第四章：陷阱和系统调用

有三种事件会导致CPU搁置普通指令的执行，强制将控制权转移给处理该事件的特殊代码。一种情况是***系统调用***，当用户程序执行**ecall**指令要求内核为其做某事时。另一种情况是***异常***：一条指令（用户或内核）做了一些非法的事情，如除以零或使用无效的虚拟地址。第三种情况是设备***中断***，当一个设备发出需要注意的信号时，例如当磁盘硬件完成一个读写请求时。

本书使用***trap***作为这些情况的通用术语。通常，代码在执行时发生trap，之后都会被恢复，而且不需要意识到发生了什么特殊的事情。也就是说，我们通常希望trap是透明的；这一点对于中断来说尤其重要，被中断的代码通常不会意识到会发生trap。通常的顺序是：trap迫使控制权转移到内核；内核保存寄存器和其他状态，以便恢复执行；内核执行适当的处理程序代码（例如，系统调用实现或设备驱动程序）；内核恢复保存的状态，并从trap中返回；代码从原来的地方恢复执行。

xv6内核会处理所有的trap。这对于系统调用来说是很自然的。这对中断来说也是合理的，因为隔离要求用户进程不能直接使用设备，而且只有内核才有设备处理所需的状态。这对异常处理来说也是合理的，因为xv6响应所有来自用户空间的异常，并杀死该违规程序。

Xv6 trap 处理分为四个阶段：RISC-V CPU采取的硬件行为，为内核C代码准备的汇编入口，处理trap的C 处理程序，以及系统调用或设备驱动服务。虽然三种trap类型之间的共性表明，内核可以用单一的代码入口处理所有的trap，但事实证明，为三种不同的情况，即来自用户空间的trap、来自内核空间的trap和定时器中断，设置单独的汇编入口和C trap处理程序会更方便的。

4.1 RISC-V trap machinery

每个RISC-V CPU都有一组控制寄存器，内核写入这些寄存器来告诉CPU如何处理trap，内核可以通过读取这些寄存器来发现已经发生的trap。RISC-V文档包含了完整的叙述[1]。riscv.h（kernel/riscv.h:1）包含了xv6使用的定义。这里是最重要的寄存器的概述。

- `stvec`：内核在这里写下trap处理程序的地址；RISC-V跳转到这里来处理trap。
- `sepc`：当trap发生时，RISC-V会将程序计数器保存在这里（因为`PC`会被`stvec`覆盖）。`sret`（从trap中返回）指令将`sepc`复制到`pc`中。内核可以写`sepc`来控制`sret`的返回到哪里。
- `scause`：RISC -V在这里放了一个数字，描述了trap的原因。
- `sscratch`：内核在这里放置了一个值，在trap处理程序开始时可以方便地使用。
- `sstatus`：`sstatus`中的**SIE**位控制设备中断是否被启用，如果内核清除**SIE**，RISC-V将推迟设备中断，直到内核设置**SIE**。**SPP**位表示trap是来自用户模式还是supervisor模式，并控制`sret`返回到什么模式。

上述寄存器与在特权态模式下处理的trap有关，在用户模式下不能读或写。对于机器模式下处理的trap，有一组等效的控制寄存器；xv6只在定时器中断的特殊情况下使用它们。

多核芯片上的每个CPU都有自己的一组这些寄存器，而且在任何时候都可能有多个CPU在处理一个trap。

当需要执行trap时，RISC-V硬件对所有的trap类型（除定时器中断外）进行以下操作：

1. 如果该trap是设备中断，且`sstatus` **SIE**位为0，则不执行以下任何操作。
2. 通过清除SIE来禁用中断。
3. 复制`pc`到`sepc`。
4. 将当前模式（用户态或特权态）保存在`sstatus`的**SPP**位。
5. 在`scause`设置该次trap的原因。
6. 将模式转换为特权态。

7. 将 `stvec` 复制到 `pc`。
8. 从新的 `pc` 开始执行。

注意，CPU不会切换到内核页表，不会切换到内核中的栈，也不会保存pc以外的任何寄存器。内核软件必须执行这些任务。CPU在trap期间做很少的工作的一个原因是为了给软件提供灵活性，例如，一些操作系统在某些情况下不需要页表切换，这可以提高性能。

