Xv6 源码导读

Dec 6 2022 NCUT 计实验20

1. **王康**: 虚拟内存
2. **苏靖博**: 系统调用
3. **董安杰**: 锁
4. **王帅帅**: 文件系统

虚拟内存

地址空间

页表介绍

Xv6跑在 Sv39 RISC-V下，该模式下虚拟地址只用到低39位其中前27位作会被解释索引找到物理块号再加上后12位得到物理地址。

（注：这27位会被分为单级和多机索引）

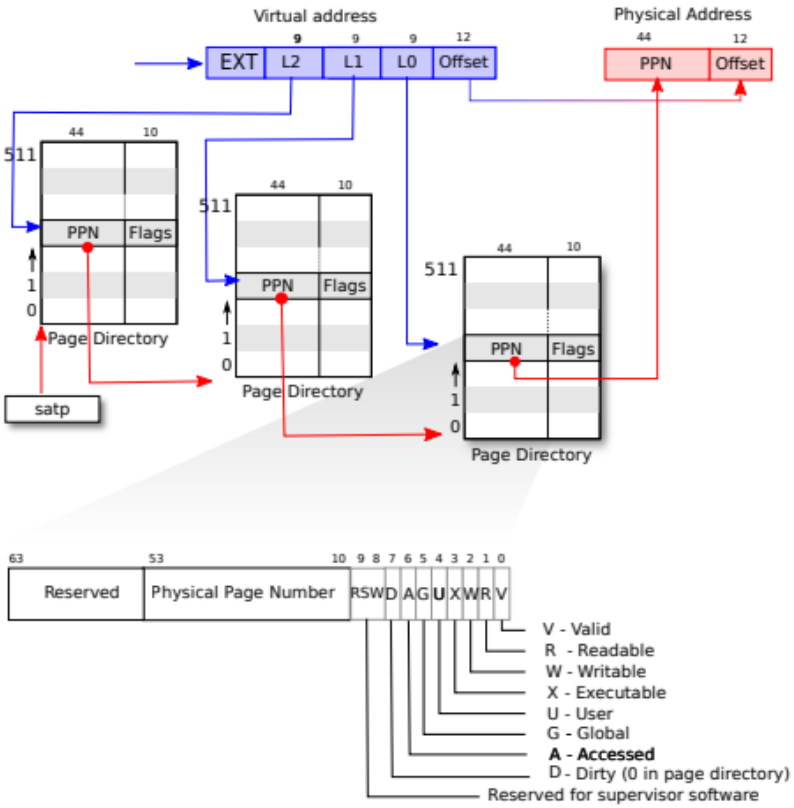


Figure 3.2: RISC-V address translation details.

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // user can access
```

内核空间

Xv6会为每个进程维护一个页表，内核也有一个页表方便内核以可预测的速度访问物理空间和硬件资源。

相关的数据定义

```
//where in kernel/riscv.h
#define PGSIZE 4096 // bytes per page
#define PXMASK          0x1FF // 9 bits
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

#define PXSHIFT(level)  (PGSHIFT+(9*(level)))
#define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK)
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
#define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)

#define PTE2PA(pte) (((pte) >> 10) << 12)
```

地址空间

最重要的函数

```
//pagetable_t 指向根页表 kernel or process


//找到相应的页表项，没有则分配
// where in kernel/vm.c
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

kvminit为内核申请一页的空间，然后调用kvmmap，在即将装载的内核页表上建立一系列的直接映射，包括I/O设备、内核代码和数据、内核空闲内存段等。

```c++
kvminit(void)
{
  kernel_pagetable = kvmmake();
}

pagetable_t
kvmmake(void)
{
  pagetable_t kpgtbl;

  kpgtbl = (pagetable_t) kalloc();
  memset(kpgtbl, 0, PGSIZE);

  // uart registers
  kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // PLIC
  kvmmap(kpgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  kvmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  // allocate and map a kernel stack for each process.
  proc_mapstacks(kpgtbl);

  return kpgtbl;
```

```
}

void
kvmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(kpgtbl, va, sz, pa, perm) != 0)
    panic("kvmmap");
}
```

```
//为页表建立映射项
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  if(size == 0)
    panic("mappages: size");

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      panic("mappages: remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

在调用kvminit申请初始化空间过后，main调用kvminithart来装载内核页表的根页表地址到satp寄存器中

```
void
kvminithart()
{
  // wait for any previous writes to the page table memory to finish.
  sfence_vma();

  w_satp(MAKE_SATP(kernel_pagetable));

  // flush stale entries from the TLB.
  sfence_vma();
}
```

在内核空间下，main马上就调用procinit，为每个用户进程分配一个内核栈，该内核栈将被映射到内核虚拟地址空间的高地址部分，位于trampoline下方。生成虚拟地址的步长为2页，而且只处理低的那一页，这样高的一页就自动成了保护页（PTE_V无效）。更新了所有内核栈的PTE之后，最后调用kvminithart更新一次satp寄存器，分页硬件就能使用新的页表。

```
void
procinit(void)
{
  struct proc *p;
  initlock(&pid_lock, "nextpid");

