



中山大學

操作系统期末大作业（3）

姓名：刁琪

学号：18364020

邮箱：1294581108@qq.com

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

(1) 实验目的:

在线程切换之间，为用户级线程系统设计上下文切换机制，设计一个创建线程的计划，并保存和恢复寄存器，以实现线程之间的切换。

首先在 xv6 中有两个文件 `user/uthread.c` 和 `user/uthread_switch.S`，以及 `Makefile` 中的一个用于构建 `uthread` 程序的规则。`Uthread.c` 文件中包含大多数用户及线程包，以及三个简单测试线程（`thread_a`、`thread_b`、`thread_c`）的代码。单个测试线程中，每个测试线程有一个循环，该循环打印一行，然后将 CPU 传递给其他线程。但是线程包缺少一些用于创建线程和在线程之间切换的代码。这就使得设计之前运行 `uthread` 时，看不到任何输出。

为实现实验目的，我们需要在 `user/uthread.c` 的 `thread_create()` 和 `thread_schedule()` 中，以及 `user/uthread_switch.S` 文件中添加代码。实现两个目标：

- 1) 确保当 `thread_schedule()` 第一次运行给定线程时，该线程在自己的堆栈上执行传递给 `thread_create()` 的函数；
- 2) 确保 `uthread_switch` 保存要切走的线程的寄存器，恢复要切换到线程的寄存器，并返回到后者线程指令中上次中断的位置；

为实现保存/恢复寄存器，我们通过修改线程结构以保存寄存器。在 `thread_schedule` 中添加对 `thread_switch` 的调用，将所需的任何参数传递给 `thread_switch`。

(2) 实验步骤:

Step1: 切换分支

```
$ git fetch
$ git checkout thread
$ make clean
```

Step2: 保存当前线程的上下文

参考 `kernel/swtch.S` 文件中保存/恢复寄存器的代码，在 `user/uthread_switch.S` 文件添加同样的保存/加载进程寄存器代码，保存线程的上下文，如下图 1-1 所示：

其中 `ra` 寄存器保存的是当前函数的返回地址；`sp` 寄存器保存的是当前线程的内核栈地址。

这步是保存要切走的线程的寄存器到存储器 `a0`，从存储器 `a1` 恢复（加载）要切换到线程的寄存器。`a0` 会被内核刷新成当前进程结构体起始地址，所以对 `a0`（`a0` 对应函数的第一个参数，也就是原线程）做偏移以存放 13 个寄存器（`ra`、`sp`、`s0-s11`）。因为这些寄存器都是 64 位，所以每次都要偏移 8 个字节。`sd` 表示将寄存器的内容存储到存储器中。`ld` 相反，会从存储器 `a1`（`a1` 对应函数的第一个参数，也就是切换到线程）中读取 13 个寄存器，完成线程的上下文切换。

函数最后的 `ret` 会将 `ra` 的内容设置为 `pc`（程序计数器），由于 `ra` 已经被刷新为新线程（要切换到线程）的起始地址，所以 CPU 会跳转到新线程指定语句执行新线程。这样用户就可以通过调用 `thread_switch`，实现线程之间的切换。

下面有几个补充说明：

- 1) `thread_switch` 函数上半部分保存了 `ra`、`sp` 等寄存器，但是没有保存程序计数器

pc。一般来说，线程的状态包含了三个部分：

- a) 程序计数器 pc：表示当前线程执行指令的位置；
- b) 保存变量的寄存器；
- c) 程序的栈 stack：每个线程拥有属于字节的栈，栈记录了函数调用的记录，并反映当前线程的执行点；

但是在 thread_switch 函数中，上半部分保存了 ra、sp 等等寄存器，但是并没有保存程序计数器 pc，因为程序计数器并没有有效信息。我们在 thread_switch 函数中关心的是我们从那里调用进入 thread_switch 函数，因为当我们通过 thread_switch 恢复执行当前线程并且从 thread_switch 函数返回时，我们希望能够从调用点继续执行。ra 寄存器保存了 thread_switch 函数的调用点，所以这里保存的是 ra 寄存器；

2) RISC-V 中有 32 个寄存器，但是 thread_switch 中只保存并恢复了 14 个寄存器。

因为 thread_switch 函数是从 C 代码调用的，所以 Caller Saved Register 会被 C 编译器保存在当前的栈上。我们在 thread_switch 函数中只需要处理 C 编译器不会保存，但是对于函数又有一些寄存器。所以在切换线程的时候，我们只需要保存 Callee Saved Register 寄存器。

```
.globl thread_switch
thread_switch:
/* YOUR CODE HERE */
/*参考kernel/switch.S保存当前线程的寄存器到a0中*/
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)

/*从a1恢复（加载）即将要切换到线程（新线程）的寄存器*/
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 /* 将ra内容返回到pc中，让CPU执行新线程内容 */
```

图 1-1 user/thread_switch.S

Step3: 修改线程结构

在 user/uthread.c 文件中的 struct thread 结构体中添加寄存器的字段，修改线程结构，如下图 1-2 所示；使得一个线程结构中包含该线程的寄存器内容，保存该线程的寄存器。

其中 ra 指向该线程的返回地址，sp 指向该线程的栈顶，剩下的 s0-s11 是需要保存 Callee Saved Register 寄存器。

```
//在struct thread中添加寄存器的字段
struct thread {
    // YOUR CODE HERE
    //保存该线程的寄存器
    uint64 ra; //返回地址
    uint64 sp; //栈顶地址
    // 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;

    char    stack[STACK_SIZE]; /* the thread's stack */
    int     state;             /* FREE, RUNNING, RUNNABLE */
};
```

图 1-2 user/thread.c/struct thread

Step4: 创建新线程

在 user/uthread.c 文件中的 thread_create() 函数中添加代码，将新创建的线程指针的 ra 寄存器保存该线程的入口地址；sp 寄存器指向该线程的栈顶，如下图 1-3 所示。

即创建线程 thread_a、thread_b、thread_c，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，即刚开始运行的时候，将线程 a、b、c 的线程状态改为 runnable，以便后面使用 thread_schedule 函数选择下一个切换的线程时，可以被选择到。

```
94
95 void
96 thread_create(void (*func)())
97 {
98     struct thread *t;
99     //遍历线程数组all_thread,如果有个线程工作完全结束了 (be free) 跳出循环
100     for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
101         if (t->state == FREE) break;
102     }
103     //刚开始的时候将状态为free的线程t设置为runnable
104     //表示该线程目标已经准备好，期待被调度器选中，占用CPU
105     t->state = RUNNABLE;
106     // YOUR CODE HERE
107     //添加保存新线程返回地址和栈指针
108     t->ra = (uint64)func; //ra指向线程入口地址
109     //栈指针sp指向struct thread->stack的最后一个元素的地址（栈顶），因为栈指针是从高地址向低地址增长的
110     t->sp = (uint64)&t->stack[STACK_SIZE-1];
111
112
113 }
```

图 1-3 user/thread.c/thread_create()

