Chapter4 Traps and System Calls

CPU通常有三种特殊事件会暂停当前指令流的执行，并强制跳转到一段特定代码来处理这些事件。

- **系统调用**，当进程执行RISC-V的**ecall**指令时，让内核为它做一些事情。
- **异常**，用户或内核指令进行了一些非法操作，例如除以0或者使用无效的虚拟地址时。
- **设备中断**，即设备发出需要注意的信号，如当磁盘硬件完成读或写请求，需要CPU及时处理时。

本书将这些情况统称为`trap`。

通常情况下，在陷阱发生时执行的任何代码稍后都需要恢复，而且他们不需要知道陷阱发生过程中发生了什么特殊的事情。也就是说，我们通常希望陷阱是透明的，即保存和恢复相关寄存器、并返回发生陷阱的地址应该由内核负责，而不让用户操心，这对于中断尤其重要，可以让用户进程感受不到trap的发生，因为中断代码通常不期望中断。

处理陷阱的通常的流程是：

1. trap强制从用户空间切换到内核空间;
2. 在内核空间下，内核保存寄存器和其他状态，以便恢复执行被暂停的进程指令流;
3. 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);
4. 内核恢复保存的状态并从trap中返回;
5. 原来的代码会在停止的地方继续。

xv6内核来处理所有的trap。系统调用显然应该如此。对于设备中断也说得通，因为隔离要求用户进程不直接使用设备，而且只有内核拥有设备处理所需的状态信息。对于异常，xv6的响应很简单，对于所有来自用户空间的异常，直接kill该进程。

Xv6陷阱处理分四个阶段进行:由RISC-V CPU执行的hardware actions、为内核C代码做准备的assembly "vector"、决定如何处理陷阱的C trap handler、system call 或device-driver service routine。虽然这三种trap类型的共性表明内核可以用single code path处理所有的trap，但对于三种不同的情况:来自用户空间的trap、来自内核空间的trap、计时器中断，使用独立的 assembly vectors 和 C trap handler 会更方便。

以用户空间下发生trap为例，trap所经历的完整流程，从代码路径来看就是：`uservec`->`usertrap`->`usertrapret`->`userret`

- `uservec`：位于`trampoline`的前半部分汇编代码，用于做一些陷入内核之前的准备工作，例如保存用户空间下的一系列寄存器，加载内核栈、内核页表等设置，然后跳转到`usertrap`。这一部分，我们称为**trap vector**。
- `usertrap`：位于内核中的一段C代码，判断引起trap的事件类型，并决定如何处理该trap，如跳转到系统调用函数、设备驱动程序等。我们一般也称其为**trap handler**。
- `usertrapret`：位于内核中的另一段C代码，trap被处理完之后，就会跳转到`usertrapret`，保存内核栈、内核页表等内核相关信息，进行一些设置，然后跳转到`userret`。
- `userret`：位于trampoline的后半部分汇编代码，用于做一些返回用户空间的恢复工作，恢复之前保存的用户空间寄存器，最后返回用户空间，恢复用户进程指令流的执行。

这里值得注意的一个点是，为什么没有将`uservec`和`usertrap`两个代码路径合并在一起。按照我们的解决方案，用两段代码分别负责**保存**和**处理**的工作，好处在于，现在我们可以区分以下三种不同的情况：来自**用户空间的trap**，来自**内核空间的trap**和**计时器中断**。

RISC-V trap machinery

每个RISC-V CPU都有一些**控制寄存器**的集合，内核通过写入这些寄存器，告诉CPU如何处理这次trap；内核通过读入这些寄存器，能够发现一次trap的发生。

这里介绍一些最重要的控制寄存器：

- **stvec**：内核将trap handler（在用户空间下的trap，是trampoline的**uservec**；在内核空间下的trap，是kernel/kernelvec.S中的**kernelvec**）的地址写到stvec中。当trap发生时，RISC-V就会跳转到stvec中的地址，准备处理trap。你可能发现，被写入stvec的是uservec或者kernelvec，而不是usertrap，其实我们也可以将两个trap vector认为是trap handler的准备工作部分。
- **sepc**：当trap发生时，RISC-V就**将当前pc的值保存在sepc中**（例如，指令ecall就会做这个工作），因为稍后RISC-V将使用stvec中的值来覆盖pc，从而开始执行trap handler。稍后在userret中，sret指令将sepc的值复制到pc中，内核可以设置sepc来控制sret返回到哪里。
- **scause**：RISC-V在这里存放一个数字，代表引发trap的原因。
- **sscratch**：一个特别的寄存器，通常在用户空间发生trap，并进入到uservec之后，sscratch就装载着指向进程**trapframe**的指针（该进程的trapframe，在进程被创建，并从userret返回的时候，就已经被内核设置好并且放置到sscratch中）。RISC-V还提供了一条**交换指令**（csrrw），可以将任意寄存器与sscratch进行值的交换。sscratch的这些特性，便于在uservec中进行一些寄存器的保存、恢复工作。
- **sstatus**：该寄存器中的**SIE**位，控制设备中断是否开启，如果SIE被清0，RISC-V会推迟期间的设备中断，直到SIE被再次置位；**SPP**位指示一个trap是来自用户模式下还是内核模式下的，因此也决定了sret要返回到哪个模式下。