你可能会想CPU的trap处理流程是否可以进一步简化。例如，假设CPU没有切换程序计数器（pc）。那么trap可以切换到监督者模式时，还在运行用户指令。这些用户指令可以打破用户空间/内核空间的隔离，例如通过修改 `satp` 寄存器指向一个允许访问所有物理内存的页表。因此，CPU必须切换到内核指定的指令地址，即 `stvec`。

4.2 Traps from user space

在用户空间执行时，如果用户程序进行了系统调用（`ecall` 指令），或者做了一些非法的事情，或者设备中断，都可能发生trap。来自用户空间的trap的处理路径是 `uservec`（kernel/trampoline.S:16），然后是 `usertrap`（kernel/trap.c:37）；返回时是 `usertrapret`（kernel/trap.c:90），然后是 `userret`（kernel/trampoline.S:16）。

来自用户代码的trap比来自内核的trap更具挑战性，因为 `satp` 指向的用户页表并不映射内核，而且栈指针可能包含一个无效甚至恶意的值。

因为RISC-V硬件在trap过程中不切换页表，所以用户页表必须包含 `uservec` 的映射，即 `stvec` 指向的trap处理程序地址。`uservec` 必须切换 `satp`，使其指向内核页表；为了在切换后继续执行指令，`uservec` 必须被映射到内核页表与用户页表相同的地址。

Xv6用一个包含 `uservec` 的trampoline页来满足这些条件。Xv6在内核页表和每个用户页表中的同一个虚拟地址上映射了trampoline页。这个虚拟地址就是 `TRAMPOLINE`（如我们在图2.3和图3.3中看到的）。`trampoline.S` 中包含trampoline的内容，（执行用户代码时）`stvec` 设置为 `uservec`（kernel/trampoline.S:16）。

当 `uservec` 启动时，所有32个寄存器都包含被中断的代码所拥有的值。但是 `uservec` 需要能够修改一些寄存器，以便设置 `satp` 和生成保存寄存器的地址。RISC-V通过 `sscratch` 寄存器提供了帮助。`uservec` 开始时的 `csrrw` 指令将 `a0` 和 `sscratch` 的内容互换。现在用户代码的 `a0` 被保存了；`uservec` 有一个寄存器（`a0`）可以使用；`a0` 包含了内核之前放在 `sscratch` 中的值。

`uservec` 的下一个任务是保存用户寄存器。在进入用户空间之前，内核先设置 `sscratch` 指向该进程的 `trapframe`，这个 `trapframe` 可以保存所有用户寄存器（kernel/proc.h:44）。因为 `satp` 仍然是指用户页表，所以 `uservec` 需要将 `trapframe` 映射到用户地址空间中。当创建每个进程时，xv6为进程的 `trapframe` 分配一页内存，并将它映射在用户虚拟地址 `TRAPFRAME`，也就是 `TRAMPOLINE` 的下面。进程的 `p->trapframe` 也指向 `trapframe`，不过是指向它的物理地址[1]，这样内核可以通过内核页表来使用它。

因此，在交换 `a0` 和 `sscratch` 后，`a0` 将指向当前进程的 `trapframe`。`uservec` 将在 `trapframe` 保存全部的寄存器，包括从 `sscratch` 读取的 `a0`。

`trapframe` 包含指向当前进程的内核栈、当前CPU的hartid、`usertrap` 的地址和内核页表的地址的指针，`uservec` 将这些值设置到相应的寄存器中，并将 `satp` 切换到内核页表和刷新TLB，然后调用 `usertrap`。

`usertrap` 的作用是确定trap的原因，处理它，然后返回（kernel/ trap.c:37）。如上所述，它首先改变 `stvec`，这样在内核中发生的trap将由 `kernelvec` 处理。它保存了 `sepc`（用户PC），这也是因为 `usertrap` 中可能会有一个进程切换，导致 `sepc` 被覆盖。如果trap是系统调用，`syscall` 会处理它；如果是设备中断，`devintr` 会处理；否则就是异常，内核会杀死故障进程。`usertrap` 会把用户 `pc` 加