  // 开始时p=proc，即p的地址是proc数组的最开始位置
  // 每次遍历p就指向下一个进程结构
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");
      // Allocate a page for a kernel stack, for each process
      // Map it high in memory at the va generated by KSTACK, followed by an invalid
guard page.
      char *pa = kalloc();
      if(pa == 0)
        panic("kalloc");
      // 指针相减就是地址相减，获取当前进程p和proc数组最开始位置的偏移量
      // 比如第一次，从p-proc=0开始，KSTACK生成虚拟地址：TRAMPOLINE - 2*PGSIZE
      // 因此TRAMPOLINE的下面第一页是guard page，第二页是kstack，也就是va指向的位置
      // 后面也以此类推，被跳过而未被处理的guard page，PTE_V是无效的
      uint64 va = KSTACK((int) (p - proc));
      // adds the mapping PTEs to the kernel page table
      // 内核栈可读可写，但在用户态不可访问，也不能直接执行
      kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
      p->kstack = va;
  }
  // 将更新后的内核页表重新写入到satp中
  kvminithart();
}
```

物理空间分配

内核在运行时会分配和释放很多物理内存，xv6将一部分的物理内存，从kernel data结束开始，到PHYSTOP为止，这一部分称为free memory，用于运行时的内存分配。每次分配和回收都以页为单位，一页大小4KB，通过一个空闲物理帧链表free-list，将空闲的物理帧串起来保存。页表、用户内存、内核栈、管道缓冲区等操作系统组件需要内存时，内核就从free-list上摘下一页或者多页分配给它们；在回收已经分配出去的内存时，这些被回收的物理帧，内核将它们一页页地重新挂到free-list上。

```
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;

extern char end[]; // first address after kernel.
                   // defined by kernel.ld.

//initialize the allocator
void
kinit()
{
  // initializes the free list to hold every page between the end of the kernel and
PHYSTOP
  // xv6 assumes that the machine has 128MB of RAM
  initlock(&kmem.lock, "kmem");
  // kernel data之后到PHYSTOP之前都可以用于分配
  // add memory to the free list via per-page calls to kfree
  freerange(end, (void*)PHYSTOP);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  //kfree是头插法
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}

void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  // casts pa to a pointer to struct run, which records the old start of the free list
in r->next,
  // and sets the free list equal to r
  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}

void *
kalloc(void)
{
  // removes and returns the first element in the free list.
  // When a process asks xv6 for more user memory, xv6 first uses kalloc to allocate
physical pages.
  struct run *r;
```

```
  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

Trap & System call

Isolation

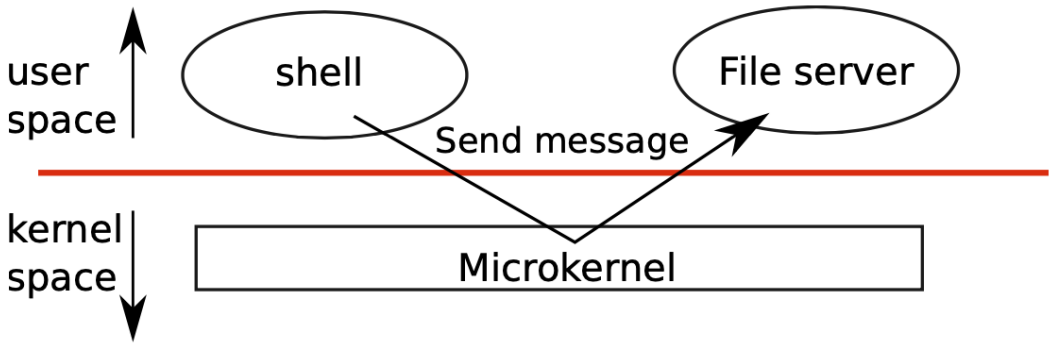


Figure 2.1: A microkernel with a file-system server

1. Trap

> Kernel code (assembler or C) that processes a trap is often called a ***handler***; the first handler instructions are usually written in assembler (trampoline.S, rather than C) and are called a ***vector***.

Popular names

- **kernel mode**: 内核态
- **user mode**: 用户态
- **supervisor**: 在内核态工作的具有较高权限的管理者身份
- **user**: 与计算机进行***交互***的普通用户身份 👨‍💻 <=> 💻
- **stvec**: The ***kernel*** writes the ***address*** of its trap handler here; the RISC-V jumps to the address in *stvec* to handle a trap. (指向了内核中处理 trap 的指令的起始地址)
- **sepc**: When a trap occurs, RISC-V saves the ***program counter*** here (since the pc is then overwritten with the value in *stvec*).
- **sret**: The sret (***return*** from trap) instruction copies *sepc* to the pc. The ***kernel*** can write *sepc* to control where *sret* goes.
- **scause**: RISC-V puts a number here that describes the ***reason*** for the trap. (***exception***, ***system call***, ***device interrupt***)
- **sscratch**: (Supervisor mode scratch) Helps trap handler avoid over writing user registers before saving them. (🌟 OS 留了一个 reg 在自己手上, 对用户进程不可见.)
- **sstatus**: The ***SIE*** bit in sstatus controls whether device interrupts are enabled. If the kernel clears SIE (**0**), the RISC-V will defer device interrupts until the kernel sets SIE (**1**). The ***SPP*** bit indicates whether a trap came from user mode (**0**) or supervisor mode (**1**), and controls to what mode *sret* returns.

```
SSTATUS [SPP _ _ SPIE UPIE _ _ SIE UIE]
          ^  ^ ^  ^    ^   ^ ^  ^   ^
          8  7 6  5    4   3 2  1   0
```

- **satp**: (Supervisor mode address translation and protection) Tells trampoline the ***user/supervisor page table*** to switch to. (包含了指向 page table 的物理内存地址)
- **uservec**: Saves ***user registers*** in the trapframe, an assembly function.
- **TRAPFRAME** (`0x3ffffe000` , 保存进程寄存器现场的内存) and **TRAMPOLINE** (`$stvec = 0x3ffffff00` , Read-only, 跳板)