以上这些控制寄存器不能在用户模式下访问，只有在内核中，在内核模式下处理trap时，才能访问或设置这些寄存器。值得一提的是，机器模式下也有完全对等的一系列控制寄存器，不同的是这些寄存器以m开头命名，而且只有在机器模式下处理trap时，才能用到它们。在xv6中，只有计时器中断这一特殊情况会用到机器模式下的控制寄存器。

在多核芯片上，每个CPU都有各自的上述控制寄存器集合，因此在任一时刻，可能有多个CPU同时都在处理trap。

当trap确实发生了，**RISC-V CPU硬件**按以下步骤处理所有类型的trap（除了计时器中断）：

1. 如果造成trap的是设备中断，将sstatus中的SIE位清0，然后跳过以下步骤。
2. （如果trap的原因不是设备中断）将sstatus中的SIE位清0，关闭设备中断。
3. 将pc的值复制到sepc中。
4. 将发生trap的当前模式（用户模式或内核模式）写入sstatus中的SPP位。
5. 设置scause的内容，反映trap的起因。
6. 设置模式为内核模式。
7. 将stvec的值复制到pc中。
8. 从新的pc值开始执行。

值得注意的是，当trap发生了之后，硬件做的事情实际上很少，CPU没有切换到内核页表，没有切换到内核栈，也没有保存任何的寄存器（除了PC）。因此，内核中的代码必须完成以上这些工作。让CPU完成尽可能少的工作，其中一个原因是为内核代码提供灵活性。例如，一些操作系统在trap发生之后可能并不需要切换页表（在它们的设计里，用户空间和内核空间使用同一个页表，按照例如用户空间在低地址，内核空间在高地址等方式设计），如果我们的内核可以自己选择不切换页表，就省去了相关操作，从而提升了性能。

你可能会想，trap发生时CPU硬件的工作流程还能再简化一些吗？例如，一些CPU可能不改变pc的值，trap可以直接切换到内核模式，然而依然运行用户指令。但是，这又打破了我们前几章一直在强调的**隔离**机制，这次是打破了用户和内核之间的隔离。问题就在于，你现在能够用内核模式来运行用户指令，这肯定是不好的。例如你现在可以更改satp寄存器的值，使用内核的页表，然后你就能访问所有的物理

内存。因此有些工作，出于隔离性和安全性考虑，我们不再做简化，CPU确实应该从用户指令流中脱离出来，然后执行一段位于内核中的代码，从而保证我们执行的是内核指令，我们可以通过stvec中的值来完成pc值的切换。

Traps From User Space

如果用户程序执行了系统调用(ecall指令)，或者做了非法的事情，或者设备中断，则在用户空间中执行时可能会发生trap。

用户空间发生trap的路径为：`uservec`(kernel/trampoline.S:16), `usertrap`(kernel/trap.c:37)，当返回时，有`usertrapret`(kernel/trap.c:90), `userret`(kernel/trampoline.S:16).

来自用户代码的陷阱比来自内核的陷阱更具挑战性，因为`satp`指向的用户页表不映射内核，而且堆栈指针可能包含无效甚至恶意的值。

uservec

因为**在trap过程中，RISC-V硬件不会切换页表**，因此，用户页表必须包含`uservec`的映射，即`stvec`所指向的trap vector指令。`uservec`必须切换`satp`来指向内核页表；并且为了在切换到内核页表后继续执行指令，必须**将`uservec`映射到内核页表中与用户页表中相同的虚拟地址**。

xv6通过包含`uservec`的`trampoline`页面来实现以上的约束。Xv6将`trampoline`页映射到内核页表和每个用户页表中相同的虚拟地址。虚拟地址是TRAMPOLINE。`trampoline`内容设置在`trampoline.S`中，当执行用户代码的时候，`stvec`被设置为`uservec`(kernel/trampoline.S:16)。

当`uservec`启动时，32个寄存器存着被中断代码的值。但是`uservec`需要能够修改一些寄存器，以便设置`satp`并生成保存寄存器的地址。RISC-V以`sscratch`寄存器的形式提供了帮助。

`uservec`的下一个任务是**保存用户寄存器**。

在进入用户空间之前，内核将`sscratch`设置为指向当前进程的`trapframe`的指针，`trapframe`提供空间保存所有用户空间下的寄存器(kernel/proc.h:44)。因为现在`satp`仍指向了用户页表，而我们在用户虚拟地址空间中需要这项映射，所以`uservec`就要求将`trapframe`映射到用户地址空间中（在创建每个进程时，xv6为进程的`trapframe`分配一个页面，并安排它始终映射到用户虚拟地址TRAPFRAME，就在`trampoline`下一页；同时，进程的`p->trapframe`也指向`trapframe`，但指向的是物理地址，因为内核页表中free memory是直接映射，因此内核也可以通过直接访问`p->trapframe`在内核页表使用`trapframe`）。

`uservec`开头的`csrrw`指令交换`a0`和`sscratch`的内容，此时`sscratch`的内容是被中断进程的`trapframe`。现在用户代码的`a0`被保存在`sscratch`中; 而`a0`包含内核先前放在`sscratch`中的值。因此，在交换`a0`和`sscratch`之后，`a0`持有一个指向当前进程的`trapframe`的指针。`uservec`现在将所有用户寄存器保存在那里（包括从`sscratch`读取的用户的`a0`）。

```
uservec:
        #
        # trap.c sets stvec to point here, so
        # traps from user space start here,
        # in supervisor mode, but with a
        # user page table.
        #
        # sscratch points to where the process's p->trapframe is
        # mapped into user space, at TRAPFRAME.
        #
```

```
    # swap a0 and sscratch
        # so that a0 is TRAPFRAME
        csrrw a0, sscratch, a0
```