Step5: 实现真正的线程切换

在 user/uthread.c 文件中的 thread_schedule() 函数中添加 thread_switch() 的调用, 如下图 1-4 所示。

在线程调用 thread_switch 函数时, 可以切走 t 线程, 切换到 next_thread, 即下一个线程。如果没有这个代码, schedule 函数并没有实现实际意义上的切换, 只是将指针指向的线程修改了一下, 并没有切换掉 CPU 上正在运行的线程, 要经过 thread_switch 中的 sd、ld 操作, 刷新 ra 指针指向的线程起始地址, CPU 才会在另一个线程上工作, 实现真正意义上的线程切换。

```
thread_schedule(void)
{
    struct thread *t, *next_thread; // 创建一个指向下一个线程的指针

    /* 找到一个状态为runbale的线程 */
    next_thread = 0;
    t = current_thread + 1; // 遍历线程数组的指针
    // 遍历线程数组中的线程, 保证数组中每个线程都被遍历到
    for(int i = 0; i < MAX_THREAD; i++){
        if(t >= all_thread + MAX_THREAD)
            t = all_thread; // 循环遍历一遍
        // 如果找到一个状态为runable的线程, 将它设为下一个线程next_thread
        if(t->state == RUNNABLE) {
            next_thread = t;
            break;
        }
        t = t + 1;
    }

    // 如果遍历后找不到RUNABLE的线程, 表示当前没有需要CPU的线程
    if (next_thread == 0) {
        printf("thread_schedule: no runnable threads\n");
        exit(-1);
    }

    // 当找到的RUNABLE线程不是当前线程时, 可以进行线程的切换
    if (current_thread != next_thread) {
        next_thread->state = RUNNING; // 将下一个线程的状态设置为RUNNING
        t = current_thread;           // 将当前线程指针保存在t中
        current_thread = next_thread; // 并将下一个线程设为当前线程
        /* YOUR CODE HERE
        * 添加调用thread_switch, 把进程从t切换到next_thread
        */
        thread_switch((uint64)t, (uint64)next_thread);
    } else
        next_thread = 0;
}
```

图 1-4 user/thread.c/thread_schedule()

Step6: 验证线程切换

按上述步骤修改好代码之后，启动 qemu，运行 uthread，输出结果如下图 1-5 所示。

从实验结果中可以看出，线程 a、b、c 之间一直在切换交替工作，每个线程输出一行之后，交出 CPU，让调度器程序 thread_schedule 将 CPU 调度给另外两个线程使用。三个线程的任务交替完成，最后几乎在同时完成所有任务，给用户一种三个线程并行计算的“假象”。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ 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_b 2
thread_c 3
thread_a 3
thread_b 3

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 1-5 uthread 测试结果

(3) 实验总结及内容补充

- 1) 回答 Lab 中提出的问题：为什么 thread_switch 只需要保存/恢复被调用者保存的寄存器？

thread_switch 是参考 kernel/swtch.S 文件编写的，和 swtch.S 实现同样的功能。thread_switch 只保存了 callee-saved 寄存器。callee-saved 寄存器是非易失性寄存器，用于保存应在每次调用中保留长寿命值。当调用者进行过程调用时，可以期望这些寄存器在被调用者返回后将保持相同的值。

callee_saved 寄存器由调用的 C 代码保存在堆栈上。thread_switch 知道 struct context 中每个寄存器字段的偏移量。它不保存 pc（程序计数器），相反 thread_switch 保存 ra 寄存器，即保存了 thread_switch 应该返回的地址。现在，thread_switch 从新的上下文中恢复寄存器，新的上下文中保存着前一次 thread_switch 所保存的寄存器值。当 thread_switch 返回时，它返回到被恢复的 ra 寄存器所指向的指令，也就是新线程之前调用 thread_switch 的指令。此外，它还会在新线程的栈上返回。

2) 代码跟踪调试

为更详细地了解 thread.c 文件实现进程切换的流程，在代码中添加输出代码，输出代码运行状况，跟踪代码运行步骤，结果如下图 1-6 所示。

```
hart 2 starting
hart 1 starting
init: starting sh
$ uthread

thread_init_current_thread:3832

thread_create
thread 12144 is free
thread_create
thread 20456 is free
thread_create
thread 28768 is free
#####finish create 3 threads#####

current_thread:3832 next_thread:12144
thread_a started
thread_a call yield!
current_thread:12144 next_thread:20456
thread_b started
thread_b call yield!
current_thread:20456 next_thread:28768
thread_c started
thread_c 0
current_thread:28768 next_thread:12144
thread_a 0
current_thread:12144 next_thread:20456
thread_b 0
current_thread:20456 next_thread:28768
thread_c 1
current_thread:28768 next_thread:12144
thread_a 1
current_thread:12144 next_thread:20456
thread_b 1
current_thread:20456 next_thread:28768
thread_c 2
current_thread:28768 next_thread:12144
thread_a 2
current_thread:12144 next_thread:20456
```

图 1-6 调试结果

根据 main 函数,如图 1-7 所示,图中第一行输出“thread_init_current_thread:3832”可以知道一开始进入 thread_init() 函数,现在的 current_thread 是 all_thread 线程数组中的第一个线程;

```
int
main(int argc, char *argv[])
{
    a_started = b_started = c_started = 0;
    a_n = b_n = c_n = 0;
    thread_init(); //初始化
    //创建三个线程
    thread_create(thread_a);
    thread_create(thread_b);
    thread_create(thread_c);
    printf("#####finish create 3 threads#####\n\n");
    //创建好线程之后进入调度程序
    thread_schedule();
    exit(0);
}
```

图 1-7 user/thread.c/main()

后续红框中的三个连续的“thread_create”输出就是创建三个线程：thread_a、thread_b、thread_c。三个线程的指针分别为：12144、20456、28768。随后进入 thread_schedule() 调度函数。在调度函数中，我们需要在 all_thread 列表中找到一个状态为“RUNNABLE”的线程作为下一个 CPU 去运行的线程。在大小为 MAX_THREAD=4 的列表中顺序循环查找，如果找到一个 RUNNABLE 的线程，将它设为下一个线程 next_thread。如果找到的 next_thread 不是当前的线程 current_thread，可以进行线程切换。将 next_thread 的状态设置为“RUNNING”，然后调用 thread_schedule() 函数将 current_thread 和

next_thread 切换。

如果遍历完列表之后找不到状态为“RUNNABLE”的线程，说明当前没有需要使用 CPU 的线程，退出。代码如上图 1-4 所示。

在第一次调用 thread_schedule() 函数时，会选择线程 thread_a，进入 thread_a() 函数中，一开始会启动 thread_a，然后判断另外两个线程是否有一个还没启动，如果剩下两个线程中有一个没有启动，该线程就会调用 thread_yield() 函数，让出 CPU，让其他的线程先启动。所以在图 1-6 的红色框中，thread_a 启动后会让出 CPU，CPU 给到 thread_b，b 启动后由于 thread_c 还没有启动，所以继续出让 CPU 给 thread_c。代码如下图 1-8 所示。

```
void
thread_a(void)
{
    int i;
    //启动线程a
    printf("thread_a started\n");
    a_started = 1;
    //当线程b、c有没有开始（start）的，主动让出CPU，让他们也先开始
    while(b_started == 0 || c_started == 0)
    {
        printf("thread_a call yield!\n");
        thread_yield(); //让出CPU
    }

    //当a、b、c三个线程都启动之后，a就可以进入自己线程的运行
    //这里设定三个线程的任务都是输出100行
    for (i = 0; i < 3; i++) {
        printf("thread_a %d\n", i);
        a_n += 1; //表示线程a的任务进度
        thread_yield(); //线程a输出了一行自己的进度（i）就主动让出CPU
    }
    //完成了100行的输出，整个线程的任务就完成了
    printf("thread_a: exit after %d\n", a_n);
    //线程完成全部任务（输出100行）之后，状态设置为free
    current_thread->state = FREE;
    //然后调用线程调度函数
    thread_schedule();
}
```

图 1-8 user/thread.c/thread_a()

在 thread_c 启动后，线程都已启动，所以 thread_c 此时不需要让出 CPU，继续占用 CPU 完成自己的工作，所以后面紧跟着输出一行“thread_c 0”。在 thread_c 完成一个输出后，就会让出 CPU 给 thread_a 使用，输出“thread_a 0”后，thread_a 又会交出 CPU 调度给 thread_b。以此类推，所以结果输出都是以“cab”顺序。

