XV6源码阅读报告

BootLoader

涉及文件

- bootasm.S
- bootmain.c
- asm.h
- mmu.h
- xv6.h

Boot程序流程

bootasm.S:完成保护模式的开启

1. 关闭中断
2. 为段寄存器赋初值
3. 通过键盘端口启用A20地址线，让内存突破1Mb的限制
4. 加载GDT段描述符
5. 开启保护模式
6. 使用跳转指令让cs寄存器加载段选择子，并根据段选择子选择GDT段描述符，完成保护模式的开启
7. 为其他段寄存器赋值
8. 设置esp寄存器的值，跳转至c程序bootmain

bootmain.c:完成内核的加载并跳转至内核

1. 读取内核文件的ELF头部到指定位置
2. 获取ELF文件的程序段表起始位置即程序段数目
3. 根据程序头表加载所有内核文件至内存中
4. 根据ELF文件头获取内核入口地址
5. 跳转至内核入口，结束引导程序，进入内核程序

总结

- 引导程序位于硬盘的第一个扇区。
- 在计算机启动时，BIOS读取引导分区的第一个扇区，将其加载进内存0x7c00处，将控制器转交给引导程序
- 引导程序完成加载内核前的一系列引导工作，如开启保护模式，将内核加载进内存中等
- 若一个扇区512byte的容量不足以让引导程序完成这些工作，可在硬盘中再增加个程序，令引导程序先加载该程序，再由该程序完成剩余工作
- 引导程序完成工作后，跳转进内核程序的入口处，将计算机控制权交予内核程序

附录

其他A20地址线开启方式

- 键盘控制器
  Keyboard Controller:
  This is the most common method of enabling A20 Gate.The keyboard micro-controller provides functions for disabling and enabling A20.Before enabling A20 we need to disable interrupts to prevent our kernel from getting messed up.The port 0x64 is used to send the command byte.

Command Bytes and ports
0xDD Enable A20 Address Line
0xDF Disable A20 Address Line

0x64 Port of the 8042 micro-controller for sending commands

Using the keyboard to enable A20:

```
EnableA20_KB:
cli                 ;Disables interrupts
push    ax          ;Saves AX
mov al, 0xdd  ;Look at the command list
out 0x64, al   ;Command Register
pop ax          ;Restore's AX
sti                 ;Enables interrupts
ret
```

- 中断处理函数 **Using the BIOS functions to enable the A20 Gate:**
  The INT 15 2400,2401,2402 are used to disable,enable,return status of the A20 Gate respectively.

  Return status of the commands 2400 and 2401(Disabling,Enabling)
  CF = clear if success
  AH = 0
  CF = set on error
  AH = status (01=keyboard controller is in secure mode, 0x86=function not supported)

  Return Status of the command 2402
  CF = clear if success
  AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)
  AL = current state (00: disabled, 01: enabled)
  CX = set to 0xffff is keyboard controller is no ready in 0xc000 read attempts
  CF = set on error

```
Disabling_A20:
push ax
mov ax, 0x2400
int 0x15
pop ax

Enabling_A20:
push ax
mov ax, 0x2401
int 0x15
pop ax

Checking_A20:
push ax
push cx
mov ax, 0x2402
int 0x15
pop cx
pop ax
```

- 系统端口
  Using System Port 0x92
  This method is quite dangerous because it may cause conflicts with some hardware devices forcing the system to halt.
  Port 0x92 Bits
  - **Bit 0** - Setting to 1 causes a fast reset
  - **Bit 1** - 0: disable A20, 1: enable A20
  - **Bit 2** - Manufacturer defined
  - **Bit 3** - power on password bytes. 0: accessible, 1: inaccessible
  - **Bits 4-5** - Manufacturer defined
  - **Bits 6-7** - 00: HDD activity LED off, 01 or any value is "on"

```
enable_A20_through_0x92:
push ax
mov al, 2 out   0x92, al pop ax
```

32位GDT段描述符格式

- 每个描述符占8byte

```
// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
// 32位段描述符格式：
// 0~15位 段界限的0至15位，即(((lim) >> 12) & 0xffff)
// 16~31位 段基址的0至15位，即((base) & 0xffff)
// 32~39位 段基址的16至23位，即(((base) >> 16) & 0xff)
// 40~43位 描述符的子类型，共4bit
// 43位为X位，代表该段是否可执行，0x8代表可执行即为代码段，反之为数据段
// 42位，对于代码段来说为C位，代表是否为特权级依从，0x4代表是依从的
//         对于数据段来说为E位，代表扩展方向，0x4代表向下扩展，即向低地址方向扩展
// 41位，对于代码段来说为R位，代表是否可读，0x2代表可读，代码段皆不可写
//         对于数据段来说为W位，代表是否可写，0x2代表可写，数据段皆可读
// 40位为A位，代表是否访问过，0x1代表已访问
// 44~47位，在SEG_ASM中即为0x9
// 47位为P位，1代表描述符所代表的段处在内存中
// 45~46位为DPL，代表特权级0，1，2，3，0为最高，这里作为boot程序，特权级为最高级0
// 44位为S位，0为系统段，1为代码段或数据段，这里为数据段或代码段，故设为1
// 48~51位 段界限16~19位，即(((lim) >> 28) & 0xf)
// 52~55位 在SEG_ASM中为0xc
// 55位为G位，代表段界限的基本单位，0代表已字节为单位，1代表已4KB为单位，
//             这里设为1，故在计算段界限时，都要将参数右移12位
// 54位为D/B位，代表默认操作数大小或默认栈指针大小或上部边界的标记
//                 对于代码段称为D位，0代表操作数和偏移量都是16位的，1代表32位的操作数和偏移量
//                 对于数据段称为B位，0代表使用pop，push，call等操作时，使用sp寄存器，1则使用esp寄存器
//                 在这里设为1
// 53位为L位，预留给64位处理器，这里设置为0
// 52位为AVL，供操作系统使用，这里直接设为0
// 故52~55位为1100即0xc
// 56~63位为段基址的24~31位，即(((base) >> 24) & 0xff)
```

32位段选择子格式

```
# 段选择子格式：
# 段选择子共16位
# 高13位为选择子编号
# 第0~1位为PRL，代表权限级别
# 第2位为TI位，0代表查找GDT，1代表查找LDT
```

通过IO端口读取硬盘数据

- 样例程序

```
//使用LBA28逻辑编址方式从硬盘中读取扇区
//扇区地址为28位
// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  // 0x1f2端口，8位输入，控制读取扇区数目
  outb(0x1F2, 1);   // count = 1
  // 0x1f3端口，输入LBA编址的0~7位
  outb(0x1F3, offset);
  // 0x1f4端口，输入LBA编址的8~15位
  outb(0x1F4, offset >> 8);
  // 0x1f5端口，输入LBA编址的16~23位
  outb(0x1F5, offset >> 16);
  // 0x1f6端口，低四位为输入LBA编址的24~27位，高四位为读取方式
  // 高3位111代表使用LBA，101代表使用CHS，第4位为0代表主硬盘，为1代表从硬盘
  // 这里0xe代表使用LBA编址并从主硬盘读取数据
  outb(0x1F6, (offset >> 24) | 0xE0);
  // 0x1f7端口，硬盘控制端口，0x20为读硬盘指令
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  // 读取一扇区的数据放入dst中
  // insl相当于insd，一双字为单位进行读取，即一次读取32位，故参数cnt为SECTSIZE/4
  insl(0x1F0, dst, SECTSIZE/4);
}
```

Boot部分所需ELF文件头的数据解析

- ELF文件头格式解析

```
//ELF 文件头格式
// File header
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
```

```
  ushort machine;
  uint version;
  uint entry;  //程序入口地址
  uint phoff;  //第一个程序段头部在文件中的偏移位置
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum; //程序段数目
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};
```

- 程序段头部格式解析

```
// Program section header
struct proghdr {
  uint type;
  uint off;    // 程序段在文件中的偏移量
  uint vaddr;
  uint paddr;  // 程序段载入内核的内存物理地址
  uint filesz; // 程序段在文件中的大小
  uint memsz;  // 程序段在内存中的大小
  uint flags;
  uint align;
};
```

关于内核ELF文件内存地址的一些总结

- 内核程序各代码地址由链接脚本指定，使用重定向符号0x80100000，即每条语句的地址为偏移量+0x80100000
- 在链接脚本中指定了入口地址为_start，程序实际入口为entry
- 引导程序将内核载入物理地址0x100000
- _start=V2P_WO(entry)，宏V2P_WO作用是将输入参数减去0x80000000，在分页机制启动前完成内存地址的转换
- 故内核程序的入口地址并非生成的虚拟地址，而是物理地址，从而实现了在还未开启分页功能时引导程序能够正确跳转进入口程序

内核初始化

涉及文件

- entry.S
- main.c

流程

entry.S

1. 扩展页大小至4MB，使得一页能容纳整个内核程序
2. 设置页表目录项地址至CR3寄存器

3. 启动分页机制
4. 分配栈空间
5. 跳转至main.c中的main函数，使用c语言完成剩下启动工作

main.C

```
int main(void)
{
  //内存初始化，设备初始化，子系统初始化
  kinit1(end, P2V(4*1024*1024)); // phys page allocator 分配物理页面
  kvmalloc();      // kernel page table 内核页表分配
  mpinit();        // detect other processors 检测其他处理器
  lapicinit();     // interrupt controller initialization 中断控制器初始化
  seginit();       // segment descriptors initialization 段描述符号初始化
  picinit();       // disable pic initialization 停用图片
  ioapicinit();    // another interrupt controller initialization 另一个中断控制器
初始化
  consoleinit();   // console hardware initialization 控制台硬件初始化
  uartinit();      // serial port initialization 串行端口初始化
  pinit();         // process table initialization 进程表初始化
  tvinit();        // trap vectors initialization 中断向量初始化
  binit();         // buffer cache initialization 缓冲区缓存初始化
  fileinit();      // file table initialization 文件表初始化
  ideinit();       // disk initialization 磁盘初始化
  startothers();   // start other processors 启动其他处理器
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  //创建第一个进程（所有进程的父进程），完成CPU设置并调度进程
  userinit();      //完成初始化后,调用userinit()创建第一个进程
  mpmain();        // finish this processor's setup 完成此处理器的设置，开始调度进程
}
```

如上述代码所示，main.c通过调用其他组件提供的初始化API完成整个内核的初始化工作

自旋锁组件

介绍

概念

是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

特点

- 自旋锁是忙等锁，当申请资源未释放时，当前进程不会睡眠而是不断循环查看自旋锁是否是否
- 被自旋锁保护的临界区代码执行时不能睡眠，不能被抢占，也不能被其他中断打断

死锁

自旋锁容易引发死锁问题

- 中断处理程序引发的死锁

    - 引发原因
      当某个进程申请占有了一个自旋锁，然后触发中断处理程序，该中断处理程序同样使用该自旋锁，导致中断处理无法结束，造成死锁
    - 解决方案
      在完成加锁操作前，关闭中断，使得在申请玩自旋锁后的操作中不会触发中断处理程序
- 同一CPU连续申请同一自旋锁造成的死锁
    - 引发原因
      当某一CPU申请占有一个自旋锁A后，在未释放锁A的情况下再次申请锁A，就会导致始终处在自旋状态，造成死锁
    - 解决方案
      在完成上锁操作前判断该CPU是否已申请过该锁，如有，则抛出panic异常
- 其他由于代码编写问题造成的死锁

结构

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

- 其中locked用于判断是否上锁
- 剩余部分用于debug，提供debug信息，包括以下信息
    - 锁的名字
    - 占有该锁的cpu
    - 函数调用栈

提供API

- initlock函数
    - 函数原型

      ```
      void initlock(struct spinlock *lk, char *name)
      ```

    - 初始化自旋锁，为自旋锁命名
- acquire函数
    - 函数原型

      ```
      void acquire(struct spinlock *lk)
      ```

    - 申请占有自旋锁lk
    - 函数解析

      ```
      // Acquire the lock.
      // Loops (spins) until the lock is acquired.
      // Holding a lock for a long time may cause
      // other CPUs to waste time spinning to acquire it.
      void
      ```

```
acquire(struct spinlock *lk)
{
  // 关闭中断
  pushcli(); // disable interrupts to avoid deadlock.
  // 如果锁lk已被当前cpu占有，抛出panic异常
  if(holding(lk))
    panic("acquire");

  // 进入自旋状态直到锁被释放
  // xchg函数是原子性操作，交换两个参数的值，并返回第一个参数的旧值
  // 当返回值为0时，代表锁已释放，并通过与1交换上锁
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  // 告知编译器避免对这段代码进行乱序优化，保证临界资源在上锁后访问
  __sync_synchronize();

  // 获取当前使用的cpu
  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  // 获取递归调用栈
  getcallerpcs(&lk, lk->pcs);
}
```

- release函数
  - 函数原型

    ```
    void release(struct spinlock *lk)
    ```

  - 释放自旋锁
  - 函数解析

```
// Release the lock.
void
release(struct spinlock *lk)
{
  // 若锁已被释放，抛出panic异常
  if(!holding(lk))
    panic("release");
  // 清空占有锁的cpu及递归调用栈
  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
```

```
  // not be atomic. A real OS would use C atomics here.
  // 使用汇编保证解锁操作是原子的
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );
  // 开启中断
  popcli();
}
```

辅助函数

- pushcli和popcli函数
  - 函数原型

    ```
    void pushcli(void)
    ```

    ```
     void popcli(void)
    ```

  - 用于代替cli，sti指令，封装了cli和sti指令
  - 保证了当pushcli指令和popcli指令一一对应且当前cpu为实现关闭中断的cpu时才进行sti操作，避免了在自旋锁还处在申请状态下打开中断
  - 对不恰当的调用顺序进行检测，并抛出异常

```
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  // 获取eflags
  eflags = readeflags();
  // 关闭中断，将eflags寄存器的IF位置0
  cli();
  if(mycpu()->ncli == 0)
    // 当第一次调用cli指令且IF位未置0时，对mycpu()->intena做上标记
    mycpu()->intena = eflags & FL_IF;
  // cli次数加1
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  // 若中断已经开启，IF标志位为1，抛出panic异常
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  // popcli操作次数大于pushcli次数，抛出panic异常
  if(--mycpu()->ncli < 0)
    panic("popcli");
  // 保证了当pushcli指令和popcli指令一一对应且当前cpu为实现关闭中断的cpu时才进行sti操作
  // 避免了在自旋锁还处在申请状态下打开中断
```

```
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

- hoding函数
  - 函数原型

    ```
    int holding(struct spinlock *lock)
    ```

  - 用于判断自旋锁lock是否被当前cpu占有
- getcallerpcs函数
  - 函数原型

    ```
    void getcallerpcs(void *v, uint pcs[])
    ```

  - 用于获取递归调用栈的信息，将其储存至pcs数组中
  - 函数解析