以下代码紧接着上面的`uservec`代码，用来保存用户寄存器。

```
        # sd a1, a2  --  将a1中的值存入a2
        # 此时a0的值为当前进程trapframe地址
        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
        sd s4, 192(a0)
        sd s5, 200(a0)
        sd s6, 208(a0)
        sd s7, 216(a0)
        sd s8, 224(a0)
        sd s9, 232(a0)
        sd s10, 240(a0)
        sd s11, 248(a0)
        sd t3, 256(a0)
        sd t4, 264(a0)
        sd t5, 272(a0)
        sd t6, 280(a0)

    # save the user a0 in p->trapframe->a0
    # 现在sscratch中保存a0的值，将a0的值存入t0寄存器
        csrr t0, sscratch
    # 将t0(即a0)也存入trapframe
        sd t0, 112(a0)
```

到了这里，所有用户寄存器都保存完成。

`trapframe`里面还保存了一些与内核信息相关的指针，现在将它们恢复出来，`trapframe`中包含内核栈指针、当前CPU的`hartid`、`usertrap`的地址和内核页表指针。`uservec`获取这些值，**将`satp`切换到内核页表**，并调用`usertrap`。值得注意的是，在我们加载内核页表到`satp`之后，`a0`就无效了，因为`trapframe`只映射在用户虚拟地址空间内，所以我们之前就把`usertrap`的位置加载到`t0`，并在最后跳转到`t0`。

```
        # ld a1, a2 -- 将a2中的值加载进a1
        # 下面将trapframe中内核信息相关的信息加载入寄存器
        # restore kernel stack pointer from p->trapframe->kernel_sp
        # 恢复内核栈sp
        ld sp, 8(a0)

        # make tp hold the current hartid, from p->trapframe->kernel_hartid
        # 恢复hartid
        ld tp, 32(a0)

        # load the address of usertrap(), p->trapframe->kernel_trap
        # 先将usertrap的地址放入t0
        ld t0, 16(a0)

        # restore kernel page table from p->trapframe->kernel_satp
        # 将用户的内核页表跟页表物理地址(satp)先存入t1
        ld t1, 0(a0)
        csrw satp, t1 # 将t1值加载入satp寄存器, 从这里开始使用进程的内核页表!!!!!
        sfence.vma zero, zero

        # a0 is no longer valid, since the kernel page
        # table does not specially map p->tf.
        # 从这里开始, a0就无效了, 因为trapframe只映射在用户进程页表中, 现在正在使用内核页表
        # 所以要提前将usertrap存起来

        # jump to usertrap(), which does not return
        jr t0
```

之后，就跳转到内核的trap handler——`usertrap`下。

usertrap

`usertrap`的任务是确定trap的原因、处理它并返回(代码在kernel/trap.c:37)。

它首先更改`stvec`寄存器的值。之前`stvec`的值是`uservec`，代表在用户空间下发生trap时，就跳转到`uservec`；而现在我们是在内核空间下，所以修改为`kernelvec`，专门处理内核空间下的trap。

接着，保存`sepc`寄存器(被保存的用户程序计数器，例如在ecall的时候会自动保存`pc`到`sepc`)。硬件保存当前的用户`pc`值到`sepc`中。保存`sepc`是因为，即使在`usertrap`下，也可能发生进程上下文切换，可能使`sepc`被覆写。

接着，根据trap的类型，如果是系统调用，调用syscall处理；如果是设备中断，调用devintr处理；如果是其它的异常，直接杀掉用户进程。值得注意的是，对于系统调用的情况，我们将保存起来的pc值加4，因为在RISC-V硬件因系统调用而保存pc值的时候，保存的是ecall的位置。对该pc值加4，返回用户空间后，会执行系统调用的下一条指令。

