及对功能的增强、bug 的修复、配置的修改等变更。这些变更有可能影响被修改过的模块本身或者是其它模块，甚至是触发过去存在但是没有被触发的异常，导致引入 bug，触发原有 bug，或者是性能受到影响等。因此在操作系统代码或者配置被修改后，即使作出的修改与这些测试涵盖的内容没有直接关系，也需要及时完整地重新执行上述测试，以此保障操作系统在不断的开发迭代过程后仍然正常工作，这种测试方法被称为**回归测试**。

但可以发现，如果每次修改代码后都需要手动测试的话，特别是在测试流程较为复杂和繁琐的条件下，会给程序员带来较大的额外负担。因此我们需要**持续集成**（Continuous Integration，CI）的支持，即开发者较为频繁地将代码合入主线，通过**自动化测试**和**门禁系统**保障回归测试的有效性（如图17.10所示）。具体来讲，当代码被推送到远端分支或远端分支合并进入主线等情况发生时，持续集成框架会触发自动化的软件编译及部署，然后进行回归测试，测试结果会及时反馈到开发者。回归测试的规模和样例选择可以依据修改的规模而定。例如在小规模的代码修改后可以只进行基本功能测试，而在主线代码有较大改动后使用完整的压力测试（如 LTP）和性能测试（如 Phoronix Test Suite）。同时为了确保主线代码的有效性，门禁系统会确保代码在通过预定测试后才能合并进入主线。

17.4.2 内核测试辅助手段

在操作系统测试，特别是功能性测试中，通过简单地运行测试程序、观察有无异常的测试方式有一定局限性。首先，虽然测试过程中内核可能触发异常，但并不一定能够观测到，需要在测试过程中通过一些辅助手段帮助测试程序暴露内核的异常。其次，测试的有效性难以衡量，有可能出现运行了大量测

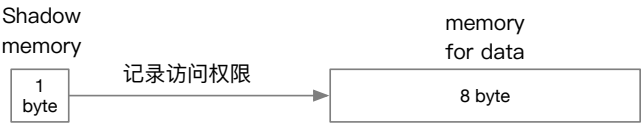

图 17.11: KASAN 内存元数据存储。

试但是仍有模块几乎未被测试的情况，因此需要内核和测试工具提供衡量测试有效性的机制并指导应该补充何种测试。本小节将简单介绍几种内核提供的用以缓解上述两个问题的支持。

静态分析工具

在功能性测试、压力测试等通过运行的方式确定操作系统是否正常运行的方法中，测试往往需要较高的时间成本，并且依赖于测试过程中触发异常。而静态分析则能够在编译时高效地检查代码是否存在特定模式的错误，当然由于通常只能对特定错误进行检测，因此在功能性上有一定局限性。我们可以使用 sparse [10] 工具对 Linux 内核代码进行静态分析。该工具能够检测出部分较为简单的代码错误，例如将用户态地址标记上 __user，在直接访问该地址内容时会触发异常报警（应当使用类似 copy_to/from_user 的接口访问）。再比如可以检测函数是否忘记放锁，实现方法是在 lock 和 unlock 操作时，分别对引用计数加一和减一（初始计数为零），当抵达函数结尾时如果引用计数不为零，则说明有重复上锁或没有放锁的情况。该方法较难处理循环或函数调用中使用的锁，但能够检测出一部分错误的情况。可见，静态测试方法有较大的使用和功能上的局限性，但是在动态测试前先使用静态检测还是能在一定程度上降低测试的难度，提前找到部分异常。

内存检查工具

在对软件进行功能性测试时，有可能代码触发了异常行为，但是并没有产生可见的错误，此时测试程序并不能发现异常。即使经过长时间的运行，该异常行为确实引发了可见的错误，但是由于发生异常和错误的时间和位置往往相距较远，很难从错误信息追溯到出现异常的位置，分析出异常发生的具体原因。这类异常行为中，比较有代表性的是内存访问异常，包括使用一段已经被释放的内存，数组访问越界等。例如当数组访问越界时，如果越界部分内存刚好没有被使用，则不会触发错误。为了能及时检测到这类内存访问出现异常，