```
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;
  // 函数调用时栈指针结构
  // ebp栈底指针
  // esp栈顶指针
  // 本函数栈指针结构
  //
  //       较早的栈
  // +12   参数pcs
  // +8    参数v
  // +4    返回地址
  // 0     旧的ebp
  //       当前函数使用的栈
  //       esp
  //
  // 故地址v=epb+8byte
  // 又由于uint* 偏移一个单位即偏移8byte
  // 故 ebp = (uint*)v - 2;
  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;
    // ebp[1]即为当前函数的返回地址
    // ebp[0]为返回的函数的ebp指针
    // 故使用该循环可以自底向上获取递归调用顺序
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  // 剩余部分用0填充
  for(; i < 10; i++)
    pcs[i] = 0;
}
```

附录

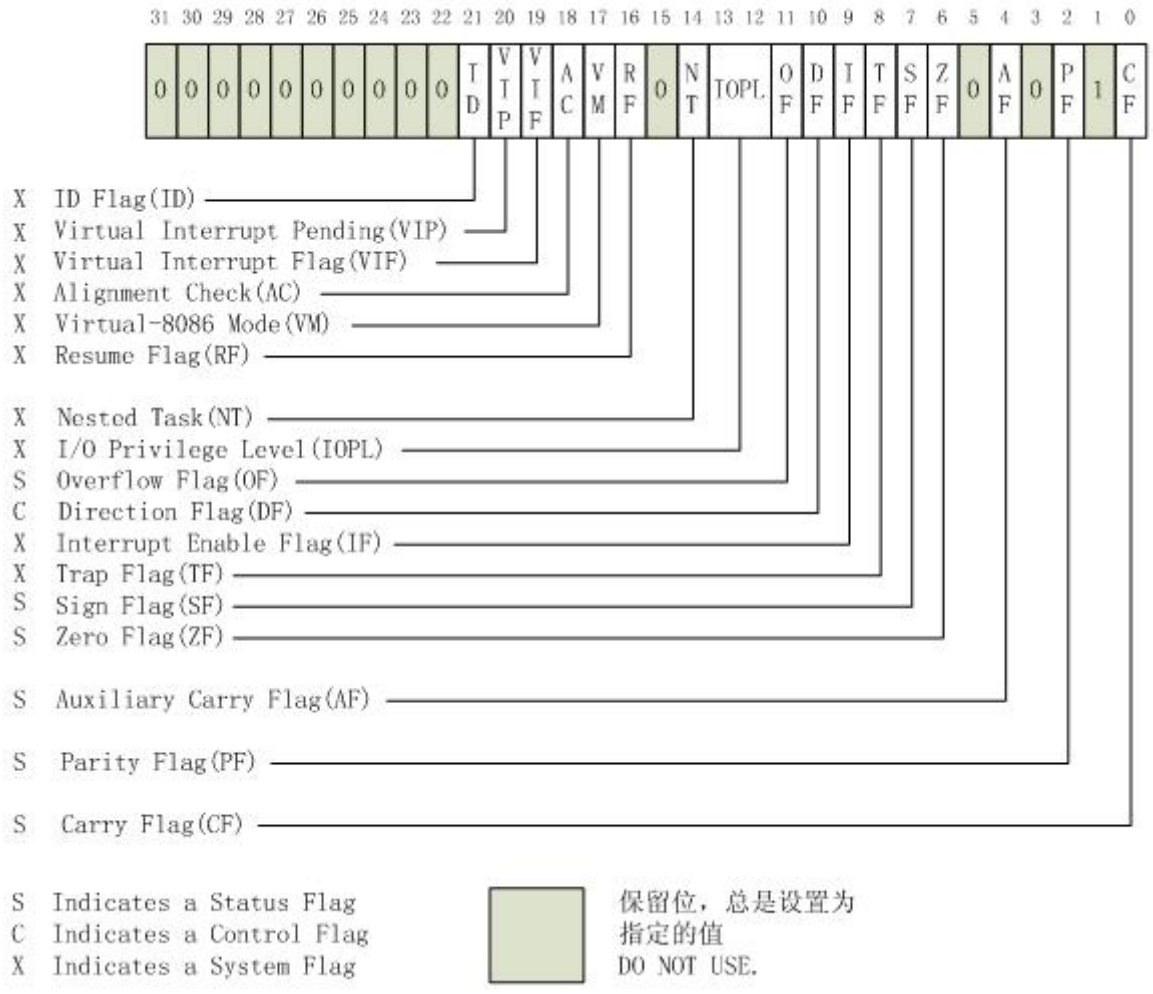
xchg函数

- xchgl汇编指令是一个原子操作，用于置换两个操作数的值
- 两个操作数至少有一个是寄存器操作数
- 但xchgl不能保证在多核情况下也是原子性操作
- 故在执行xchgl前须使用lock汇编指令保证该指令在多核情况下也是原子操作

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  // lock操作，保证xchgl在多核情况下是原子操作
  // 内联汇编输出参数
  // +m:内存操作数，可读写
  // =a, 寄存器操作数，将eax的值读入
  // 1:输入操作数，该操作数使用的寄存器与%1参数使用的寄存器一致
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

EFLAGS寄存器



内存管理

内存分配器

涉及文件

- kalloc.c

功能

- 对空闲内存进行管理
- 封装了内存分配和释放操作，用户只需使用虚拟内存即可完成对物理内存的分配

实现思想

- 使用结构体

  ```
  // 可使用内存链表节点
  // 同时也可作为分配内存的首地址
  // 未使用时储存下一内存链表节点地址
  struct run {
    struct run *next;
  };
  ```

  作为内存分配的基本单元

- 使用结构体

  ```
  // 管理空闲内存的结构
  struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;
  } kmem;
  ```

  作为管理内存的结构

  - 该结构维护一个空闲内存链表
  - 同时提供一个自旋锁保护空闲链表
  - 并且提供是否启用自旋锁的选项

提供API

- kinit1函数
  - 函数原型

    ```
    void kinit1(void *vstart, void *vend)
    ```

  - 在未启动其他CPU核心时使用
  - 释放部分内存供初始化使用
- kinit2函数

    - 函数原型

      ```
      void kinit2(void *vstart, void *vend)
      ```

    - 多核启动结束后以及加载完完整页表后调用
    - 释放剩余内存
    - 由于此时已启用多CPU核心，故启用自旋锁维护空闲内存链表
- kfree函数
    - 函数原型

      ```
      void kfree(char *v)
      ```

    - 释放4kb内存
    - 函数解析

      ```
      //PAGEBREAK: 21
      // Free the page of physical memory pointed at by v,
      // which normally should have been returned by a
      // call to kalloc().  (The exception is when
      // initializing the allocator; see kinit above.)
      // 释放4kb内存
      void
      kfree(char *v)
      {
        struct run *r;
        // 判断释放内存地址是否合法，end为内核尾后虚拟地址，PHYSTOP为最大物理地址
        if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
          panic("kfree");

        // Fill with junk to catch dangling refs.
        // 清空内存内容
        memset(v, 1, PGSIZE);

        // 若使用自旋锁维护空闲内存列表
        // 在访问kmem需要完成申请自旋锁
        // 结束访问后释放
        if(kmem.use_lock)
          acquire(&kmem.lock);
        r = (struct run*)v;
        // 将释放的内存添加到列表头部
        r->next = kmem.freelist;
        kmem.freelist = r;
        if(kmem.use_lock)
          release(&kmem.lock);
      }
      ```

- kalloc函数
    - 函数原型

      ```
      char* kalloc(void)
      ```

    - 分配4kb内存
    - 返回分配内存的首地址
    - 函数解析

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  // 从可使用列表中分配内存并返回内存首地址
  // 若无可使用内存，则返回0
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

虚拟内存管理

涉及文件

- vm.c

页表

- 操作系统通过页表机制实现了对内存空间的控制。页表使得 xv6 能够让不同进程各自的地址空间映射到相同的物理内存上，还能够为不同进程的内存提供保护。 除此之外，我们还能够通过使用页表来间接地实现一些特殊功能。xv6 主要利用页表来区分多个地址空间，保护内存。另外，它也使用了一些简单的技巧，即把不同地址空间的多段内存映射到同一段物理内存（内核部分），在同一地址空间中多次映射同一段物理内存（用户部分的每一页都会映射到内核部分），以及通过一个没有映射的页保护用户栈。
- xv6使用x86架构提供的分页硬件来完成虚拟地址到物理地址转换
- 涉及寄存器
  - CR3寄存器
    该寄存器存储页表目录项的地址以及一些配置信息，且由于CR3寄存器只使用高20位来存放地址，故页表目录入口地址必须4K对齐
  - CR4寄存器
    将该寄存器的PSE位置1，可运行硬件使用超级页
- x86架构的页表结构
  - x86架构使用两级页表
    - 第一级为页表目录项，其指向存放页表项的物理地址，长度为4Kb，一个页表项占4byte，故一个页表目录项指向的内存可存放1024个页表项
    - 页表目录项同样存放在4Kb的内存中，一个页表目录项占4byte，共由1024个页表项，故32位x86计算机支持最大内存为1024*1024*4Kb=4Gb内存
  - x86硬件寻址方式

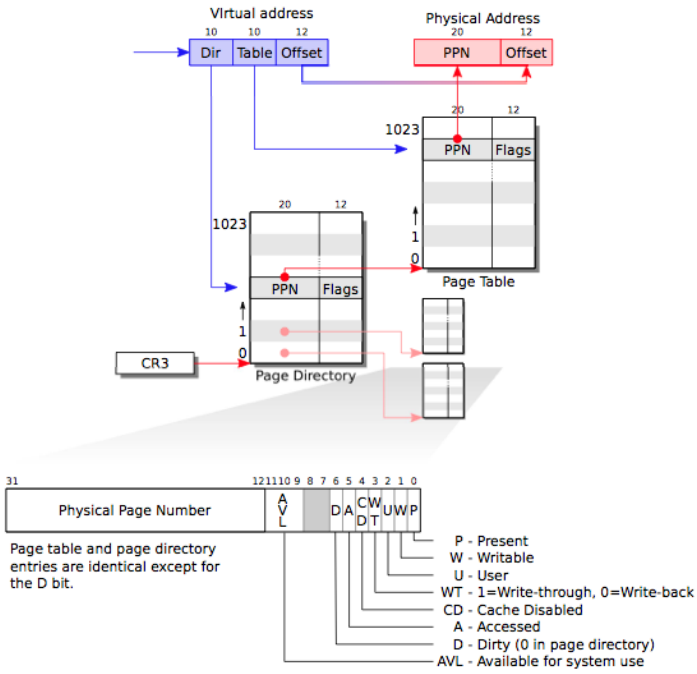


Figure 2-1. x86 page table hardware.

如图，对于一个32位的虚拟地址

1. 分页硬件从CR3寄存器存储的页表目录入口地址找到页表目录，并根据虚拟地址高10位找到在页表目录中对应的页表目录项
2. 分页硬件再根据虚拟地址中10位找到页表项，得到对应页的物理地址
3. 最后根据虚拟地址低12位的偏移量确定具体内存地址

- 页表项格式

31–22	21	20–13	12	11–9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]				Ignored					PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame	Reserved (must be 0)	Bits 39:32 of address[2]	PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table				Ignored		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored													0	PDE: not present
Address of 4KB page frame				Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored													0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

如图为各种情况下页表项的格式

xv6内存映射模型

定义常量及含义

- KERNBASE 0x80000000 代表此地址以上的虚拟地址由内核使用
- EXTMEM　0x100000　　扩展内存的起始地址
- DEVSPACE 0xFE000000 其他设备使用的高位地址
- PHYSTOP　0xE000000　假定物理内存的大小
- data　　在链接脚本中指定　内核数据段起始地址

内存映射

虚拟地址范围	映射的物理地址	功能
0~KERNBASE	具体映射的物理内存由内核分配	供用户进程使用
KERNBASE~KERNBASE+EXTMEM	0~EXTMEM	供IO设备使用
KERNBASE+EXTMEM~data	EXTMEM~V2P(data)	存放内核代码段及只读数据段
data~KERNBASE+PHYSTOP	V2P(data)~PHYSTOP	内核数据段及其他未分配内存
0xfe000000~0	直接映射	供其他设备使用，如ioapic

xv6内存管理机制

- xv6使用段页式存储机制进行内存管理
- 内核具有所有内存的权限，并对空闲内存进行分配
- 每一个进程拥有对应的页表，当切换到某一进程时，为该进程设置段描述符，并加载该进程使用的页表
- xv6使用kalloc和kfree函数完成物理内存分配，上层内存管理接口只需关注虚拟内存地址
- 内核根据用户进程的页表调用kalloc和kfree函数来完成内存分配和释放工作
- 每个用户进程的内存都是从0开始线性编址的
- KERNBASE以上的虚拟地址为内核所独有
- xv6内存管理系统还提供了loaduvm接口从磁盘文件加载数据至内存，作为文件系统与内存管理系统的桥梁
- xv6还提供了copyuvm接口，用于复制一个新的页表并分配布局和旧的完全一样的新内存，用于实现fork为父进程创建子进程

使用结构体

- 内核内存映射

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

存放内核使用的虚拟内存到物理内存的映射方式

- 用户进程

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
```

```
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

存放用户进程信息

提供API

初始化API

- API列表

```
void            seginit(void);
void            kvmalloc(void);
void            inituvm(pde_t*, char*, uint);
```

- seginit函数
  - 每个cpu核心启动时调用
  - 为每个cpu设置内核态的GDT表并加载它
- kvmalooc函数
  - 调用setupkvm为内核建立页表，将内核页表目录入口地址存至kpgdir
  - 调用switchkvm加载内核页表
- inituvm函数
  - 为第一个用户进程创建页表
  - 先为该进程分配内存并建立内存映射
  - 再将该进程代码赋值到分配的内存处

内存管理API

- API列表

```
pde_t*          setupkvm(void);
char*           uva2ka(pde_t*, char*);
int             allocuvm(pde_t*, uint, uint);
int             deallocuvm(pde_t*, uint, uint);
void            freevm(pde_t*);
int             loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint
offset, uint sz);
pde_t*          copyuvm(pde_t*, uint);
void            switchuvm(struct proc *p)
void            switchkvm(void);
int             copyout(pde_t*, uint, void*, uint);
void            clearpteu(pde_t *pgdir, char *uva);
```

- setupkvm函数

    - 设置内核页表并返回内核页表目录入口地址
    - 通过调用mappages完成kmap内的所有内核映射关系
- uva2ka函数
    - 将用户进程的虚拟地址转换为内核虚拟地址
    - 物理地址到对应的内核虚拟地址可通过P2V宏来实现
    - 物理地址可通过用户进程的页表来得到
    - 从而实现将用户进程的虚拟地址到内核虚拟地址的转换
- allocuvm函数
    - 进程需求内存变多时为其分配内存
    - 函数解析

```
// Allocate page tables and physical memory to grow process from oldsz
to
// newsz, which need not be page aligned.  Returns new size or 0 on
error.
// 为用户进程分配更多内存，将进程内存从oldsz增加至newsz
// 该函数基于以下假定：
// 1. 进程内存虚拟地址从0开始编址
// 2. 进程内存呈线性编址
// 3. 进程使用内存大小等同与进程使用的内存的最高地址
// 参数说明：
// pdgdir：用户进程的页表目录入口地址
// oldsz：进程原大小
// newsz：新的进程大小
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  // 对进程新大小合法性进行检测
  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  // 由于内存是以4kb为单位分配的
  // 故若oldsz不是4k对齐的，则必定多分配了内存
  // 故向上4k对齐
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    // 分配新的4k内存
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    // 初始化内存
    memset(mem, 0, PGSIZE);
    // 为页表目录项建立虚拟内存到新分配的内存间的映射关系
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
```

```
      return 0;
    }
  }
  // 返回新的进程大小以便判断内存是否分配成功
  return newsz;
}
```

- deallocuvm函数
  - 进程释放部分内存时调用此函数，属于allocuvm的相反操作
  - 函数解析