在`usertrap`的最后，即处理完这些trap分支之后，进行一些额外的检查。如果内核认为该进程应该被杀掉，那么就终止它；如果设备中断的类型是计时器中断，即时间片到期，那么就调用yield让出CPU。而至于其它的情况，则跳转到`usertrapret`。

```
void
usertrap(void)
{
  int which_dev = 0;
```

```
  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  // 将stvet寄存器的指赋为kernelvec
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  // 在进入trap处理之前，RISC-V硬件将pc的值复制到sepc中
  // 在这里将sepc的值保存到p->trapframe->epc中，防止有其他进程切换进来，导致sepc丢失
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    // 对于系统调用来说，返回后执行的命令是下一条命令
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
     // 系统调用将返回值记录在p->trapframe->a0中
     // 一般返回值为负数，表明出现错误；返回值为0或者正数，表明运行成功
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    // scause寄存器的值表示page fault的类型
    // stval寄存器包含不能被翻译的地址
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

接下来的`usertrapret`(代码在kernel/trap.c:90)是**从内核空间返回到用户空间的第一步**。无论是内核处理完trap之后，还是新进程被创建之后，又或是另一进程被调度执行之后，第一步都要经过`usertrapret`。`usertrapret`设置一系列的RISC-V控制寄存器，以便在用户空间下发生的下一次trap依然能按照我们常规的代码路径来处理。

usertrapret

首先，重新设置`stvec`的值，使其指向`uservec`。这里特别重要的一点是，在此之前要先**关闭中断**。因为我们设置了使`stvec`指向`uservec`，但我们当前仍在内核空间下执行，那么内核空间下发生的trap，就会从`uservec`开始处理，这显然是不正确的。所以在一切都准备就绪之前，在我们真正能处理用户空间的trap之前，亦即在我们正式返回用户空间之前（通过`sret`），我们应该保持设备中断关闭，不能让trap发生在这个过渡期期间，打扰内核的正常执行。

接着，我们之前在`uservec`中恢复出了一些内核相关的信息，现在我们即将要返回用户空间，要重新把它们保存回去，一遍用户进程下一次可以进入内核。包括内核页表、内核栈、`usertrap`的位置、CPUid这些，都保存到用户进程的`trapframe`中。

然后，完成`sstatus`寄存器的设置。我们前面提过，SPP位控制`sret`返回的模式，清空SPP位所以下次调用`sret`返回时，返回的是用户模式；SPIE位则控制中断的开关，在我们通过`sret`成功返回用户空间之后，我们希望之后中断能够产生，因此置位SPIE位。

然后，将此前保存在`trapframe`中的，用户空间下trap发生时的`pc`值加载到`sepc`中，在`sret`返回时，会将`sepc`的值复制到`pc`中。

最后，我们准备好用户页表，然后调用`userret`，完成最后的工作，并返回到用户空间中。值得注意的是，这里通过函数指针调用`userret`，并且将`trapframe`的虚拟地址作为参数`a0`，用户页表作为参数`a1`传入。

```
//
// return to user space
//
void
usertrapret(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(), so turn off interrupts until
  // we're back in user space, where usertrap() is correct.
  // 关闭中断
  intr_off();

  // send syscalls, interrupts, and exceptions to trampoline.S
  // 改变寄存器stvec的值，指向TRAMPOLINE中的uservec
  w_stvec(TRAMPOLINE + (uservec - trampoline));

  // set up trapframe values that uservec will need when
  // the process next re-enters the kernel.
  // 将那些内核相关的信息保存回trapframe
  p->trapframe->kernel_satp = r_satp();         // kernel page table
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel stack
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid()
```

```
  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x); // 设置sstatus寄存器,清空SPP以返回用户模式,并且设置SPIE位以使可以在用户模式中有中断

  // set S Exception Program Counter to the saved user pc.
  // 将发生trap时用户空间下的pc加载到sepc中
  // 在sret返回时, 会将sepc的值赋值到pc中
  // 用户程序被创建时, 也返回到这里, 该用户的epc在exec()里已被初始化
  // 这样用户程序被创建并且返回后, sret会将sepc即epc的内容复制到pc中, 然后从pc开始继续执行
  w_sepc(p->trapframe->epc);

  // tell trampoline.S the user page table to switch to.
  // 刚刚在上面已经存了内核页表的跟页表地址在trapframe里面
  // 这里的satp是用户页表
  uint64 satp = MAKE_SATP(p->pagetable);

  // jump to trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 fn = TRAMPOLINE + (userret - trampoline);
  // 利用地址调用userret, 参数a0为TRAPFRAME的虚拟地址, 参数a1为用户页表的satp
  ((void (*)(uint64,uint64))fn)(TRAPFRAME, satp);
}
```

下一步就是进到`userret`中

userret

首先，`userret`马上就更换成用户页表。这里我们将看到和`uservec`一样的原理，`userret`也放置在`trampoline`中，在用户和内核虚拟地址空间下映射到相同的位置。因此即使我们用的是用户页表，执行这段内核代码也不会出错。

注意，这里的`a0`、`a1`就是指传入的参数了，`a0`指的是trapframe地址，`a1`指的是pagetable地址。

```
userret:
        # userret(TRAPFRAME, pagetable)
        # switch from kernel to user.
        # usertrapret() calls here.
        # a0: TRAPFRAME, in user page table.
        # a1: user page table, for satp.

        # switch to the user page table.
        csrw satp, a1
        sfence.vma zero, zero
```

接着，取出此前保存在`trapframe`的用户`a0`值（如果trap是系统调用造成的，那么内核将使用系统调用的返回值，因为系统调用的返回值存在`p->trapframe->a0`，也就是新的`a0`替换旧的`a0`），通过间接的寄存器`t0`，将新的用户`a0`值写入到`sscratch`寄存器中。注意， 这里并不是交换，`a0`仍然是`trapframe`。

```
        # put the saved user a0 in sscratch, so we
        # can swap it with our a0 (TRAPFRAME) in the last step.
        ld t0, 112(a0) # 将p->trapframe->a0读入t0
        csrw sscratch, t0 # 将p->trapframe->a0写入sscratch寄存器中
```

然后，恢复所有此前在`uservec`中保存的寄存器。

```
        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
        ld s5, 200(a0)
        ld s6, 208(a0)
        ld s7, 216(a0)
        ld s8, 224(a0)
        ld s9, 232(a0)
        ld s10, 240(a0)
        ld s11, 248(a0)
        ld t3, 256(a0)
        ld t4, 264(a0)
        ld t5, 272(a0)
        ld t6, 280(a0)
```

然后，再次使用RISC-V的`csrrw`指令，交换`a0`和`sscratch`，所以交换之后，`a0`寄存器里面是我们要返回的用户`a0`值，而`sscratch`中则是`trapframe`。