4，因为RISC-V在执行系统调用时，会留下指向`ecall`指令的程序指针[2]。在退出时，`usertrap`检查进程是否已经被杀死或应该让出CPU（如果这个trap是一个定时器中断）。

回到用户空间的第一步是调用`usertrapret`（kernel/trap.c:90）。这个函数设置RISC-V控制寄存器，为以后用户空间trap做准备。这包括改变`stvec`来引用`uservec`，准备`uservec`所依赖的`trapframe`字段，并将`sepc`设置为先前保存的用户程序计数器。最后，`usertrapret`在用户页表和内核页表中映射的trampoline页上调用`userret`，因为`userret`中的汇编代码会切换页表。

`usertrapret`对`userret`的调用传递了参数`a0`，`a1`，`a0`指向`TRAPFRAME`，`a1`指向用户进程页表（kernel/trampoline.S:88），`userret`将`satp`切换到进程的用户页表。回想一下，用户页表同时映射了trampoline页和`TRAPFRAME`，但没有映射内核的其他内容。同样，事实上，在用户页表和内核页表中，trampoline页被映射在相同的虚拟地址上，这也是允许`uservec`在改变`satp`后继续执行的原因。`userret`将`trapframe`中保存的用户`a0`复制到`sscratch`中，为以后与`TRAPFRAME`交换做准备。从这时开始，`userret`能使用的数据只有寄存器内容和`trapframe`的内容。接下来`userret`从trapframe中恢复保存的用户寄存器，对`a0`和`sscratch`做最后的交换，恢复用户`a0`并保存`TRAPFRAME`，为下一次trap做准备，并使用`sret`返回用户空间。

4.3 Code: Calling system calls

第2章以`initcode.S`调用`exec`系统调用结束（user/initcode.S:11）。让我们来看看用户调用是如何在内核中实现`exec`系统调用的。

用户代码将`exec`的参数放在寄存器`a0`和`a1`中，并将系统调用号放在`a7`中。系统调用号与函数指针表`syscalls`数组（kernel/syscall.c:108）中的项匹配。`ecall`指令进入内核，执行`uservec`、`usertrap`，然后执行`syscall`，就像我们上面看到的那样。

`syscall`（kernel/syscall.c:133）从trapframe中的`a7`中得到系统调用号，并其作为索引在`syscalls`查找相应函数。对于第一个系统调用`exec`，`a7`将为`SYS_exec`（kernel/syscall.h:8），这会让`syscall`调用`exec`的实现函数`sys_exec`。

当系统调用函数返回时，`syscall`将其返回值记录在`p->trapframe->a0`中。用户空间的`exec()`将会返回该值，因为RISC-V上的C调用通常将返回值放在`a0`中。系统调用返回负数表示错误，0或正数表示成功。如果系统调用号无效，`syscall`会打印错误并返回-1。

4.4 Code: System call arguments

内核的系统调用实现需要找到用户代码传递的参数。因为用户代码调用系统调用的包装函数，参数首先会存放在寄存器中，这是C语言存放参数的惯例位置。内核trap代码将用户寄存器保存到当前进程的trap frame中，内核代码可以在那里找到它们。函数`argint`、`argaddr`和`argfd`从trap frame中以整数、指针或文件描述符的形式检索第n个系统调用参数。它们都调用`argraw`来获取保存的用户寄存器（kernel/syscall.c:35）。

一些系统调用传递指针作为参数，而内核必须使用这些指针来读取或写入用户内存。例如，`exec`系统调用会向内核传递一个指向用户空间中的字符串的指针数组。这些指针带来了两个挑战。首先，用户程序可能是错误的或恶意的，可能会传递给内核一个无效的指针或一个旨在欺骗内核访问内核内存而不是用户内存的指针。第二，xv6内核页表映射与用户页表映射不一样，所以内核不能使用普通指令从用户提供的地址加载或存储。

