

一、基于事件驱动的 Client-Server 问题

1.1 任务要求

1.1.1 Client-Server

任务首先要求实现 Client-Server 模型，创建客户 Client 和服务器 Server 两个进程，Client 进程派生 3 个生产者线程，Server 进程派生 3 个消费者线程。所有进程和线程间通信均通过管道实现。

1.1.2 事件驱动

任务要求使用 epoll 监听管道的收发，将消息在合适的管道中读取或发送。关于 epoll 的原理和使用方法不在这里赘述。

1.1.3 非阻塞

任务要求所有的管道收发都要严格执行非阻塞操作，这就要求建立合适大小的缓冲区，将收发的数据手动对齐，减小管道之间收发速度不匹配造成数据发送截断的情况。

1.2 设计思路

下面按照任务要求分为主要的三步，逐步实现。

1.2.1 Client-Server

Client-Server 模型主要基于作业一中的模型改编而来。主要改动的地方有三个。第一个是将原来的 FIFO 管道线程的执行任务，放置到主函数中，作为主线程的任务。第二个是将原来的生产者/消费者线程与管道线程之间的共享内存改为匿名管道收发的模式。第三个是将原来发送的 10 字节大小的字符串消息，修改为 8 字节大小的线程 src_id 和 4096 字节大小的由 0 填充的字符串，两种类型数据组成的结构体消息，总计 4104 字节大小，这样更容易产生管道堵塞的问题，由此使用 epoll、非阻塞管道和缓冲池来优化。


1.1.2 事件驱动

完成 Client-Server 的管道收发模型改造后，使用 epoll 监听管道的收发。每个进程中，epoll 需要监听 4 个事件，在 Client 进程中，epoll 在主线程的位置监听 3 个生产者线程和主线程通信的 3 条匿名管道的读取端，以及与 Server 进程通信的 FIFO 管道的发送端。在 Server 进程中，epoll 在主线程的位置监听 3 个消费者线程和主线程通信的 3 条匿名管道的发送端，以及与 Client 进程通信的 FIFO 管道的读取端。综上，两个进程中的 epoll 都需要监听 4 个管道事件，同时都包含管道文件的发送和读取。

epoll 在死循环中不断监听可以唤醒的事件，挑选出合适的管道，从而触发管道的读写。

1.1.3 非阻塞

由于管道的缓存大小是 64kb，在阻塞模式下最多能同时容纳 15 条消息。而在非阻塞模式下，最多能同时容纳 15 条消息和若干被截断的消息。因此，如果所有管道开启非阻塞模式时，一旦管道收发两端的速率不一致，一段时间之后就很容易产生堵塞，如下图所示。

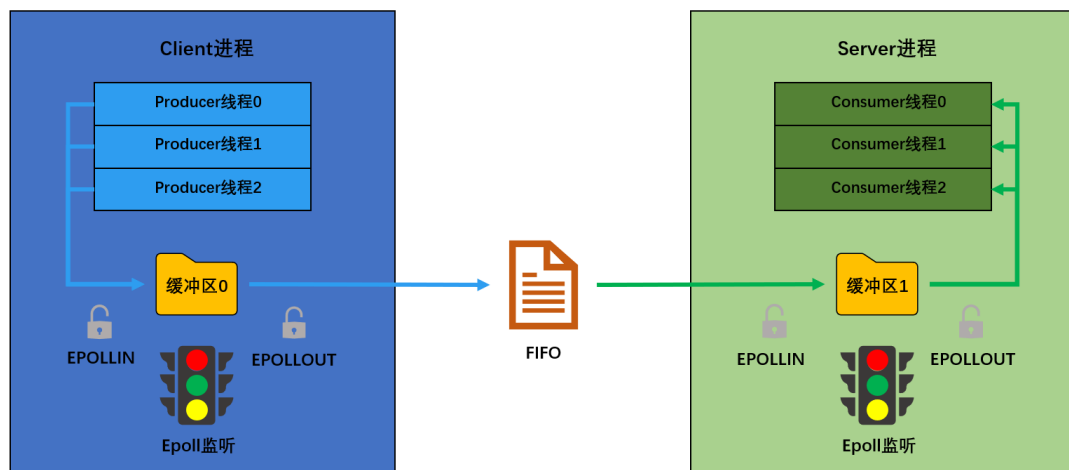


```
os@ubuntu:~/os-assignment3$ ./server 1
read_buf:14064578 4014
read_buf: 4014
read_buf: 4014
read_buf: 4014
read_buf:14064578 4014
read_buf: 4014
read_buf:14064579 4014
read_buf: 4014
read_buf:14064579 4014
read_buf: 4014
read_buf:14064578 4014
read_buf: 4014
read_buf:14064579 4014
read_buf: 4014
read_buf:14064578 4014
read_buf: 4014
read_buf:14064578 4014
read_buf: 4014
^C
```