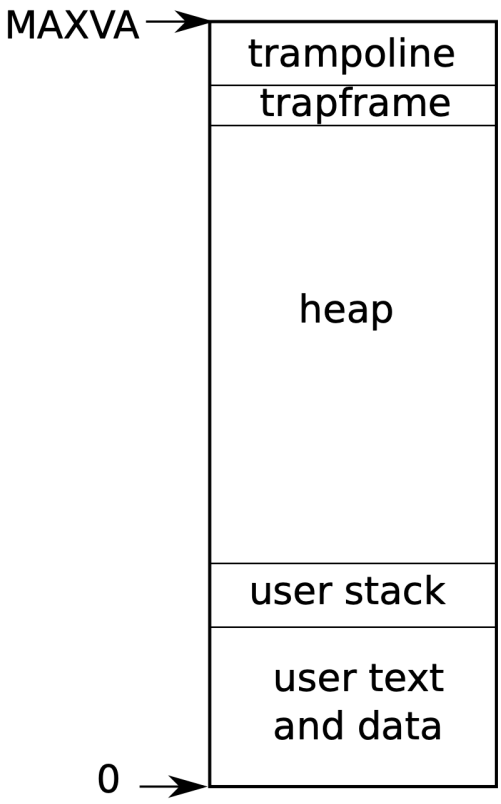


Figure 2.3: Layout of a process's virtual address space

- **memlayout.h**

```
// User memory layout
// Address zero first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
//   ...
//   TRAPFRAME (p->trapframe, used by the trampoline)
//   TRAMPOLINE (the same page as in the `kernel`)

// Map the trampoline page to the highest address,
// in both user and kernel space.
#define TRAMPOLINE (MAXVA - PGSIZE)

// TRAPFRAME is the next page of TRAMPOLINE
#define TRAPFRAME (TRAMPOLINE - PGSIZE)
```

- **PTE_U**: flag determines whether the user mode can use current page table. ***PTE_U = 0: Supervisor, 1: User***

- **proc.h**

struct trapframe & 32 regs

Good practice

2. User mode

对 ecall 瞬间的状态做快照 (trampoline.S)

- 填充 struct trapframe (*proc.h*) <= `sd regs` (page position definition: *memlayout.h*)
- 利用 `$sscratch` (S-mode scratch reg) 保存所有 register
- 切换到 *kernel stack* (切换进程对应的"内核线程")
- 切换到 *kernel address space*
  - 修改 `$satp` 指向 (`csrw satp, t1`)
  - `csrw`
  - `sfence.vma`
- 跳转 (`jr t0`)到 *usertrap* 进入c代码!

> usertrap: determine trap **cause**, process it, and return; it changes **stvec** so that kernel <= **kernelvec** rather than uservec; it saves **sepc** (saved user **pc**)

RISC-V user-level ecall 指令 (trap.c: usertrap)

- 打开中断 `intr_on();`
- 设置 `$sstatus` 为 `S-mode`
- 更改 `$stvec` 指向 `kernelvec` (`w_stvec((uint64)kernelvec);`)
- 复制 `$pc` 到 `$sepc`；`$sepc += 4`
- 设置 `$scause` 为 trap 的原因 (*ecall, 8*)
- `$pc` 跳转到 `$stvec` (let `$pc = $stvec`) 并执行

> **ps. *ecall* 不能** switch page table.
>
> **Q.** pc->virtual address, 当 switch page table 时为什么程序没有crash或产生其他垃圾?

3. System call

- **user.h**

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

- **syscall.h**

System call	System call number
fork	1
exit	2
wait	3
pipe	4
read	5
kill	6
exec	7
fstat	8
chdir	9
dup	10
getpid	11
sbrk	12
sleep	13
uptime	14
open	15

System call	System call number
write	16
mknod	17
unlink	18
link	19
mkdir	20
close	21

- **syscall.c**

```
// Prototypes for the functions that handle system calls.
extern uint64 sys_func(void);
...

static uint64 (*syscalls[])(void) = {
  [SYS_fork] sys_fork,
  ...
};

void syscall(void) {
  int num;
  struct proc *p = myproc();
  num = p->trapframe->a7;
  ...
}
```

4. Kernel (Supervisor) mode

- **usertrapret (trap.c)**: Sets up the RISC-V control registers to prepare for a future trap from user space. (ecall 的逆操作)
  - 关中断 `intr_off();`
  - 更新 `stvec` 指向用户空间的 trap 处理代码, 设置了 `stvec` 指向 trampoline, 在那里执行 `sret` 返回到 user address space
  - 填入 trapframe 内容 (恢复现场)
    - 存储 kernel page table pointer (`kernel_satp`)
    - 存储当前用户进程的 kernel stack (`kernel_sp` , stack pointer)
    - 存储 usertrap 函数指针, 使得 trampoline 代码能够跳转到 (`kernel_trap = usertrap`)
    - 从 `tp` 中读取当前的CPU核编号 (`kernel_hartid`), 存储在 trapframe 中, 使得 trampoline 代码能够恢复这个数字, 因为用户代码可能会修改它
- **userret (trampoline.S)**: Switches satp to the process's user page table. kernel 中最后一条指令
  - 程序切换回 user mode
  - `$sepc` 的数值会被 copy 到 `pc`
  - sret 重新打开中断