```
// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
// 释放进程内存，将进程内存从oldsz降至newsz，allocuvm的反操作
// 该函数基于以下假定：
// 1. 进程内存虚拟地址从0开始编址
// 2. 进程内存呈线性编址
// 3. 进程使用内存大小等同与进程使用的内存的最高地址
// 参数说明：
// pdgir：进程页表目录入口地址
// oldsz：进程原大小
// newsz：新的进程大小
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  // 由于释放内存也是以4k为单位的
  // 若向下对齐，会将还需要的内存释放
  // 向上4k对齐
  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    // 获取待释放的页表项地址
    pte = walkpgdir(pgdir, (char*)a, 0);
    // 返回0代表当前页表目录项对应的页表不存在
    // 故直接跳到下一条目录项
    if(!pte)
      a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
    // 若该页表项处于使用状态，将其对应的物理内存释放
    // 并将该页表项置空
    else if((*pte & PTE_P) != 0){
      // 获取页表项对应的物理地址
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      // 将物理地址转换为虚拟地址并释放
      char *v = P2V(pa);
      kfree(v);
      // 清空页表项
      *pte = 0;
```

```
    }
  }
  return newsz;
}
```

- freevm函数
  - 释放整个页表以及分配的所有物理内存
  - 函数解析

```
// Free a page table and all the physical memory pages
// in the user part.
// 释放整个页表以及分配的所有物理内存
// 参数说明：
// pgdir：待释放的页表目录入口地址
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  // 释放页表分配的所有内存
  // 由于在函数内有对内存是否分配进行检测，保证内存不会重复释放
  // 故deallocuvm的oldsz可直接为虚拟地址最大值
  deallocuvm(pgdir, KERNBASE, 0);
  // 释放所有页表项占据的内存
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      // 获取页表项的物理地址
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  // 释放页表目录的物理地址
  kfree((char*)pgdir);
}
```

- loaduvm函数
  - 将文件程序段装载至虚拟地址为addr的内存中
  - 函数解析

```
// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
// 加载程序段进内存中
// 参数说明
// pgdir：页表目录项入口地址
// addr：加载的目标虚拟地址，需要4k对齐
// ip：待加载文件的inode
// offset：读取文件的起始位置偏移值
// sz：读取的大小
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint
sz)
{
```

```
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  // 创建页表项
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    // 获取页表项指向的物理地址
    pa = PTE_ADDR(*pte);
    // 剩余数据大小未满PGSIZE，就读取剩余的数据
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    // 从磁盘中读取文件数据存放至物理地址pa处
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

- copyuvm函数
  - 根据给定进程页表复制进程所有内容
  - 用于父进程创建子进程使用
  - 函数解析

```
// Given a parent process's page table, create a copy
// of it for a child.
// 根据给定进程页表复制进程所有内容
// 用于父进程创建子进程使用
// 参数说明：
// pgdir：父进程页表
// sz：进程大小
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  // 创建新的页表目录项
  // 此时页表目录映射关系为内核的映射关系
  // 之后重新建立页表的映射关系
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    // 获取父进程页表项地址
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    // 获取父进程页表项对应物理地址
    pa = PTE_ADDR(*pte);
```

```
    // 获取父进程页表项属性信息
    flags = PTE_FLAGS(*pte);
    // 为子进程分配内存
    if((mem = kalloc()) == 0)
      goto bad;
    // 将父进程页表项对应的内存内的数据复制到子进程的对应内存中
    memmove(mem, (char*)P2V(pa), PGSIZE);
    // 建立子进程的内存映射关系
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;
// 内存分配错误，释放子进程已分配内存
bad:
  freevm(d);
  return 0;
}
```

- switchuvm函数
  - 切换用户进程时调用
  - 切换cpu的任务状态段
  - 切换至进程p使用的页表，向cr3寄存器加载储存进程页表目录入口地址
  - 页表入口地址储存在p->pgdir
  - 函数解析

```
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  // 判断该进程指针是否合法
  if(p == 0)
    panic("switchuvm: no process");
  // 判断栈指针是否合法
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  // 判断页表目录入口地址是否设置
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  // 关闭中断
  pushcli();
  // 为该进程建立任务状态段的段描述符
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  // 将该段的s位设置为系统位
  mycpu()->gdt[SEG_TSS].s = 0;
  // 设置cpu目前的任务状态值
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
```

```
  // 加载段描述符
  ltr(SEG_TSS << 3);
  // 加载进程的页表
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

- switchkvm函数
  - 切换至内核页表，向cr3寄存器加载储存内核页表目录入口地址的指针kpgdir
- copyout函数
  - 从虚拟地址p中拷贝len个字节至用户进程的va地址中
  - 函数解析

```
// Copy len bytes from p to user address va in page table pgdir.
// Most useful when pgdir is not the current page table.
// uva2ka ensures this only works for PTE_U pages.
// 从虚拟地址p中拷贝len个字节至用户进程的va地址中
// 参数说明：
// pgdir：用户进程页表
// va：用户进程虚拟地址
// p：待复制数据的虚拟地址
// len：复制字节长度
int
copyout(pde_t *pgdir, uint va, void *p, uint len)
{
  char *buf, *pa0;
  uint n, va0;

  buf = (char*)p;
  while(len > 0){
    va0 = (uint)PGROUNDDOWN(va);
    // 使用uva2ka函数保证若va地址运行在内核态，返回0
    // 从而避免向内核拷贝内容，破坏内核
    pa0 = uva2ka(pgdir, (char*)va0);
    if(pa0 == 0)
      return -1;
    // 计算要拷贝的字节长度
    n = PGSIZE - (va - va0);
    if(n > len)
      n = len;

    // va - va0为拷贝内存地址在该页中的偏移量
    // pa0为该页的物理地址
    // 通过memmove函数从buf（p）指针处拷贝n字节数据到va对应的内核虚拟内存上
    memmove(pa0 + (va - va0), buf, n);
    len -= n;
    buf += n;
    va = va0 + PGSIZE;
  }
  return 0;
}
```

- clearpteu函数
  - 将虚拟地址uva对应的页表项的U位置0
  - 即让该页表仅运行特权级为0，1，2的用户访问
  - 用于创建不可被用户进程访问的页

辅助函数

- 函数列表

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc);
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

- walkpgdir函数
  - 返回虚拟地址所对应的页表项地址
  - 当alloc=1时，若虚拟地址对应的页表不存在，创建该页表项并分配内存
  - 函数解析

```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
// 返回页表项所在地址
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  // 页表项属于第几个页目录项
  pde = &pgdir[PDX(va)];
  // 若当前页目录项有效，直接得到对应页表项地址
  if(*pde & PTE_P){
    // 页表项起始位置对应的物理内存的虚拟地址
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    // 如果alloc=1，为页表项分配内存
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    // 初始化内存
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    // 完成页目录项初始化
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

- mappages函数
  - 为页表创建某一段虚拟内存到物理内存的映射

- 函数解析

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
// 为具有一段长度的内存创建页表项
// 参数说明
// pgdir:页表项地址
// va:虚拟地址
// size:内存大小
// pa:映射的起始物理内存
// perm:设置页表项的属性
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  // 完成4k对齐，舍去虚拟地址中的低位值
  a = (char*)PGROUNDDOWN((uint)va);
  // 同样对最末一项完成4k对齐
  // last为最后一项的起始地址
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  // 使用向下对齐完成4k对齐的原因：
  //       分配充足的内存避免内存不足
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    //配置页表项属性及对应物理地址，这里使用4kb页表，故没有将PS位置1
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

程序运行

涉及文件

- exec.c

流程

1. 接受程序路径及程序命令行参数作为参数
2. 根据文件路径获取文件inode
3. 检测inode指向的文件是否是可执行的ELF文件
4. 为该文件创建页表目录
5. 通过页表目录为该文件分配内存并将文件从文件系统中装载入内存
6. 额外分配两个页面，第二个页面作为程序栈，第一个页面作为程序代码段和程序栈之间的缓冲，避免栈溢出及其他安全问题
7. 将程序命令行参数加载至内存对应位置

8. 初始化程序栈的其他部分
9. 调整栈指针
10. 保存调试信息
11. 设置进程信息
12. 切换进程页表及任务状态段
13. 释放旧页表内存
14. 完成程序启动

提供API

- exec函数
  - 函数原型

    ```
    int exec(char *path, char **argv)
    ```

  - 功能：该函数根据运行程序的路径及命令行参数，将程序从文件系统中加载至内存并完成进程切换
  - 作为内存管理、进程管理及文件系统三者间的粘合剂，完成一个初始化一个待运行的程序
  - 函数解析

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  // 获取当前cpu运行的进程
  struct proc *curproc = myproc();

  // 使用文件系统前调用
  // 内部使用自旋锁保护
  // 保证同一时间只有一个cpu调用文件系统接口
  begin_op();

  // 获取路径path所对应文件的inode
  if((ip = namei(path)) == 0){
    // 如果该文件不存在，结束文件系统调用
    end_op();
    // 抛出异常
    cprintf("exec: fail\n");
    return -1;
  }
  // 为inode加锁
  // 如果ip->valid==0，将inode加载至内存
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  // 检查文件的ELF头，判断是否是可执行的ELF头文件
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
```

```
    goto bad;

  // 为新进程创建页表
  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;

  // 加载文件所有程序段至内存中
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    // 检测文件程序头是否损坏
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    // 检查文件程序头类型是否需要装载
    if(ph.type != ELF_PROG_LOAD)
      continue;

    // 检查程序头数据是否有误
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    // 为该段分配内存
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    // 检查段起始地址
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    // 将该段加载至分配的内存中
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  // 更新inode信息并解锁inode
  iunlockput(ip);
  // 结束对文件系统的调用
  end_op();
  ip = 0;

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  // 再分配两页内存给进程
  // 其中第一页不允许用户访问，避免栈溢出或是其他安全问题
  // 第二页作为用户使用的栈
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  // 栈是向下扩展的
  // 故栈指针指向最后一个地址
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  // 加载程序的命令行参数
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
```

```
    // 将参数拷贝至对应内存位置
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  // 栈顶为0xffffffff(-1)，代表函数递归调用到头了
  ustack[0] = 0xffffffff;  // fake return PC
  // 保存命令行参数个数
  ustack[1] = argc;
  // 保存命令行参数argv起始地址
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  // 调整栈指针
  sp -= (3+argc+1) * 4;
  // 加载用户栈至内存中
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  // 保存程序名作为调试信息
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;

  // 设置进程状态
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  // 切换进程
  switchuvm(curproc);
  // 释放旧页表内存
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

进程管理

xv6 操作系统实现了一个基于进程 (**没有实现线程**) 的简单进程管理机制。

一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程，操作系统中资源分配的基本单位是进程，在引入了线程的操作系统中，线程是处理机调度的基本单位，而在`xv6 os`中，进程是调度的基本单位。

`xv6`中进程和`CPU`的数据结构见`proc.h`

数据结构

在proc.h头文件中定义了与进程管理有关的数据结构`procstate、context、proc`和`cpu`。

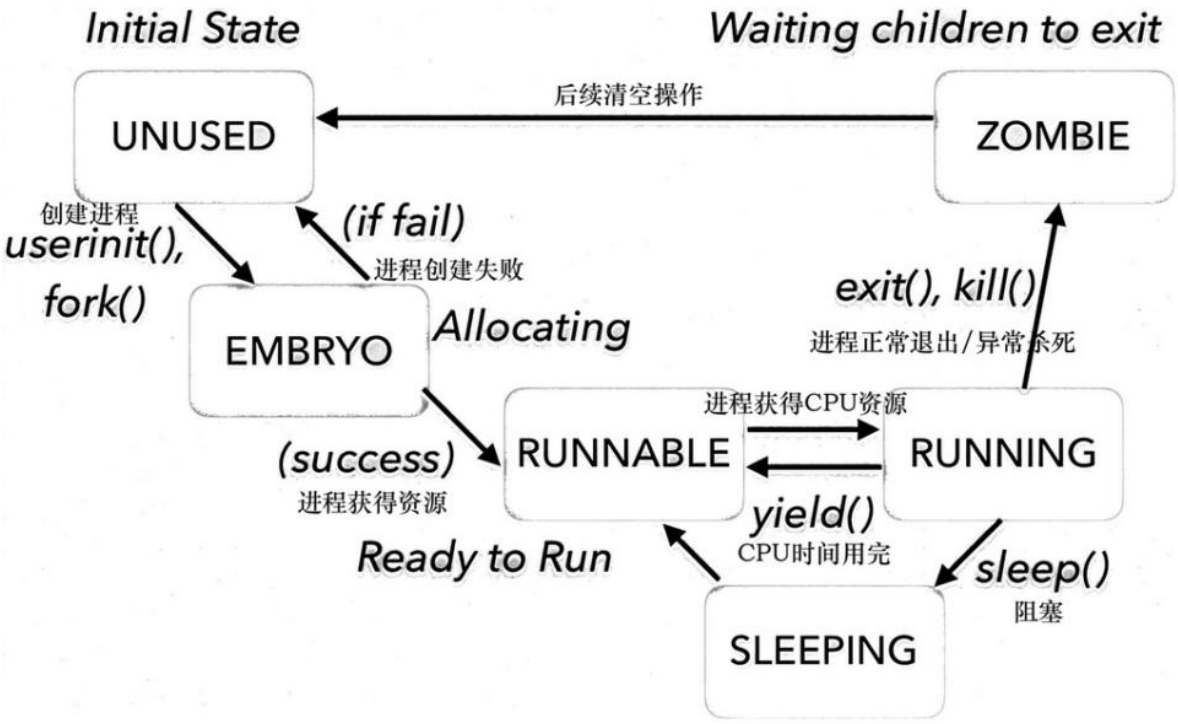
进程状态

为准确描述进程的状态，在`proc.h`中定义了`procstate(State of process)`枚举类型，描述`xv6 OS`中进程的六种状态：`EMBRO, SLEEPING, RUNNABLE, RUNNING`和`ZOMBIE`六种状态，分别为未使用态、初始态、等待态、就绪态、运行态和僵尸态。

```
/*枚举类型procstate,分别代表一个进程的六种状态 未分配、创建态、阻塞态、就绪态、运行态、结束态
*/
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

进程状态简介

- *UNUSED*：进程未被创建（PCB空闲）时的状态；
- *EMBRYO*：代表进程的创建态；
- *SLEEPING*：进程由于等待除处理机之外的资源，进入阻塞状态，即等待态/睡眠态/阻塞态；
- *RUNNABLE*：进程获得了除处理机之外的资源，等待分配处理机时间片，处于就绪态；
- *RUNNING*：进程获得处理机，正在执行进程，即运行态；
- *ZOMBIE*：进程执行完毕，即结束态。



进程状态转换示意图

状态转换情况

1. *UNUSED→EMBRYO*：执行一个程序，创建一个子进程。
2. *EMBRYO→RUNNABLE*：当操作系统完成了进程创建的必要操作，并且当前系统的性能和虚拟内存的容量均允许。
3. *RUNNING→ZOMBIE*：当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结。
4. *RUNNBALE→RUNNING*：当一个进程通过`scheduler()`CPU调度获得CPU时间片，则进程进入RUNNING态开始执行进程程序段。
5. *RUNNING→SLEEPING*：当一个进程程序段执行过程中需要等待某资源或等待某一事件发生时，应当让出CPU资源，从而进入SLEEPING状态，等待事件的发生/资源的获得。
6. *SLEEPING→RUNNABLE*：当一个进程程序段等待的资源已经就绪或等待的某一事件发生时，应当进入CPU时间片等待队列，由SLEEPING态转为RUNNABLE态，等待CPU的调度 。

进程结构体