```
    # restore user a0, and save TRAPFRAME in sscratch
    csrrw a0, sscratch, a0 # 交换a0和sscratch
```

最后一步，我们调用`sret`结束整个漫长的trap过程。这里`sret`完成三个工作：复制`sepc`中的值到`pc`中，返回原指令流执行；根据`sstatus`寄存器的设置，切换到用户模式；并重新开放中断。`sret`完成的工作，与trap发生时硬件做的工作有对应的部分，以系统调用为例，RISC-V的`ecall`指令也完成三个工作：将`pc`的值保存到`sepc`中，从用户模式切换到监管者模式，跳转到`stvec`中指向的指令。

```
        # return to user mode and user pc.
        # usertrapret() set up sstatus and sepc.
        sret
```

这之后，被暂停的用户指令流将重新执行，我们讨论的整个trap流程对于进程来说是透明的。

Code: Calling system calls

在第二章中，我们介绍了`initcode.S`在结束的时候使用系统调用**exec**，加载用户进程的虚拟地址空间。现在，我们将从系统调用的执行路径来重新看一遍。

首先，在用户代码执行exec，请求系统调用函数exec时，用户参数按顺序分别被放置到寄存器`a0`和`a1`，而系统调用号被放置在`a7`，然后执行ecall指令，一个用户空间下的trap正式发生。

下图是`user/usys.S`中代码片段，展示了当调用exec时的动作。

```
.global exec
exec:
 li a7, SYS_exec
 ecall
 ret
```

调用ecall后，经过`uservec`的准备工作后，进入`usertrap`开始处理，发现trap的起因是系统调用，于是执行了`sysvall`函数。`syscall`函数根据传入的系统调用号(`p->trapframe->a7`)，索引syscalls系统调用函数数组，找到对应的系统调用，通过函数指针执行相应的系统调用。以下代码是`kernel/syscall.c`中的：

```
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
```

```
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};

void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

在`initcode.S`这个例子中，执行了系统调用实现函数`sys_exec`。结束之后，返回值存到`p->trapframe->a0`，稍后在`userret`中会取出，然后作为最终返回用户的`a0`值。非负值表示调用成功，负值表示调用失败。

Code: System Call Arguments

除了系统调用号，系统调用实现函数还需要知道用户代码传入的其他参数。

trap在发生时，`uservec`将用户寄存器保存到`trapframe`中了，内核在`trapframe`中可以找到它们。内核使用`argint`，`argaddr`，`argfd`函数，来获取出用户参数值， 并把它们作为整数、指针或文件描述符。他们调用`argraw`(kernel/syscall:35)这个函数来获取对应的用户寄存器。

```
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n) {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
```

```
}
```

一些系统调用会传递指针作为用户参数，内核必须使用这些指针，读或写这些属于用户进程的物理内存。例如，`exec`就给内核传递了一个`argv`指针，指向一系列的命令行参数。

使用这些用户指针有两个挑战。一是用户进程可能是有漏洞或者有恶意的，因此它会传递一个无效的指针，甚至是一个企图访问属于其它用户进程内容、或者属于内核内容的指针。二是内核页表和用户页表的不同所造成的，因为各自的映射不同，在使用内核页表时，不能用简单的指令访问这些用户地址。

幸运的是，内核实现了专门的函数，用于安全地从用户地址中复制内容到内核缓冲区中。

例如`fetchstr`(kernel/syscall.c)这个例子，众多系统调用，如exec，就是用它来复制位于用户空间的参数。`fetchstr`将主要的工作交给`copyinstr`来完成。

```
// Fetch the nul-terminated string at addr from the current process.
// Returns length of string, not including nul, or -1 for error.
int
fetchstr(uint64 addr, char *buf, int max)
{
  struct proc *p = myproc();
  int err = copyinstr(p->pagetable, buf, addr, max);
  if(err < 0)
    return err;
  return strlen(buf);
}
```

`copyinstr`(kernel/vm.c)的作用是，给定一个用户页表，从用户虚拟地址`srcva`（例如用户缓冲区），安全地拷贝最多max个字节到内核的`dst`位置中。首先`copyinstr`调用`walkaddr`来为`srcva`找到对应的物理地址`pa0`，利用内核采用**直接映射**的特性，我们将`pa0`作为虚拟地址，便可以直接地从`pa0`中拷贝字节流到`dst`中。

```
// Copy a null-terminated string from user to kernel.
// Copy bytes to dst from virtual address srcva in a given page table,
// until a '\0', or max.
// Return 0 on success, -1 on error.
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  uint64 n, va0, pa0;
  int got_null = 0;

  while(got_null == 0 && max > 0){
    va0 = PGROUNDDOWN(srcva);
    // walkaddr 可以检查用户提供的虚拟地址是不是进程的，因此应用程序不能欺骗内核访问其他内存
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (srcva - va0);
    if(n > max)
      n = max;
    // 因为kernel将物理RAM地址恒等映射到内核虚拟地址上，所以copyinstr可以直接从a0到dst拷贝字符串
    char *p = (char *) (pa0 + (srcva - va0));
```

```
    while(n > 0){
      if(*p == '\0'){
        *dst = '\0';
        got_null = 1;
        break;
      } else {
        *dst = *p;
      }
      --n;
      --max;
      p++;
      dst++;
    }

    srcva = va0 + PGSIZE;
  }
  if(got_null){
    return 0;
  } else {
    return -1;
  }
}
```