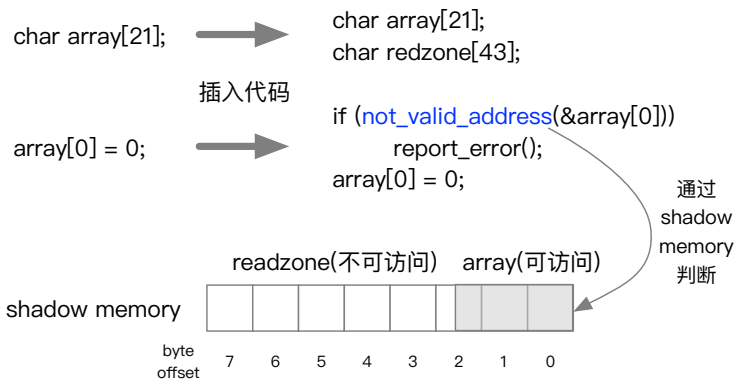

图 17.12: KASAN 自动插入的检查代码。

人们提出了众多方法，其中比较有代表性的方法是 Address Sanitizer [1]。它通过在内存访问的前后加入检查代码的方式找出部分内存访问异常，该方法在 Linux 内核中的实现被称为Kernel Address Sanitizer (KASAN)。下面我们简单讨论 KASAN 是如何检查数组访问越界的，如图17.11所示，KASAN 需要记录各个内存区域是否是可访问的，每 8 个字节的数据需要用 1 个字节来表示这 8 个字节中哪些是可以访问的，这 1 个字节被称为 shadow memory，具体的表示方法此处不赘述。数组越界检查实现如图17.12所示，首先在定义数组时，KASAN 会为在数组 array 后生成不可访问的 redzone 区域（大小保持 32 字节对齐），如果步长为 1 的连续访问数组越界的话，一定会访问到 redzone 区域（当然如果步长太大，越界的数组操作访问到了另一块有效内存，那 KASAN 无法识别该异常）。其次，在读写数组的代码前加入检查代码，通过查看 shadow memory 的方式查看即将读写的内存是否是可访问的，如果不能访问则汇报错误。

另外，KASAN 也支持对动态分配的内存区域进行内存检查。访存信息格式以及访存前的检查和上述过程相似，区别主要在于在分配时标记整块区域为可访问，而释放内存时标记区域为不可访问，因此使用已经被释放的内存时会触发报警。

注意，由于 KASAN 插桩了大量内存检查的代码，内核在使用 KASAN 后会导致不可忽视的性能下降，另外由于存储了多余的代码和内存信息元数据，存储空间也有不少消耗，因此通常仅限于在测试环境下使用该技术，正常部署时应当保持关闭。

代码覆盖检测

为了尽可能对所有代码及其产生的行为进行测试，一个健壮的测试应当覆盖绝大部分分支路径，测试样例应该尽可能地触发边界条件，因此代码覆盖率是衡量测试的有效性的重要指标之一，它表示在测试过程中执行过的代码占总代码的比例为多少。在 Linux 用户态应用和内核中，可以使用 gcov 工具 [4] 生成进行代码覆盖率统计。它的基本原理是在代码编译至汇编后，在每个**基本代码段**（basic block）前加入统计该基本代码段执行次数的指令，统计结果暂存在内存中。需要读取覆盖率时依靠所有基本代码段执行次数统计信息，即可获取对应源代码各行被执行的次数。

在操作系统测试，特别是功能测试中，需要确保测试程序在执行过程中覆盖了大部分的代码路径，尽可能排除未测试到异常代码的情况。以LTP为例，LTP 内置的许多测试样例声明了该测试在某个内核版本的某些代码中覆盖率为多少。但是由于内核版本、配置等不同该值仅供参考使用，如果测试者非常关心测试样例的代码覆盖率如何，需要自行开启 gcov 功能并在测试后收集代码覆盖信息。