```
struct proc {
  uint sz; // 进程的内存大小（以byte计）
  pde_t* pgdir; // 进程页路径的线性地址。
  char *kstack; // 进程的内核栈底
  enum procstate state; // 进程状态（包括六种UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE）
  int pid; // 进程ID
  struct proc *parent; // 父进程
  struct trapframe *tf; // 位于x86.h，是中断进程后，需要恢复进程继续执行所保存的寄存器内容。
  struct context *context; // 切换进程所需要保存的进程状态。切换进程需要维护的寄存器内容，定义在proc.h中。
  void *chan;  // 不为0时，是进程睡眠时所挂的睡眠队列
  int killed; // 当非0时，表示已结束
  struct file *ofile[NOFILE]; // 打开的文件列表
  struct inode *cwd; // 进程当前路径
  char name[16]; // 进程名称
};
```

进程上下文切换现场

```
struct context {
  //变址寄存器
  //分别叫做"源/目标索引寄存器"
  //在很多字符串操作指令中，DS:ESI指向源串,而ES:EDI指向目标串.
  uint edi;
  uint esi;
  uint ebx;//基地址寄存器，在内存寻址时存放基地址。
  uint ebp;//寄存器存放当前线程的栈底指针
  //寄存器存放下一个CPU指令存放的内存地址，当CPU执行完当前的指令后
  //从EIP寄存器中读取下一条指令的内存地址，然后继续执行。
  //EIP不会被显式设置，它在对swtch()函数的call和ret时被设置
  uint eip;
};
```

`struct context`实际上是五个寄存器的值，也就是在上下文切换时，主要做的事情就是保存并更新寄存器值。同时根据惯例，调用者会保存`%eax,%ecx,%edx`的值。xv6在进程调度中主要通过切换context上下文结构进行的，正常情况下c函数被调用时编译器会自动为其加上保存context的代码。

CPU结构体

```
 struct cpu {
    uchar apicid;                // Local APIC ID 每个cpu都有一个唯一硬件ID，这个ID可以lapicid()函数进行获取，然后存放于这个字段中。
    struct context *scheduler;   // scheduler context，即scheduler运行环境信息
    struct taskstate ts;         // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS];   // x86 global descriptor table
    volatile uint started;       // Has the CPU started?
    int ncli;                    // Depth of pushcli nesting.
    int intena;                  // Were interrupts enabled before pushcli?
    struct proc *proc;           // The process running on this cpu or null
  };
```

XV6操作系统可以运行在使用多核处理机的计算机上，在`proc.c`中使用`struct cpu`来记录每一个CPU状态。

进程创建

第一个进程`initproc`

```
int main(void)
{
  //内存初始化，设备初始化，子系统初始化
  kinit1(end, P2V(4*1024*1024)); // phys page allocator 分配物理页面
  kvmalloc();      // kernel page table 内核页表分配
  mpinit();        // detect other processors 检测其他处理器
  lapicinit();     // interrupt controller initialization 中断控制器初始化
  seginit();       // segment descriptors initialization 段描述符号初始化
  picinit();       // disable pic initialization 停用图片
  ioapicinit();    // another interrupt controller initialization 另一个中断控制器初始化
  consoleinit();   // console hardware initialization 控制台硬件初始化
  uartinit();      // serial port initialization 串行端口初始化
  pinit();         // process table initialization 进程表初始化
  tvinit();        // trap vectors initialization 中断向量初始化
  binit();         // buffer cache initialization 缓冲区缓存初始化
  fileinit();      // file table initialization 文件表初始化
  ideinit();       // disk initialization 磁盘初始化
  startothers();   // start other processors 启动其他处理器
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  //创建第一个进程（所有进程的父进程），完成CPU设置并调度进程
  userinit();      //完成初始化后,调用userinit()创建第一个进程
  mpmain();        // finish this processor's setup 完成此处理器的设置，开始调度进程
}
```

在`main.c`中的主函数，在完成了初始化一些设备和子系统后，它通过调用`userinit()`建立了第一个进程，以下着重分析第一个进程的创建过程，该进程是所有进程的父进程。

```
//PAGEBREAK: 32
// Set up first user process.创建第一个用户进程
//这个函数只调用一次，创建的init process是所有进程的父进程
/*
 init进程的内存布局：
 +--------------------+ 4GB
 |                    |
 |                    |
 |                    |
 +--------------------+ KERNBASE+PHYSTOP(2GB+224MB)
 |                    |
 |   direct mapped    |
 |   kernel memory    |
 |                    |
 +--------------------+
 |    Kernel Data     |
 +--------------------+ data
 |    Kernel Code     |
 +--------------------+ KERNLINK(2GB+1MB)
 |   I/O Space(1MB)   |
 +--------------------+ KERNBASE(2GB)
 |                    |
 |                    |
 |                    |
 |                    |
 |                    |
 |                    |
 |                    |
 |                    |
 +---------+----------+ PGSIZE <-- %esp
 |         v          |
 |       stack        |
 |                    |
 |                    |
 |     initcode.S     |
 +--------------------+ 0  <-- %eip
*/
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];//这是initcode.S
的加载位置和大小

  /* 分配进程数据结构并初始化 */
  p = allocproc(); //在内存中分配一个proc结构，并初始化进程内核堆栈以及一系列内核寄存器

  initproc = p;   //将新分配内存的进程p赋值给initproc，代表所有进程的父进程
    /* 创建页表，将进程的kernel部分页映射进来 */
  if((p->pgdir = setupkvm()) == 0) //创建页表
    panic("userinit: out of memory?"); //若创建失败则可能内存不够，抛出异常
  /*将initcode.S第一个进程的代码加载到进程p中，并分配一页物理内存，将虚拟地址0映射到该物理地
址，实现虚拟地址初始化*/
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE; //设置进程大小为一页
  memset(p->tf, 0, sizeof(*p->tf));
```

```
    /* 设置初始的用户模式状态 */

  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;//cs寄存器指向代码段并处于用户模式
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;//ds寄存器指向数据段并处于用户模式
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;// 允许硬件中断
  p->tf->esp = PGSIZE;// 用户栈大小为1页
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);//获取进程锁，确保原子操作

  p->state = RUNNABLE;  //将进程状态置RUNNABLE

  release(&ptable.lock);//释放进程锁
}
```

`userinit()` 函数通过调用 `allocproc()` 函数在 `ptable` 中找到一个闲置 `PCB(即proc)`，然后初始化该进程的内核栈。内存大小是有限的，因而进程的个数也是有限的，`allocproc()` 函数的作用是从 `ptable` 进程表中寻找到一个空闲的 `PCB`，若找到则将进程状态从 `UNUSED` 置为 `EMBRYO`，为其分配 `pid` 并释放进程表锁，防止死锁或长时间等待；然后分配内核栈，设置进程上下文使新进程从 `forkret` 开始执行，并返回到中断管理 `trapret`，然后返回这个结构体的指针。

在这个寻找与分配的过程中如果没有空闲的 `PCB` 可以使用，或者分配内核栈失败，则返回0代表没有分配 `proc` 结构体失败。

```
//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);//获取进程表锁

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED) //寻找未使用的进程控制块
      goto found;

  release(&ptable.lock); //未找到，释放进程锁
  return 0; //返回0

found:
  p->state = EMBRYO; //设置进程状态为创建态
  p->pid = nextpid++; //设置进程ID（自增）
```

```
  release(&ptable.lock); //释放进程锁

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){ //从内存链表上分配4096字节的一页内存作为内核栈空间
    p->state = UNUSED; //分配失败，重置进程状态
    return 0; //返回0，进程内存分配失败
  }
  sp = p->kstack + KSTACKSIZE; //将栈顶指针设置为内存高地址处

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp; //设置trapframe的栈底指针

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp; //设置进程上下文指针
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret; //init进程将从forkret处开始执行
  return p;
}
```

分配完进程槽以后，需要创建进程的页表，设置进程的若干属性：进程空间，打开中断等，并加载第一个进程的程序`initcode.S`。

```
    /* 创建页表，将进程的kernel部分页映射进来 */
  if((p->pgdir = setupkvm()) == 0) //创建页表
    panic("userinit: out of memory?"); //若创建失败则可能内存不够，抛出异常
  /*将initcode.S第一个进程的代码加载到进程p中，并分配一页物理内存，将虚拟地址0映射到该物理地址，实现虚拟地址初始化*/
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE; //设置进程大小为一页
  memset(p->tf, 0, sizeof(*p->tf));

    /* 设置初始的用户模式状态 */

  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;//cs寄存器指向代码段并处于用户模式
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;//ds寄存器指向数据段并处于用户模式
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;// 允许硬件中断
  p->tf->esp = PGSIZE;// 用户栈大小为1页
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");
```

这里我们特别关注一下新的可执行程序`initcode.S`是如何执行的，实际上在`xv6 os`中是通过修改`trapframe`的`%eip`为新程序的起点即可，`p->tf->eip = 0;  // beginning of initcode.S`。

```
# Initial process execs /init.
# This code runs in user space.
```

```
#include "syscall.h"
#include "traps.h"

#第一个进程运行程序的汇编代码
# exec(init, argv)
# 触发 exec 系统调用
#将 argv，init，0三个值压入栈
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax #将系统调用编号存放在%eax寄存器中
  int $T_SYSCALL  # 调用编号为T_SYSCALL即64号中断进入系统调用

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

`initcode`的作用实际上就是通过系统调用`exec`执行`/init`程序。

完成初始化工作后，`userinit`函数将进程的状态设置为就绪态`RUNNABLE`。

```
p->state = RUNNABLE;  //将进程状态置RUNNABLE
```

第一个进程的内核栈示意图如下：

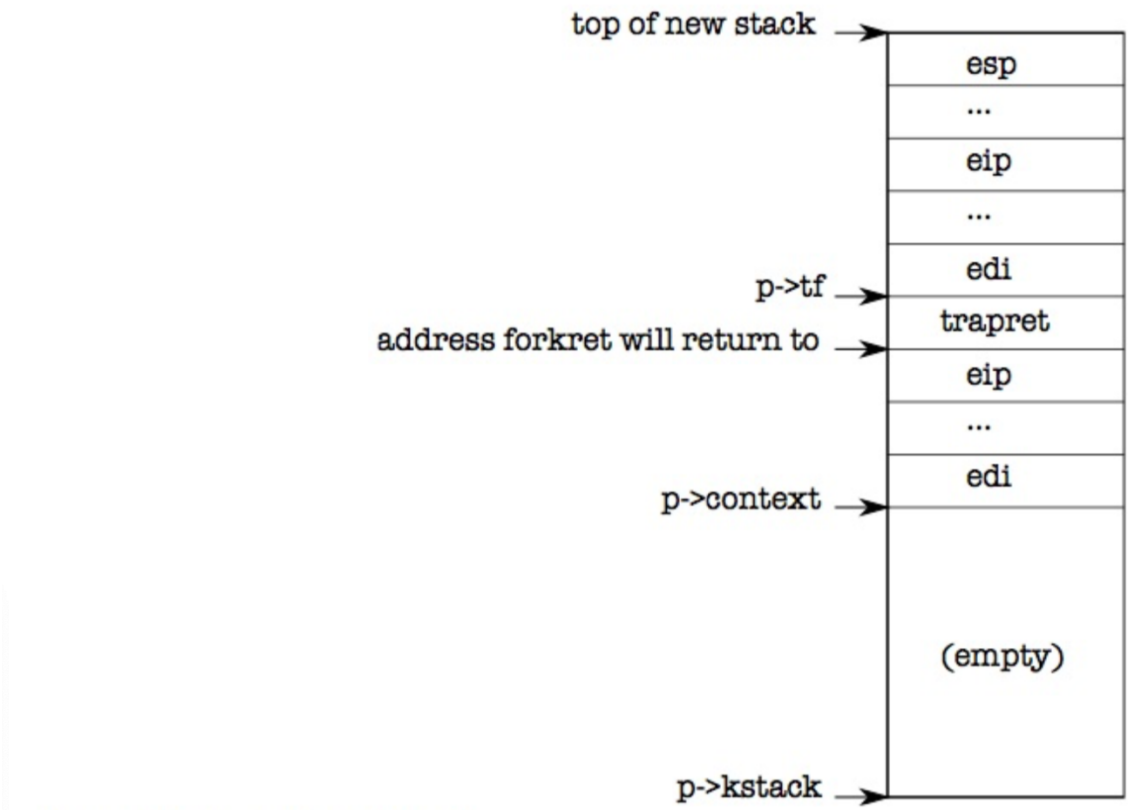


Figure 1-3. A new kernel stack. 新创建进程的内核栈示意图

`XV6 OS`可以运行于多处理机环境(上限为8个`CPU`)，每一个处理机都需要完成上述相应的初始化，最后调用`scheduler()`开始进程的调度，`scheduler()`作为当前`CPU`的内核调度器线程，循环从`ptable`中寻找已就绪的进程运行在当前`CPU`上。具体的调度方式参见下文的***进程调度***篇章。

创建子进程`fork()`

除了第一个进程之外，其他进程（子进程）都是由父进程`initproc`创建的。即父进程调用`fork()`函数。过程如下：

1. 获取当前的进程，其过程为在调用`myproc`函数中调用`mycpu`来表示当前的cpu。
2. 调用`allocproc`初始化该proc。
3. 以一次一页的方式复制父进程地址空间，若失败则回收空间并报报错。
4. 子进程继承父进程的大小，设置父进程为当前进程，继承`trapframe`。
5. 设置`trapframe`的`%eax`为0，这样通过中断返回的时候，会返回0。
6. 子进程继承父进程的打开的文件和当前工作目录，以及父进程的名字
7. 设置进程当前状态为就绪`RUNNABLE`
8. 最后返回子进程的`pid`

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc(); //获取当前进程curproc

  // Allocate process.
```

```
  if((np = allocproc()) == 0){ //分配proc结构体，初始化该proc
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){ //一次一页复制父进程
地址空间
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED; //复制失败，回收空间
    return -1;
  }
  np->sz = curproc->sz; //继承父进程大小
  np->parent = curproc; //设置父进程为curproc
  *np->tf = *curproc->tf; //继承trapframe

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0; //清除寄存器%eax内容，中断返回0

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]); //继承打开文件
  np->cwd = idup(curproc->cwd);  //继承运行目录

  safestrcpy(np->name, curproc->name, sizeof(curproc->name)); //继承父进程名

  pid = np->pid;

  acquire(&ptable.lock); //获取进程表锁

  np->state = RUNNABLE; //置进程状态为就绪态

  release(&ptable.lock);

  return pid; //返回新进程pid
}
```

进程创建学习总结

`XV6 OS`中，主要通过第一个进程的创建过程（在`main.c`中完成设备与子系统初始化以后调用`userinit()`创建所有进程的父进程），之后的各进程则通过`fork()`创建子进程。子进程与父进程将执行相同的程序文本段，但是各自拥有不同的栈段、数据段和堆栈数据拷贝。

在进程创建的学习过程中，涉及到了大量的内存堆栈与汇编语言的知识，在理解过程中存在一定的困难，在教学周的操作系统课程上对于进程创建仅停留在一个最基本、概括的创建过程"申请`PCB`->分配资源->初始化`PCB`->将新进程插入队列"，而在`XV6`中，通过具体操作系统的研习，对于其中的底层原理有了更深刻的理解，收获颇丰。

进程切换`swtch()`

基本概念

操作系统为了控制进程的执行，提高系统资源的利用率，必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行，这种行为被称为进程切换，任务切换或上下文切换。进行进程切换就是从正在运行的进程中收回处理器，然后再使待运行进程来占用处理器。

`swtch()`实现方式

在XV6中，通过`swtch()`函数来实现进程之间的切换。在该操作系统中，进程切换的过程实际上可分为以下几步

- 将当前`CPU`执行进程的寄存器的内容`%eip,%epb,%ebx,%esi,%edi`保存在栈上
- 创建一个新的`struct context`进程上下文结构体，保存当前进程寄存器的值(实际上就是刚刚压入栈中的五个寄存器)
- 将原进程的`context`上下文环境栈顶地址保存，压入进程栈中
- 切换堆栈环境(实则是切换了进程)
- 将寄存器内容`%eip,%epb,%ebx,%esi,%edi`（即新进程的上下文环境）退出栈，实现进程上下文的恢复

至此，通过上述几步，完成进程的切换。

在XV6的源代码中，swtch()函数是通过汇编代码的形式给出的，源码位于swtch.S文件中。

`swtch.S`代码分析