到最后三个线程都完成 100 次的输出任务的时候，状态设置为“FREE”，调度程序就找不到状态为“RUNNABLE”的线程。

2、Xv6 lab: locks/Memory allocator

程序 `user/kalloctest` 强调了 xv6 的内存分配器：三个进程增加和缩小了它们的地址空间，从而导致对 `kalloc` 和 `kfree` 的许多调用。`kalloc` 和 `kfree` 获得 `kmem.lock`。`kalloctest` 打印由于尝试获取另一个内核已经持有的锁（对于 `kmem` 锁和一些其他锁）而导致的获取循环迭代的次数。也就是多个 CPU 在争夺同一个锁时，在此处会浪费很多时间。所以，获取中循环迭代的次数是锁争用的粗略度量。

对于每个锁，`acquire` 都会维护获取该锁的调用次数，以及获取中的循环尝试但未设置锁的次数。`kalloctest` 调用一个系统调用，该调用导致内核为 `kmem` 和 `bcache` 锁（这是本练习的重点）以及 5 个最有争议的锁打印这些计数。如果存在锁争用，则获取循环的迭代次数将很大。系统调用返回 `kmem` 和 `bcache` 锁的循环迭代总数。

(1) 实验目的：

出现“竞争锁”的根本原因是 `kalloc.c` 文件中是具有单个空闲列表 `freelist`，并受单个锁 `kmem.lock` 保护。当多个 CPU 需要分配或者释放内存页的时候，都需要获取到 `kmem.lock` 锁才可以对空闲列表进行增减操作，由此造成了争抢“锁”的情况。

要消除“竞争锁”，我们可以重新设计内存分配器以避免单个锁和列表。基本思想是为每个 CPU 维护一个空闲列表，每个列表都有自己的锁。不同 CPU 上的分配和释放内存页可以并行运行，即每个 CPU 将在各自的列表上运行。，这样大家都有“锁”和“列表”，不用进行争抢。

但还要考虑一种情况：一个 CPU 的空闲列表为空（无空闲的内存页），而另一个 CPU 的列表具有空闲内存的情况。在这种情况下，一个 CPU 必须“窃取”另一 CPU 空闲列表的一部分。窃取可能会引入锁争用，但是这种情况的争抢比全部 CPU 去争用一个锁情况好多了。所以本次实验的目的是：**实现每个 CPU 的空闲列表，并在 CPU 的空闲列表为空时进行窃取。**

(2) 实验步骤：

Step1: 切换分支

```
$ git fetch
$ git checkout lock
$ make clean
```

Step2: 修改 `kmem` 结构

让每一个 CPU 都有自己的空闲列表 `freelist` 和锁 `kmem.lock`，如下图 2-1 所示。



```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU]; //将kmem修改为数组，这样每个CPU对应一份freelist和lock
```

图 2-1 kernel/kalloc.c/struct `kmem`

`kmem` 结构中的 `freelist` 是 XV6 中的一个非常简单的数据结构，它会将所有的可用的内

存页保存于一个列表中。这样当 kalloc 函数需要一个内存 page 时，它可以从 freelist 中获取。Freelist 是一个指针，指向空闲列表中的第一个空闲页。

lock 是一个自旋锁，用于保护空闲列表，禁止两个 CPU 同时在同一个空闲列表中进行内存页的分配和释放。如果没有锁的保护，面对一些“锁争抢”情况，会导致丢失内存页。

Step3: 修改 kinit 函数

Kinit 函数中要对上面为每个 CPU 创建的 kmem 结构体进行初始化。该函数的主要作用还是为每个 CPU 的 kmem 调用 freerange 函数。freerange 函数会将 end~PHYSTOP 之间的内存一页页地加入到空闲列表 freelist 中管理，为每个 CPU 形成自己的空闲页表。

end 是内核 kernel 之后地第一个地址，在 kernel.sym 文件中有定义。PHYSTOP 是 kern_base+128M。Freerange()函数会将所有可用的内存分配给运行 freerange 的 CPU。但由于 end 并不是 kern_base，所以真正能分配出去地物理内存并没有 128M。

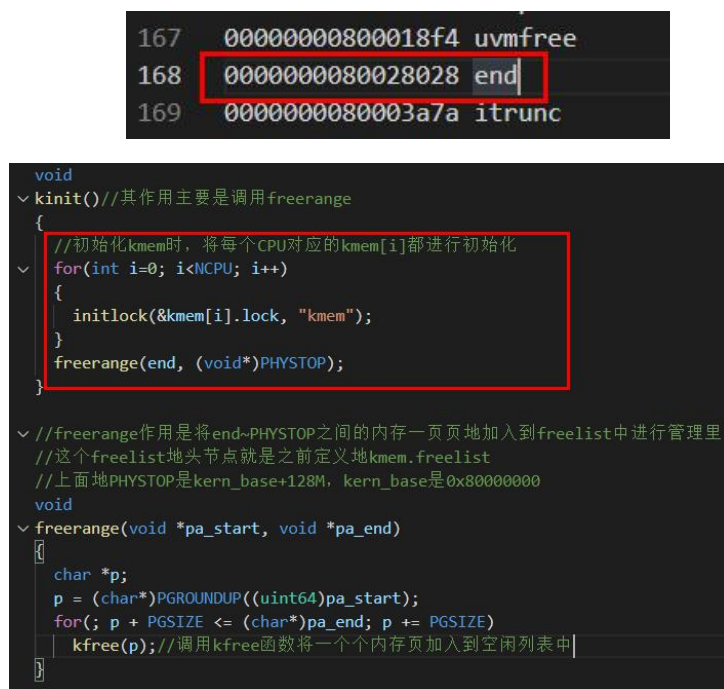


图 2-2 kernel/kalloc.c/kinit

Step4: 修改 kfree 函数

kfree 函数的主要作用是：将参数，即物理内存的地址，加入到空闲列表的开头，如下图 2-3 所示。

Kfree 函数的参数有两种情况：

- 1、在初始化 kinit 的时候，函数的参数是从 end-PHYSTOP 之间的地址，此时的地址都是按页对齐的；
- 2、Kalloc 函数返回，即有时候需要释放一些内存页，就可以直接将内存页的地址传入 kfree 函数中

这里代码的修改重点是要对所属于当前 CPU 的空闲列表进行内存页的插入和取出。所以代码中，先使用 cpuid()函数获得当前 CPU 的编号，用其编号来索引到当前 CPU 的 kmem 信息，即当前 CPU 的空闲列表和锁。然后才去获得属于当前 CPU 的锁，再操作当前 CPU 的空闲列表。

这里为了保证使用 `cpuid()` 函数返回当前 CPU 编号的结果安全，我们要关闭中断，插入空闲页之后，再打开中断。

并且，为了标志这个内存页已经被释放，可作为空闲页使用，将内存页中的内容用 1 填充。

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

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

    // 为了标记这个页被释放（空闲页），将页的内容用1来填充
    memset(pa, 1, PGSIZE);
    r = (struct run*)pa; //让r指向释放的页 pa

    push_off(); //关闭中断避免死锁
    int i = cpuid(); //CPU编号
    acquire(&kmem[i].lock); //获得该CPU对应的锁
    r->next = kmem[i].freelist; //将释放的内存页插到freelist的表头
    kmem[i].freelist = r; //更新freelist为r，（将头指针指向r，释放的这一页）
    release(&kmem[i].lock); //释放锁
    pop_off(); //打开中断
    //acquire(&kmem.lock);
    //r->next = kmem.freelist;
    //kmem.freelist = r;
    //release(&kmem.lock);
}
```

图 2-3 kernel/kalloc.c/kfree

Step5: 修改 kalloc 函数

在 `kalloc` 函数中，我们要实现的主要目标是：优先考虑在当前 CPU 的空闲列表中分配空闲内存页。只有在当前 CPU 的空闲列表指针指向 `NULL`，即没有空闲内存页的时候，我们才考虑去另外 CPU 的空闲列表中“窃取”一个内存页来使用。具体代码如下图 2-4 所示。

该函数分配一个 4096 字节的物理内存页，并返回内核可用使用的指针，该指针指向分配好的内存页。如果当前 CPU 的空闲列表中有空闲内存页（`freelist` 指针不等于 `NULL`），就直接在当前的空闲列表中分配；反之，需要在其他 CPU 的空闲列表中“窃取”一个空闲的内存页。

从空闲列表中取到一个空闲的内存页后，不管是从哪个 CPU 的空闲列表中窃取的，为了标记该内存页已经被分配出去（不是空闲页），把页的内容用 5 填满。然后再返回分配页的指针。