4. Kernel (Supervisor) mode

- **usertrapret (trap.c)**: Sets up the RISC-V control registers to prepare for a future trap from user space. (ecall 的逆操作)
  - 关中断 `intr_off();`
  - 更新 `stvec` 指向用户空间的 trap 处理代码, 设置了 `stvec` 指向 trampoline, 在那里执行 `sret` 返回到 user address space
  - 填入 trapframe 内容 (恢复现场)
    - 存储 kernel page table pointer (`kernel_satp`)
    - 存储当前用户进程的 kernel stack (`kernel_sp` , stack pointer)
    - 存储 usertrap 函数指针, 使得 trampoline 代码能够跳转到 (`kernel_trap = usertrap`)
    - 从 `tp` 中读取当前的CPU核编号 (`kernel_hartid`), 存储在 trapframe 中, 使得 trampoline 代码能够恢复这个数字, 因为用户代码可能会修改它
- **userret (trampoline.S)**: Switches satp to the process's user page table. kernel 中最后一条指令
  - 程序切换回 user mode
  - `$sepc` 的数值会被 copy 到 `pc`
  - sret 重新打开中断

5. Summary

System call entry/exit is far more complex than function call.

系统调用进入/退出比函数调用复杂得多.

Much of the complexity is due to the requirement for isolation and the desire for simple and fast hardware mechanisms.

大部分的复杂性是由于对隔离的要求以及对简单快速的硬件机制的需求.

6. References

[1] (Read) xv6-book: xv6: a simple, Unix-like teaching operating system

[2] (Read) Lecture note: 6.1810 2022 Lecture 6: System Call Entry/Exit

[3] (Read) Yanyan's Wiki: 操作系统: 设计与实现 (2022 春季学期)

[4] (Video) MIT Course: MIT 6 S081 Fall 2020 Lecture 6 Isolation & System Call Entry Exit

[5] (Video) NJU Course: 南京大学2022操作系统-设计与实现

[6] (Lab) MIT Lab: Xv6, a simple Unix-like teaching operating system

[7] (Code) Latest xv6: xv6-riscv

锁与并发

为什么需要锁

在操作系统内核中有大量的数据结构都是可以被并发访问的。并发地去访问同一片数据，可能会导致读写错误。 要让这些并发不安全的操作被序列化，可以使用锁这种同步原语。

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

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

在运行过程中，内核的内存分配器维护一个全局的链表 `freelist` 。 如果调用函数 `kalloc()` 成功，会从代表可用内存的链表中弹出一页内存。 如果调用函数 `kfree()` 成功，会向代表可用内存的链表推入一页内存。

在 xv6 中所使用的链表结构是线程不安全的，如果同时进行插入或弹出操作，可能会导致操作乱序。

在哪里使用了锁

用到锁的文件	为什么需要锁
bcache.lock	保护块缓冲区缓存条目的分配
cons.lock	序列化了对控制台硬件的访问，避免了混合的输出
ftable.lock	序列化文件表中文件结构体的分配
itable.lock	保护内存中 `inode` 的分配
vdisk_lock	序列化对磁盘硬件和 DMA 描述符队列的访问
kmem.lock	对内存的分配进行序列化
log.lock	序列化对事务日志的操作
pipe’s pi->lock	序列化对每个管道的操作
pid_lock	序列化了 `next_pid` 的增量
proc’s p->lock	序列化对进程状态的改变
wait_lock	避免 `wait` 失去唤醒
tickslock	序列化对 `ticks` 计数器的操作

用到锁的文件	为什么需要锁
inode's ip->lock	序列化对每个 `inode` 和其内容的操作
buf's b->lock	序列化对每个块缓冲区进行的操作

锁的实现

Xv6 中实现并使用了两种锁。

自旋锁

在 `kernel/spinlock.h` 中定义了自旋锁的结构体。

```
// 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.
};
```

自旋锁是对多处理器互斥的。也就是说，两个 CPU 不能同时对 `lk->locked` 进行修改，要达到这个目的，需要一些**原子化**的操作。 我们所学习的 xv6 操作系统目标的 RISC-V 架构指令集提供了满足这个需求的原子化原语 `amoswap r, a` 。 函数 `__sync_lock_test_and_set` 便是基于这一指令定义的。 在此基础上，一个上锁函数的定义思路就出现了：将封装好的设定 `lk->locked` 的操作放在条件循环中，当条件不满足时进行循环地自旋， 以达到阻塞下一步操作的目的。

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
acquire(struct spinlock *lk)
{
  push_off(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
  //   a5 = 1
  //   s1 = &lk->locked
  //   amoswap.w.aq a5, a5, (s1)
  while(__sync_lock_test_and_set(&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 strictly after the lock is acquired.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Record info about lock acquisition for holding() and debugging.
  lk->cpu = mycpu();
}
```

注意到，上面锁的上锁操作定义中，除了调试信息之外还有一些额外的代码。

- 条件检查 `if(holding(lk))` 获取锁的操作不应该在当前 CPU 已经持有锁的基础上发生。 Xv6 的锁实现是不可重入的。可重入锁又被称为递归锁，意思是如果当前的线程已经持有了锁， 这个线程还尝试再次上锁，内核是允许这种情况发生的，而不是像现在的实现一样 panic 掉。
- 调用 `push_off` 函数（与 `pop_off` 配套）会追踪在当前 CPU 上嵌套调用多个锁的层数。当 调用多层锁的层级被减低到 0 时，系统将会重新启动中断功能，反之启用。 这是因为在 xv6 中，锁保护的对象，既可以被线程使用，也可以被中断处理所使用。 在使用受保护的数据之前应该对其上锁，这是如果中断启用，可能会导致死锁。
- 调用 `__sync_synchronize` 函数：现代的编译器会对指令进行重排和优化，也就是说， 程序执行指令的方式和在文本中定义的顺序可能是不同或并行的；如果发生了寄存器缓存的情况， 程序甚至可能不执行生成需要的 `load` 和 `store` 指令。如果在当前的操作中，指令有序和不被省略 对程序的行为是关键的，可以使用函数 `__sync_synchronize` 来告诉编译器不要做这些优化。 对需要保护的对象上锁是这一场景的典型例子。

考虑到上面的要点和实现方式，实现解锁的过程也是类似的。

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->cpu = 0;

  // Tell the C compiler and the CPU to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other CPUs before the lock is released,
  // and that loads in the critical section occur strictly before
  // the lock is released.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
  __sync_lock_release(&lk->locked);

  pop_off();
}
```

