

操作系统期末大作业——实验报告

岳毅然 18364112 2018 级 智科二班

yueyr3@mail2.sysu.edu.cn

1. Xv6 lab: Multithreading/Uthread: switching between threads

1.1 问题重述

本实验将为用户级线程系统设计并实现上下文切换机制。xv6 有两个文件 `user/uthread.c`、`user/uthread_switch.S`、`Makefile`。`uthread.c` 包含大多数用户级线程包，以及三个简单测试线程的代码。线程包缺少一些用于创建线程和在线程之间切换的代码。`Makefile` 用于构建 `uthread` 程序。

实验的具体内容是创建线程并保存或恢复寄存器以在线程之间切换。需要在 `user / uthread.c` 中的 `thread_create()` 和 `thread_schedule()` 函数以及在 `user / uthread_switch.S` 中的 `thread_switch` 函数中添加代码。

一个目标是确保当 `thread_schedule()` 首次运行给定线程时，该线程在自己的堆栈上执行传递给 `thread_create()` 的函数。另一个目标是确保 `thread_switch` 保存要远离的线程的寄存器，恢复要切换到的线程的寄存器，并返回到后者线程指令中上次中断的位置。需要在 `thread_schedule` 中添加对 `thread_switch` 的调用，可以将所需的任何参数传递给 `thread_switch`，但目的是从线程 `t` 切换到 `next_thread`。

完成后，在 `xv6` 上运行 `uthread` 时，应该会看到相应的程序输出（三个线程可能以不同的顺序启动）。

1.2 解决方案

本实验可以参考进程上下文切换的方式来完成线程上下文的切换。大致思路为给线程添加上下文信息用于保存状态，然后在 `thread_schedule` 里添加交换上下文代码。

首先添加上下文结构到 `thread` 结构中。在 `kernel/proc.h` 中，对进程上下文的定义如图 1 所示，其切换上下文的汇编代码（图 2）在 `kernel/swtch.S` 中，仿照其结构，在 `user/uthread.c` 的 `struct thread` 添加寄存器字段，添加后的代码如图 4 所示。在 `uthread_Switch.S` 中补充线程上下文的切换如图 3 所示。

```
// Saved registers for kernel context switches.
struct context {
    uint64 ra;
    uint64 sp;

    // callee-saved
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
};
```

图 1-`proc.h` 原始进程上下文定义

```
# void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.

.globl swtch
swtch:
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret

.globl pthread_switch
pthread_switch:
    /* YOUR CODE HERE */

    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret
```

图 2-汇编代码

图 3- `uthread_Switch.S`

```
struct thread {
    uint64 ra;
    uint64 sp;
    char stack[STACK_SIZE];
    int state;
    // callee registers
    // thread_switch needs to save/restore only the callee-save registers.
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
};
```

图 4-`thread` 结构添加寄存器代码

之后修改 `thread_create`，添加保存新线程返回地址和栈指针，使之能够记录线程的返回地址 `ra` 与栈地址 `sp`。其中返回地址表示当切换线程时线程应该返回到的地址，即传入函数的入口 `func`。要注意的是存储栈指针时要注意 `sp` 一开始要指向栈底，为函数参数存储分配空间。具体代码见图 5。

```
void
thread_create(void (*func)())
{
    struct thread *t;

    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == FREE) break;
    }
    t->state = RUNNABLE;
    // YOUR CODE HERE
    t->ra = (uint64)func;
    /** 栈是倒着生长 -- addi sp, sp, -STACK_SIZE */
    t->sp = (uint64)(t->stack + STACK_SIZE);
}
```

图 5-修改 `thread_create()`

在 `thread_schedule` 中添加 `thread_switch` 调用，具体代码见图 6。

```
static void
thread_schedule(void)
{
    thread_p t;

    /* Find another runnable thread. */
    next_thread = 0;
    for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
        if (t->state == RUNNABLE && t != current_thread) {
            next_thread = t;
            break;
        }
    }

    if (t >= all_thread + MAX_THREAD && current_thread->state == RUNNABLE) {
        /* The current thread is the only runnable thread; run it. */
        next_thread = current_thread;
    }

    if (next_thread == 0) {
        printf("thread_schedule: no runnable threads\n");
        exit();
    }

    if (current_thread != next_thread) {          /* switch threads? */
        next_thread->state = RUNNING;
        t = current_thread;
        current_thread = next_thread;
        thread_switch((uint64)t, (uint64)next_thread);
    } else
        next_thread = 0;
}
```