```
# Context switch
#   进程切换函数汇编代码
#   void swtch(struct context **old, struct context *new);
#   函数原型void swtch(struct context **old, struct context *new);
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch: #寄存器参数赋值
  # 进程堆栈如下：
  #          +----------------+
  #          | context *new   |
  #  edx --> +----------------+
  #          | context **old  |
  #  eax --> +----------------+
  #          | eip            |
  #  esp --> +----------------+
  movl 4(%esp), %eax #将Reg[4+esp]的内容赋值给Reg[eax]
  movl 8(%esp), %edx #将Reg[8+esp]的内容赋值给Reg[edx]

  # Save old callee-saved registers
  #            +----------------+
  #            | context *new   |
  #  edx -->   +----------------+
  #            | context **old  |
  #  eax -->   +----------------+
  #          / | eip            |
  #          | +----------------+
  #          | | ebp            |
  #          | +----------------+
  #  context | | ebx            |
  #          | +----------------+
  #          | | esi            |
  #          | +----------------+
```

```
  #          \ | edi             |
  #  esp -->   +-----------------+
  pushl %ebp #保存存放当前线程的栈底指针的寄存器，压入栈中
  pushl %ebx #保存基地址寄存器，压入栈中
  pushl %esi #保存源索引寄存器，压入栈中
  pushl %edi #保存目标索引寄存器，压入栈中

  # Switch stacks 切换堆栈环境
  movl %esp, (%eax)    #将Reg[esp]的内容赋值给Reg[eax]，保存原进程栈顶指针至eax寄存器，
*old = old_proc->context = esp
  movl %edx, %esp      #将Reg[edx]的内容赋值给Reg[esp]，即建立新的进程栈环境，esp =
new_proc->context

  # Load new callee-saved registers 恢复上下文环境
  popl %edi  #弹出新进程context上下文的寄存器内容，恢复。
  popl %esi
  popl %ebx
  popl %ebp
  ret #返回函数调用点
```

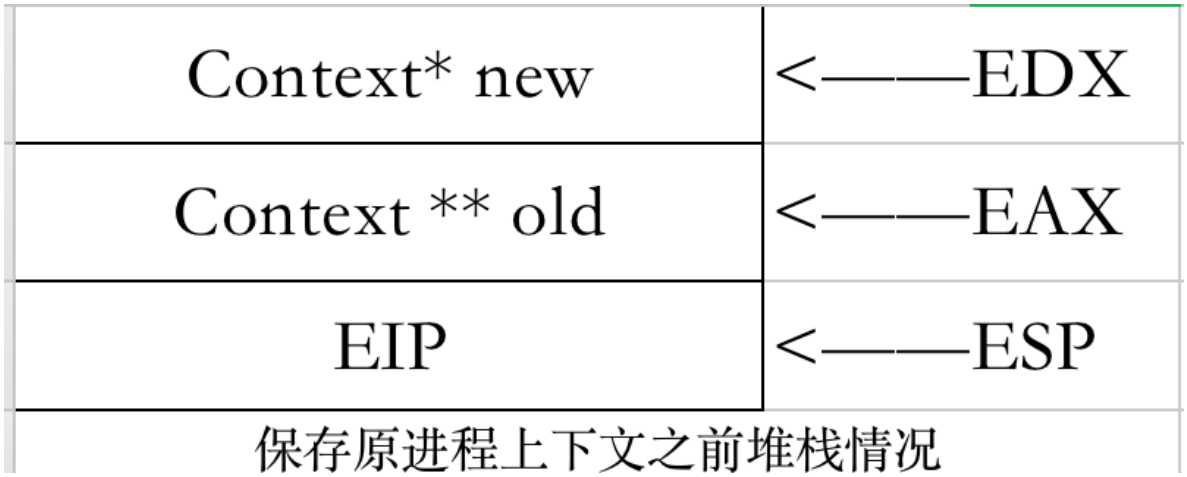
1. 函数原型`void swtch(struct context **old, struct context *new)`

   在swtch.S中通过汇编语言的方式定义了swtch函数，第一参数为进程切换前占有CPU的进程上下文地址指针，第二参数为进程切换后占有CPU的进程上下文地址指针

   调用示例

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      c->proc = p;                                                    //找到处于就绪
态RUNNABLE的进程p
      switchuvm(p);
      p->state = RUNNING;
      //切换进程,调用swtch函数,第一参数为&(c->scheduler),为原进程的上下文环境
      //第二参数为p->context,为欲调度进程的上下文环境
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

2. 汇编代码分析



保存原进程上下文之前堆栈情况

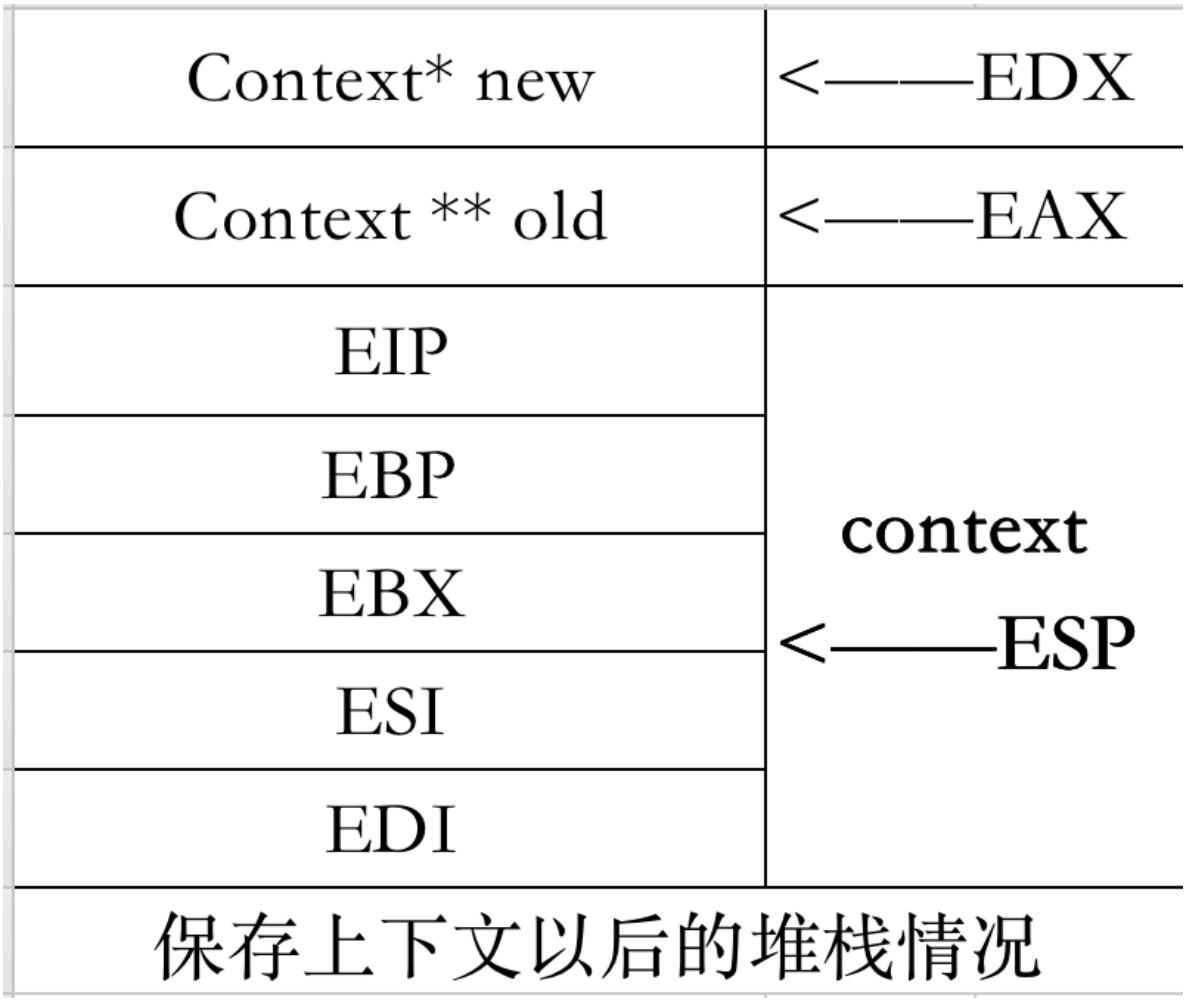
`%eax`寄存器所存储的是原进程的上下文堆栈地址，`%edx`寄存器所存储的是切换后进程的上下文堆栈地址，因此调用`swtch()`函数以后，首先需要给`%edx、%eax`寄存器赋值

```
//swtch()汇编代码部分:寄存器参数赋值
  movl 4(%esp), %eax #将Reg[4+esp]的内容赋值给Reg[eax]
movl 8(%esp), %edx #将Reg[8+esp]的内容赋值给Reg[edx]
```

调用`swtch()`以后，会自动将调用指令的下一条指令地址即`EIP`压入栈中，以便`swtch()`执行完毕以后能够返回到函数调用点。

在切换到目标进程之前，需要保存原进程的`context`上下文环境(即五个寄存器的值，由于这里`EIP`已经在调用时保存，所以只要依次保存四个寄存器即可)，以便再次切换到该进程可以继续执行该进程。

```
//swtch()汇编代码部分:保存原进程上下文环境
pushl %ebp #保存存放当前线程的栈底指针的寄存器，压入栈中
pushl %ebx #保存基地址寄存器，压入栈中
pushl %esi #保存源索引寄存器，压入栈中
pushl %edi #保存目标索引寄存器，压入栈中
```



完成上下文环境寄存器的保存以后，堆栈情况如图。最后五个寄存器正好构成了原进程的`context`，`%esp`即指向原进程的上下文堆栈地址。

之后，需要进行进程上下文环境的切换，将`%esp`的地址保存至`%eax`中，便于恢复进程时能够恢复进程的上下文环境；同时，将当前的栈顶地址切换为新进程的栈顶地址。这通过以下汇编代码实现：

```
# Switch stacks 切换堆栈环境
movl %esp, (%eax)    #将Reg[esp]的内容赋值给Reg[eax]，保存原进程栈顶指针至eax寄存器
movl %edx, %esp      #将Reg[edx]的内容赋值给Reg[esp]，即建立新的进程栈环境
```

这样，就完成了进程堆栈的切换，但是现在进程的上下文环境还没有恢复，还需要从堆栈中弹出寄存器的内容，才能完全实现上下文环境的恢复，完成进程的切换。

```
# Load new callee-saved registers 恢复上下文环境
popl %edi   #弹出新进程context上下文的寄存器内容，恢复。
popl %esi
popl %ebx
popl %ebp
```

至此，整个进程切换的过程就完成了，最后只需要返回`swtch()`函数的调用点，执行下一条指令即可。

```
ret #返回函数调用点
```

进程切换学习总结

- 为了控制进程的执行，内核必须有能力挂起正在`CPU`上执行的进程，并恢复以前挂起的某个进程的执行，这叫做进程切换、任务切换、上下文切换；
- 挂起正在`CPU`上执行的进程，与中断时保存现场是不同的，中断前后是在同一个进程上下文中，只是由用户态转向内核态执行；
- 进程上下文包含了进程执行需要的所有信息

```
用户地址空间：包括程序代码，数据，用户堆栈等
控制信息：进程描述符，内核堆栈等
硬件上下文（注意中断也要保存硬件上下文只是保存的方法不同）
```

在`XV6 OS`中，模拟了一个进程上下文`context`结构体，其中只有五个寄存器的值`%eip,%epb,%ebx,%esi,%edi`，与实际的大型操作系统如`Linux`仍然有很大的区别，但是通过对于进程切换的源码分析与研究，对于汇编语言知识、以及操作系统中维护的内核堆栈与寄存器等内容有了更加深刻的理解。

进程调度

基本概念

无论是在批处理系统还是分时系统中，用户进程数一般都多于处理机数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用处理机。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程,以使之执行。

原理分析

`XV6 OS`中，进程调度分为两个部分：进程调出`sched()`和进程调入`scheduler()`。

进程调出是指，当一个进程异常中断、时间片用完、睡眠等情况下需要剥夺或者让出该进程的处理机资源，主要在`sched()`中实现。

进程调入是指，在`scheduler()`函数（*CPU*的内核调度器线程）中，*CPU*不断从`ptable`页表中寻找**可调度**的进程（即:进程状态处于*RUNNABLE*的进程），并且赋予其处理机资源，投入运行。

进程调度实则就是一个从原进程剥夺处理机（调出进程），找到可以运行的进程（调入进程），切换内核栈，完成上下文环境的切换(`old context->new context`)，然后赋予新进程处理机资源的过程。

`XV6 OS`不会直接从一个进程的上下文切换到另一个进程的上下文，而是通过一个中间的内核线程实现的:内核调度器线程，其中调用`swtch(struct context **old, struct context *new)`完成上下文切换。具体如图：

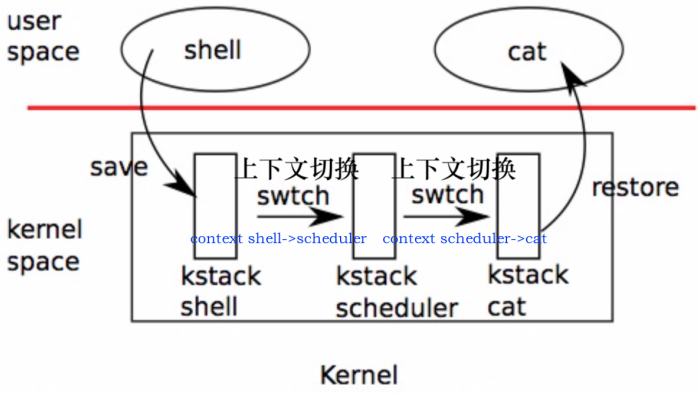


Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread). 从一个进程切换到另一个进程，涉及到从用户态与内核态之间的转换

`XV6 OS`中，在`main.c`中经过一系列初始化之后，内核会进入`scheduler`中，开始第一次进程调度。首先内核会允许中断，然后加锁，在循环体中找到一个就绪状态的进程。然后调用`switchuvm()`切换到该进程的页表。然后使用`swtch()`切换到该进程中运行，该函数会按照`struct proc`中保存的上下文去切换上下文。然后再切回内核的页表最后再释放锁。

在进程运行的过程中，时钟会发送一个中断的信号给内核，强制结束进程只能在某时间片上运行(时间片轮转)，若一个进程的时间片用完，操作系统则调用`trap`函数调用`yield`函数，`yield`调用`sched()`陷入内核态，调用内核的`scheduler()`调度器，切换上下文环境实现进程调度。除此之外，进程异常中断、睡眠、正常退出等都会引发进程的调度，具体请见下文详述。

进程调入`scheduler()`分析

`XV6 OS`中，由***进程创建***分析部分中可见，在操作系统的主函数中，需要完成第一个进程的创建（所有进程的父进程）。