现在，用户将无法传递一个尝试访问内核的物理内存的指针，因为`walkaddr`会检查该地址是否在用户虚拟地址空间中。因此，一个尝试访问内核物理内存的非法访问，显然不会出现在用户页表的映射项里，`walkaddr`会返回失败，从而不让恶意的用户进程得逞。同理，尝试访问其它用户进程的物理内存的指针也不会通过`walkaddr`的严格检查。

以上是从用户空间（如用户缓冲区）复制数据到内核中，从内核中复制数据到用户空间下也遵循相似的原则，内核提供了`copyout`来完成这个任务。同样地，目的地址必须是用户进程自己拥有的物理内存页，不然`walkaddr`也会阻止这次复制。

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.(user
pagetable)
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
```

```
  }
  return 0;
}
```

Trap from kernel space

上面的讨论都是发生在用户空间下的trap，下面来讨论发生在内核空间下的trap。

对于这两种模式下的trap，xv6配置相关寄存器的方式也略有不同。当CPU在内核空间下执行内核代码的时候，`stvec`被设置为指向`kernelvec`（kernel/kernelvec.S），用于处理内核空间下的trap。

kernelvec

由于此时已经是在内核空间下，就算trap发生，也不需要修改`satp`和`sp`，内核页表和内核栈都可以继续使用的。因此`kernelvec`的工作很简单，当前正在运行的内核线程因trap而被暂停，直接在该**内核线程的内核栈上**，保存相关的寄存器即可。保存在线程自己的内核栈上这一点很重要，因为内核下的trap可能会导致线程的切换，然后内核会在另一个线程的内核栈上执行，而原线程的相关寄存器则被安全地放置在自己的内核栈上。

```
kernelvec:
        // make room to save registers.
        addi sp, sp, -256

        // save the registers.
        sd ra, 0(sp)
        sd sp, 8(sp)
        sd gp, 16(sp)
        sd tp, 24(sp)
        sd t0, 32(sp)
        sd t1, 40(sp)
        sd t2, 48(sp)
        sd s0, 56(sp)
        sd s1, 64(sp)
        sd a0, 72(sp)
        sd a1, 80(sp)
        sd a2, 88(sp)
        sd a3, 96(sp)
        sd a4, 104(sp)
        sd a5, 112(sp)
        sd a6, 120(sp)
        sd a7, 128(sp)
        sd s2, 136(sp)
        sd s3, 144(sp)
        sd s4, 152(sp)
        sd s5, 160(sp)
        sd s6, 168(sp)
        sd s7, 176(sp)
        sd s8, 184(sp)
        sd s9, 192(sp)
        sd s10, 200(sp)
        sd s11, 208(sp)
        sd t3, 216(sp)
        sd t4, 224(sp)
```

```
        sd t5, 232(sp)
        sd t6, 240(sp)

    // call the C trap handler in trap.c
        call kerneltrap
```

内核空间下的trap发生时，当前线程在自己的栈上保存了相关的寄存器之后，内核从`kernelvec`跳转到`kerneltrap`（kernel/trap.c）中。

kerneltrap

`kerneltrap`处理两种trap，**设备中断和异常**。内核调用`devintr`检查设备中断类型，并处理它。如果发现不是设备中断，就一定是异常。这个异常在xv6内核中发生，总是一个致命的错误，这会直接导致内核停止执行。

如果是计时器中断，而且进程的内核线程正在running(而不是scheduler thread)，这就表明当前线程的时间片用完了，应该切换让别的内核线程运行，`kerneltrap`调用`yield`来放弃CPU，在某一时刻，其他的线程可能调用`yield`给其他线程运行的机会。

你可能会担心，我们的`kerneltrap`还没有正式执行完，不用担心，我们已经事先保存过了其它线程可能会修改的寄存器，而且这个被挂起的线程，稍后就会被重新唤醒到上次的位置继续执行。

```
// interrupts and exceptions from kernel code go here via kernelvec,
// on whatever the current kernel stack is.
void
kerneltrap()
{
  int which_dev = 0;
  // 在开始trap之前先将sepc、sstatus
  // 因为时间片中断导致的yield可能会摧毁sepc和sstatus
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if((which_dev = devintr()) == 0){
    printf("scause %p\n", scause);
    printf("sepc=%p stval=%p\n", r_sepc(), r_stval());
    panic("kerneltrap");
  }

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING)
    yield();

  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instruction.
  // 因为yield() 可能会覆盖这些寄存器，所以恢复它们
  w_sepc(sepc);
  w_sstatus(sstatus);
}
```

`kerneltrap`的主要工作执行完了之后，现在正在执行的内核线程，可能和之前是同一个，也可能是被调度运行的最新一个，无论如何，我们重新加载`sepc`和`sstatus`寄存器中的值，确保不会出现错误。现在`kerneltrap`正式结束，由于没有别的函数调用（注意，没有kerneltrapret），它会返回`kernelvec`中。

kernelvec

从当前内核线程的内核栈上，恢复所有保存过的寄存器，并执行`sret`，将`sepc`复制到`pc`上并恢复被中断的内核代码。

下面的代码紧接着之前`kernelvec`的代码