```

void *
kalloc(void)
{
    struct run *r;
    push_off(); // 关闭中断
    int i = cpuid(); // 返回当前的内核号，一定要在中断关闭的时候调用它并使用其结果才是安全的
    acquire(&kmem[i].lock);
    r = kmem[i].freelist; // 当前空闲页的指针
    if(r) // 如果当前CPU的空闲列表中还有空闲页，就取下来使用
    {
        kmem[i].freelist = r->next;
    }
    release(&kmem[i].lock); // 释放锁
    if(!r) // 如果当前cpu对应freelist为空，没有空闲内存页
    {
        for(int j=0; j<NCPU; j++) // 从其他CPU的空闲列表中借用
        {
            if(j!=i) // 从不同的CPU中借用
            {
                acquire(&kmem[j].lock); // 获得选中CPU的锁
                if(kmem[j].freelist) // 如果有空闲页
                {
                    r = kmem[j].freelist; // 借用一个空闲页
                    kmem[j].freelist = r->next; // 指针指向下一个空闲页
                    release(&kmem[j].lock); // 释放该CPU的锁
                    break;
                }
                release(&kmem[j].lock);
            }
        }
    }
    pop_off(); // 打开中断

    if(r) // 成功分配到内存页时
    // 为了标记这个页已经被分配出去，将内存页内容填充为5
    memset((char*)r, 5, PGSIZE);
    return (void*)r; // 返回获得的内存页的指针r
}

```

图 2-4 kernel/kalloc.c/kalloc

Step6: 验证代码

按上述步骤修改好代码之后，启动 qemu，运行 kallocetest 测试文件，结果如下图 2-5 中的红框所示。测试 usertests sbrkmuch，结果如下图的红框显示。测试 usertests 的结果如下图的绿框显示。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ kallocetest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 91108
lock: kmem: #fetch-and-add 0 #acquire() 164312
lock: kmem: #fetch-and-add 0 #acquire() 177602
lock: bcache: #fetch-and-add 0 #acquire() 1248
--- top 5 contended locks:
lock: proc: #fetch-and-add 24983 #acquire() 163870
lock: virtio_disk: #fetch-and-add 7238 #acquire() 114
lock: proc: #fetch-and-add 6501 #acquire() 163856
lock: proc: #fetch-and-add 4454 #acquire() 163913
lock: proc: #fetch-and-add 4357 #acquire() 163912
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6283
sepc=0x000000000000022cc stval=0x000000000000fb90
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

图 2-6 修改后测试结果

(3) 实验总结及分析

对比未修改代码之前的 `kalloctest` 测试结果，如下图 2-6 所示。其中修改代码前后的“top 5 contended locks”中的竞争锁的次数有很大的差别。未修改代码之前，`xv6` 中 8 个 CPU 使用的是同一个空闲列表 `freelist`，竞争锁的次数高达五十多万。修改代码之后，为每一个 CPU 实现一个自己的空闲列表 `freelist` 和锁，此时的竞争锁的次数下降到了十六万左右。并且循环迭代的次数也从 6939 降到了 0，说明为每个 CPU 设计自己的空闲列表可以大大减少出现“竞争锁”的概率。让操作系统的效率更高。

```
hart 1 starting
hart 2 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 6939 #acquire() 433016
lock: bcache: #fetch-and-add 0 #acquire() 1248
--- top 5 contended locks:
lock: proc: #fetch-and-add 63691 #acquire() 506945
lock: proc: #fetch-and-add 38983 #acquire() 507227
lock: proc: #fetch-and-add 28761 #acquire() 507228
lock: proc: #fetch-and-add 12538 #acquire() 506931
lock: proc: #fetch-and-add 12163 #acquire() 506932
tot= 6939
test1 FAIL
start test2
total free number of pages: 32499 (out of 32768)
```

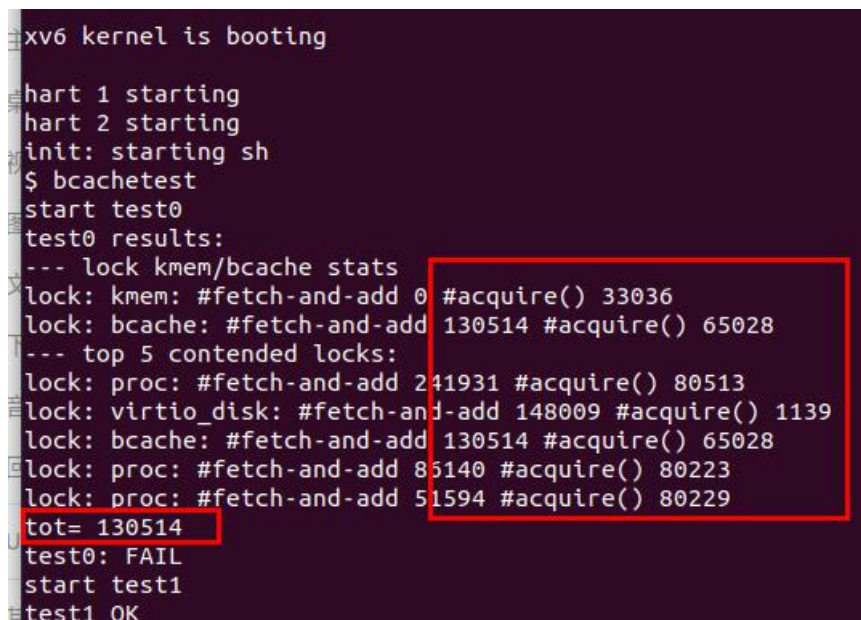
图 2-6 修改前测试结果

3、Xv6 lab: Locks/Buffer cache

本次实验和 2 中一样，主要是重新设计代码以提高并行度的经验。多核计算机上并行性差的常见症状是锁争用较高。通过修改数据结构和锁定策略来改善并行性，以减少争用。

比如，在多个进程密集使用文件系统时，它们可能会争用 `bcache.lock`。因为进程需要占用到 `bcache.lock` 这个锁，才能在磁盘块的缓存区进行读取或写入的操作。在原来的 `bio.c` 文件中所有的 `buffer` 缓冲区都是组织在一条链表中，因此如果有多个进程想要使用 `buffer`，它们并发请求 `buffer`。而这个组织 `buffer` 的共享链表为了维持数据安全，使用锁来保护，保证每次只处理一个进程的请求，这样多个进程的并发请求只能按顺序处理。即每次只有一个进程能马上获取到锁和发配 `buffer`，其余的进程需要等待，获取锁的动作循环迭代，耗费很多的时间。

实验在 `bcachetest` 文件创建了多个进程，这些进程重复读取不同的文件，以便在 `bcache.lock` 上产生争用。未修改代码前的 `bcachetest` 测试结果如下图 3-1 所示。从下图容易看出，在未修改代码之前运行 `bcachetest` 文件时，循环迭代次数最多的 5 个锁的循环迭代次数和总的迭代次数都非常大，进程在等待获得锁的情况下花费了很多时间。