```
//创建第一个进程（所有进程的父进程），完成CPU设置并调度进程
  userinit();      //完成初始化后,调用userinit()创建第一个进程
  mpmain();        // finish this processor's setup 完成此处理器的设置，开始调度进程
```

其中，`mpmain()`函数的定义如下

```
// Common CPU setup code.
static void
mpmain(void)
{
  //完成CPU的配置，输出参数，通知其他处理机（若为多核处理机）本处理机初始化完毕
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes 调用schedluer()函数，开始无限循环寻找RUNNABLE状态的进程赋予其处理机资源
}
```

`XV6 OS`可以运行于多处理机环境(上限为8个`CPU`)，每一个处理机都需要完成相应的初始化，并调用`scheduler()`开始进程的调度，`scheduler()`作为当前`CPU`的内核调度器线程，循环从`ptable`中寻找已就绪的进程运行在当前`CPU`上。

```
//void scheduler(void)源码分析
//PAGEBREAK: 42
```

```
// 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(); //获取当前CPU
  c->proc = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti(); // 在每次执行一个进程之前，需要调用sti()函数开启CPU的中断
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);  //获取进程表锁，与其他CPU的scheduler线程互斥
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE) //扫描进程表，找到一个进程状态为RUNNABLE的进程
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
       /*切换到选择的进程，释放进程表锁，当进程结束时，再重新获取*/
      c->proc = p; //将找到的进程设置为CPU当前执行的进程
      switchuvm(p); //切换至目标进程页表，并通知硬件
      p->state = RUNNING; //切换进程状态为RUNNING，将进程p置为运行态
      /*
       * this perform a context switch to the target process's kernel thread
       * the current context is not a process but rather a special per-cpu
scheduler context,
       * so scheduler tells swtch() to save the current hardware registers in
       * per-cpu storage(cpu->scheduler) rather than in any process's kernel
thread context
       */
      swtch(&(c->scheduler), p->context);//从当前cpu->scheduler现场切换至进程p的上下
文环境
      switchkvm();//切换至内核页表
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      //当前cpu没有正在运行的进程
      c->proc = 0;
    }
    release(&ptable.lock); //释放进程锁
  }
}
```

- 需要注意的是，在 `XV6` 的 `scheduler()` 中每一次循环之前都会执行 `acquire(&ptable.lock)` 以获取进程表锁，与其他 `CPU` 的 `scheduler` 线程互斥，每一次循环之后都必须释放进程锁 `release(&ptable.lock)`，这是为了防止 `CPU` 闲置时，由于当前 `CPU` 一直占有锁，其他 `CPU` 无法调度运行导致的死锁；同时总是要在调度时打开中断 `sti()`，这是为了防止当所有进程都在等待 `I/O` 时，由于关闭中断而产生的死锁问题。
- 在切换进程上下文之前与进程结束之后，都要分别切换至目标页表 `switchuvm(p)` 与内核页表 `switchkvm()`。进程切换切换的是进程页表(即将新进程的pgd(页目录)加载到CR3寄存器中)，而内核页表是所有进程共享的，每个进程的“进程页表”中内核态地址相关的页表项都是“内核页表”的一个拷贝。

- 进程的调度实则是三个上下文环境之间的切换：`old process context, kernel context, new process context`，在调出和调入之间，需要kernel进行调度工作。

进程调出分析

进程的调度包含进程调出过程，也就是从用户态进程进入内核，如进程`sleep()，exit()`或者中断，调出主要在`sched()`中进行，它是进程从用户态进入内核态的入口。与直接调用`scheduler()`不同的是，`sched()`中对于进程上下文切换的条件进行了判断，确保进程释放`CPU`资源前已经获取了进程表锁、执行过了`pushcli`，原进程处于非***RUNNING***状态，以及`CPU`处于不可中断状态；所有条件满足以后才会进入内核态，将上下文切换至`cpu`的`scheduler`，否则即抛出异常。

```
//判断是否满足陷入内核态的各条件，进入内核态
void sched(void)
{
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))   // 是否获取到了进程表锁
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)   // 是否执行过pushcli
    panic("sched locks");
  if(p->state == RUNNING)  // 执行的程序应该处于结束或者睡眠状态
    panic("sched running");
  if(readeflags()&FL_IF)  // 判断中断是否可以关闭
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);    // 上下文切换至当前cpu的scheduler
  mycpu()->intena = intena;
}
```

时间片用完`yield()`

如果一个进程属于时间片用完的状态，需要进行调度，则调用`yield()`函数，让出一个`cpu`调度时间片。

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock 获取进程表锁
  myproc()->state = RUNNABLE; //当前进程时间片用完，置就绪状态
  sched();  //请求调度
  release(&ptable.lock); //释放进程表锁
}
```

进程等待（等待子进程退出）`wait()`

当前进程需要等待子进程退出时，会调用`wait()`函数，从而引发进程的调度。

`wait()`函数的工作流程如下：

1. 用`myproc()`获取当前进程，加锁后进入无限循环

2. 查找当前进程的子进程
3. 如果没有或者当前进程被杀死了返回-1
4. 否则判断子进程是否处于僵尸状态，若是则设置其为`UNUSED`，并且重置该进程上所有的内容，包括释放栈空间等。
5. 返回该子进程的pid

```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc(); //获取当前进程

  acquire(&ptable.lock); //进程表加锁
  for(;;){ //无限循环，寻找当前进程的子进程
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        //重置该进程上所有的内容，释放堆栈，置进程状态为UNUSED
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){ //没有子进程或者当前进程被杀死了返回-1
      release(&ptable.lock); //释放进程表锁
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep 等待子进程退出
  }
}
```

调用`sleep()`置睡眠态，进入内核态调度进程。

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
```

```
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc(); //获取当前进程

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING; //置进程状态为SLEEPING

  sched();  //调度进程，陷入内核态

  // Tidy up.
  p->chan = 0; //释放进程等待链

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

进程退出`exit()`

进程正常退出时，也会引发进程的调度，在`exit()`函数中调用`sched()`陷入内核态完成进程调度。

`exit()`进程终止函数工作流程如下：

1. 用`myproc()`获取当前进程，检查是否是初始进程，若是则对控制台输出`init exiting`再调用exit()自身。

然后关闭所有文件。

2. 调用`iput`函数把对当前目录的引用从内存中删除。
3. 上锁后唤醒当前进程的父进程，以及以上的进程。
4. 把当前进程的所有子进程都划归为用户初始进程`initproc`的子进程中。
5. 把当前进程改为僵尸进程，调用`sched()`陷入内核态使用进程调度器调度下一个可执行进程。

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
```

```
void
exit(void)
{
  struct proc *curproc = myproc(); //获取当前进程
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){ //关闭当前进程打开的所有文件
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);        //把对当前目录的引用从内存中删除
  end_op();
  curproc->cwd = 0;          //当前运行目录置空

  acquire(&ptable.lock);     //获取进程表锁

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent); //唤醒当前进程的父进程，以及以上的进程

  // Pass abandoned children to init.
  //把当前进程的所有子进程都划归为用户初始进程initproc的子进程中。
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE; //把当前进程改为僵尸进程
  sched(); //陷入内核态，开始调度
  panic("zombie exit");
}
```

进程调度学习总结

通过对几个与进程调度有关的重点函数`exit()&sched()&yield()&scheduler()`的分析，可以发现进程的调度（即调入&调出）也就是以下进程上下文环境的切换过程：

```
调入：cpu->scheduler => proc->context
调出：proc->context => cpu->scheduler
```

而进程之间上下文的切换，则需要用到`swtch()`，保存旧的context，然后切换到新的context并弹出寄存器值恢复，具体已在上文详细阐述。

`xv6 os`的进程调度简化了实际大型应用操作系统的进程调度，在代码中仅通过循环扫描进程表选择一个可调度进程实现调度算法，在进程调度中极其影响系统效率的调度算法被削减、弱化，而且在`xv6 os`中也没有实现线程，没有线程之间的调度，但其也向我们展示了一个十分清晰而又森严的进程调度流程，可以很好的防止进程死锁现象，易于理解与学习，收获颇多。

文件系统

文件系统的构成

xv6的文件系统一共分成5层，这五层分别是：

1. buffer层。与其他操作系统类似，xv6采用cache-主存-辅存 三级存储结构。所有接近CPU的操作都应先存放在buffer中。buffer层解决了，存储在buffer中的block，应该如何被读写和保护。
2. log层。每一次buffer和memory之间的数据交换，都会通过log进行。这是为了防止突然断电导致的数据不一致，保证了数据的安全性。
3. file层。包括inode分配，文件读写等一些元操作。
4. directory层。这里会设计一些特殊的inode，用来记录其他的inode以及一些特殊数据。
5. name层。最后我们是通过路径和文件名，来访问各个文件的。file和directory中的一些数据，比如在memory中的存储位置等信息，被封装了起来。

涉及的文件

vx6中，涉及文件系统的相关文件罗列如下。

源文件	头文件	描述
bio.c	buf.h	处理buffer层
log.c		处理log层
file.c	file.h	文件在内存中的处理层
fs.c	fs.h	文件在磁盘中的存放层
sysfile.c		宏观的文件系统操作层

结构体

buffer

buffer作为缓冲的基本单元，其结构体的定义如下。

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
```

其中需要注意的几点

1. 这里使用blockno，而没有使用block所处在的位置（即buffer[xxx]）来呈现block的序号。首先这样可以考虑分成不同区域存储的buffer，使得block占用缓冲的方式更加便捷。另外是block在时间上，是可以被随机销毁和生成的，因此block在空间上也不具有连续性。
2. 正因为block在空间上是不连续的，我们采用链表的形式，使得程序可以遍历整个cache。并且为了方便，这里使用了双向链表。在实现上使用了prev和next两个变量。当然正向遍历更普遍，所以next使用的更多一些。
3. 这里设计了一个sleeplock，为的是在一个线程使用buffer的时候，其他线程不可以使用该buffer，对buffer起保护作用。

log

log负责保护数据，安全的在cache和memory间交换。基本结构如下：

```
struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev;
  struct logheader lh;
};
```

log中的lh成员，全名叫log header。它相当于data block，是存储转移数据的中转站。其结构如下.

```
struct logheader {
  int n;                                  // 有效block的个数
  int block[LOGSIZE];
};
```

file

file结构体作为文件块在kernel中的呈现形式，其基本结构如下。

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

其中需要注意的几点

1. 这里的type需要在以下三个中选择：FD_NONE(普通文件)、FD_PIPE(管道文件)、FD_INODE(索引节点文件)
2. ref指的是引用计数。ref的存在，在于unix系统设计时候，允许多个文件指向同一个inode。所以在删除的时候，并不是删除一个文件，就会在磁盘中删除相应的file，而是等所有的file都删除完成，ref=0的时候，才会清除文件。

inode

inode结构体的数据，使用在内存中。结构体信息包括设备号，Inode编号，引用计数等。基本结构如下：

```
struct inode {
  uint dev;           // 设备ID
  uint inum;          // Inode编号
  int ref;            // 引用计数
  struct sleeplock lock; // Inode锁
  int valid;          // inode是否被从磁盘中读入？

  short type;         // 从dinode结构体得到的拷贝
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

事实上，inode结构体在磁盘中，以dinode形式存储。其基本结构如下。

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

几件事情需要注意

1. 我们可以看到，相比于inode结构体，dinode结构体的组成更加简洁。这主要是考虑到内存中inode需要存储和多线程引用和访问的问题的。
2. 初始的addrs的大小，设置为NDIRECT。通过查询，我们可以知道NDIRECT=12。按照一个block0.5k来计算，一个文件初始的最大size是6k。如果超过6k，在addrs不扩容的情况下，是无法存储所有数据的。这一点需要注意。

superblock

superblock超级块，存储总块数，以及各分类型块数等数据。结构如下。

```
struct superblock {
  uint size;         // 文件系统中一共有多少blocks?
  uint nblocks;      // 数据block的块数
  uint ninodes;      // inode的块数
  uint nlog;         // log block的块数
  uint logstart;     // 第一块logblock的起点
  uint inodestart;   // 第一块inode的起点
  uint bmapstart;    // 第一块block map的起点
};
```

函数过程

binit

这是一个针对bcache的操作函数。然而，更多的时候，这个函数在初始化buffer。为什么要初始化buffer呢？因为buffer的双向链表不是自然联通的。

我们可以看到，函数遍历bcache.buf，来设置其中的next和prev两个属性。

```
void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");
//PAGEBREAK!
  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    initsleeplock(&b->lock, "buffer");
    // buffer采用的都是休眠锁，可以参考buf.h
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

bget

这个函数要求输入设备号和block编号，来找到缓冲区中buffer的位置，并返回其指针。

值得玩味的是，这是一个把查询和创建buffer合二为一的函数。其函数逻辑如下：

1. 函数花了前10行去查询满足指定设备号和block编号的buffer。
2. 如果没有找到，则遍历bcache，去找是否有空闲的buffer来存储待查找的buffer.
3. 如果有，则分配和初始化该buffer，并返回buffer
4. 如果没有，返回panic报错。这时类似于stack overflow，需要扩容或停止执行当前任务解决。

```
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);
  // 代表了现在的cache已经被占用。
  // 在此期间，其他所有对cache的请求都讲无法执行。

  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock);                // 解锁cache
      acquiresleep(&b->lock);               // 但还要锁住找到的buffer
      return b;        // 在目前的cache中已经找到了目标buffer，并返回
    }
  }

  // Not cached; recycle an unused buffer.
  // Even if refcnt==0, B_DIRTY indicates a buffer is in use
```

```
  // because log.c has modified it but not yet committed it.
  // 执行到这里，函数已经遍历了一遍buffer，没有找到
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->flags = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;                          // 找到一个没有用的buffer。替代原有废置的
buffer并返回
    }
  }
  panic("bget: no buffers");             // 没有可以用的buffer，panic报错
}
```

bread和bwrite

bread读一个已经存在于cache中的buffer。函数如下：

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b;
}
```

与bread函数对应的bwrite函数，把一个buffer的内容写回磁盘。函数如下：

```
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

brelse

brelse针对已经使用完缓冲的情况。这时系统需要释放缓冲。

需要注意的是，释放完缓冲的程序，是不应该再次使用缓冲区的

函数如下：

```
void
brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
```

```
    panic("brelse");

  releasesleep(&b->lock);

  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0) {
    // no one is waiting for it.
    b->next->prev = b->prev;
    b->prev->next = b->next;
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }

  release(&bcache.lock);
}
```

initlog

初始化log。函数如下。

```
void
initlog(int dev)
{
  if (sizeof(struct logheader) >= BSIZE)
    panic("initlog: too big logheader");

  // 这里申请一个superblock，是为了获取superblock中start, size等信息
  // 参考fs.h，我们知道一共本操作系统中有4种block，分别是superblock, logblock,
datablock, mapblock
  struct superblock sb;
  initlock(&log.lock, "log");
  readsb(dev, &sb);
  log.start = sb.logstart;              // 注意这里的start指的是磁盘地址
  log.size = sb.nlog;
  log.dev = dev;
  recover_from_log();
}
```

install_trans

把log block的信息，提取到log结构体中

```
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1);
    // read log block起点，是superblock指向的log位置
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]);
    // read dst终点，取lh第tail个block的地址
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
```

```
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

read_head

把磁盘中的数据读入logheader中

```
static void
read_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *lh = (struct logheader *) (buf->data);
  int i;
  log.lh.n = lh->n;
  for (i = 0; i < log.lh.n; i++) {
    log.lh.block[i] = lh->block[i];       // 把buffer中的block地址，拷贝给log
header
  }
  brelse(buf);
}
```

在把block通过logheader写入磁盘的过程中，这个函数真正在执行写操作