内核实现了安全地将数据复制到用户提供的地址或从用户提供的地址复制数据的函数。例如`fetchstr`（kernel/syscall.c:25）。文件系统调用，如`exec`，使用`fetchstr`从用户空间中检索字符串文件名参数。`fetchstr`调用`copyinstr`来做这些困难的工作。

`copyinstr`（kernel/vm.c:406）将用户页表`pagetable`中的虚拟地址`srcva`复制到`dst`，需指定最大复制字节数。它使用`walkaddr`（调用`walk`函数）在软件中模拟分页硬件的操作，以确定`srcva`的物理地址`pa0`。`walkaddr`（kernel/vm.c:95）检查用户提供的虚拟地址是否是进程用户地址空间的一部分，所以程序不能欺骗内核读取其他内存。类似的函数`copyout`，可以将数据从内核复制到用户提供的地址。

4.5 Traps from kernel space

Xv6根据用户还是内核代码正在执行，对CPU陷阱寄存器的配置略有不同行为。当内核在CPU上执行时，内核将`stvec`指向`kernelvec`上的汇编代码（kernel/kernelvec.S:10）。由于xv6已经在内核中，`kernelvec`可以使用`satp`，将其设置为内核页表，以及引用有效内核的堆栈指针。`kernelvec`保存所有寄存器，以便中断的代码最后可以在没有中断的情况下恢复。

`kernelvec`将寄存器保存在中断内核线程的堆栈上，因为寄存器值属于该线程，这是合理的。如果trap导致切换到另一个线程—在这种情况下，trap将实际返回到新线程的栈上，将中断线程保存的寄存器安全地保留在其堆栈上。

`kernelvec`在保存寄存器后跳转到`kerneltrap`（kernel/trap.c:134）。`kerneltrap`是为两种类型的陷阱准备的：设备中断和异常。它调用`devintr`（kernel/trap.c:177）来检查和处理前者。如果trap不是设备中断，那么它必须是异常，如果它发生在xv6内核中，则一定是一个致命错误；内核调用`panic`并停止执行。

如果由于计时器中断而调用了`kerneltrap`，并且进程的内核线程正在运行（而不是调度程序线程），`kerneltrap`调用`yield`让出CPU，允许其他线程运行。在某个时刻，其中一个线程将退出，并让我们的线程及其`kerneltrap`恢复。第7章解释了线程让出CPU控制权。

当`kerneltrap`的工作完成时，它需要返回到被中断的代码。因为`yield`可能破坏保存的`sepc`和在`sstatus`中保存的之前的模式。`kerneltrap`在启动时保存它们。它现在恢复那些控制寄存器并返回到`kernelvec`（kernel/kernelvec.S:48）。`kernelvec`从堆栈恢复保存的寄存器并执行`sret`，`sret`将`sepc`复制到`pc`并恢复中断的内核代码。

可以思考一下，如果因为时间中断，`kerneltrap`调用了`yield`，trap return是如何发生的。

当CPU从用户空间进入内核时，Xv6将CPU的`stvec`设置为`kernelvec`；可以在`usertrap`（kernel/trap.c:29）中看到这一点。内核运行但`stvec`被设置为`uservec`时，这期间有一个时间窗口，在这个窗口期，禁用设备中断是至关重要的。幸运的是，RISC-V总是在开始使用trap时禁用中断，xv6在设置`stvec`之前不会再次启用它们。

4.6 Page-fault exceptions

Xv6对异常的响应是相当固定：如果一个异常发生在用户空间，内核就会杀死故障进程。如果一个异常发生在内核中，内核就会**panic**。真正的操作系统通常会以更有趣的方式进行响应。

举个例子，许多内核使用页面故障来实现***写时复制（copy-on-write，cow）*fork**。要解释写时复制fork，可以想一想xv6的`fork`，在第3章中介绍过。`fork`通过调用`uvmcopy`（kernel/vm.c:309）为子进程分配物理内存，并将父进程的内存复制到子程序中，使子进程拥有与父进程相同的内存内容。如果子进程和父进程能够共享父进程的物理内存，效率会更高。然而，直接实现这种方法是行不通的，因为父进程和子进程对共享栈和堆的写入会中断彼此的执行。