在 `kernel/spinlock.c` 中定义了自旋锁的初始化与基本操作，和上面提及的辅助函数。

睡眠锁

自旋操作会让 CPU 处于忙于等待的状态，适合用于进行一些需要时间短，顺序关键的操作。 如果符合以上要求的特性，这样的锁操作可以做到低延迟。此外，这里实现的自旋锁是和硬件相关的，最基础的高层同步原语。 如果有的上锁操作需要执行相当一段时间，例如文件操作时，便需要一类锁能够不消耗太多系统资源， 不让 CPU 忙于等待，让调度器知道当前任务在至少多少时间内无法推进。于是便引入了睡眠锁。

在 `kernel/sleeplock.h` 中定义了睡眠锁的数据结构。

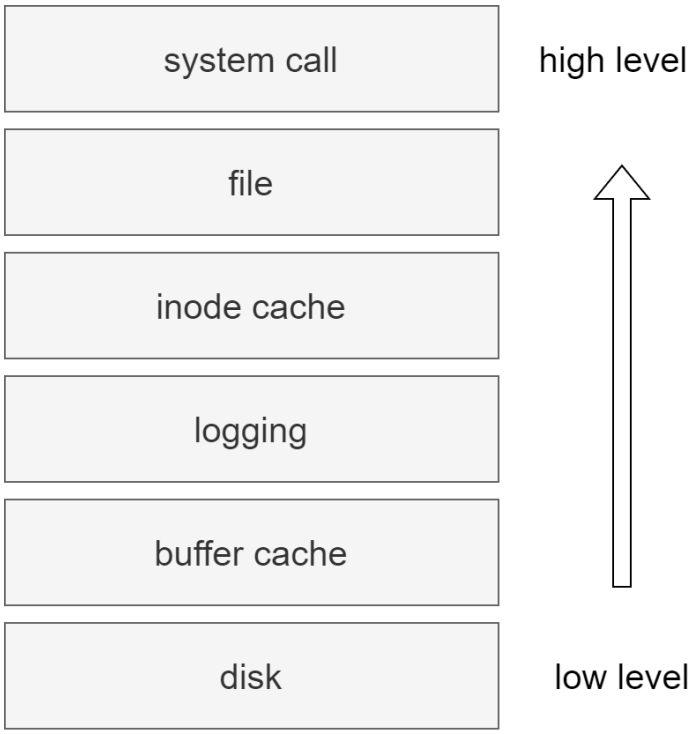
```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

在睡眠锁的结构体中包含了自旋锁的字段，这是为了让对睡眠锁的操作序列化。

在 `kernel/sleeplock.c` 中定义的对睡眠锁的基本操作中，除了初始化函数外， 都使用内部的自旋锁来保持序列化性质。

```
void
initsleeplock(struct sleeplock *lk, char *name)
{
  initlock(&lk->lk, "sleep lock");
  lk->name = name;
  lk->locked = 0;
  lk->pid = 0;
}

void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}

int
holdingsleep(struct sleeplock *lk)
{
  int r;

  acquire(&lk->lk);
  r = lk->locked && (lk->pid == myproc()->pid);
  release(&lk->lk);
  return r;
}
```

睡眠锁所能进行的基本操作与自旋锁是类似的。 在定义睡眠锁时，使用了 `sleep` 与 `wakeup` 这一对操作。 操作 `sleep` 告诉内核，线程将会停止运行，直到等待到特定事件发生； 操作 `wakeup` 告诉内核，事件发生，线程将会继续推进。



xv6 文件系统层次结构示意图

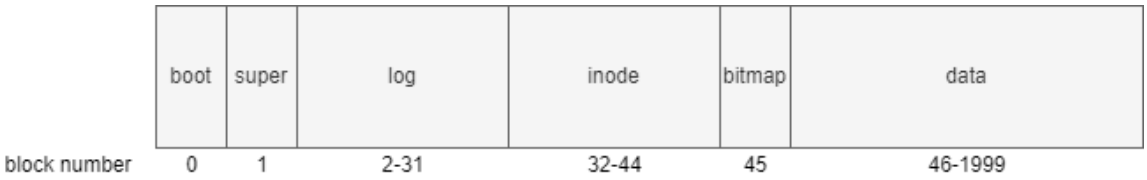
- 物理磁盘，可持久化存储文件
- buffer cache，缓存了磁盘中的盘块，避免频繁读取磁盘
- logging，文件系统的持久性
- inode cache，缓存使用到的inode
- file，管理不同类型文件，文件描述符
- syscall，文件系统接口

disk (磁盘)

布局

sector: 磁盘存取的最小单位。在xv6中为1kb

block: 文件系统存取的最小单位，为sector的任意整数倍。在xv6中为1kb



- boot block: 启动操作系统的代码
- super block: 描述文件系统信息
- log blocks:
- inode blocks: 存放所有inode
- bitmap block: 记录data block是否空闲
- data blocks: 存储文件和目录的内容

supber block

