启动流程

RISC-V计算机打开电源上电后，初始化自己并运行一个在ROM中的boot loader。Boot loader将xv6的内核加载到物理地址为`0x80000000`内存中（`0x0~0x80000000`的地址范围里包含了I/O设备）。

然后在**machine mode**下，CPU从`_entry`（位于`kernel/entry.S`）开始运行xv6。

```
.section .text
.global _entry
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0
        li a0, 1024*4
        csrr a1, mhartid
        addi a1, a1, 1
        mul a0, a0, a1
        add sp, sp, a0
        # jump to start() in start.c
        call start
spin:
        j spin
```

初始栈stack0的声明在`kernel/start.c`中

```
__attribute__ ((aligned (16))) char stack0[4096 * NCPU];
```

表明了每个CPU的栈是4096个字节。

RISC-V的栈是向下扩展的，高地址为栈底，低地址为栈顶，所以`sp = stack0 + (hartid * 4096)`，将高地址加载到`sp`寄存器中。

什么是hart？RISC-V处理器对底层提供了一种抽象，叫Hardware Thread，简称hart，中文可以翻译为硬件线程。可以把hart理解为是真实CPU提供的一种模拟，关于hart、core、CPU的一些区别并不是操作系统层面需要关心的，我们在这里可以简单地将三者视为同样的概念，把hartid看作是cpuid。

之后程序跳转到了函数`start`（位于`kernel/start.c`）中。

函数`start`执行一些仅在**machine mode**下允许的配置，然后才会切换到supervisor mode。

```
void
start()
{
  // set M Previous Privilege mode to Supervisor, for mret.
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK;
  x |= MSTATUS_MPP_S;
  w_mstatus(x);

  // set M Exception Program Counter to main, for mret.
```

```
  // requires gcc -mcmodel=medany
  w_mepc((uint64)main);

  // disable paging for now.
  w_satp(0);

  // delegate all interrupts and exceptions to supervisor mode.
  w_medeleg(0xffff);
  w_mideleg(0xffff);
  w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);

  // configure Physical Memory Protection to give supervisor mode
  // access to all of physical memory.
  w_pmpaddr0(0x3fffffffffffffull);
  w_pmpcfg0(0xf);

  // ask for clock interrupts.
  timerinit();

  // keep each CPU's hartid in its tp register, for cpuid().
  int id = r_mhartid();
  w_tp(id);

  // switch to supervisor mode and jump to main().
  asm volatile("mret");
}
```

首先切换工作模式，`r_mstatus()`函数的功能相当于执行了一个`csrr`指令，读取了`mstatus`寄存器的值存储在了一个变量x中。接下来是对寄存器中的位进行修改，此处修改涉及到了RISC-V中`mstatus`寄存器的结构。修改完之后，同样的，`w_mstatus()`函数的功能其实是相当于执行了一个`csrw`指令，将x值写入了`mstatus`寄存器中。

之后，将`main`函数的地址写入`mepc`寄存器，由此将返回地址设为`main`函数，以便于在`main`函数中执行代码。向页表寄存器`satp`写入0来**禁止虚拟地址转换**，然后赋予supervisor mode对所有物理内存的访问权限，还有将中断和异常委托给supervisor mode。此外，还需要对时钟芯片进行编程以产生计时器中断。

最后，`start`就可以通过调用`mret`返回，然后进入到到**supervisor mode**了，此时PC的值将更改为`main`函数的地址。

```
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
```

```
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

在`main`函数中，启动操作系统之前我们需要做一些初始化配置。首先调用了`consoleinit`函数，事实上，这个函数内部有一个对UART进行初始化的操作，然后连接到读和写的系统调用。接着再对`printf`进行初始化，就可以在屏幕打印信息了。

接下来，进行了对一些设备和系统中一些必要模块的初始化。完成了上面的初始化之后，就可以调用`userinit`函数来创建第一个用户进程了，我们总是需要有一个用户进程在运行，这样才能实现与操作系统的交互。第一个进程会执行一个小程序，`kernel/proc.c`的`uchar initcode[]`中展现了这个程序的二进制形式，事实上它对应了一段汇编代码，在`user/initcode.S`中。它**通过调用`exec`系统调用来重新进入内核**，然后`exec`会**用一个新程序`/init`来替换当前进程的内存和寄存器**，一旦`exec`完成，就会**返回`/init`进程中的用户空间**。`init`会在控制台上**启动一个shell**，具体细节在`user/init.c`中。

系统调用流程