```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33036
lock: bcache: #fetch-and-add 130514 #acquire() 65028
--- top 5 contended locks:
lock: proc: #fetch-and-add 241931 #acquire() 80513
lock: virtio_disk: #fetch-and-add 148009 #acquire() 1139
lock: bcache: #fetch-and-add 130514 #acquire() 65028
lock: proc: #fetch-and-add 85140 #acquire() 80223
lock: proc: #fetch-and-add 51594 #acquire() 80229
tot= 130514
test0: FAIL
start test1
test1 OK
```

图 3-1 未修改前的测试结果

(1) 实验目的

为解决上面的 `bcache` 锁循环迭代次数很高的情况，我们通过修改块缓存，以便在运行 `bcachetest` 的时候，`bcache` 中所有锁的获取循环迭代次数接近 0，并通过 `bcachetest` 和 `usertests`。

于 2 中的 `kalloc` 相比，减少块缓存之间的争用更为棘手，因为 `bcache` 缓冲区确实在进程之间共享，因此也在 CPU 之间共享。而 `kalloc` 中通过为每个 CPU 分配自己的分配器来消除大多数的争用的方法不适用于块缓存，所以我们使用一个哈希表在缓存中查找块号，该哈希表每个哈希存储桶均具有锁定状态。

在某些情况下，如果我们的解决方案存在锁冲突，我们可以采取以下措施：

- 1) 当两个进程同时使用相同的块号时，这种情况在 `bcachetest` 的 `test0` 中不会这样做；
- 2) 当两个进程同时在高速缓存中丢失时，需要找到一个未使用的缓存块来使用，

bcachetest 的 test0 中也不会出现这种情况。

- 3) 当两个进程同时使用你用来对块和锁进行区分的方案冲突的缓存块时。例如，两个进程使用其块号哈希到哈希表中的同一插槽的块，尝试调整修改方案的详细信息避免冲突，比如更改哈希表的大小。

为实现上述目的，我们按照下列实验步骤进行修改。

(2) 实验思路

根据 Lab 给出的一些提示：

- 1) 可以使用固定数量的存储桶，而不动态调整哈希表大小。使用主要数量的存储桶来减少散列冲突的可能性。
- 2) 在哈希表中搜索缓冲区，并在找不到缓冲区的时候为缓冲区分配一个条目。这个操作必须是原子操作。
- 3) 删除所有缓冲区的列表 (bcache.head 等)，并使用上次使用的时间，即 kernel/trap.c 中的滴答声来标记时间戳。通过此更改，brelse 不需要获得 bcache 的锁，并且 bget 可以根据时间戳选择最近最少使用的块。
- 4) 可以在 bget 函数中对逐出部分进行序列化，即当在缓存中未找到查询时，bget 的一部分会选择要重用的缓存区。
- 5) 在某些情况下，我们可能需要按住两个锁。例如，在驱逐期间，我们需要按住 bcache 锁和每个存储桶的锁，确保避免死锁。
- 6) 更换块的时候，我们可以将 struct buf 从一个存储桶移动到另一个存储桶，因为新的块可能回和散列到与旧块相同的存储桶。我们要确保在这种情况下避免死锁

根据提示，在本次实验有两个实现思路，一个没有使用时间戳，通过缓存块在不同存储桶之间的转移来实现实验目的。

另一个尝试使用时间戳来分配缓存块，实现实验目的。

下面就分别来实现两种思路。

(3) 思路一实验（不使用时间戳）步骤及测试结果

Step1: 修改 bcache 结构

将原来的链表结构修改为哈希表，如下图 3-2 所示：

```
#define NBUCKETS 13

struct {
    //将原来的链表结构修改为哈希表的结构
    struct spinlock lock[NBUCKETS]; //哈希表的每个bucket有一个锁
    struct buf buf[NBUF]; //NBUF=30, 30个缓冲区
    struct buf hashbucket[NBUCKETS]; //哈希表的bucket也用buf结构来表示
} bcache;
```

图 3-2 bcache 结构（无时间戳）

根据哈希表的结构，我们选择缓存块的块号 `blockno` 作为哈希表的 `key` 值，即将块号余数相同的缓存块放入同一个 `bucket` 中，如下图 3-3 所示：

```
int idx(int blockno){  
    //根据哈希表的结构特点，使用缓存块的块号作为哈希表的key值  
    //通过对块号取余，得到缓存块对应的bucket  
    //即余数相同的缓存块放在同一个bucket中  
    return blockno%NBUCKETS;  
}
```

图 3-3

哈希表的结构如下图 3-4 所示，（图来源于网络，以 `NBUCKETS=3` 为例）

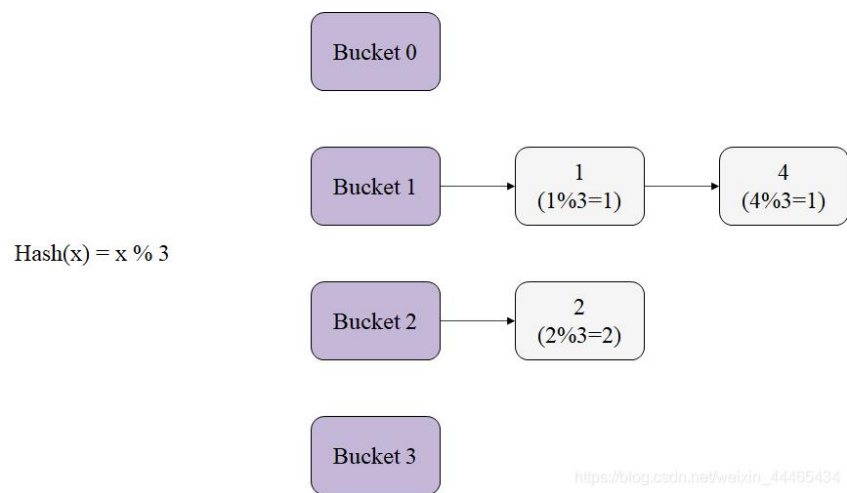


图 3-4 哈希表结构

Step2: 修改 binit 函数

初始化修改的哈希表，如下图 3-5 所示

```
void
binit(void)
{
    struct buf *b;
    //初始化、遍历哈希表
    for(int i=0;i<NBUCKETS;i++){
        initlock(&bcache.lock[i], "bcache.bucket");
        b=&bcache.hashbucket[i];
        //将每个bucket的头结点指向自己
        b->prev = b;
        b->next = b;
    }

    //此时的缓存块还没有和磁盘块对应起来，
    //所以缓存块的blockno先初始化为0，即都在哈希表中0 bucket上
    //下面便使用头插法，将缓存块先全部插到hashbucket的头部分
    for(b = bcache.buf; b < bcache.buf+NBUF; b++){
        //插到hashbucket头部
        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 3-5 binit 函数（无时间戳）