```
// kernel/fs.h
struct superblock {
  uint magic;        // Must be FSMAGIC
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
```

```
#define FSSIZE       2000  // size of file system in blocks
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NINODES 200
```

```
// kernel/fs.c
// there should be one superblock per disk device,
// but we run with only one device
struct superblock sb;

// Init fs
void
fsinit(int dev) {
  readsb(dev, &sb);
  if(sb.magic != FSMAGIC)
    panic("invalid file system");
  initlog(dev, &sb);
}

// Read the super block.
static void
readsb(int dev, struct superblock *sb)
{
  struct buf *bp;

  bp = bread(dev, 1);
  memmove(sb, bp->data, sizeof(*sb));
  brelse(bp);
}
```

buffer cache

struct buf

```
// kernel/buf.h
struct buf {
  int valid;   // has data been read from disk?
  int disk;    // does disk "own" buf?
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;      //
  struct buf *prev; // LRU cache list
  struct buf *next;
  uchar data[BSIZE];
};
```

bcache

```
// kernel/bio.c
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
  // 双向循环链表
  struct buf head;
} bcache;
```

bread()

```
// kernel/bio.c
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if(!b->valid) {
    virtio_disk_rw(b, 0);
    b->valid = 1;
  }
  return b;
}
```

bget()

```
// kernel/bio.c
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

  // 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);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  // 逆序遍历
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0) {
      b->dev = dev;
      b->blockno = blockno;
      // 还未从磁盘读取数据
      b->valid = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

brelese()

```
// kernel/bio.c
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移动到链表表头
    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);
}
```

logging

why

- case 1

  ```
  // kernel/sysfile.c
  static struct inode*
  create(char *path, short type, short major, short minor)
  {
  ...
    if((ip = ialloc(dp->dev, type)) == 0){
      iunlockput(dp);
      return 0;
    }
    <- crashed here, what will happen
  ...
  }
  ```

  crash会导致我们会丢失这个inode

- case 2

  在为文件分配block时

  1. 从 data blocks 中找到一块空闲 block
  2. 将该 block number 写入到文件的 inode 中
  3. 在bitmap中标记该block已使用

如果2，3之间 crash 会怎么样

crash 可能会导致这个 block 被分配给多个文件

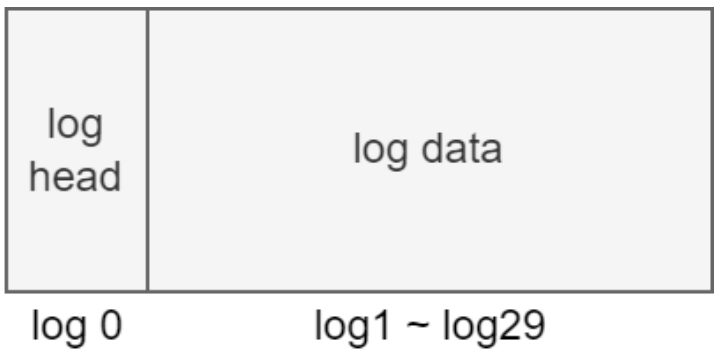
fatal !

what

buffer cache 之上的一种机制，用来保证系统调用的原子性，同时能够在系统 crash 之后进行 Fast Recovery

how

```
// kernel/log.c
struct logheader {
  int n;
  int block[LOGSIZE];
};

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



log 实现

- log write4

  当需要更新 inode block 或 bitmap block 或 data block 时，我们并不直接写入到磁盘对应的位置，而是记录一条 log 到磁盘的 log 分区

```
// kernel/log.c
void
log_write(struct buf *b)
{
  int i;

  acquire(&log.lock);
  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");
  // 要写入的 block number 已存在
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // log absorption
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n) {  // Add new block to log?
    bpin(b);
    log.lh.n++;
  }
  // i != log.lh.n
  // log 已存在并且未 commit, nothing to do
  release(&log.lock);
}
```

- commit

```
// kernel/log.c
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    write_head();    // Write header to disk -- the real commit
    install_trans(0); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

对单个 disk block 的读写具有原子性

commit 可保证系统调用的原子性

```
// kernel/log.c
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); // log block
    struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
    // 将 log 中记录的缓冲块号的缓冲块复制到 log 缓冲块
    memmove(to->data, from->data, BSIZE);
    // 将 log 缓冲块写出到磁盘
    bwrite(to);  // write the log
    brelse(from);
    brelse(to);
  }
}
```

```
// kernel/log.c
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  // 将内存中的 logheader 复制到 log head 的缓冲块
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  // 将 log head 的缓冲块写出到磁盘
  bwrite(buf);
  brelse(buf);
}
```

- install trans

```
// kernel/log.c
static void
install_trans(int recovering)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    if(recovering == 0)
      bunpin(dbuf);
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- clean log

```
// kernel/log.c
static void
commit()
{
...
  log.lh.n = 0;
  write_head();    // Erase the transaction from the log
}
```

- recovery

```
// kernel/log.c
static void
recover_from_log(void)
{
  read_head();
  install_trans(1); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}
```

usage