Linux 内核除了支持使用 gcov 统计代码覆盖情况外，为了支持后文所述的模糊测试，还引入了另外一种代码覆盖统计机制 kcov，详见第17.4.3小节。

17.4.3 其它测试方法

模糊测试

模糊测试（fuzzing）是一种自动化的软件测试方法，它将自动生成的数据输入到程序中，并监视程序的表现是否符合预期。决定模糊测试有效性关键之一在于如何生成交给测试程序的数据，大部分模糊测试工具使用两种方法：1. 按照特定方式**生成**（generation-based）：许多测试输入需要考虑语义信息，例如对操作系统系统调用生成随机输入时，应注意参数类型。例如有的参数为指针，有的为整型数字。2. 在原有数据基础上随机**变异**（mutation-based）：该方法收集已有的输入数据，在此基础上做较小的调整再次输入。例如对于系统调用，每次改变一个输入参数的值。但是使用此方法可能产生大量属于同一等价类的输入，即这些输入触发完全相同的代码路径，进行多次这样的测试意义不大。为了缓解这个问题，许多工具使用了**语料库**（corpus）的方法，语料库记录了产生不同代码路径的各种输入，具体来说，每次测试时从语料库中取出一个输入作为**种子**（seed），在它基础上变异产生多个输入后进行测试，如果新产生的输入触发的代码路径和种子产生的代码路径不同，则将新产生的输入

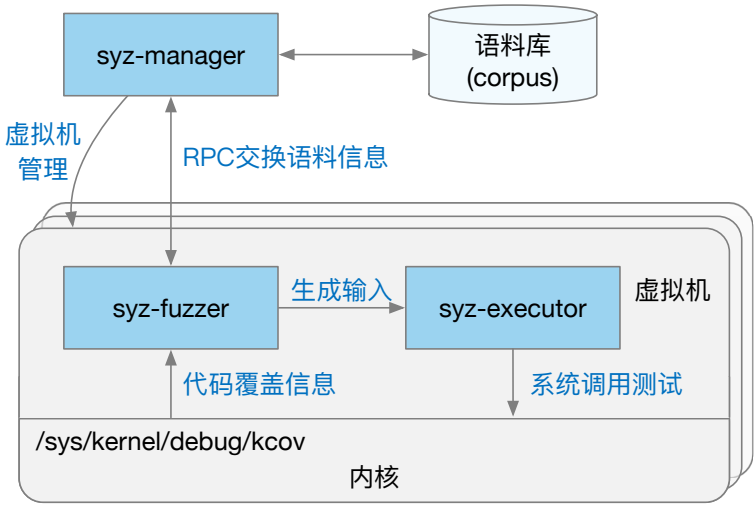

图 17.13: syzkaller 架构。

放入语料库中，等待之后在它基础上再次进行变异，反之如果代码路径相同则不做额外处理。这种启发式的方法主要考虑点在于：使用尽可能少的输入达到较大的代码覆盖范围，避免在类似的输入上做重复无用的测试。

模糊测试已经被广泛应用到操作系统中验证功能的正确性，Google 实现的 syzkaller [11] 是其中比较有代表性的工作，支持对 FreeBSD、Linux、Windows 等系统进行模糊测试。syzkaller 是 syscaller 的谐音，顾名思义，它从系统调用的层面上随机生成数据输入操作系统。如图17.13所示，syzkaller 主要包含三个部分，它们的功能和大致运行流程如下：

- syz-manager：负责维护全局的语料库，在测试过程中它会启动多个虚拟机进行测试，将语料库的种子传输到虚拟机内部的 syz-fuzzer 进行进一步处理。
- syz-fuzzer：它从 syz-manager 收集到输入种子后，在此基础上进行生成和变异，产生一连串的系统调用及其对应的参数，这些系统调用会交给 syz-executor 执行。在执行完毕后，syz-fuzzer 会通过内核提供的代码覆盖统计接口收集系统调用的代码覆盖情况，并和种子输入的代码覆盖情况对比，如果变异后的输入访问了新的代码分支则将该变异传输给 syz-manager，syz-manager 会将该变异加入语料库中。
- syz-executor：它是 syz-fuzzer 临时产生的用于执行系统调用的进程。

在 syzkaller 运行过程中至关重要的一点是如何收集系统调用产生的代码

覆盖路径，使用 gcov 方法（如章节17.4.2所述，即统计每个基本代码段执行次数）会面临如下问题：

- 一次典型的模糊测试过程是重复执行如下序列：1. 重置覆盖统计信息；2. 执行系统调用；3. 收集覆盖统计信息。在重置和收集 gcov 覆盖信息时需要修改或访问所有基本代码段的统计信息，但其中仅有跟此次系统调用相关的代码段是有用的，因此额外开销较高。