```
        // restore registers.
        ld ra, 0(sp)
        ld sp, 8(sp)
        ld gp, 16(sp)
        // not this, in case we moved CPUs: ld tp, 24(sp)
        ld t0, 32(sp)
        ld t1, 40(sp)
        ld t2, 48(sp)
        ld s0, 56(sp)
        ld s1, 64(sp)
        ld a0, 72(sp)
        ld a1, 80(sp)
        ld a2, 88(sp)
        ld a3, 96(sp)
        ld a4, 104(sp)
        ld a5, 112(sp)
        ld a6, 120(sp)
        ld a7, 128(sp)
        ld s2, 136(sp)
        ld s3, 144(sp)
        ld s4, 152(sp)
        ld s5, 160(sp)
        ld s6, 168(sp)
        ld s7, 176(sp)
        ld s8, 184(sp)
        ld s9, 192(sp)
        ld s10, 200(sp)
        ld s11, 208(sp)
        ld t3, 216(sp)
        ld t4, 224(sp)
        ld t5, 232(sp)
        ld t6, 240(sp)

        addi sp, sp, 256

        // return to whatever we were doing in the kernel.
        sret
```

最后一步，我们同样执行`sret`，宣告我们对于内核空间下的trap处理完毕，`sret`将`sepc`复制到`pc`中，根据`sstatus`寄存器的设置，切换到内核模式，并重新开放中断。因此，我们最后会回到上次被打断的地方，继续执行内核代码。

值得思考的是，如果内核trap由于计时器中断而调用yield，该如何返回trap？.....

当CPU从用户态进入内核态时，xv6将`stvec`寄存器赋值为`kernelvec`。在内核开始运行到将`stvec`赋值为`kernelvec`有一段时间窗口，在这段窗口内设备中断被禁止是很重要的。幸运的是RISC-V在它启动trap的时候总是禁止中断的，并且xv6直到设置`stvec`之前都不启用。

计时器中断

实际上计时器中断在**机器模式**下处理。在xv6的启动阶段中，仍处于机器模式下，在机器模式下我们通过`timerinit`（kernel/start.c）初始化了计时器，`CLINT`将负责产生计时器中断，更重要的是，我们通过改写`mtvec`寄存器的值（和`stvec`类似）将机器模式下的trap handler设置为`timervec`。因此，当`CLINT`产生一个计时器中断时，硬件会自动陷入机器模式，并跳转到`timervec`开始处理。

`timerinit`代码如下，

```
// set up to receive timer interrupts in machine mode,
// which arrive at timervec in kernelvec.S,
// which turns them into software interrupts for
// devintr() in trap.c.
void
timerinit()
{
  // each CPU has a separate source of timer interrupts.
  int id = r_mhartid();

  // ask the CLINT for a timer interrupt.
  int interval = 1000000; // cycles; about 1/10th second in qemu.
  *(uint64*)CLINT_MTIMECMP(id) = *(uint64*)CLINT_MTIME + interval;

  // prepare information in scratch[] for timervec.
  // scratch[0..3] : space for timervec to save registers.
  // scratch[4] : address of CLINT MTIMECMP register.
  // scratch[5] : desired interval (in cycles) between timer interrupts.
  uint64 *scratch = &mscratch0[32 * id];
  scratch[4] = CLINT_MTIMECMP(id);
  scratch[5] = interval;
  w_mscratch((uint64)scratch);

  // set the machine-mode trap handler.
  w_mtvec((uint64)timervec);

  // enable machine-mode interrupts.
  w_mstatus(r_mstatus() | MSTATUS_MIE);

  // enable machine-mode timer interrupts.
  w_mie(r_mie() | MIE_MTIE);
}
```

`timervec`如下所示（kernel/kernelvec.S），主要做的事情就是对计时器芯片重新编程，使其开始新的一轮计时，并且产生向内核模式发出一个**软件中断**。最后，`timervec`通过`mret`返回内核模式下，如果在内核模式下，我们的中断是开放的，那么内核就能够捕捉到这个软件中断，内核会跳转到`kernelvec`，保存相关寄存器，然后在`kerneltrap`中，发现这是一个设备中断而且是计时器中断，从而执行相应的处理。

```
timervec:
        # start.c has set up the memory that mscratch points to:
        # scratch[0,8,16] : register save area.
        # scratch[32] : address of CLINT's MTIMECMP register.
        # scratch[40] : desired interval between interrupts.

        csrrw a0, mscratch, a0
        sd a1, 0(a0)
        sd a2, 8(a0)
        sd a3, 16(a0)

        # schedule the next timer interrupt
        # by adding interval to mtimecmp.
        ld a1, 32(a0) # CLINT_MTIMECMP(hart)
        ld a2, 40(a0) # interval
        ld a3, 0(a1)
        add a3, a3, a2
        sd a3, 0(a1)