```
uint64
sys_open()
{
...
  // 合法性检查
  begin_op();
  ...
  log_write();
  ...
  log_write();
  ...
  end_op();
...
}
```

```
// kernel/log.c
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    // 有系统调用正在 commit
    if(log.committing){
      sleep(&log, &log.lock);
      // 可能超出 log 大小限制
    } 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;
      release(&log.lock);
      break;
    }
  }
}
```

```
// kernel/log.c
void
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  // 所有系统调用都已经 end_op()
  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.
    wakeup(&log);
  }
  release(&log.lock);

  // if log.outstanding != 0
  // noting to do

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

一些结构

```
// kernle/file.h
// 文件控制块
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

```
// kernel/fs.h
// 磁盘索引结点
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
#define NDIRECT 12
```

```
// kernel/file.h
// 内存索引结点
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

```
// kernel/fs.h
// 目录项
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

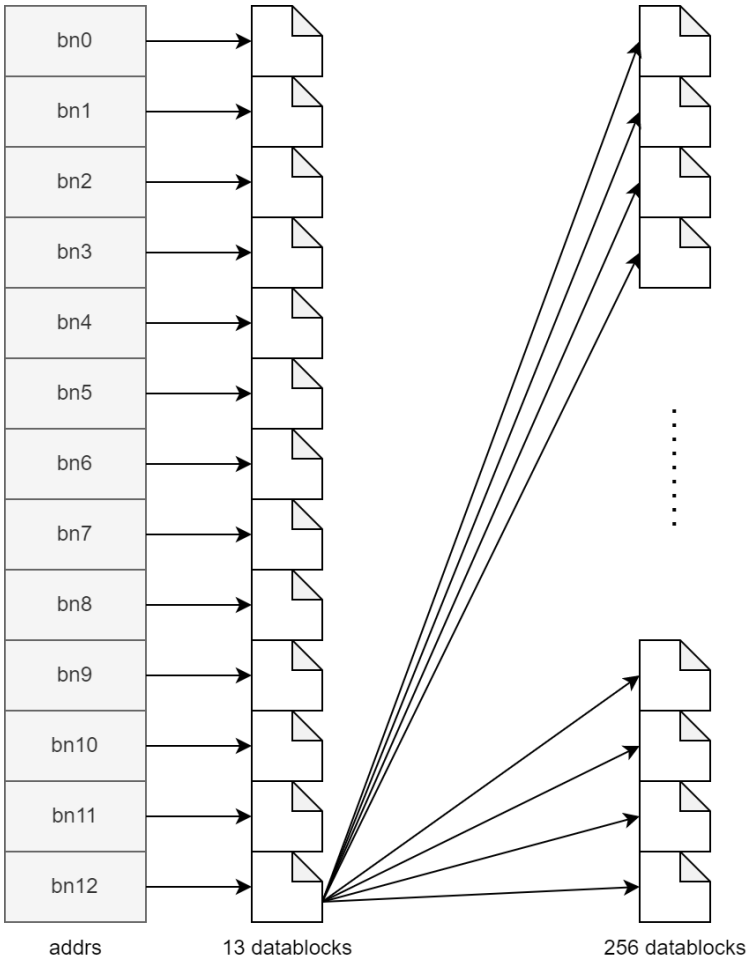
addition

Major number (主设备号)

> Traditionally, the major number identifies the driver associated with the device. A major number can also be shared by multiple device drivers.

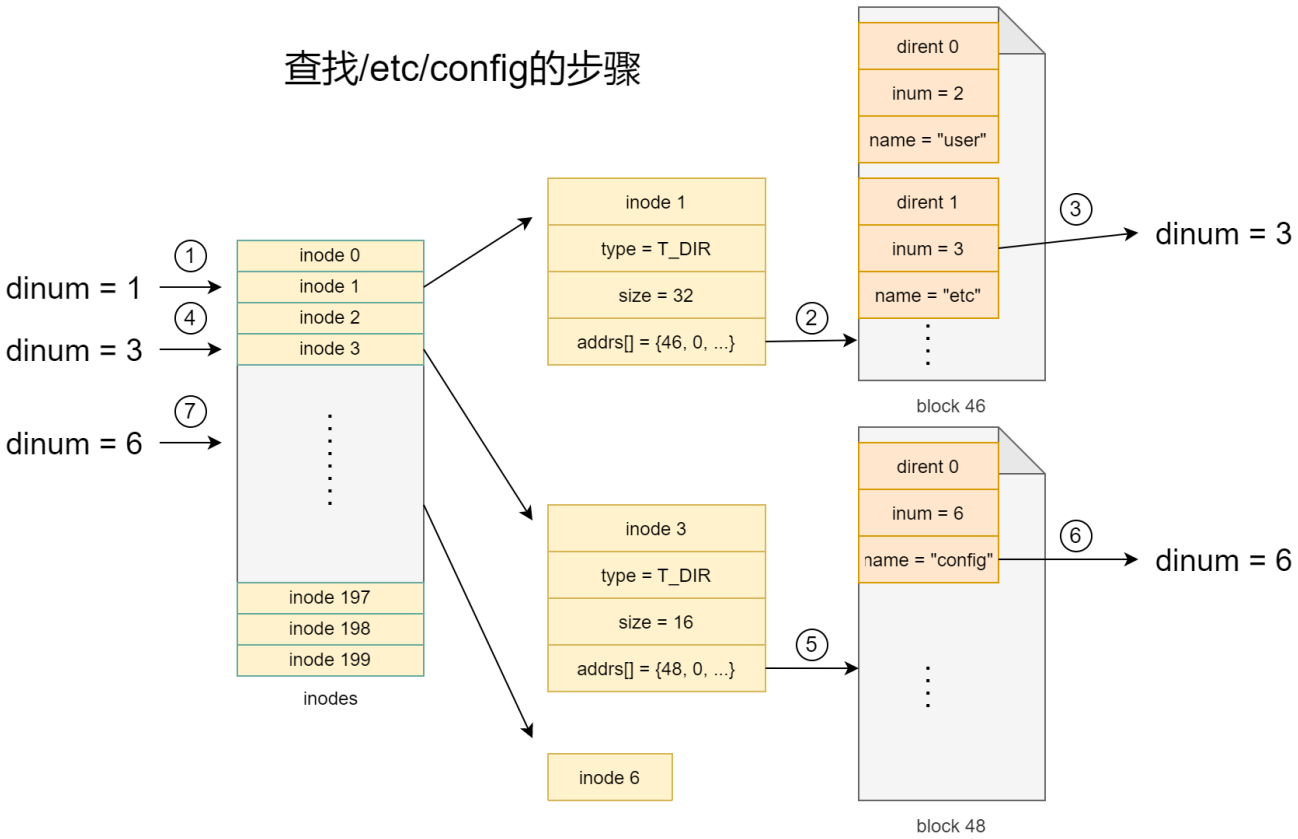
Minor number (次设备号)

> The major number is to identify the corresponding driver. Many devices may use the same major number. So we need to assign the number to each device that is using the same major number.



xv6文件系统增量索引示意图

目录查询