Step3: 修改 bget 函数

bget 函数主要实现在哈希表中获取合适的缓冲块并。修改后我们要实现的是，首先在 blockno 对应的 bucket 中寻找合适的缓冲块。如果在词 bucket 下没有找到合适的缓冲块，我们就需要去别的 bucket 中“窃取”一个缓冲块使用，并插到 blockno 对应的 bucket 中。

```
75 static struct buf*
76 bget(uint dev, uint blockno)
77 {
78     struct buf *b;
79     int i=idx(blockno);
80     acquire(&bcache.lock[i]);
81
82     //首先在blockno对应的bucket中查找缓存块
83     for(b = bcache.hashbucket[i].next; b != &bcache.hashbucket[i]; b = b->next){
84         //如果找到合适缓存块，将其锁定返回
85         if(b->dev == dev && b->blockno == blockno){
86             b->refcnt++; //refcnt=1表示buffer处于忙碌状态，进锁定要使用它
87             release(&bcache.lock[i]); //释放bucket的锁
88             acquiresleep(&b->lock); //获得缓存块的锁
89             return b; //返回锁定的缓存块
90         }
91     }
92 }
```

```

92
93 //下面处理的情况是，当我们无法在当前对应的存储中找到合适的缓存块
94 //参考上一个Lab的做法，我们去其他的bucket中“窃取”一个缓存块
95 int ni=(i+1)%NBUCKETS;
96 //遍历其他的bucket
97 while(ni!=i){
98     acquire(&bcache.lock[ni]); //获得其他bucket的锁
99     //遍历该bucket上的所有缓存块
100     for(b = bcache.hashbucket[ni].prev; b != &bcache.hashbucket[ni]; b = b->prev){
101         if(b->refcnt == 0) {
102             b->dev = dev;
103             b->blockno = blockno;
104             b->valid = 0;
105             b->refcnt = 1;
106             //在该bucket下找到合适的缓存块时，将其从原来的bucket中取下
107             b->next->prev=b->prev;
108             b->prev->next=b->next;
109             release(&bcache.lock[ni]);
110             //然后使用头插法，将其插入到blockno对应的bucket中
111             b->next=bcache.hashbucket[i].next;
112             b->prev=&bcache.hashbucket[i];
113             bcache.hashbucket[i].next->prev=b;
114             bcache.hashbucket[i].next=b;
115             release(&bcache.lock[i]);
116             acquiresleep(&b->lock);
117             return b; //返回锁定的缓存块
118         }
119     }
120     release(&bcache.lock[ni]);
121     ni=(ni+1)%NBUCKETS;
122 }
123 //如果没有找到合适的缓存块，出现panic
124 panic("bget: no buffers");
125 }

```

图 3-6 bget 函数（无时间戳）

Step4: 实验测试结果

修改好代码之后，我们运行 bcachetest 进行测试，结果如下图组 3-7 所示。从图(a)可以看出修改之后，竞争 bcache 锁的情况得到很大的改善，最后的循环迭代总次数也下降到了 0。


```

hart 2 starting
hart 1 starting
init: starting sh
$ bachetest
exec bachetest failed
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33087
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4142
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2112
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4276
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4327
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6333
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6323
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6607
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6622
lock: bcache.bucket: #fetch-and-add 0 #acquire() 7075
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6199
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4142
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4143
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2134
--- top 5 contended locks:
lock: proc: #fetch-and-add 210188 #acquire() 75146
lock: proc: #fetch-and-add 143697 #acquire() 74812
lock: virtio disk: #fetch-and-add 75237 #acquire() 1154
lock: proc: #fetch-and-add 62897 #acquire() 74847
lock: proc: #fetch-and-add 46132 #acquire() 74786
tot= 0
test0: OK
start test1
test1 OK

```

(a)

```

$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK

```

(b)

```

OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitiput: OK
test input: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED

```

(c)

图组 3-7 bcache 实验结果（没有使用时间戳）

(4) 思路二实验（使用时间戳）步骤及测试结果

根据思路一可实现实验目的，但在实验的 hints 中提出可以使用时间戳来实现，下面便尝试使用第二种思路完成实验。具体实验步骤如下所示：

Step1: 修改 bcache 结构

修改 bcache 为一个结构体数组，数组中每个元素对应哈希表的每个 bucket，每个 bucket 都有一个锁 lock 和一个 buf 缓冲块数组，如下图 3-8 所示。

这里使用的哈希表的 bucket 数量会小一点，因为锁的数目有限，而 binit 至少需要 $NBUF \times NBUCKET + NBUCKET$ 个锁，所以这里选择的 bucket 数量为 7。

```
#define NBUCKET 7
struct {
    //修改bcache的结构，将其从链表结构修改为一个哈希表结构
    //bcache为一个数值，可认为是一个含有7个存储桶的哈希表
    //哈希表的每个存储桶都有一个锁 lock
    //并且每个存储桶含有一个可存放NBUF个缓存块的缓存块数组
    struct spinlock lock;
    struct buf buf[NBUF]; //NBUF=30
} bcache[NBUCKET];
```

图 3-8 bcache 结构（使用时间戳）

Step2: 修改 buf.h 文件

为了使用时间戳，我们需要在 struct buf 里面添加 time_stamp 时间戳，用于记录每个缓存块 buf 的上次使用时间，如下图 3-9 所示。

```
//xv6中磁盘块数据结构，块大小为512字节
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];
    uint time_stamp; //上次使用的时间，时间戳
};
```

图 3-9 buf 结构体修改

Step3: 修改 binit 初始化函数

```
void
binit(void)
{
    //初始化哈希表
    struct buf *b;
    //NBUCKET=7,遍历bcache哈希表的7个存储桶
    for (int i=0; i<NBUCKET;i++)
    {
        //遍历该存储桶的缓存块（30个），初始化每个缓存块的锁lock，
        for(b=bcache[i].buf; b<bcache[i].buf+NBUF; b++){
            initsleeplock(&b->lock, "buffer");//将缓存块的锁打开（置0）
        }
        //初始化每个存储桶的锁，置为0，表示未锁住
        initlock(&bcache[i].lock, "bcache.bucket");
    }
}
```

图 3-10 修改 binit 函数（使用时间戳）

Step4: 修改 bget 函数

使用 bget 函数请求一个合适的缓存块会遇到两种情况：命中、未命中。

命中是指能在 blockno 对应的 bucket 中找到合适的缓存块，可以直接锁定返回。未命中则是找到最小时间戳对应的 buf 缓存块，即最近使用最少的缓存块返回，并将缓存块加入到该 blockno 对应的 bucket 中，如图 3-11 所示。

```
61 //通过缓存区缓存查找设备dev上的块，命中。
62 //如果找不到，则分配一个缓冲区。
63 //在这两种情况下，返回锁定的缓冲区。
64 static struct buf*
65 bget(uint dev, uint blockno)
66 {
67     struct buf *b=0; //缓存块buffer的指针
68     int i=idx(blockno); //缓存块的号对应的哈希表的key值
69     uint min_time_stamp=-1; //最小时间戳
70     struct buf *min_b=0;
71
72     //获取到哈希表中该缓存块对应存储桶的锁，避免竞争
73     acquire(&bcache[i].lock);
74     //遍历该对应存储桶的缓存块
75     for(b=bcache[i].buf; b<bcache[i].buf+NBUF; b++){
76         if(b->dev==dev && b->blockno==blockno)
77         {
78             b->refcnt++;
79             release(&bcache[i].lock);
80             acquiresleep(&b->lock); //获得选中缓存块的锁
81             return b; //返回锁定的缓存块buffer
82         }
83     }
84     //找到最小的时间戳对应的buf缓存块
85     //根据时间戳，选择最近最少使用的块
86     if(b->refcnt==0 && b->time_stamp<min_time_stamp)
87     {
88         min_time_stamp = b->time_stamp;
89         min_b = b;
90     }
91 }
```

```

91
92     b=min_b;
93     //如果在高速缓存中未找到查询,
94     //选择要重新使用的缓冲区部分,调入缓冲区
95     if(b!=0)
96     {
97         b->dev = dev;
98         b->blockno = blockno;
99         b->valid = 0;
100        b->refcnt = 1;
101        release(&bcache[i].lock);
102        acquiresleep(&b->lock);
103        return b; //返回锁定的缓冲区
104    }
105
106    panic("bget: no buffers");
107 }