        # raise a supervisor software interrupt.
    li a1, 2
        csrw sip, a1

        ld a3, 16(a0)
        ld a2, 8(a0)
        ld a1, 0(a0)
        csrrw a0, mscratch, a0

        mret
```

Page-fault exceptions

xv6对于异常情况的处理非常简单粗暴，如果是发生在用户空间下的异常，内核直接终止该进程；如果是发生在内核空间下的异常，内核会panic停止执行。实际上，现在的操作系统通常有很多不同的方法来响应这些异常。

例如，**缺页错误Page-fault**这种异常，可以被很多操作系统内核利用，从而实现**写时复制Copy-On-Write**方法，例如copy-on-write(COW) fork。

COW fork

先回忆xv6的系统调用`fork`（kernel/proc.c），对一个进程调用`fork`就会创造一个新的子进程，原来的进程自然成为父进程。最重要的是，`fork`会把父进程的整个内存布局，通过`uvmcopy`，为子进程分配物理内存，然后拷贝到子进程的空间中。

出于效率考虑，整份的内存拷贝可能不是必须的，如果子进程很快就会关闭，或者很快就要调用`exec`加载新的内存映像，这些复制就显得多余。一个简单的想法是，让父子进程共享父进程的物理内存，但这个解决方案显然不够好，因为两个进程现在共用堆区域和栈区域，父子进程因此很容易就干扰对方的正常运行。

当CPU无法转换一个虚拟地址时，即相应的用户页表里没有这一项映射，或者相关权限要求不满足时，就会产生我们熟知的缺页错误。根据执行指令的不同，缺页错误又可以细分为三种：

- **Load Page Faults**：无法转换的虚拟地址位于一条加载（读）指令中。Scause：13
- **Store Page Faults**：无法转换的虚拟地址位于一条存储（写）指令中。Scause：15
- **Instruction Page Faults**：无法转换的虚拟地址位于一条执行指令中。Scause：12

COW fork的基本方案是，开始的时候，父子进程共享所有的物理页，但是这些物理页只标为**只读**。当父子进程尝试执行一个写操作的时候，RISC-V CPU抛出一个缺页异常。作为对这个异常的回应，内核对包括这个错误地址的页制作一个副本，然后将一个副本映射到子进程的地址空间中**读/写**，另一个副本映射到父进程的地址空间中**读/写**，（开放这两个页的读写权限，更新两个进程的页表）。在更新页表之后，内核在引起错误的指令处恢复执行。

COW的方法对fork很有效，因为子进程经常在fork之后立即调用exec，用一个新的地址空间替换它的地址空间。在这种情况下，子进程只会遇到一些缺页错误，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任何修改就可以使其受益。

Lazy Allocation

另一个利用缺页错误而实现的很棒的功能是**懒惰分配Lazy Allocation**，它可以分两步实现。

首先，用户进程可以调用`sbrk`请求更多的物理内存，这点在前一章已经展示过，只不过，现在内核只是简单地标记该进程的大小已增长（例如增加`p->sz`），却不急着分配物理内存给它，因此，那些新增长的虚拟地址在页表中是无效的。接着，使用这些虚拟地址就会引发缺页错误，现在内核再为该虚拟地址分配一页物理内存，并且更新页表。

因为用户进程会经常向分配器请求比它们实际上所需要的更多的物理内存。在这种情况下，内核只在用户进程实际用到它们的时候，再来为它们分配物理页。通过这种懒惰的思想，内核尽可能地推迟物理内存的分配，可以避免很多因用户进程的贪婪而带来的不必要的分配。同样地，懒惰分配对用户进程也应该是透明的。

Paging from Disk

另一个广泛使用的利用缺页错误的特性是**从磁盘分页**。

当一个用户进程需要装载一些页到物理内存中，但RAM的空闲空间不足时，内核此时可以**逐出evict**一些物理页，将这些页写到诸如磁盘等地方，同时在相应页表中标记PTE为无效。之后，当别的进程读写这些被逐出的页时，就会产生缺页错误。内核发现这些页已经被换出到磁盘上，因此首先在RAM中为它们找到空间，然后从磁盘上重新读进这些页，改写PTE为有效，更新页表，并且重新执行读写指令。内核在RAM为这些物理页找空间的时候，可能又要逐出别的页。但是，整个过程对用户进程仍然是透明的

请求调页避免了在进程运行时，直接将所有进程的物理页载入物理内存中，而是在进程需要的时候，才从磁盘上加载这些页面。这种模式，在满足**访存局部性**的情况下表现很好，进程在一个时间段内只访问固定一部分物理页，这避免了频繁地换入换出页。

还有其它一些功能特性也利用了分页和缺页错误来实现，例如自动增长的栈Automatically extending stacks和内存映射文件Memory-mapped files等。

Real World

我们已经看到，通过将`trampoline`页同时映射到用户和内核空间下，可以解决很多棘手的问题。但是，如果我们直接**将内核的映射也加入用户页表**中，甚至可以彻底解决切换页表这样的棘手问题，尤其是我们反复穿梭于内核和用户空间之间，只需要适当地设置相关PTE中的标志位，保证我们建立起来的隔离机制不被破坏。系统调用也能从中受益，例如内核可以直接利用当前进程的用户内存映射，允许内核代码直接解引用用户指针。

事实上，很多操作系统就是这么设计的，在效率上的提升很显著。xv6不这样设计的原因是，内核的代码需要相当小心地编写，否则就会出现一些严重的安全漏洞（例如用户指针使用不当），而且这会使得内核代码会相当复杂，毕竟xv6只是一个简单的教学操作系统。