- 模糊测试希望仅仅收集当前测试的系统调用产生的代码覆盖信息，但是 gcov 会同时收集其他无关线程系统调用的覆盖情况，也会统计如中断处理、调度器等产生的较为随机的代码访问。

为了解决在模糊测试中代码覆盖统计遇到的问题，Linux 使用了 kcov 代码覆盖统计机制。kcov 的实现方式是在每一次调用跳转指令时，都先调用一个跟踪函数记录当前跳转的位置。因此 kcov 重置和收集覆盖信息时所访问的内存大小不取决于总的分支数量，而仅取决于实际运行到的分支数量，避免了访问所有基本代码段统计信息的开销。其次，kcov 记录的信息是每个线程独立的，并且只会记录系统调用相关函数内的代码覆盖情况，保证收集系统调用代码覆盖信息时不会受无关线程访问的影响。用户态可以使用 debugfs 重置和获取当前线程系统调用产生的代码覆盖信息。

安全测试

安全测试期望能够排除程序中的安全漏洞，通常会从程序的机密性、完整性、可用性不受攻击者影响的角度衡量系统的安全性是否符合规范。如第十六章所述，安全性在操作系统中是不可或缺的，因此安全测试在操作系统中也扮演着至关重要的角色。常见的安全测试通常基于既定规约从两个角度进行：一种是进行安全审计，检查操作系统的各类配置和安全策略是否符合约定。例如Security Content Automation Protocol (SCAP)协议约定了一系列安全配置、安全评估、漏洞披露等方面的标准。遵循该标准的安全审计软件（如 OpenSCAP [7]）使用该协议定义各类安全相关的配置，例如是否能使用 sysctl 指令操作某项内核参数、是否能够读取内核日志输出等。在进行安全审计时，测试软件会根据安全配置的清单依次检查当前系统是否符合规范，不符合则提出对应的建议以加强安全性。

另一种则是反向的渗透测试，此类测试会尝试攻击待测系统，观察系统行为是否违反既定安全规约。例如 Metasploit [6] 收集了上千种针对各种系统部

件的攻击方式，使用者可以尝试在各个系统上复现已有的攻击，观察是否有攻击成功并在安全性方面做相应改进。

17.5 案例分析：ChCore 内核测试流程

在本书配套的教学操作系统 ChCore 的实现过程中，我们使用了持续集成的方式完成了该操作系统的开发。开发人员周期性地将各自的工作合并进入主线中，在合并时必须通过自动化的构建、部署与测试，验证加入新功能后 ChCore 仍然正常工作，这种开发方式能够在保持软件快速迭代的同时保障代码的可靠性。

作为一个测试驱动的开发流程，在开发过程中不断运行的回归测试是确保错误能被及时发现的关键。在 ChCore 开发中，新加入的代码首先需要提交至远端仓库非主线分支，每次提交都会触发一次自动测试，该测试主要内容为：

- **构建测试和静态检查**：确保代码能够被正确编译，在编译流程中使用 fbinfer [2] 静态地检查是否存在使用未初始化变量、访问空指针等情况；
- **单元测试**：单独编译内存管理、调度器、内置数据结构等模块，在不需要运行完整内核的条件下对这些模块进行独立的测试，初步验证这些模块的功能正确性；
- **集成测试**：完整地构建并部署内核及用户态程序，验证操作系统能够正常启动。

这些由代码提交触发的测试较为轻量级，测试深度和广度较为有限，因此在代码被合入主线前，会触发一次更完整的测试：

- **单元测试**：与代码提交时触发的单元测试不同，本单元测试将会部署一个完整的内核，在内核启动完毕后开始对锁、调度器、内存分配器等进行测试，验证这些模块在实际的内核环境中仍能正常工作；
- **集成测试**：代码提交时触发的集成测试主要用于验证操作系统能够正常启动，而本集成测试将会运行大量用户态应用，包括 socket 测试、文件系统测试、同步原语测试等，确保内核和一些用户态操作系统服务正常运行；
- **性能测试**：为了确保新加入的代码没有对性能产生较大影响，会针对 IPC、系统调用等的性能进行测试，如果性能低于阈值将会触发预警；

