第二部分：内核的启动与组织

现在我们关注内核的具体的实现。其实，内核也只是一个“程序”，它遵循着硬件的规范与硬件沟通，从而启动、运行等等。

在这一部分，我们将从代码层面解释内核程序的启动，以及启动之后内核如何组织进程和内存。

本部分对应于 xv6 book 的 chapter 2 。

启动

我们的第一个程序只能由硬件来唤醒，这里是软硬件的接口。

这里涉及到的文件分别是：`kernel/entry.S`，`kernel/kernel.ld`，`kernel/start.c`，`main.c` 。

kernel/kernel.ld：链接描述文件

```
OUTPUT_ARCH( "riscv" )
ENTRY( _entry )

SECTIONS
{
  /*
   * ensure that entry.S / _entry is at 0x80000000,
   * where qemu's -kernel jumps.
   */
  . = 0x80000000;

  .text : {
    *(.text .text.*)
    . = ALIGN(0x1000);
    _trampoline = .;
    *(trampsec)
    . = ALIGN(0x1000);
    ASSERT(. - _trampoline == 0x1000, "error: trampoline larger than one page");
    PROVIDE(etext = .);
  }
```

```
  /* ... */
}
```

这是链接描述文件，负责在链接环节精细地调整程序的构成。

第一行指明了程序的目标架构是 riscv，第二行则告知链接器，程序的入口是 _entry 标签，链接器会把 _entry 标签所在的代码放置到存放着所有代码的 （.text 段）的最前面。`. = 0x80000000;` 语句树立了一个地址的标签，随后 .text 段就以此地址开始（因此 `_entry` 标签的地址也就是 `0x80000000`），而 `*(.text .text.*)` 就会把编译出来对象文件的 .text 代码段全都放到一起。

QEMU（作为虚拟的硬件设备）指定的程序入口位置为地址为 `0x80000000` 的位置。

因此，当电脑一接上电源，我们的操作系统便会从 _entry 标签开始执行。

不将操作系统入口放在 `0x0` 的地址位置，是因为 QEMU 在 `0x0` 到 `0x80000000` 之间预留了 I/O 设备的“地址”。（事实上，QEMU 模拟的 RAM 的硬件地址就是从 `0x80000000` 开始的，到 `0x86400000` 结束）

值得提醒的是，在启动的时候，页表硬件并没有被启动，所以此时地址并不会经过翻译，我们会直接操作真实的物理内存。（值得提到的是，即使页表之后启动了，内核的虚拟的内存位置和真实的物理内存位置仍然一致，这是“直接映射”，参见 book p35）

kernel/entry.S：_entry 入口

```
        # qemu -kernel loads the kernel at 0x80000000
        # and causes each CPU to jump there.
        # kernel.ld causes the following code to
        # be placed at 0x80000000.
.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
```

这段代码如注释所说，是给每一个 cpu 都分配一个 4K 大小的栈空间，也就是 start.c 里面的 stack0，同时把初始化的栈顶指针，根据 cpu 的 id （从 hartid 中读取出来）放置到正确的位置。

需要提醒的是，这则代码对于每一个 cpu 都会执行。

kernel/start.c

```
// entry.S jumps here in machine mode on stack0.
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");
}
```

这一段代码执行了以下的操作，总体来说是利用 machine mode，假装自己是在一个 supervisor mode 中引发的中断中，并利用 mret 指令跳转到 kernel/main.c 中，以 supervisor mode 继续执行。具体来说：

1. `w_mstatus`：将机器从 machine mode 调整为 supervisor mode
2. `w_mepc`：将返回地址设置为 main 函数的地址
3. `w_satp`：禁用页表
4. 把所有的中断都交给 supervisor mode 去运行，设置内存边界
5. 调用 timerinit，初始化时钟中断，让之后的时钟中断都可以成为软件中断从而能够被处理
6. mret 指令（进入 main 函数）

需要提醒的是，每一个 cpu 都会执行一遍上述的代码。

kernel/main.c

```
volatile static int started = 0;

// start() jumps here in supervisor mode on all CPUs.
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();
}
```

这段代码初始化了整个操作系统的各个子系统。

这里代码分成了两个部分逐 cpu 进行初始化。

在 0 号 cpu 上进行绝大部分工作，同时也创建了第一个用户进程（`userinit()`），而在其他 cpu 上则只是进行基本的配置。

这里用到了 `__sync_synchronize();`，这是由 gcc 提供的原子指令，称为一个 memory barrier，告知编译器调整指令执行循序时不要越过这一指令，从而保证 started 变量确实能够控制初始化的顺序。