```
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];       // 把log header地址，拷贝给buffer中的
block
  }
  bwrite(buf);
  brelse(buf);
}
```

begin_op

开始对log进行读写操作，锁住log，并在outstanding成员上做标记，令其为1

这样如果断电重启，我们可以通过outstanging是否非0来判断，这次的operation是否完成。

```
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);             // 设置log的锁
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
```

```
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;                 // 对outstanding做上锁标记
      release(&log.lock);
      break;
    }
  }
}
```

end_op和commit

log读写结束，解除锁定。

需要注意的是，只有当outstanding=0的时候，才将其提交。因为此时所有的log任务都已完成

相反的，如果发现当前log正在commiting，也就是说在该log读写结束前outstanding已经为0了，代表现在outstanding--后，其值将小于0。需要panic报错。

而如果其他log还处在挂起状态，则可以唤醒他们执行。

```
void
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space,
    // and decrementing log.outstanding has decreased
    // the amount of reserved space.
    // 还有其他的begin_op()在排队等待执行
    wakeup(&log);
  }
  release(&log.lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    // 现在outstanding=0，可以提交了
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}
```

commit函数是一个看起来非常简洁的函数，但其调用了其他功能复杂的函数，完成了commit功能。其函数如下。

```
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // 将修改后的块从缓存写入日志
    write_head();    // 将标头写入磁盘 (这里是真正的提交)
    install_trans(); // 调用install_trans函数
    log.lh.n = 0;
    write_head();    // 从日志中删除事务
  }
}
```

write_log

把buffer从cache向log转移的函数。

```
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev, log.start+tail+1);      // 终点是log block
    struct buf *from = bread(log.dev, log.lh.block[tail]);  // 起点是cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to);  // write the log
    brelse(from);
    brelse(to);
  }
}
```

log_write

乍看名字，你会感觉log_write和write_log两个函数一模一样。但事实上，他们在执行不同的功能。

我们可以从两个函数的输入和输出来分析

两个函数的输出都是void。区别的是log_write的输入是一个struct buf类型的数据，而write_log的输入是void。

这表明，log_write是在buffer层级上与log互动的函数。官方给出的注释告诉我们，log_write函数可以代替bwrite函数，其基本用法是：

- bp = bread(...)
- *modify bp->data[]
- log_write(bp)
- brelse(bp)

```
void
log_write(struct buf *b)
{
  int i;

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");
```

```
  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // buffer的数据读入log header
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

异常/中断/系统调用处理机制

中断/异常机制

涉及文件

- vector.S
- trapasm.S
- trap.c

概念

中断（interrupts）和异常（Exception）是计算机系统，处理器，程序，任务运行时发生的“事件”（Event），代表发生了某些情况，此时处理器会强制从执行当前程序或任务转向执行中断处理程序（Interrupt Handler）和异常处理程序（Exception Handler）来对发生的事件进行处理。

中断发生在程序或任务运行的任意时间，用于回应硬件设备发出的信号（Signal），系统通过中断来处理外来事件，如对外围设备发出的请求。程序也可通过 INT n 指令主动产生中断请求调用中断处理程序。

异常指处理器在执行某些指令时发现错误，如除零错误，当发生异常时，和中断类似，处理器调用异常处理程序来对发生的异常进行处理

中断/异常处理流程

1. 处理器接收到中断信号或发现异常时，暂停当前运行的程序或任务并保存现场
2. 处理器根据中断或异常的类别调用相应的中断/异常处理程序
3. 中断/异常处理程序完成运行后，处理器恢复现场继续运行暂停的程序（除非异常导致程序不可恢复或中断导致程序终止）

异常分类

Intel将计算机发生的异常分为三类，陷阱（Trap），故障（Fault），中止（Abort）

陷阱

陷阱指在触发陷阱的指令运行后立刻报告的异常，它允许发生该异常的程序在不失连续性的情况下继续运行，该异常处理程序的返回地址是触发陷阱的指令的下一条指令

故障

故障是指可修复的异常，当发生故障时，允许程序以不失去连续性的方式重新启动，该异常处理程序的返回地址为触发故障的那一条指令

中止

中止是一种有时无法报告触发该异常确切位置的异常，该异常不允许程序在异常处理程序执行完毕后继续运行

系统调用

系统调用是操作系统提供给用户程序的一系列API，运行在内核态，完成一系列用户态无法执行的特权指令，如从硬盘读取文件等。

故当用户程序需要使用系统调用提供的功能时，则通过相应指令主动进入内核态，这个过程称为陷入（trap）

x86体系的中断处理程序

由于异常，中断即系统调用的处理方式十分相似，x86架构对这些事件采用同样的硬件机制处理，为了简便起见，下面对这些事件都统称为中断。

x86提供了INT n指令来直接产生一个中断以便用户程序从用户态进入内核态调用中断处理程序。

IDT

中断描述符表，最多有256项，每一个表项称为门描述符

用于将中断向量及门描述符联系起来

其中门描述符可分为中断门，陷阱门和任务门。

门描述符格式

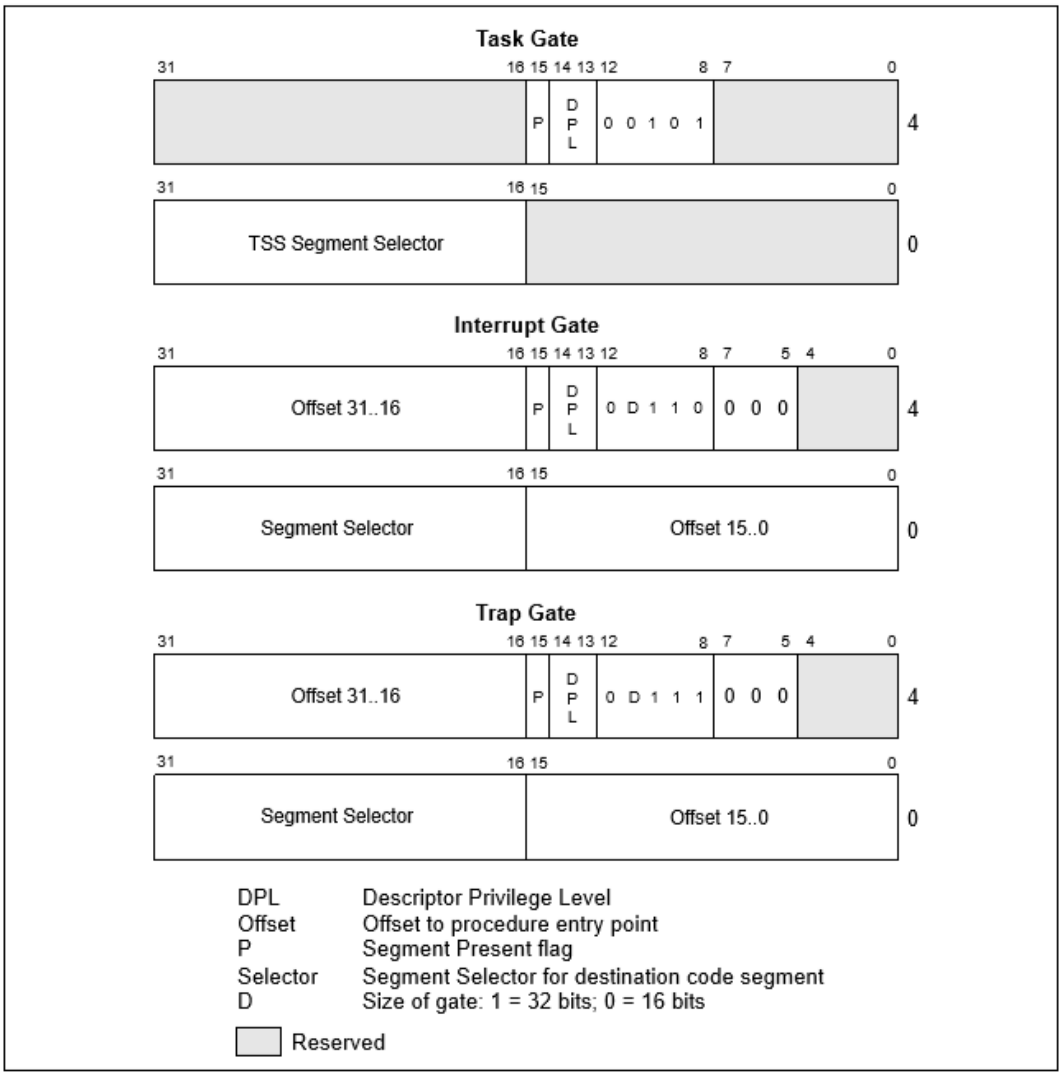


Figure 6-2. IDT Gate Descriptors

IRQ号映射到中断向量

IRQ指中断请求，即硬件向处理器发出，由中断控制器芯片控制

xv6使用APIC来完成中断控制，并且使用内存映射的方式对APIC进行访问，APIC默认地址为**0xFEC00000**，APIC为每个IRQ分配了一个寄存器来存放对应IRQ所对应的中断向量号

xv6使用如下代码完成IRQ号的映射

```
ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
```

INT n指令流程

- 从 IDT 中获得第 n 个描述符，n 就是 int 的参数。
- 检查 %cs 的域 CPL <= DPL，DPL 是描述符中记录的特权级。
- 如果目标段选择符的 PL < CPL，就在 CPU 内部的寄存器中保存 %esp 和 %ss 的值。
- 从一个任务段描述符中加载 %ss 和 %esp。
- 将 %ss 压栈。
- 将 %esp 压栈。
- 将 %eflags 压栈。
- 将 %cs 压栈。
- 将 %eip 压栈。
- 清除 %eflags 的一些位。
- 设置 %cs 和 %eip 为描述符中的值。

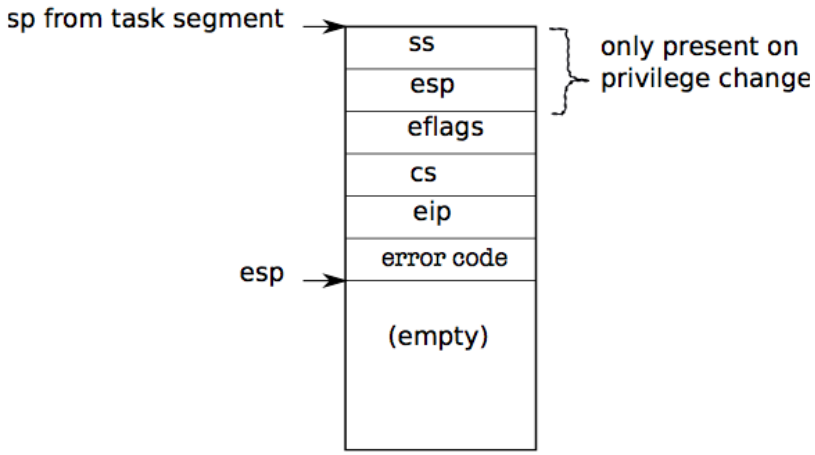


Figure 3-1. Kernel stack after an int instruction.

xv6 中断/异常/系统调用机制

xv6对于所有中断处理程序都采取同样的措施，完成现场保存后均跳转至alltraps函数来实现相应的具体功能

中断向量表

- xv6使用脚本vector.pl完成中断向量表代码vector.S的生成
- 脚本生成的代码形如

```
# sample output:
#   # handlers
#   .globl alltraps
#   .globl vector0
#   vector0:
#     pushl $0
#     pushl $0
#     jmp alltraps
#   ...
#
#   # vector table
#   .data
#   .globl vectors
#   vectors:
#     .long vector0
#     .long vector1
#     .long vector2
#   ...
```

- 每段代码先往栈内push errorcode（若非异常时，手动push以保证trapframe结构完整）和trapno参数
- 再调用alltraps函数
- vetors为建立的中断向量表，内容为每个中断处理函数入口地址

alltraps和trapret函数

- 用于保存现场和恢复现场

- alltraps将通用寄存器存入栈中，构造出trapframe结构体，设置段选择子，再将栈顶指针esp作为指向结构体的指针参数存入栈中，调用trap函数
- 待trap函数执行完毕，再执行trapret，完成恢复现场的工作

trap函数

- 根据参数选择执行相应的中断处理程序
- 函数解析

```
void
trap(struct trapframe *tf)
{
  // 如果为中断类型为系统调用的话，调用相关系统调用
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }


  switch(tf->trapno){
  // 触发计时器中断
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      // 计时器+1
      ticks++;
      // 唤醒相应进程
      wakeup(&ticks);
      release(&tickslock);
    }
    // 表示中断已接收
    lapiceoi();
    break;
  // 触发主从硬盘中断
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  // 触发键盘中断
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  // 触发串口设备中断
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
```

```
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpuid(), tf->cs, tf->eip);
    lapiceoi();
    break;

  //PAGEBREAK: 13
  // 剩余中断则默认为异常触发的
  // 打印出异常信息完成异常处理，结束进程
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  // 若进程状态为killed且处于用户态，结束进程
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  // 当中断为计时器中断时，进行调度
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

系统调用

涉及文件

- syscall.c
- sysproc.c
- sysfile.c

系统调用流程

- 用户先将要调用的系统调用编号存入eax寄存器中

- 调用中断指令 INT T_SYSCALL;进入中断处理程序
- 中断处理程序调用trap函数，trap函数对于系统调用执行syscall函数
- syscall函数根据当前进程trapframe属性获取系统调用编号，从而执行相应系统调用

系统调用实现

- xv6系统调用的参数均存放在栈中，由alltrap函数将其存放于结构体trapframe中
- xv6对访问参数的操作进行了封装，使用arint，arstr，arptr三个函数获取整型，字符串，指针三种不同类型的参数
- 每个系统调用都是对相应模块的API进行封装，通过调用arint，arstr，arptr函数获取参数，然后再调用相应模块提供的API

系统调用种类

- xv6主要提供了进程管理和文件管理两种类型的系统调用，分别位于sysproc.c和sysfile.c中
- sysproc.c中的系统调用封装了进程管理提供的API
- sysfile.c中的系统调用封装了文件系统提供的API

管道

概念

管道是一个小的内核缓冲区，它以**文件描述符对**的形式提供给进程，一个用于写操作，一个用于读操作。从管道的一端写的数据可以从管道的另一端读取。管道提供了一种进程间交互的方式。

文件描述符是一个整数，它代表了一个进程可以读写的被内核管理的对象。进程可以通过多种方式获得一个文件描述符，如打开文件、目录、设备，或者创建一个管道（pipe），或者复制已经存在的文件描述符。简单起见，我们常常把文件描述符指向的对象称为“文件”。文件描述符的接口是对文件、管道、设备等的抽象，这种抽象使得它们看上去就是字节流。

pipe实现概述

- Pipe的主要部分实际上是一小段规定长度的连续数据存储，读写操作将其视为无限循环长度的内存块。
- 初始化时，将给定的文件输入、输出流与该结构体关联。
- 关闭时，释放内存，解除文件占用。
- 读写操作时，则分别需要判断是否超出读写的范围，避免覆盖未读数据或者读取已读数据；如果写操作未执行完，则需通过睡眠唤醒的方式来完成大段数据的读取。

结构

```
struct pipe {
  struct spinlock lock;//spinlock的作用相当与当进程请求得到一个正在被占用的锁时，将进程处于循环检查，等待锁被释放的状态
  char data[PIPESIZE];//保存pipe的内容，PIPESIZE为512
  uint nread;     // 读取的byte的长度
  uint nwrite;    // 写入的byte的长度
  int readopen;   // 是否正在读取
  int writeopen;  // 是否正在写入
};
```

函数

- pipealloc

  该函数实现了pipe的创建，并将pipe关联到两个文件f0, f1上。如果创建成功，返回0，否则返回-1