通过使用写时复制fork，可以让父进程和子进程安全地共享物理内存，通过页面故障来实现。当CPU不能将虚拟地址翻译成物理地址时，CPU会产生一个页面故障异常（page-fault exception）。RISC-V有三种不同的页故障：load页故障（当加载指令不能翻译其虚拟地址时）、stote页故障（当存储指令不能翻译其虚拟地址时）和指令页故障（当指令的地址不能翻译时）。`scause`寄存器中的值表示页面故障的类型，`stval`寄存器中包含无法翻译的地址。

COW fork中的基本设计是父进程和子进程最初共享所有的物理页面，但将它们映射设置为只读。因此，当子进程或父进程执行store指令时，RISC-V CPU会引发一个页面故障异常。作为对这个异常的响应，内核会拷贝一份包含故障地址的页。然后将一个副本的读/写映射在子进程地址空间，另一个副本的读/写映射在父进程地址空间。更新页表后，内核在引起故障的指令处恢复故障处理。因为内核已经更新了相关的PTE，允许写入，所以现在故障指令将正常执行。

这个COW设计对`fork`很有效，因为往往子程序在fork后立即调用exec，用新的地址空间替换其地址空间。在这种常见的情况下，子程序只会遇到一些页面故障，而内核可以避免进行完整的复制。此外，COW fork是透明的：不需要对应用程序进行修改，应用程序就能受益。

页表和页故障的结合，将会有更多种有趣的可能性的应用。另一个被广泛使用的特性叫做***懒分配(lazy allocation)***，它有两个部分。首先，当一个应用程序调用`sbrk`时，内核会增长地址空间，但在页表中把新的地址标记为无效。第二，当这些新地址中的一个出现页面故障时，内核分配物理内存并将其映射到页表中。由于应用程序经常要求获得比他们需要的更多的内存，所以懒分配是一个胜利：内核只在应用程序实际使用时才分配内存。像COW fork一样，内核可以对应用程序透明地实现这个功能。

另一个被广泛使用的利用页面故障的功能是从***磁盘上分页(paging from disk)***。如果应用程序需要的内存超过了可用的物理RAM，内核可以交换出一些页：将它们写入一个存储设备，比如磁盘，并将其PTE标记为无效。如果一个应用程序读取或写入一个被换出到磁盘的页，CPU将遇到一个页面故障。内核就可以检查故障地址。如果该地址属于磁盘上的页面，内核就会分配一个物理内存的页面，从磁盘上读取页面到该内存，更新PTE为有效并引用该内存，然后恢复应用程序。为了给该页腾出空间，内核可能要交换另一个页。这个特性不需要对应用程序进行任何修改，如果应用程序具有引用的位置性（即它们在任何时候都只使用其内存的一个子集），这个特性就能很好地发挥作用。

其他结合分页和分页错误异常的功能包括自动扩展堆栈和内存映射文件。

4.7 Real world

如果将内核内存映射到每个进程的用户页表中（使用适当的PTE权限标志），就不需要特殊的trampoline页了。这也将消除从用户空间trap进入内核时对页表切换的需求。这也可以让内核中的系统调用实现利用当前进程的用户内存被映射的优势，让内核代码直接去间接引用（对地址取值）用户指针。许多操作系统已经使用这些想法来提高效率。Xv6没有实现这些想法，以减少由于无意使用用户指针而导致内核出现安全漏洞的机会，并减少一些复杂性，以确保用户和内核虚拟地址不重叠。

4.8 Exercises

1. 函数`copyin`和`copyinstr`在软件中walk用户页表。设置内核页表，使内核拥有用户程序的内存映射，`copyin`和`copyinstr`可以使用`memcpy`将系统调用参数复制到内核空间，依靠硬件来完成页表的walk。
2. 实现内存的懒分配。
3. 实现写时复制 fork。

---

1. 内核中物理地址和虚拟地址时直接映射的，所以可以在启用分页时，通过物理地址访问。
2. 执行系统调用时，进程的pc会指向ecall指令，这里需要加4清除，因为进程栈的地址空间是从高到低。