一、COW是做什么的：

Copy-On-Write（写时复制，简称COW）是一种在计算机领域中广泛应用的优化策略，主要用于提高内存和存储的使用效率

1.kernel/proc.c——fork()

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc(); // 获取当前进程(父进程)的PCB

  if((np = allocproc()) == 0){ // 给新进程(子进程)申请一个PCB
    return -1;
  }

  // 子进程数据初始化

  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){ // 将父进程p的数据复制给子进程np ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
    // 失败了则...
      freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz; // 子进程size初始化

  np->parent = p; // 子进程parent=父进程

  *(np->trapframe) = *(p->trapframe); // 将父进程的寄存器复制给子进程

  np->trapframe->a0 = 0; // 子进程返回值为0

  // 文件相关，不管
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid; // 父进程返回值为子进程的pid

  np->state = RUNNABLE; // 将子进程设定为可执行状态，下一次调度就会执行子进程了

  release(&np->lock);

  return pid;
}
```

2.kernel/vm.c——uvmcopy()

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){ // 从0开始，每次访问一个页
    if((pte = walk(old, i, 0)) == 0) // 获取old页表中目标页面的最后一级pte，因为最后一级pte才记录着目标页面的物理地址
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte); // 获取到目标页面对应的物理地址
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0) // 申请一个新物理页面
      goto err;
    memmove(mem, (char*)pa, PGSIZE); // 将pa中的数据copy到新页面mem中
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){ // 再将mem映射到new页表的对应位置中
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

二、如何知道有人在修改共享页面?

```
// sample
int main()
{
    int x = 10;
    x = 1; // 此时未共享，虽然在修改，但是无所谓
    int pid=fork();
    if(pid == 0) // 子进程
    {
        x = 5; // 此时已经共享，修改行为要被发现
    }
    else // 父进程
    {
        x = 15;// 此时已经共享，修改行为要被发现
    }
}
```

当对某个页面进行写时，系统会检查该页是否可写——pte的 `PTE_W`

若 `PTE_W=0`，即为不可写，那么就会产生缺页中断

kernel/riscv.h

```
#define PTE_V (1L << 0) // 存在？
#define PTE_R (1L << 1) // 可读？
#define PTE_W (1L << 2) // 可写？
#define PTE_X (1L << 3) // 可执行？
#define PTE_U (1L << 4) // 用户可访问？
```

三、缺页中断时会怎么样

1. 当外部设备或内部定时器产生**中断**时，会向处理器发送中断信号
2. **保存上下文**：保存一下context、pc等 （硬件自己完成了，我们不用管）
3. **跳转到中断处理程序**：处理器会跳转到由 `stvec` 寄存器指定的中断处理程序的起始地址开始执行：uservec
4. **恢复上下文并返回**：处理完成后，会恢复之前保存的上下文，并通过执行 `sret` 指令返回到原来的程序执行点。

跳转到中断处理程序：

1.kernel/trampoline.S——uservec

```
uservec:
        # 记性好的同学应该记得，上节课我们说过：每个进程在执行第一个函数forkret()后会跳转到userret处，我们将sscratch的值改成了TRAPFRAME
        # 所以sscratch记录着TRAPFRAME的虚拟地址，此操作交换sscratch与a0，于是a0就记录着trapframe的地址
        # 想了解40(a0)到底是什么，可以去proc.h中查询trapframe结构体的定义，注释中有标注
        csrrw a0, sscratch, a0

        # 将之前说的寄存器都存起来
        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)

    # a0的初始值存在sscratch里，现在交换sscratch和t0，再将t0(a0的初始值)存到trapframe->a0 即:112(a0)里
        csrr t0, sscratch
        sd t0, 112(a0)

        # 8(a0)是kernel_stack_pointer，由于现在要转入内核模式，所以要将sp换成 ksp
        ld sp, 8(a0)

        # 32(a0)是cpu的id，将该id存到tp寄存器中
        ld tp, 32(a0)

        # 16(a0)是kernel_trap
        # 再次，记性好的同学应该还记得，在trap.c的usertrapret()中，我们将kernel_trap的值改成了usertrap()函数地址
        # 所以此时t0即为usertrap()函数的地址
        ld t0, 16(a0)

        # 0(a0)是内核页表，由于现在要转入内核模式，所以要将页表换成内核页表
        ld t1, 0(a0)
        csrw satp, t1
        sfence.vma zero, zero

        # 之前t0中已经存储了usertrap()函数地址，现在跳转到该地址去执行usertrap()
        jr t0 # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
```

2.kernel/trap.c——usertrap()

```
void
usertrap(void)
{
  int which_dev = 0;
  // status中有一个bit叫做SPP，SPP位=1代表是内核模式，=0代表是用户模式，此处要求SPP=0因为usertrap只能接受来自用户的请求
  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  w_stvec((uint64)kernelvec);

  // 由于现在进入内核态了，内核不知道之前是哪个进程调用这个函数
```

```
12    // myproc()函数获取当前正在执行的进程指针，以便能够访问该进程的上下文信息。
13    struct proc *p = myproc();
14
15    // 将当前程序计数器（PC）的值（即sepc寄存器的值）保存到当前进程的trapframe中的epc字段。这对于之后恢复用户程序执行是必要的
16    p->trapframe->epc = r_sepc();
17
18    if(r_scause() == 8){ // scause是什么？
19      // system call
20
21      if(p->killed)
22        exit(-1);
23
24      // 之前我们将PC的值存在epc中，该值为ecall所在地址，之后返回要返回到ecall的下一条指令中，所以+4
25      p->trapframe->epc += 4;
26
27      intr_on();
28
29      syscall(); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
30    } else if((which_dev = devintr()) != 0){
31      // ok
32    } else {
33      printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
34      printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
35      p->killed = 1;
36    }
37
38    if(p->killed)
39      exit(-1);
40
41    // give up the CPU if this is a timer interrupt.
42    if(which_dev == 2)
43      yield();
44
45    usertrapret(); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
46  }
```

scause是什么：一个寄存器

当RISC-V处理器遇到中断时，会停止当前程序的执行，并跳转到中断处理程序。在这个过程中，`scause`寄存器会被更新以反映中断的原因或类型。

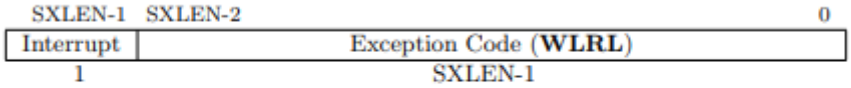

Figure 4.11: Supervisor Cause register scause.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

我们发现决定如何处理中断的工作是交给usertrap()函数的，所以我们应该在usertrap()中确定好对应的处理方法

四、释放页面时存在的问题

- 在原先的xv6中，当一个进程被杀死，它需要释放掉它申请的所有页面
- 进程也可以主动调用free()函数去释放页面
- 这样的设计显然没有考虑到页面的共享问题：如果共享的页面被某个进程释放掉，其他进程想要的数据就丢失了，这显然不合理！
- 所以系统应该统计一下，每个页面被多少个进程共享，当这个页面没有进程共享它时，才可以真正把它释放掉