```

图 3-11 修改 bget 函数（有时间戳）

Step5: 修改 brelse 函数

修改 brelse 函数, 在每个缓冲块 buf 释放之后, 我们要求更新它的时间戳, 即当前的 tick 时间作为它时间戳, 表示这个缓存块刚使用不久, 具体代码如下图 3-12 表示。

```

// Release a locked buffer.
// Move to the head of the most-recently-used list.
void
brelse(struct buf *b)
{
    release(&bcache.lock); /*
    if(!holdingsleep(&b->lock))
        panic("brelse");

    releasesleep(&b->lock);
    int i=idx(b->blockno);
    acquire(&bcache[i].lock);
    b->refcnt--;

    if(b->refcnt == 0){
        //no one is waiting for it
        //更新时间戳
        b->time_stamp = ticks;
    }
    release(&bcache[i].lock);
}

```

图 3-12 brelse 函数（有时间戳）

Step6: 测试结果

使用时间戳来处理哈希表的 `bcachetest` 实验结果如下图组 3-13 所示。实验结果也表明循环迭代获取锁的总次数大大降低。

```
$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32987
lock: bcache.bucket: #fetch-and-add 0 #acquire() 10302
lock: bcache.bucket: #fetch-and-add 47 #acquire() 10298
lock: bcache.bucket: #fetch-and-add 8 #acquire() 8434
lock: bcache.bucket: #fetch-and-add 22 #acquire() 8696
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8476
lock: bcache.bucket: #fetch-and-add 0 #acquire() 9588
lock: bcache.bucket: #fetch-and-add 0 #acquire() 8298
--- top 5 contended locks:
lock: proc: #fetch-and-add 126206 #acquire() 70741
lock: virtio_disk: #fetch-and-add 93151 #acquire() 1014
lock: proc: #fetch-and-add 61074 #acquire() 70406
lock: proc: #fetch-and-add 58680 #acquire() 70427
lock: proc: #fetch-and-add 55409 #acquire() 70428
tot= 77
test0: OK
start test1
test1 OK
```

(a)

```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
```

(b)

```
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

(c)

图组 3-13 `bcachetest` 测试结果

(5) 实验总结及分析

完成实验的两种思路都是通过修改 **bcache** 结构，改善多个进程并发争取 **bcache** 锁时出现的循环迭代现象。总的来说都是将原来的 **bcache** 的链表结构转变成哈希表的结构，让进程可以同时去哈希表的不同 **bucket** 中获取合适的缓存块，减少竞争现象。实验显示改善效果非常显著。

4、Xv6 lab: File System/Large files

在本次实验中我们将向 xv6 文件系统添加大文件，即增加 xv6 文件的最大大小。

Xv6 的文件系统系统结构如下图 4-1 所示，其中最重要的就是 inode 部分。Inode 是代表一个文件的对象，并且它并不依赖于文件名。实际上，inode 是通过自身的序列号来区分的，这里的序列号就是一个帧数。所以文件系统内部通过一个数字，而不是通过文件路径引用 inode。

同时，基于 inode 必须有一个 link count 来跟踪指向这个 inode 的文件名的数量。一个文件（inode）只能在 lonk count=0 的时候才能被删除。总的来说文件系统的核心数据就是 inode 和文件描述符（file descriptor）。

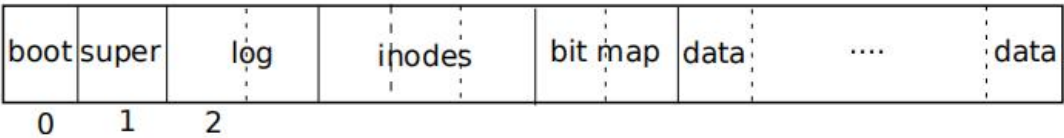


图 4-1 文件系统结构（图源 book-riscv）

在本次实验中主要使用到的是 inode，内核在内存中维护的 inode 是磁盘中的结构体 dinode 在内存中的拷贝，内存只会在有指针指向一个 i 节点的时候才会把这个 i 节点保存到内存中。磁盘上的 dinode 结构记录了 i 节点的大小和数据块的块号数组，其结构如下图 4-2 所示。

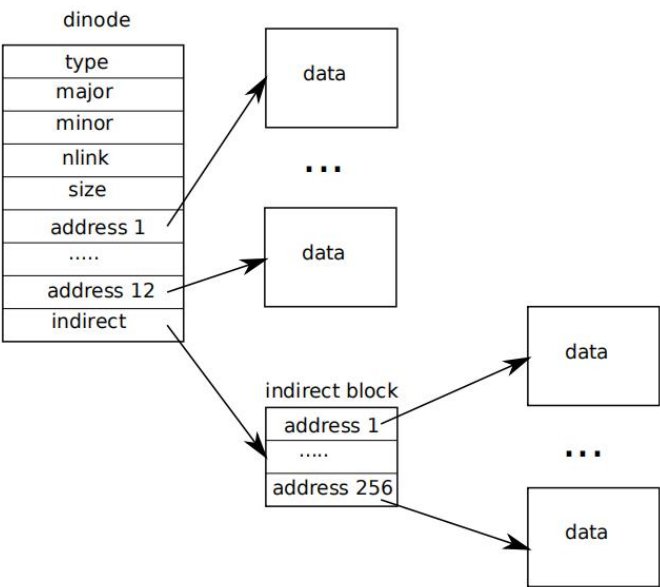


图 4-2 dinode 结构

从上图可以看出，这是一个 64 字节的数据结构。通常来说，它有一个 type 字段，用来表明 inode 是文件还是目录。nlink 字段就是 nlink 计数器，用来跟踪究竟有多少文件名指向当前的 inode。Size 字段用来表明数据有多少个字节。

dinode 结构在不同文件系统中表达方式可能不一样，在 xv6 的文件系统中，接下来的是一些 block 的编号，例如编号 0、编号 1 等。Xv6 的 inode 中总共有 12 个 block 编号，这些被称为 direct block number（直接块）。这 12 个 block 编号指向了构成文件的前 12 个 block。在这 12 个 block 之后还有一个 indirect block number，它对应了磁盘上的一个 block，这个 block 包含了 256 个 block number。所以 inode 中的 block number 从 0-11 都是 direct block number，而 12 开始就保存在一个 indirect block number 指向另一个 block。

所以，当前 xv6 文件限制为 268 个块，即 xv6 中文件的最大长度是 256×1024 字节。本次实验要求我们能够创建 65803 个块的文件，即文件的最大长度要扩展到 65803×1024 字节。所以根据上面对 inode 的分析，我们可以对 13 个 block number 进行修改。

(1) 实验目的

实验目的是支持更大的文件，我们可以通过将直接块 direct block number 中取一项支持“三级表”，这样，xv6 单个文件的大小最大可扩展到 $11 + 256 + 256 \times 256$ 块。即修改之后，block 编号前 11 个 block 为直接块，第 12 个作为一级间接块，最后的第 13 个 block 实现二级间接块。

(2) 实验步骤

Step1: 修改 inode 块编号在 fs.h 文件中的定义，如下图 4-3 所示

```
//定义直接块只有11个
#define NDIRECT 11
//一级间接块256
#define NINDIRECT_1 (BSIZE / sizeof(uint))
//二级间接块256*256
#define NINDIRECT_2 (NINDIRECT_1*NINDIRECT_1)
#define NINDIRECT (NINDIRECT_1 + NINDIRECT_2)
//文件最大的长度|
#define MAXFILE (NDIRECT + NINDIRECT)
```

图 4-3 编号修改

Step2: 修改 inode 和 dinode 结构体

因为 step1 中将直接块的大小 NDIRECT 改为了 11，所以这里 dinode 和 inode 的 addr 数组大小需要加上一。