图 6-修改 `thread_schedule`

1.3 运行实例

运行实例见图 7、图 8，此输出来自三个测试线程（main 函数中），每个测试线程都有一个循环，该循环打印一行，然后将 CPU 传递给其他线程。

```
leakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprec

xv6 kernel is booting

virtio disk init 0
hart 2 starting
hart 1 starting
init: starting sh
$ Uthread
exec Uthread failed
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_a 92
thread_b 92
thread_c 93
thread_a 93
thread_b 93
thread_c 94
thread_a 94
thread_b 94
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$ |
```

图 7&8-输出结果

2. Xv6 lab: Lock/Memory allocator

2.1 问题重述

多核计算机上并行性差的一个常见症状是高锁争用。在未修改前锁竞争剧

烈，因为 xv6 上只有一个内存链表供于多个 CPU 使用。申请内存时添加锁，其他 CPU 需要切换进行内存申请必须等待当前线程释放内存锁。

指导书中给出一个建议：让每个 CPU 拥有自己的内存链表进行分配，从而无需获取当前线程的内存链表。同时还添加了一个要求：当一个 CPU 内存链表不足的时候，可从其他链表使用内存（或换一个线程）来分配内存。

本实验中需要重新设计代码以提高并行性。本实验使用 `kalloc` 来测试在多线程环境下获取 `kmem` 和 `bcache` 锁的竞争次数。为了减少争用，提高并行性通常需要同时改变数据结构和锁定策略。这需要我们为 xv6 内存分配器和块缓存执行此操作。

2.2 解决方案

首先根据题意，出现大量竞争情况是因为 `kalloc()` 只有一个被单个锁保护的 `freelist`。为消除锁竞争需重新设计内存分配器，为每个 CPU 维护多个空闲列表，每个列表都有自己的锁，以避免单一的锁和列表（图 9）。修改结构体时参考 `param.h` 的常量声明（图 10），`param.h` 主要用于声明基本的一些常量，包括内核栈大小等。

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU];
```

图 9-修改 `kalloc.c` 结构体

```
#define NPROC      64 // maximum number of processes
#define NCPU       8  // maximum number of CPUs
#define NOFILE     16 // open files per process
#define NFILE      100 // open files per system
#define NINODE     50 // maximum number of active i-nodes
#define NDEV       10 // maximum major device number
#define ROOTDEV    0  // device number of file system root disk
#define MAXARG     32 // max exec arguments
#define MAXOPBLOCKS 10 // max # of blocks any FS op writes
#define LOGSIZE    (MAXOPBLOCKS*3) // max data blocks in on-disk log
#define NBUF       (MAXOPBLOCKS*3) // size of disk block cache
#define FSSIZE     1000 // size of file system in blocks
#define MAXPATH    128 // maximum file path name
#define NDISK      2
```

图 10-`param.h` 声明

由于我们修改了 `kmem`，对于 `kinit`、`kalloc`、`kfree` 函数中对内存的操作，我们都需要做出相应修改。修改 `kinit()`，设置循环遍历设置各个表的锁。修改 `kfree()` 获取和释放相应 CPU 的锁，这里在获取 CPU id 的时候把中断关闭防止

线程切换。然后再给 `kalloc()` 多添加一个可以偷其他 CPU 内存链表的功能。三个函数具体修改情况见图 11、图 12、图 13。

```
void
kinit()
{
    for(int i = 0; i < NCPU; i++)
        initlock(&kmem.lock[i], "kmem");
    freerange(end, (void*)PHYSTOP);
}
```

图 11- `kinit()` 函数修改

```
// initializing the allocator; see kinit above.)
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);

    r = (struct run*)pa;

    push_off();
    int id = cpuid();
    pop_off();

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

图 12- `kfree()` 函数修改

需要特别注意的是，当一个 CPU 没有空闲列表但另一个 CPU 的列表有空闲内存时，该 CPU 必须“窃取”另一个 CPU 的空闲列表的一部分，这种窃取可能会导致锁争用。详见图 14 代码的判断 `if(!r)` 部分。

此外在提示中写到，在调用 `cpuid` 的时候，必须保证 `interrupts` 是 `turned off` 状态，而我们是通过 `push_off()` 和 `pop_off()` 来切换其状态的。

```

// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
    struct run *r;

    push_off();
    int id = cpuid();
    pop_off();

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

    // steal memory
    if (!r)
    {
        for (int i = 0; i < NCPU; i++)
        {
            if (i == id) continue;
            acquire(&kmem[i].lock);
            r = kmem[i].freelist;
            if (r) {
                kmem[i].freelist = r->next;
                release(&kmem[i].lock);
                break;
            }
            release(&kmem[i].lock);
        }
    }

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