```c
// kernel/fs.c
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if(*path == '/')
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(myproc()->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(nameiparent && *path == '\0'){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

open系统调用

open()

```
void
ls(char *path)
{
  char buf[512], *p;
  int fd;

  if((fd = open(path, 0)) < 0){
    fprintf(2, "ls: cannot open %s\n", path);
    return;
  }
  ...
}
```

```
int open(const char* file, int omode);
```

- file: 文件名，相对路径和绝对路径
- omode: 打开方式

omode	value	描述
O_RDONLY	0	只读
O_WRONLY	1<<0	只写
O_RDWR	1<<1	读写
O_CREATE	1<<9	新建
O_TRUNC	1<<10	删除

- 返回值: 一个整数表示文件描述符，打开失败返回-1

sys_open()

```
// kernel/sysfile.c
uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;

  // 取参数
  argint(1, &omode);
  if((n = argstr(0, path, MAXPATH)) < 0)
    return -1;

  // 表示开始一个事务
  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  }
  // 不需要新建
  else {
    // 获取目标文件的inode
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    // 目录只能以只读方式打开
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  // 设备文件
  if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
    iunlockput(ip);
    end_op();
    return -1;
  }

  // 分配文件控制块和文件描述符
  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }

  // 设备文件
  if(ip->type == T_DEVICE){
    f->type = FD_DEVICE;
    f->major = ip->major;
  }
  // 文件或目录
  else {
```

```
    f->type = FD_INODE;
    f->off = 0;
  }
  f->ip = ip;
  // 非只写
  f->readable = !(omode & O_WRONLY);
  // 只写或可读可写
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

  // 删除文件
  if((omode & O_TRUNC) && ip->type == T_FILE){
    itrunc(ip);
  }

  iunlock(ip);
  // 结束一个事务
  end_op();

  // 返回文件描述符
  return fd;
}
```

create()

```
// kernel/sysfile.c
static struct inode*
create(char *path, short type, short major, short minor)
{
  struct inode *ip, *dp;
  char name[DIRSIZ];

  // struct inode* nameiparent(char *path, char *name)
  // 返回path的父目录的inode和目标文件的name
  // e.g., path="/etc/apt/config"
  // dp = inode("/etc/apt/")
  // name = "config"
  if((dp = nameiparent(path, name)) == 0)
    return 0;

  ilock(dp);

  // struct inode* dirlookup(struct inode *dp, char *name, uint *poff)
  // 查询dp目录下name文件的inode
  if((ip = dirlookup(dp, name, 0)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && (ip->type == T_FILE || ip->type == T_DEVICE))
      return ip;
    iunlockput(ip);
    return 0;
  }

  // 若文件不存在
  // 分配inode
  if((ip = ialloc(dp->dev, type)) == 0){
    iunlockput(dp);
    return 0;
  }

  // 设置inode
  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  // 将ip写入到磁盘
  iupdate(ip);

  // 目录文件
  if(type == T_DIR){  // Create . and .. entries.
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      goto fail;
  }

  // 添加目录项
  if(dirlink(dp, name, ip->inum) < 0)
    goto fail;

  if(type == T_DIR){
    // now that success is guaranteed:
    dp->nlink++;  // for ".."
    iupdate(dp);
  }

  iunlockput(dp);

  // 未释放ip的锁
  return ip;
```

```
 fail:
  // something went wrong. de-allocate ip.
  ip->nlink = 0;
  iupdate(ip);
  iunlockput(ip);
  iunlockput(dp);
  return 0;
}
```

file descriptor (文件描述符)

```
// kernel/sysfile.c
static int
fdalloc(struct file *f)
{
  int fd;
  struct proc *p = myproc();

  // NOFILE: 每个进程最多打开文件数
  for(fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd] == 0){
      p->ofile[fd] = f;
      // 文件描述符即为下标
      return fd;
    }
  }
  return -1;
}
```

```
struct proc {
...
  int pid;                     // Process ID
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
...
};
```