- **兼容性测试**：ChCore 将会部署在 x86 架构和 AARCH64 架构上进行测试，并且在真实硬件和虚拟化平台上都进行测试。

另外，当 ChCore 开发中引入了新的功能时，开发者会及时地添加相应模块的单元测试和集成测试，确保该模块在后续迭代开发中的功能正确性。

17.6 思考题

1. perf 采样生成函数调用图时，默认采取的是 backtrace 的方式，即递归地在栈上找到上一级调用函数，但一种优化程序性能的方式是消除栈指针操作（如 gcc 中的–fomit-frame-pointer），使用此优化后不能够通过 backtrace 生成调用关系，你觉得本章节所述哪些硬件机制能够帮助你重构函数调用关系？

2. 在操作系统中使用断点调试时，有时会遇到动态装载的代码，即在设置断点地址后代码才被写入特定位置，你认为这种情况应该使用硬件断点还是软件断点，为什么？

3. perf events 为何使用共享内存传递采样信息给用户态应用，相比系统调用的接口有何优势？

4. 除了本章所述的通过系统调用、共享内存等方式获取内核信息外，Linux 中还有通过插入内核模块的方式直接获取内核信息的方法（如 SystemTap，sysdig），该方法相比与前者的优点和缺点分别是什么？

5. 如果为了验证 I/O 功能正确性，需要对其进行压力测试时，有必要增加其它资源（如处理器、内存）的负载吗？

6. 你认为在正确性测试中保持 KASAN 开启是否能让正确性测试更加可靠？

7. 在 fuzzing 测试中，syzkaller 通过构造系统调用向操作系统产生不同的输入，你认为这样的输入方式合理吗？有没有其它为操作系统产生输入的方法？

8. 本章哪些调测手段可以用于在 Linux 中追踪系统调用？

常见的系统调测工具

本章最后罗列一些常用的用于系统调试、性能监测、性能评估的工具，如表 17.1所示。

表 17.1: 常见的系统调测工具

种类	命令	用途
运行跟踪	strace	跟踪应用的系统调用和信号情况
	tcpdump	跟踪网络包相关的运行情况
	trace-cmd	使用 ftrace 跟踪内核运行
	perf	见第17.3.2章
	bpftrace、bcc	见第17.2.3章
	sysdig	综合性的跟踪工具，可以跟踪系统调用、I/O 访问等
性能监监	top、htop	监控系统中运行的各种进程占用的处理器、内存资源等
	vmstat、free	监控系统内存使用情况
	slabtop	监控内核动态内存分配情况
	iotop	监控各进程占用的文件系统读写带宽
	iostat	监控各硬盘设备当前的读写带宽
	iftop	监控与各远端机器的网络收发请求带宽
	netstat	监控网络连接及网络设备情况
	sar	综合性的系统资源监控
性能测试	netperf、iperf	网络相关的性能测试
	fio、filebench、dd	I/O 相关性能测试
	sysbench	综合性能测试，包含 I/O、CPU、内存以及 OLTP 测试等
	stress	对 I/O、CPU、内存相关的压力测试

参考文献

[1] Addresssanitizer algorithm. https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm. Accessed: 2020-04-05.

[2] fbinfer. https://github.com/facebook/infer. Accessed: 2020-04-26.

[3] Flamegraph. https://github.com/brendangregg/FlameGraph. Accessed: 2020-04-29.

[4] gcov. https://01.org/linuxgraphics/gfx-docs/drm/dev-tools/gcov.html. Accessed: 2020-06-03.

[5] kernelci. https://kernelci.org/. Accessed: 2020-06-03.

[6] metasploit. https://www.metasploit.com/. Accessed: 2020-06-30.

[7] Openscap. https://www.open-scap.org/. Accessed: 2020-08-13.

[8] Phoronix test suite. https://www.phoronix-test-suite.com/. Accessed: 2020-06-03.

[9] Posix test suite. http://posixtest.sourceforge.net/. Accessed: 2020-06-03.

[10] sparse. https://sparse.wiki.kernel.org/index.php/Main_Page. Accessed: 2020-06-03.

[11] syzkaller. https://github.com/google/syzkaller. Accessed: 2020-03-29.

[12] Why printk() is so complicated (and how to fix it). https://www.linuxplumbersconf.org/event/4/contributions/290/. Accessed: 2020-03-31.

操作系统调测：扫码反馈