```

图 13- kalloc()函数修改

2.3 运行实例

在修改前 make qemu，输入 kalloctest，测试初始状态下的锁竞争情况（图 14，287629），修改后运行发现竞争数量明显减少（图 15，5772）。

```

$ kalloctest
start test0
test0 done: #test-and-sets = 287629
start test1
total allocated number of pages: 32479 (out of 32768)
test1 done

```

图 14-初始锁竞争情况

```

xv6 kernel is booting
virtio disk init 0
hart 2 starting
hart 1 starting
init: starting sh
$ kallocstest
start test0
test0 done: #test-and-sets = 5772
start test1
total allocated number of pages: 32479 (out of 32768)
test1 done
$ |

```

图 15-修改后锁竞争情况

3. Xv6 lab: Lock/Buffer cache

3.1 问题重述

buf 对磁盘上的 block 块进行缓存。buf 是一个 LRU 链表。在多线程环境下，如果不修改磁盘缓冲区结构 buffer cache，则会导致获取缓冲区时锁竞争严重，因为该结构只有一个全局锁。为解决这个问题，可以为每个 CPU 分配属于自己的内存链表，但是不能为 CPU 分配属于自己的磁盘缓冲区，因为磁盘缓冲区本身比较大，应该让多个 CPU 访问磁盘同一块区域时访问同一块缓冲区，以减少空间浪费和设计上的耦合。

出现大量竞争情况是因为 bio.c 中缓冲区的结构体只有一个全局锁（图 16），如果多个进程密集地使用文件系统，它们可能会争用 bcache.lock，其保护着 cached block buffers 的链表和一些其他属性。因此解决方案呼之欲出，修改 bcache 的结构体定义，并对其配套函数进行修改即可。

```

struct {
    struct spinlock lock;
    struct buf buf[NBUF];

    // Linked list of all buffers, through prev/next.
    // head.next is most recently used.
    struct buf head;
} bcache;

```

图 16-bio.c 缓冲区结构体

3.2 解决方案

首先修改函数使 bcache 中不同块的并发查找和释放不太可能在锁上发生冲突。根据题目所说 “*We suggest you look up block numbers in the cache with a hash table that has a lock per hash bucket.*” 我们需要将其从链表修改为哈希表，并为

每个 bucket 都分配锁，separate chaining 的方式来组织 buffer，使用 blockno 作为 key，同时去掉 head。题目提醒建议用质数作为 hash table 大小，这里选用指导书建议的 13，将 buf 分段映射到不同的 hash 桶里。修改代码见图 17。

```
struct {
    struct spinlock lock[NBUCKETS];
    struct buf buf[NBUF];
    struct buf hashbucket[NBUCKETS];

    // Linked list of all buffers, through prev/next
    // head.next is most recently used.
    //struct buf head;
} bcache;
```

图 17-修改后的 bcache 结构

然后修改和 head 有关的 binit、bget、brelse 等函数。buffer 没有和磁盘块对应起来，如图 18 所示，在 binit()初始化函数中 blockno 全部初始化为 0，将其全部放在第一个 bucket 中。

Bget()负责寻找一个扇区的 buf，将解决其他 bucket 缺少 buffer 的问题（如图 19）。首先在 blockno 对应的 bucket 中找，如果没有找到，那么需要到其他 bucket 中找，在其他 bucket 中找到 buffer 后将其插入到原 bucket 中，插入后或未找到才能释放当前的锁。代码中用 nh 和 h 来表示下一个要查找的 bucket 和当前 bucket，当所有 nh 变为 h 说明 buffer 都被占用。

brelse()函数负责释放 buf，brelse、bpin、bunpin 具体配套修改较为简单，具体见图 20。

```

void
binit(void)
{
    struct buf *b;
    for(int i=0; i<NBUCKETS; i++){
        initlock(&bcache.lock[i], "bcache.bucket");
        // 仍然将每个bucket的头节点都指向自己
        b=&bcache.hashbucket[i];
        b->prev = b;
        b->next = b;
    }

    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
        // 插在表头
        b->next = bcache.hashbucket[0].next;
        b->prev = &bcache.hashbucket[0];
        initsleeplock(&b->lock, "buffer");
        bcache.hashbucket[0].next->prev = b;
        bcache.hashbucket[0].next = b;
    }
}

```

图 18-修改后的 binit()函数

```

// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return locked buffer.
static struct buf*
bget(uint dev, uint blockno)
{
    struct buf *b;
    int h=bhash(blockno);
    acquire(&bcache.lock[h]);

    for(b = bcache.hashbucket[h].next; b != &bcache.hashbucket[h]; b = b->next){
        if(b->dev == dev && b->blockno == blockno){
            b->refcnt++;
            release(&bcache.lock[h]);
            acquiresleep(&b->lock);
            return b;
        }
    }
}

```

图 19 (a) -修改后的 bget()函数 (1)

```

int nh=(h+1)%NBUCKETS;

while(nh!=h){
    acquire(&bcache.lock[nh]); // 获取当前bucket的锁
    for(b = bcache.hashbucket[nh].prev; b != &bcache.hashbucket[nh]; b = b->prev){
        if(b->refcnt == 0) {
            b->dev = dev;
            b->blockno = blockno;
            b->valid = 0;
            b->refcnt = 1;
            // 从原来bucket的链表中断开
            b->next->prev=b->prev;
            b->prev->next=b->next;
            release(&bcache.lock[nh]);
            // 插入到blockno对应的bucket中
            // 头插法
            b->next=bcache.hashbucket[h].next;
            b->prev=&bcache.hashbucket[h];
            bcache.hashbucket[h].next->prev=b;
            bcache.hashbucket[h].next=b;
            release(&bcache.lock[h]);
            acquiresleep(&b->lock);
            return b;
        }
    }
    // 如果当前bucket里没有找到，在转到下一个bucket之前，记得释放当前bucket的锁
    release(&bcache.lock[nh]);
    nh=(nh+1)%NBUCKETS;
}
panic("bget: no buffers");
}

```

图 19 (b) -修改后的 bget()函数 (2)

```

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

    releasesleep(&b->lock);
    int h=bhash(b->blockno);
    acquire(&bcache.lock[h]);
    b->refcnt--;
    if (b->refcnt == 0) {
        // no one is waiting for it.
        // 下面做的就是b从原来的位置取下来 放在链表开头 (头插法)
        b->next->prev = b->prev;
        b->prev->next = b->next;
        b->next = bcache.hashbucket[h].next;
        b->prev = &bcache.hashbucket[h];
        bcache.hashbucket[h].next->prev = b;
        bcache.hashbucket[h].next = b;
    }

    release(&bcache.lock[h]);
}

void
bpin(struct buf *b) {
    int h=bhash(b->blockno);
    acquire(&bcache.lock[h]);
    b->refcnt++;
    release(&bcache.lock[h]);
}

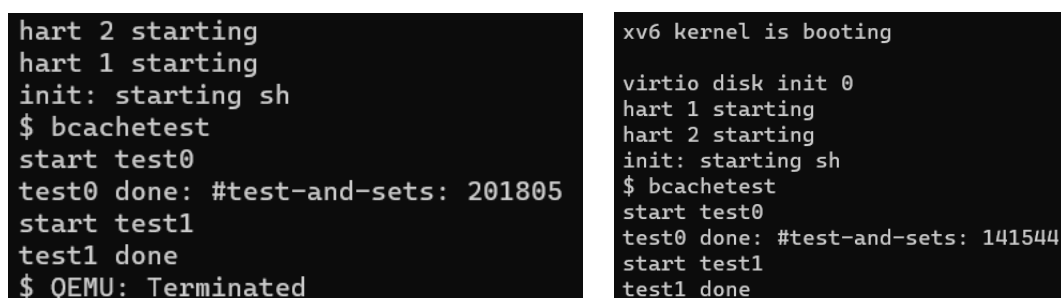
void
bunpin(struct buf *b) {
    int h=bhash(b->blockno);
    acquire(&bcache.lock[h]);
    b->refcnt--;
    release(&bcache.lock[h]);
}

```

图 20-修改后的 brelse()函数等

3.3 运行实例

如图 21 所示, (a)为修改前的 bcachetest 测试截图, (b)为修改后的截图。



```
hart 2 starting
hart 1 starting
init: starting sh
$ bcachetest
start test0
test0 done: #test-and-sets: 201805
start test1
test1 done
$ QEMU: Terminated
```

```
xv6 kernel is booting
virtio disk init 0
hart 1 starting
hart 2 starting
init: starting sh
$ bcachetest
start test0
test0 done: #test-and-sets: 141544
start test1
test1 done
```