最后，每一个 cpu 都调用了自己的 scheduler，开始准备运行用户的程序。

系统组织

这一部分稍微详细一点的解释一下，我们的操作系统是运行在什么样“抽象环境”中的。

xv6 的文件组织大体上是模块化的。每个模块向外暴露的函数的声明写在了 `kernel/defs.h` 中，供其他模块使用。

xv6 也尽力为进程抽象出了各种资源，最显著的例子就是文件描述符（file descriptor），让用户程序的编写者不用纠结于过多的细节。

进程组织

进程是计算机、操作系统中真正的核心部门，执行“计算”的核心功能，操作系统中几乎所有其他模块都是为进程组织而服务的。所以这里我们先介绍进程的组织方法。

xv6 中进行隔离和抽象的单位是进程，一个进程不能访问另一个进程的任何内容，只能通过进程间通信的方式沟通。

操作系统的任务就是让每一个进程都觉得自己占有整台机器（包括内存和 CPU），这样可以大大简化用户程序的设计。

进程结构

在 `kernel/proc.h:83` 处，xv6 用 proc 结构体维护了一个进程的关键的信息。

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

这段代码的注释非常充分，没有什么可以解释的。

内存管理

每一个进程有自己的一个页表，用于把自己的虚拟内存地址映射到物理内存地址上。具体的页表策略不在这里阐述。xv6 使用 38 位虚拟地址（最大到 `MAXVA=0x3fffffffff` 地址，见 `kernel/riscv.h:363`）。

具体的内存管理方式和页表的使用方法会在 xv6 的 chapter 3 中解释，这里简单的（对 proc 结构体中的一些变量）做一个解读。

- `state`：表示这个进程的状态
- `pagetable`：这是进程自己的页表。
- `trapframe`：这是 `trampoline.S` 中的程序在系统调用切换用户态和内核态的时候会用到的数据
- `kstack`：进程独属的内核栈，在系统调用中使用。

进程切换

上面提到，每个 cpu 在初始化之后，都会调用 scheduler 函数。

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run.
//  - swtch to start running that process.
//  - eventually that process transfers control
//    via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

在注释中可以看出，该函数的功能是：轮询是否有没有被阻塞（RUNNABLE）的进程，如果有，切换上下文运行该进程。

一个用户进程通过 exit（或者 sleep）的系统调用，或者中断之后调用 yield ，标志着放弃当前的 CPU 运行时间。调用 `sched` 函数（kernel/proc.c:467），从用户的上下文切换到 `scheduler` 的上下文，从而会接着之前的 `scheduler` 运行，继续进行一个论询（因为 pc 寄存器被保存了起来）。

用户进程与内核

系统调用是用户进程通过 `trampoline page` 中的代码和 `trapframe` 中的参数，实现控制权和数据向内核传输。

这也是用户进程与内核的沟通，具体的机制不在这里阐述。

习题

https://github.com/ChenQiqian/xv6-code-report/compare/riscv...ex2

在 `kalloc.c` 里面编写了 `calfree` 函数，并在 `syscall.h`，`syscall.c` 以及 `sysfile.c` 中补全了系统调用；在 `usys.pl` 中补全了系统调用的 `entry` 。

```
int
calfree(void)
{
  struct run *r;
  int ans = 0;
  acquire(&kmem.lock);
  r = kmem.freelist;
  while(r) {
    ans = ans + 1;
    r = r->next;
  }
  release(&kmem.lock);
  return ans * 4096;
}
```

阅后心得

这部分解释了最关键的部分：操作系统启动、操作系统初始化、操作系统最初的进程，解决了我们对整个计算机体系中可能是最模糊的一部分的认识。

参考文献

https://biscuitos.github.io/blog/LD-ENTRY/

https://www.jianshu.com/p/42823b3b7c8e

https://blog.csdn.net/shenjin_s/article/details/88712249

https://jborza.com/emulation/2021/04/04/riscv-supervisor-mode.html

https://www.cnblogs.com/bluestorm/p/12527969.html

https://www.cnblogs.com/FrankTan/archive/2010/12/11/1903377.html

https://five-embeddev.com/riscv-isa-manual/latest/supervisor.html#supervisor-scratch-register-sscratch

https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf