哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

xv6原理简析：启动过程

操作系统课程组

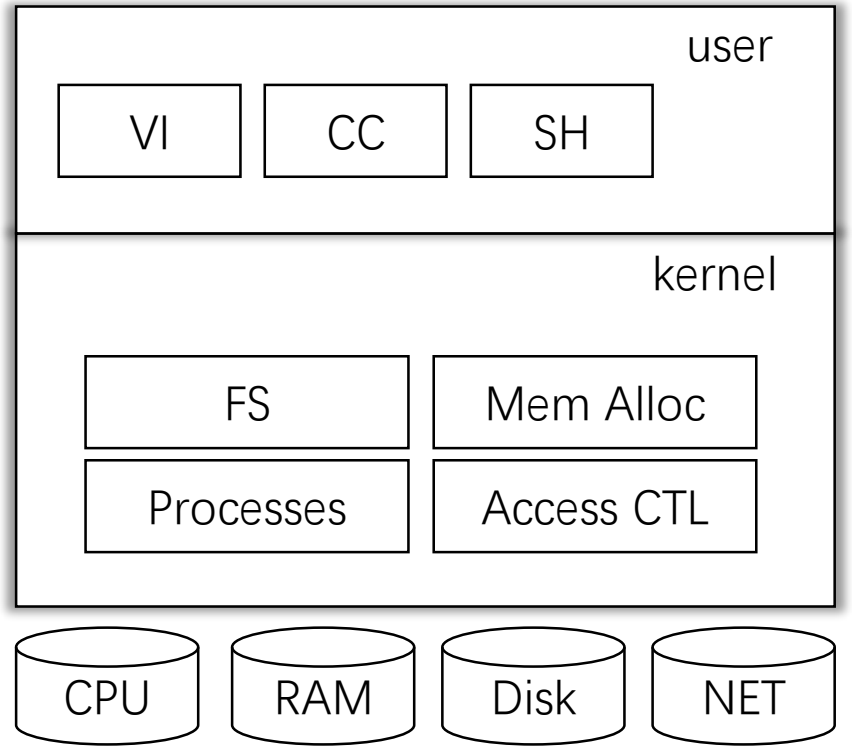
哈尔滨工业大学（深圳）

2022年秋季

Email：xiawen@hit.edu.cn

认识xv6

- xv6是由麻省理工学院（MIT）开发以教学目的的操作系统
- xv6是在x86处理器上用ANSI标准C重新实现的Unix第六版（即v6），课程编号为6.828
- 2019年被移植到RISC-V之上，并设置了6.S081课程
- **麻雀虽小**
  - 基于C语言，没有任何依赖库
  - 代码量仅2万行，方便阅读
- **五脏俱全**
  - 功能完备、开源，可理解操作系统原理及实现
  - 基于RISC-V、X86，可深入理解计算机体系结构
  - 类Unix系统，可延伸学习其他常见操作系统



RISC-V通用寄存器

- RISC-V提供32个通用寄存器
- 实验中可以忽略浮点寄存器
- 进程切换以及用户态/内核态切换时需要保存/恢复**寄存器上下文**

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

RISC-V特权寄存器

- 特权寄存器用来处理异常和环境，属于操作系统专用
- 只有内核态才能操作
- 常用特权寄存器
  - sstatus：处理器运算状态
  - stvec：异常向量
  - sepc：异常指令地址
  - stval：异常地址
  - satp：操作页表

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

使用QEMU运行xv6

- QEMU是一套处理器模拟软件，能够模拟用户模式或者系统模式
  - xv6运行在系统模式，QEMU相当于一块主板，模拟了CPU、内存、磁盘等外设
  - QEMU可以模拟不同架构的平台，以解释的方式运行，在硬件配置上更灵活（模拟）
- 为什么不能用物理机运行xv6?
  - 实验中的xv6是riscv移植版，而目前主流物理机是x86-64架构
  - 操作系统需要操作特权态环境（特权寄存器、页表等）
- 事实上，xv6是可以运行在riscv开发板上的（如k210）相关工具链：
  - Linux发行版：由Linux内核、GNU工具、附加软件和软件包管理器组成的操作系统
  - RISC-V工具链：包括一系列交叉编译的工具：gcc，binutils，glibc等
  - QEMU： 在其他架构上模拟RISC-V架构的CPU

xv6缺少的功能

- xv6是非常精简的操作系统，也因此缺少了许多现代操作系统提供的功能，例如：
  - 只支持多进程，不支持多线程
  - 不支持内存页面替换（内存不够了会分配失败）
  - 不支持信号系统
  - 不支持内存映射（mmap）
  - 只读用户页（所有页面都具备可写权限）
  - 系统调用规范不完全符合POSIX标准
  - 不支持动态链接
  - 内核态无法动态内存分配

xv6内核代码导读（kernel目录下）

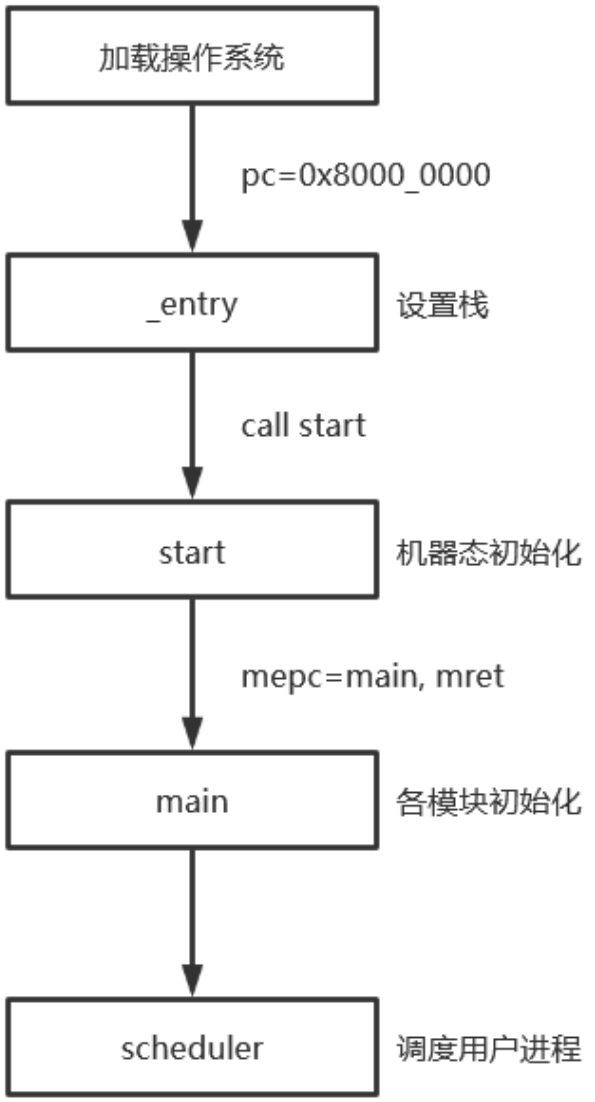
File	Description
bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	exec() system call.
file.c	File descriptor support.
fs.c	File system.
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel, and timer interrupts.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
spinlock.c	Locks that don't yield the CPU.
start.c	Early machine-mode boot code.
string.c	C string and byte-array library.
swtch.S	Thread switching.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
virtio_disk.c	Disk device driver.
vm.c	Manage page tables and address spaces.

Figure 2.2: Xv6 kernel source files.

- 启动相关核心代码
  - entry.S、start.c、main.c
- 进程管理相关核心代码
  - proc.c、sleeplock.c、spinlock.c
- 内存管理相关核心代码
  - vm.c、kalloc.c

xv6：从启动到进入控制台及其他用户进程

- 启动QEMU，QEMU从控制台参数加载资源，**将xv6整体加载到物理内存中**
- **QEMU将pc设为0x8000_0000，开始运行xv6的首个函数：_entry (kernel/entry.S)**
- entry函数设置当前CPU的栈寄存器(sp)，拥有栈后就可以调用函数了，进入start。
- start函数进行了如下操作：
  - 初始化机器态特权寄存器
  - 启用时钟中断
  - 把当前CPU的id放入tp寄存器
- start函数完成了机器态初始化，随后就进入了内核态的首个函数：main。
- main函数初始化了各种外设的驱动、内存模块、进程模块、文件系统。
- 进入scheduler函数之后，**xv6已经可以正常运行用户程序了，只要用户进程存在就会被CPU调度运行。**



放置第一个用户程序

执行控制台以及其他用户进程

相关代码简析

- 启动涉及entry.S与start.c等少数几个文件
- CPU有机器态、内核态（特权）、用户态
- entry、start两个函数运行在机器态（类似于BIOS），main运行在内核态（特权级），后面的用户程序运行在用户态

```
5   .section .text
6   _entry:
7           # set up a stack for C.
8           # stack0 is declared in start.c,
9           # with a 4096-byte stack per CPU.
10          # sp = stack0 + (hartid * 4096)
11          la sp, stack0
12          li a0, 1024*4
13          csrr a1, mhartid
14          addi a1, a1, 1
15          mul a0, a0, a1
16          add sp, sp, a0
17          # jump to start() in start.c
18          call start
19  spin:
20          j spin
21
```

kernel/entry.S

```
20  void
21  start()
22  {
23    // set M Previous Privilege mode to Supervisor, for mret.
24    unsigned long x = r_mstatus();
25    x &= ~MSTATUS_MPP_MASK;
26    x |= MSTATUS_MPP_S;
27    w_mstatus(x);
28
29    // set M Exception Program Counter to main, for mret.
30    // requires gcc -mcmodel=medany
31    w_mepc(x: (uint64)main);
32
33    // disable paging for now.
34    w_satp(x: 0);
35
36    // delegate all interrupts and exceptions to supervisor mode.
37    w_medeleg(0xffff);
38    w_mideleg(0xffff);
39    w_sie(x: r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);
40
41    // ask for clock interrupts.
42    timerinit();
43
44    // keep each CPU's hartid in its tp register, for cpuid().
45    int id = r_mhartid();
46    w_tp(x: id);
47
48    // switch to supervisor mode and jump to main().
49    asm volatile("mret");
50  }
```

kernel/start.c

kernel/main.c

```
// 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 cache
    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();
}
```

kernel/proc.c

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

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

    int found = 0;
    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;

        found = 1;
      }
      release(&p->lock);
    }
    if(found == 0) {
      intr_on();
      asm volatile("wfi");
    }
  }
}
```


- 启动过程中，从main函数进入scheduler函数（调度器）；scheduler调度到初始进程initcode，然后进入第一个进程

xv6：第一个进程的诞生

- 在上述启动过程中，也诞生了第一个用户进程
- 在main函数调用的userinit()里面，xv6在内核加载了第一个用户程序initcode，只有不到100字节。这个程序运行时只占用一个内存页（数据来自initcode），这个页兼具代码段、数据段、栈的功能。它只做一件事：运行exec(“/init”);
  - exec系统调用能够替换当前的进程至目标程序
- **init程序被放在磁盘上，它就是所有进程的祖先**
  - 从磁盘加载应用程序的过程是exec系统调用完成的
  - 它通过fork+exec的方式启动控制台程序sh
  - 它不断地调用wait(0)来回收所有僵尸进程
  - 它被启动后会输出一行“init: starting sh”
- sh程序是**控制台进程**。它被启动后就会输出'$'

这就是第一个进程（0号进程），该进程执行exec，会被替换成init程序

程序	位置	用途
initcode	内存数据段	exec启动init程序
init	磁盘	启动sh 回收僵尸进程
sh	磁盘	用户操作

这就是第二个进程（1号进程）shell控制台程序

initcode详解

- initcode的52个字节干了什么？（每条汇编4字节，所以9条汇编语句+字符串，注意括号里面是注释）

auipc a0, 0　　addi a0, a0, 36　(a0 = 36="\init\0")

auipc a1, 0　　addi a1, a1, 35　(a1 =43=&NULL)

add a7, x0, 7　　ecall (exec，这里执行init程序)

add a7, x0, 2　　ecall (exit)

jal ra, -8 (exit)　　'/'　　'i'　　'n'　　'i'

't'　　'\0'　　'\0'　　'$'　　'\0'　...

```
// a user program that calls exec("/init")
// od -t xC initcode
uchar initcode[] = {
  0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
  0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
  0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
  0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
  0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
  0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00
};
```

什么是系统调用

- 前面提到了，initcode执行了exec("/init")。这就是进行了系统调用。进行系统调用时CPU会产生异常，并将当前状态从用户态切换至内核态，从特权寄存器stvec中取出进入内核态的入口地址（物理地址），在xv6中，这个入口地址就是trampoline。
  - trampoline作为用户态和内核态的跳板：用户态陷入后，xv6先保存用户寄存器到特定的内存（trapframe页，后面内存管理会讲），然后跳入内核态的usertrap()函数里面（处理陷入...）；当系统调用完成回到用户态时，寄存器从这块内存恢复回原来的数据。

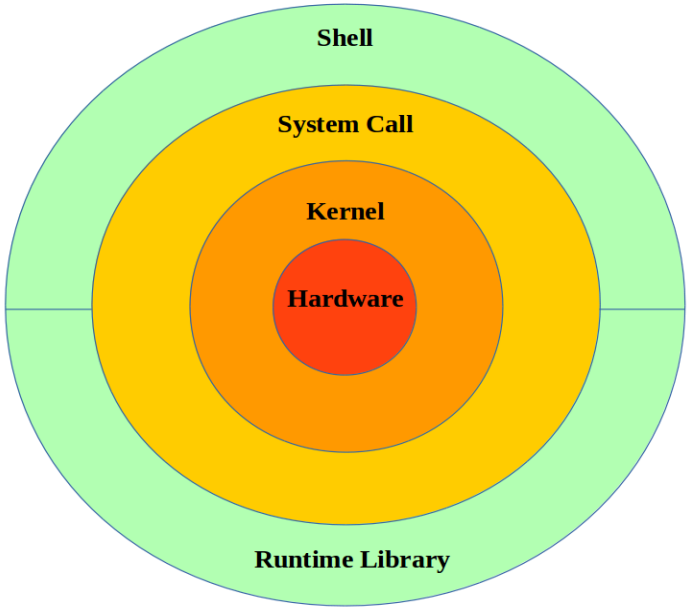
```
11          .section trampsec
12  .globl trampoline
13  trampoline:
14  .align 4
15  .globl uservec
16  uservec:
```

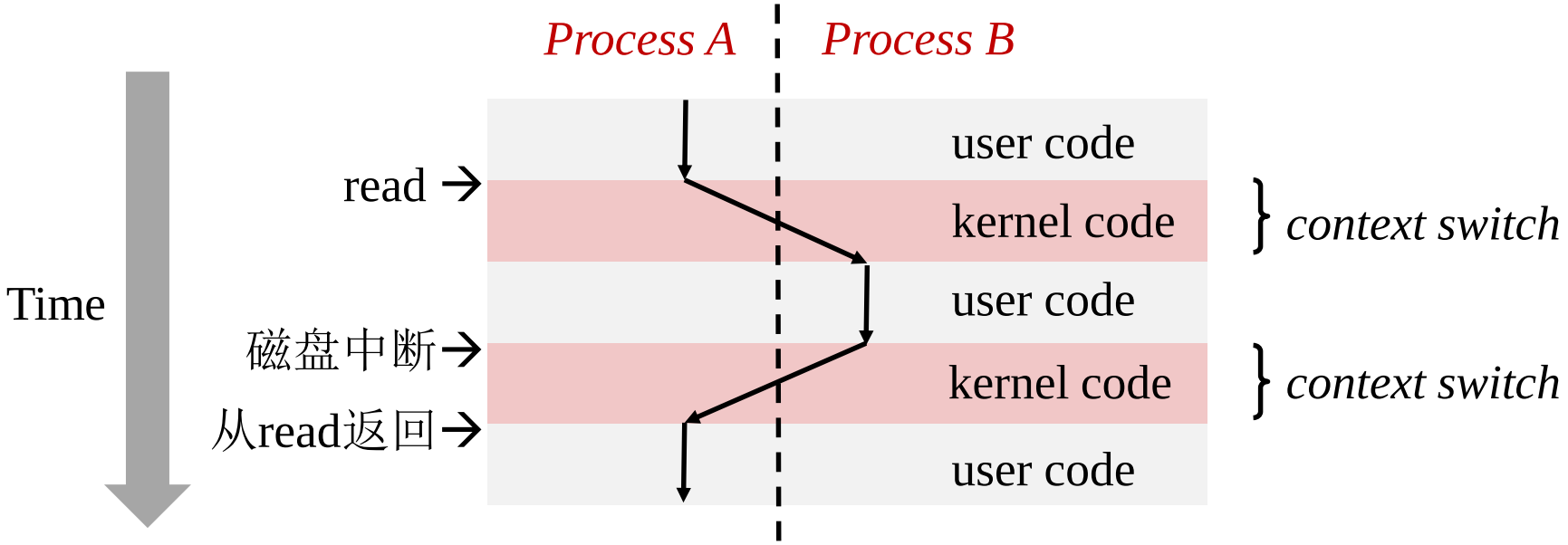
trampoline.S

- 用户态和内核态有什么区别?
  - 用户态和内核态主要体现在资源的访问权限上。例如，内核态可以操作许多的特权寄存器，例如页表寄存器satp、异常向量寄存器stvec。而用户态不能控制这些，而且只能访问U标志位的页面。

用户态与内核态以及进程切换【理论课回顾】

- 操作系统的内核是一种特殊的软件程序，控制计算机的硬件资源，例如协调CPU资源，分配内存资源，并提供稳定的环境供应用程序运行，并提供系统调用；
- 注意：用户态程序只能受限地访问内存；而内核程序并不是一个独立的进程，但是作为其他存在的进程的一部分来运行；
- 进程的切换涉及到内核态与用户态的切换，以及各种上下文切换配合进行（理论课-进程章节内的控制流部分）。





总结

- xv6操作系统概貌
  - xv6是一个精简、易于理解的操作系统
  - 缺少许多现代操作系统的功能（缺页替换等）
- 系统启动过程介绍
  - 由QEMU加载内核到物理内存
  - 完成机器态初始化
  - 初始化内核各个模块
  - 加载初始用户进程
  - 调度用户进程
- 更多的细节参照xv6相关代码以及xv6 book深入理解