```
int
pipealloc(struct file **f0, struct file **f1)
{
  struct pipe *p;

  p = 0;
  *f0 = *f1 = 0;
    //如果f0, f1不存在则返回-1
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  if((p = (struct pipe*)kalloc()) == 0)
    goto bad;
    //初始化
  p->readopen = 1;
  p->writeopen = 1;
  p->nwrite = 0;
  p->nread = 0;
  initlock(&p->lock, "pipe");
  (*f0)->type = FD_PIPE;
  (*f0)->readable = 1;
  (*f0)->writable = 0;
  (*f0)->pipe = p;
  (*f1)->type = FD_PIPE;
  (*f1)->readable = 0;
  (*f1)->writable = 1;
  (*f1)->pipe = p;
  return 0;

    //如果创建失败则将进度回滚，释放占用的内存，解除对文件的占用
//PAGEBREAK: 20
 bad:
  if(p)
    kfree((char*)p);
  if(*f0)
    fileclose(*f0);
  if(*f1)
    fileclose(*f1);
  return -1;
}
```

- pipeclose

  实现了关闭pipe的处理

```
void
pipeclose(struct pipe *p, int writable)
{
    //获取管道锁，避免在关闭的同时进行读写操作
  acquire(&p->lock);
    //判断是否有未被读取的数据
  if(writable){
      //如果存在，则唤醒pipe的读进程
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {
      //不存在就唤醒pipe的写进程
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0){
    release(&p->lock);
    kfree((char*)p);
  } else
    release(&p->lock);
}
```

- pipewrite

  实现了管道的写操作

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
    //逐字节写入
  for(i = 0; i < n; i++){
      //如果pipe已经写满
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || myproc()->killed){
          //唤醒读进程，写进程进入睡眠，并返回-1
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
    //写完之后唤醒读进程
```

```
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

- piperead

  piperead实现了pipe的读操作

```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
    //如果pipe已经读空，并且正在写入，则进入睡眠状态
  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
    //读取完毕，唤醒写进程
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
    //返回读取的字节长度
  return i;
}
```

Shell

shell 是一个普通的程序，它接受用户输入的命令并且执行它们，它也是传统 Unix 系统中最基本的用户界面。shell 作为一个普通程序，而不是内核的一部分，充分说明了系统调用接口的强大：shell 并不是一个特别的用户程序。这也意味着 shell 是很容易被替代的，实际上这导致了现代 Unix 系统有着各种各样的 shell，每一个都有着自己的用户界面和脚本特性。xv6 shell 本质上是一个 Unix Bourne shell 的简单实现。

主循环通过 `getcmd` 读取命令行的输入，然后它调用 `fork` 生成一个 shell 进程的副本。父 shell 调用 `wait`，而子进程执行用户命令。

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
```

```c
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
// Simplifed xv6 shell.

#define MAXARGS 10

// All commands have at least a type. Have looked at the type, the code
// typically casts the *cmd to some specific cmd type.
struct cmd {
  int type;          //  ' ' (exec), | (pipe), '<' or '>' for redirection
};

struct execcmd {//最基本命令
  int type;              // ' '
  char *argv[MAXARGS];   // arguments to the command to be exec-ed
};

struct redircmd {
  int type;          // < or >
  struct cmd *cmd;   // the command to be run (e.g., an execcmd)
  char *file;        // the input/output file
  int mode;          // the mode to open the file with
  int fd;            // the file descriptor number to use for the file
};

struct pipecmd {
  int type;          // |
  struct cmd *left;  // left side of pipe，输入
  struct cmd *right; // right side of pipe，输出
};

int fork1(void);  // Fork but exits on failure.
struct cmd *parsecmd(char*);

// Execute cmd.  Never returns.
void runcmd(struct cmd *cmd)
{
  int p[2], r;
  struct execcmd *ecmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    exit(0);

  switch(cmd->type){
  default:
    fprintf(stderr, "unknown runcmd\n");
    exit(-1);

  case ' ':
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit(0);
//    fprintf(stderr, "exec not implemented\n");
    // Your code here ..
```

```
    if (access(ecmd->argv[0], F_OK) == 0)//access作用检查能否对某个文件执行某个操作，mode_>R_OK(测试可读），W_OK(测试可写），X_OK(测试可执行），F_OK(测试是否存在）成功返回0，失败返回-1,execv会停止当前进程，并以pathname应用进程替换被停止的进程进程ID没有变,execv(pathname,argv[])
    {
        execv(ecmd->argv[0], ecmd->argv);//pathname文件路径,argv传给应用程序的参数列表，第一个为应用程序名字本身，最后一个为NULL
    }
    else
    {
    const char *binPath = "/bin/";
    int pathLen = strlen(binPath) + strlen(ecmd->argv[0]);
    char *abs_path = (char *)malloc((pathLen+1)*sizeof(char));
    strcpy(abs_path, binPath);//后面的字符串复制到前面
    strcat(abs_path, ecmd->argv[0]);//后面的字符串追加到前面
    if(access(abs_path, F_OK)==0)
    {
        execv(abs_path, ecmd->argv);
    }
    else
        fprintf(stderr, "%s: Command not found\n", ecmd->argv[0]);

    }
    break;

  case '>':
  case '<':
    rcmd = (struct redircmd*)cmd;
    //fprintf(stderr, "redir not implemented\n");
    // Your code here ...
    close(rcmd->fd);//关闭标准的输入输出，fd为文件描述符
    if(open(rcmd->file, rcmd->mode, 0644) < 0)//打开新的文件作为新的标准输入输出
    {
        fprintf(stderr, "Unable to open file: %s\n", rcmd->file);
        exit(0);
    }
    runcmd(rcmd->cmd);
    break;

  case '|':
    pcmd = (struct pipecmd*)cmd;
    //fprintf(stderr, "pipe not implemented\n");
    // Your code here ...
    // pipe建立一个缓冲区，并把缓冲区通过fd形似给程序调用，它将p[0]修改为缓冲区的读取端，p[1]修改为缓存区的写如端
    // dup（old_fd)产生一个fd，指向old-fd指向的文件,并返回这个fd
    if(pipe(p) < 0)
    {
        fprintf(stderr, "pipe failed\n");
        //exit(0);
    }
    if(fork1() == 0)
    {
        close(1);//关闭标准输出
        dup(p[1]);//把标准输出定向到p[1]所指文件，即管道写入端
    //去掉管道对端口的引用
        close(p[0]);
        close(p[1]);
```

```
    //left的标准输入不变，标准输入流入管道
        runcmd(pcmd->left);
    }

    if(fork1() == 0)
    {
        close(0);
        dup(p[0]);
        close(p[0]);
        close(p[1]);
        runcmd(pcmd->right);
    }

    close(p[0]);
    close(p[1]);
    wait(&r);
    wait(&r);
    break;
  }
  exit(0);
}

int
getcmd(char *buf, int nbuf)//读入命令
{

  if (isatty(fileno(stdin)))//判断标准输入是否为终端
    fprintf(stdout, "$ ");//是终端则显示提示符
  memset(buf, 0, nbuf);
  fgets(buf, nbuf, stdin);//从标准输入读入nbuf个字符到buf中
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}

int
main(void)
{
  static char buf[100];
  int fd, r;

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      //如果是cd命令，切换命令后继续等待读入
      // Clumsy but will have to do for now.
      // Chdir has no effect on the parent if run in the child.
      //一般写完命令敲回车，这里把回车改成‘\0’
      buf[strlen(buf)-1] = 0;  // chop \n，把回车改为0
      if(chdir(buf+3) < 0)//chdir is the same as cd,when sucess, it will return
0, if not, it will return -1
        fprintf(stderr, "cannot cd %s\n", buf+3);//print the message to the file
sterr(stream)
      continue;
    }
      //若不是cd命令，则fork出子程序尝试运行命令
    if(fork1() == 0)//the conmad is not cd, then fork
      runcmd(parsecmd(buf));//child process，parsecmd解析buf，结果送入runcmd
```

```
    wait(&r);//等待子进程的结束
  }
  exit(0);
}

int
fork1(void)
{
  int pid;

  pid = fork();
  if(pid == -1)
    perror("fork");
  return pid;
}

struct cmd*
execcmd(void)
{
  struct execcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = ' ';
  return (struct cmd*)cmd;
}

struct cmd*
redircmd(struct cmd *subcmd, char *file, int type)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = type;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->mode = (type == '<') ?  O_RDONLY : O_WRONLY|O_CREAT|O_TRUNC;
//o_rdonly read only 只读
//o_wronly write only 只写
//o_crEAT 若文件不存在则建新文件
//o_trunc 若文件存在则长度被截为0(属性不变)
  cmd->fd = (type == '<') ? 0 : 1;
  return (struct cmd*)cmd;
}

struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = '|';
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

```
// Parsing

char whitespace[] = " \t\r\n\v";
char symbols[] = "<|>";

int
gettoken(char **ps, char *es, char **q, char **eq)//提取出一段字符串
{
  char *s;
  int ret;

  s = *ps;
  while(s < es && strchr(whitespace, *s))//找到非空，非换行回车的第一个字符
    s++;
  if(q)
    *q = s;
  ret = *s;//返回第一个不是空格的字符指针
  switch(*s){
  case 0://到尾部了
    break;
  case '|':
  case '<':
    s++;
    break;
  case '>'://忽略此字符
    s++;
    break;
  default:
    ret = 'a';
    while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
      s++;//s指向空格或symbols字符
    break;
  }
  if(eq)
    *eq = s;
//q和eq分别指向非空非symbols字符串的首尾
  while(s < es && strchr(whitespace, *s))
    s++;//找到下一个非空字符
  *ps = s;
  return ret;
}

int
peek(char **ps, char *es, char *toks)//找到字符串中非空的第一个字符，若toks中则返回1
{
  char *s;

  s = *ps;
  while(s < es && strchr(whitespace, *s))//找到不是空格的第一个字符
    s++;
  *ps = s;
  return *s && strchr(toks, *s);//检查该字符是不是tock中的字符
}

struct cmd *parseline(char**, char*);
struct cmd *parsepipe(char**, char*);
struct cmd *parseexec(char**, char*);
```

```
// make a copy of the characters in the input buffer, starting from s through
es.
// null-terminate the copy to make it a string.
char
*mkcopy(char *s, char *es)
{
  int n = es - s;
  char *c = malloc(n+1);
  assert(c);
  strncpy(c, s, n);
  c[n] = 0;
  return c;
}

struct cmd*
parsecmd(char *s)
{//命令构造
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);//es points to the last blank char
  cmd = parseline(&s, es);
  peek(&s, es, "");//s指向第一个非空字符
  if(s != es){
    fprintf(stderr, "leftovers: %s\n", s);
    exit(-1);
  }
  return cmd;
}

struct cmd*
parseline(char **ps, char *es)//参数字符串的头指针和末尾指针
{//将字符串转化为命令
  struct cmd *cmd;
  cmd = parsepipe(ps, es);
  return cmd;
}

struct cmd*
parsepipe(char **ps, char *es)//处理管道命令
{
  struct cmd *cmd;

  cmd = parseexec(ps, es);//不同?
  if(peek(ps, es, "|")){
    gettoken(ps, es, 0, 0);
    cmd = pipecmd(cmd, parsepipe(ps, es));
  }
  return cmd;
}

struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)//重定向文件
{
  int tok;
  char *q, *eq;
```

```
  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);//提取出重定向的字符
    if(gettoken(ps, es, &q, &eq) != 'a') {//提取出重定向之后的文件名
      fprintf(stderr, "missing file for redirection\n");
      exit(-1);
    }
    switch(tok){
    case '<':
      cmd = redircmd(cmd, mkcopy(q, eq), '<');
      break;
    case '>':
      cmd = redircmd(cmd, mkcopy(q, eq), '>');
      break;
    }
  }
  return cmd;
}

struct cmd*
parseexec(char **ps, char *es)//管道左右字符串的提取
{//
  char *q, *eq;
  int tok, argc;
  struct execcmd *cmd;
  struct cmd *ret;

  ret = execcmd();
  cmd = (struct execcmd*)ret;

  argc = 0;
  ret = parseredirs(ret, ps, es);
  while(!peek(ps, es, "|")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != 'a') {
      fprintf(stderr, "syntax error\n");
      exit(-1);
    }
    cmd->argv[argc] = mkcopy(q, eq);
    argc++;
    if(argc >= MAXARGS) {
      fprintf(stderr, "too many args\n");
      exit(-1);
    }
    ret = parseredirs(ret, ps, es);
  }
  cmd->argv[argc] = 0;
  return ret;
}
```

部分硬件驱动实现

键盘

键盘端口

- CPU可使用0x60端口来获取键盘输入

扫描码

- 0x60输入的数据为键盘按键的扫描码，当按下按键或松开按键时，键盘都会传送一个扫描码至0x60端口
- 同一按键的松开和按下的扫描码关系为**松开扫描码=按下扫描码+0x80**，如a的扫描码为0x1E和0x9E
- 扫描码只有按键有关，不区分大小写
- 部分扫描码在由两个字符构成，如小键盘的扫描码都以0xE0作为开头，如右箭头的扫描码为0xE0 0x4D和0xE0 0xCD

xv6获取按键信息流程

- xv6使用kbdgetc函数从作为底层驱动直接从键盘获取信息
- 函数解析

```
int
kbdgetc(void)
{
  static uint shift;
  static uchar *charcode[4] = {
    normalmap, shiftmap, ctlmap, ctlmap
  };
  uint st, data, c;
  // 检查是否有数据
  st = inb(KBSTATP);
  if((st & KBS_DIB) == 0)
    return -1;
  // 获取扫描码
  data = inb(KBDATAP);

  // 发送的第一个数据为0xE0，即代表该扫描码有两个数字构成
  // 启用E0标记位
  if(data == 0xE0){
    shift |= E0ESC;
    return 0;
  } else if(data & 0x80){
    // Key released
    // 按键松开，还原配置
    // shiftcode使用按下扫描码来作为映射
    // 当扫描码为E0扫描码时，使用松开扫描码以确保与其他键使用不同的映射
    data = (shift & E0ESC ? data : data & 0x7F);
    shift &= ~(shiftcode[data] | E0ESC);
    return 0;
  } else if(shift & E0ESC){
    // Last character was an E0 escape; or with 0x80
    // 若是E0扫描码的按键，使用松开扫描码进行映射
    // 并且取消E0标志位
    data |= 0x80;
    shift &= ~E0ESC;
  }

  // 使用如CTRL，ALT这样按下才生效的的功能键
  shift |= shiftcode[data];
```

```
  // 使用CapsLk，NumLock这样保持状态
  shift ^= togglecode[data];
  // 根据shift的状态位来得到当前按键使用的扫描码映射
  c = charcode[shift & (CTL | SHIFT)][data];
  // 若当前为大写锁定状态，将小写字母转大写，大写转小写
  if(shift & CAPSLOCK){
    if('a' <= c && c <= 'z')
      c += 'A' - 'a';
    else if('A' <= c && c <= 'Z')
      c += 'a' - 'A';
  }
  return c;
}
```

显示

- 显卡的GCA彩色字符模式在内存中的映射为0xB8000~0xBFFFF
- xv6将显示内容输入到对应内存的位置，从而显示在屏幕上

参考资料

- xv6中文文档
- Intel 64及IA-32架构软件开发手册
- x86汇编语言：从实模式到保护模式
- 其他与xv6相关的博客