图 20 (a、b) -修改前后运行结果

4. Xv6 lab: File System/Large files

4.1 问题重述

在 xv6 系统中每个文件具有一个 inode 节点, inode 节点包含 12 个直接地址块和 1 个间接地址块, 每个间接地址块储存 256 个直接地址块。所以一个 inode 节点最多储存 $12+256=268$ 个地址块, 即一个文件的信息最多占用 268 个 block。我们要增加 xv6 文件的最大大小, 修改 xv6 的 inode 节点的格式, 以支持每个 inode 中的二级间接索引, 其中包含 256 个一级间接索引块的地址, 每个块最多可以包含 256 个数据块地址, 11 个直接地址块, 1 个间接地址块, 1 个二级间接地址块。因此一个文件最多可以包含 $256 \times 256 + 256 + 11$ 个块 (将为二级间接索引块牺牲一个直接索引块号, 故 $12-1=11$)。

4.2 解决方案

整体思路为向 xv6 添加符号链接。符号链接 (硬链接) 通过路径名引用链接文件, 当符号链接打开时, 内核将跟随链接指向引用的文件。

首先修改文件系统的 block 数目 (如图 21 白框所示), 将 kernel/param.h 中的 FSSIZE 的数值增大为 200000。

```

#define NDEV      10 // maximum major device number
#define ROOTDEV   0 // device number of file system root
#define MAXARG     32 // max exec arguments
#define MAXOPBLOCKS 10 // max # of blocks any FS op writes
#define LOGSIZE    (MAXOPBLOCKS*3) // max data blocks in on-
#define NDUP      (MAXOPBLOCKS*3) // size of disk block ca
#define FSSIZE     200000 // size of file system in blocks
#define MAXPATH    128 // maximum file path name
#define NDISK      2

```

图 21-修改块大小

inode 在 xv6 文件系统中有两种类型，一种是在系统实际运行时在内存中的 INODE（即 fsvar.h 中的 inode 结构），另一种是在磁盘上的 INODE（即 fs.h 中的 dinode）。根据题目要求，牺牲了一个直接索引，将其变为二级间接索引，所以修改 fs.h，修改一些宏定义和 dinode 结构体如图 22 所示。

```

#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDOUBLE_INDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLE_INDIRECT)
// #define MAXFILE (NDIRECT + NINDIRECT)

// On-disk inode structure
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+2]; // Data block addresses
};

```

图 22-修改定义 fs.h

根据提示 “If you change the definition of NDIRECT, you’ll probably have to change the declaration of addrs[] in struct inode in file.h.”，修改了 NDIRECT 的宏定义后需要修改 file.h 文件里 inode 结构体中 addrs[] 的定义（图 23），以确保 inode 和 dinode 的在 addrs[] 中有相同的元素个数。

```

// in-memory copy of an inode
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+2];
};

```

图 23-修改定义 file.h

修改 fs.c 文件中的 bmap() 函数，该函数的两个参数分别是分配的 inode 节点的地址和分配的文件大小，返回 inode 的第 n 个块(block)，如果第 n 块是直接块则可以马上返回，否则要加载 block 然后读到相应的块后返回。根据 fs.h 中的宏，1 个一级间接索引可以容纳 $BSIZE/sizeof(uint)=1024/4=256$ 个地址，要求增加的二级间接索引通过这 256 个地址再去寻找一个中间块，然后再找到最终的数据块，即可达到存储更大文件的目的。

因此，bmap() 函数中代表“*logical block number*”的 bn 参数需要三个 if 来判断它属于何种类型的索引类型，以避免不必要的的时间或空间上的浪费。根据题意 The block numbers in ip->addrs[]，然后分析并模仿直接索引和一级间接索引的程序，完成二级间接索引的程序如图 24 所示。

```

}
bn -= NDIRECT;

if(bn < NDOUBLE_INDIRECT){
    uint bn_level_1 = bn / NINDIRECT;
    uint bn_level_2 = bn % NINDIRECT;
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
        ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn_level_1]) == 0){
        a[bn_level_1] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    struct buf *bp2 = bread(ip->dev, addr);
    uint addr, *a2 = (uint*)bp2->data;
    if((addr = a2[bn_level_2]) == 0){
        a2[bn_level_2] = addr = balloc(ip->dev);
        log_write(bp2);
    }
    brelse(bp2);
    return addr;
}

panic("bmap: out of range");

```

图 24-修改 fs.c

4.3 运行实例

修改前输入 bigfile 显示 “.. wrote 268 blocks bigfile: file is too small”，修改后可见创建了 65803 ($256 \times 256 + 256 + 11$) 个块文件，通过测试。

```

init: starting sh
$ bigfile
..
wrote 268 blocks
bigfile: file is too small

```

图 25-修改前的 bigfile 测试

```
$ bigfile  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
wrote 65803 blocks  
bigfile done; ok  
$
```

图 26-修改后的 bigfile 测试