```
// 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 addr[NDIRECT+1]; // Data block addresses
};
```

图 4-4 dinode 结构修改

```

// 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+1+1];
};

```

图 4-5 inode 结构修改

Step3: 修改 bmap 函数

在读写文件时会调用 bmap 函数，例如在写入时，bmap 函数会根据需要分配的新的块来保存文件内容，并在需要时分配一个间接块来保存块地址。Bmap 函数处理两种块号，bn 参数时“逻辑块号”，即文件中相对于文件开头的块号；ip->addrs[]中的块号以及 bread 函数的参数是磁盘块号。所以，可以将 bmap 函数视为将文件的逻辑块号映射到磁盘块号上。

修改的代码如下图 4-6 所示。

```

377 static uint
378 bmap(struct inode *ip, uint bn)
379 {
380     uint addr, *a;
381     struct buf *bp;
382
383     //当bn<11是，可以在直接块上得到磁盘上的地址，直接块
384     if(bn < NDIRECT){
385         if((addr = ip->addrs[bn]) == 0)
386             ip->addrs[bn] = addr = balloc(ip->dev);
387         return addr; //返回磁盘上的地址
388     }
389     //如果bn>NDIRECT，将bn减去NDIRECT得到在11之后的偏移量
390     bn -= NDIRECT;
391     //如果bn在一级间接的范围[11, 11+256]内
392     if(bn < NINDIRECT_1){
393         // 加载第一间接块的内容，
394         //如果addr=0，表示还没有分配，则只用balloc为其分配缓存块
395         if((addr = ip->addrs[NDIRECT]) == 0)
396             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
397         bp = bread(ip->dev, addr);
398         //读取到一级间接的内容，
399         //其内容是第12个undirect block指向的一个block的表头
400         a = (uint*)bp->data;
401         //对指向的block进行索引，找到对应的addr，
402         //如果addr=0，为其分配一个缓存块
403         if((addr = a[bn]) == 0){
404             a[bn] = addr = balloc(ip->dev);
405             log_write(bp); //并将其写入log日志文件中
406         }
407         brelse(bp);
408         return addr; //返回磁盘地址
409     }

```

图 4-6(a) bmap 函数

```

410
411 ✓ //如果bn的值还是超过了一级间接的范围，则进入二级间接的范围中寻找
412 //同理将bn减去一级间接的大小，得到在二级间接中的偏移量
413 bn -= NINDIRECT_1;
414 //当bn在二级间接可寻址的范围内时
415 ✓ if(bn < NINDIRECT_2){
416     // 加载二级间接block，如果addr=0，为其分配缓存块
417     if((addr = ip->addrs[NINDIRECT+1]) == 0)
418         ip->addrs[NINDIRECT+1] = addr = balloc(ip->dev);
419
420     //读取作为二级间接的第13个inode内容，其内容是第一级block的表头地址
421     bp = bread(ip->dev, addr);
422     a = (uint*)bp->data;
423     uint bn_1= ( bn & 0xff00)>>8 ;//相当于将bn除以256，得到在一级表中的偏移
424 ✓   uint bn_2= bn & 0xff;//相当于bn%256，得到在二级表中的偏移
425     //如果索引到的地址为0，使用balloc为其分配缓存
426 ✓   if((addr = a[bn_1]) == 0){
427       a[bn_1] = addr = balloc(ip->dev);
428       log_write(bp);//记录到日志文件中
429   }
430   brelse(bp);
431   //读取在一级表中定位的内容，其内容指向的是二级表的表头
432   bp = bread(ip->dev, addr);
433   a = (uint*)bp->data;
434   //然后根据在二级表中的偏移量，利用二级表头索引
435 ✓   if((addr = a[bn_2]) == 0){
436       //如果索引到的地址为0，说明需要为它分配缓存块
437       a[bn_2] = addr = balloc(ip->dev);
438       log_write(bp);//写入日志
439   }
440   brelse(bp);
441   return addr;//返回磁盘地址
442 }
443 panic("bmap: out of range");
444 }
445

```

图 4-6(b) bmap 函数

Step4: itrunc 函数

itrunc 是用来释放文件中申请的缓存，如果对应的是间接块，我们需要通过循环来保证释放所有文件的块。具体代码实现如下图 4-7 所示。

```
446 // Truncate inode (discard contents).
447 // Caller must hold ip->lock.
448 void
449 itrunc(struct inode *ip)
450 {
451     int i, j, k;
452     struct buf *bp;
453     struct buf *bp2;
454     uint *a;
455     uint *a2;
456     //循环将0-10号的11个直接block释放
457     for(i = 0; i < NDIRECT; i++){
458         if(ip->addrs[i]){
459             bfree(ip->dev, ip->addrs[i]);
460             ip->addrs[i] = 0;
461         }
462     }
463
464     //如果第12个指向一个block不为0（分配了磁盘缓存）
465     //，循环释放一级间接的内容
466     if(ip->addrs[NDIRECT]){
467         //读取inode的内容，其内容是一级表的表头
468         bp = bread(ip->dev, ip->addrs[NDIRECT]);
469         a = (uint*)bp->data;
470         //得到一级表的表头，循环遍历释放表中的内容
471         for(j = 0; j < NINDIRECT_1; j++){
472             if(a[j])
473                 bfree(ip->dev, a[j]);
474         }
475         brelse(bp);
476         bfree(ip->dev, ip->addrs[NDIRECT]); //最后释放第12个undirect block
477         ip->addrs[NDIRECT] = 0;
478     }
479 }
```

图 4-7(a) itrunc 函数


```

479
480 //如果二级间接中也有内容（分配了磁盘缓存）
481 if(ip->addrs[NINDIRECT+1]){
482     //先读取第13个inode的内容，获得指向第一级表头的地址
483     bp = bread(ip->dev, ip->addrs[NINDIRECT+1]);
484     a = (uint*)bp->data;
485     //循环遍历第一级block
486     for(j = 0; j < NINDIRECT_1; j++){
487         //第一级表中的条目如果分配了缓存
488         if(a[j])
489         { //得到该条目的内容，其内容是该条目对应的第二级表头地址
490             bp2 = bread(ip->dev, a[j]);
491             a2 = (uint*)bp2->data;
492             //遍历第二级间接表的内容
493             for(k=0 ;k < NINDIRECT_1 ;k++)
494             { //如果第二级简介表中分配了缓存，将其释放
495                 if(a2[k]) bfree(ip->dev,a2[k]);
496             }
497             brelse(bp2);
498             bfree(ip->dev, a[j]); //释放一级表的条目
499         }
500     }
501     brelse(bp);
502     bfree(ip->dev, ip->addrs[NINDIRECT+1]); //释放第13个undirect block的缓存
503     ip->addrs[NINDIRECT+1] = 0;
504 }
505 ip->size = 0;
506 iupdate(ip);
507 }

```

图 4-7(b) itrunc 函数

(3) 实验总结及分析

修改代码之前 xv6 文件大小限制在 268 块（268*1024 字节）内，我们就无法生成一些大文件，如下图 4-8 所示。

修改之后，我们将 xv6 文件大小扩大到 65803 块。通过修改 dinode 结构中的 direct block 和 undirect block，增加间接块的层数，牺牲一个直接块，从而使文件最大长度增加，实现生成大文件的目的。如图 4-9 和 4-10 展示了实验测试结果。

```

xv6 kernel is booting

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

```

图 4-8 未修改前测试结果


```
{ xv6 kernel is booting  
init: starting sh  
$ bigfile  
. . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// . . . . .  
// wrote 65803 blocks  
$ bigfile done; ok
```

图 4-9 修改后 bigfile 测试结果

```
$ userstests
userstests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
```

```
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

图 4-10 修改后的 usertests 测试结果