并且容易产生消息截断，经过实践，消息常常会被分割为 8 字节大小和 4096 字节大小的两部分消息，猜测这是由于结构体导致的。

基于上述的发现，可以设计简易的缓存池，以数组队列的实现为例，当接收到 4104 字节大小的消息时，将该消息正常放入队列中；当接收到 8 字节大小消息时，使用由 0 组成的 4096 字节大小的字符串，重新构成一个正常消息，放入队列中；当接收到 4096 字节大小消息时，直接丢弃消息，不放入队列中。

总体模型架构如下图所示。



1.3 代码实现

代码实现部分将分为 Client 进程 和 Server 进程，其中再分为主线程任务和其他线程任务。

1.3.1 Client 生产者线程

生产者线程主要执行的任务时，间隔一定时间，生产一个包含线程自身 `src_id` 的消息，通过与主线程单独通信的匿名管道，发送给主线程。每个生产者线程有自己一条匿名管道，因此生产者线程中不需要使用 `epoll` 来监听管道的收发，只需要将管道的收发端同时设置为非阻塞的模式即可。

1.3.2 Client 主线程

主线程中除了一系列的变量定义、线程创建以外，主要执行的任务是，使用 `epoll` 监听与 3 个生产者线程通信的 3 条匿名管道的读取，从合适的管道中读取的消息，放入缓冲区中对齐；再使用 `epoll` 监听与 Server 进程通信的 FIFO 管道，在合适的时机，从缓冲区中取出对齐的消息，写入 FIFO 管道中。Client 主线程部分代码如下图所示。

1.3.3 Server 消费者线程

消费者线程主要执行的任务是，间隔一定时间，从与主线程单独通信的匿名管道中，读取消息，并在终端打印出来。消费者线程中同样只有一个单独的管道，不需要使用 `epoll` 来监听收发，但需要注意的是，由于管道的收发端被设置为了

非阻塞模式，消费者线程有可能读空，返回一个-1 字节大小的消息，需要手动过滤。

1.3.4 Server 主线程

server 主线程主要执行的任务是，使用 epoll 监听 FIFO 管道，从中读取消息，将消息放入缓冲区中对齐；再使用 epoll 监听 3 条匿名管道的写端，从缓冲区中取出消息放入合适的管道中，发送给对应的消费者。

1.3.5 非阻塞模式

根据任务要求，需要对所有管道的收发段 设置非阻塞模式，使用 fcntl 设置非阻塞模式的函数如下图所示。

```
67 //利用fcntl设置成非阻塞
68 int setFdNonblocking(int fd){
69     int flags;
70     flags = fcntl(fd, F_GETFL, 0);
71     if (flags == -1){
72         perror("fcntl");
73         return -1;
74     }
75     flags |= O_NONBLOCK;
76     if (fcntl(fd, F_SETFL, flags) == -1){
77         perror("fcntl");
78         return -1;
79     }
80     return 0;
81 }
```

1.3.6 epoll 初始化

本实验监听管道添加的事件触发标志是 EPOLLIN 和 EPOLLOUT，还有默认的水平触发（ET）模式，关于不设置 LT 模式的原因在之后会讲述。

1.4 测试结果

1.4.1 Client=1, Server=1

Server 速度快过 Client 时，匿名管道唤醒的次数更多，消费者线程运行的频率增加了，基本上可以和生产者同步。

1.4.3 Client=1, Server=0.5

```
os@ubuntu:~/os-assignment3$ ./server 0.5
listen fifo...FIFO get 4104 bytes 140425392137984
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425375352576
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425375352576
fd = 5, send 4104 bytes
consumer 0 read 4104 bytes 140425392137984
FIFO get 4104 bytes 140425392137984
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425383745280
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425383745280
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425392137984
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425383745280
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425375352576
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425392137984
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425383745280
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425392137984
fd = 5, send 4104 bytes
consumer 0 read 4104 bytes 140425375352576
FIFO get 4104 bytes 140425375352576
fd = 5, send 4104 bytes
FIFO get 4104 bytes 140425375352576
fd = 5, send 4096 bytes
FIFO get 4104 bytes 140425375352576
fd = 7, send 4104 bytes
FIFO get 4104 bytes 140425392137984
fd = 7, send 4104 bytes
FIFO get 4104 bytes 140425375352576
fd = 7, send 4104 bytes
FIFO get 4104 bytes 140425383745280
fd = 7, send 4104 bytes
FIFO get 4104 bytes 140425392137984
fd = 7, send 4104 bytes
consumer 0 read 4104 bytes 140425375352576
FIFO get 4104 bytes 140425383745280
fd = 7, send 4104 bytes
consumer 1 read 4104 bytes 140425375352576
consumer 0 read 4104 bytes 140425392137984
FIFO get 4104 bytes 140425375352576
fd = 7, send 4104 bytes
FIFO get 4104 bytes 140425383745280
fd = 7, send 4104 bytes
FIFO get 4104 bytes 140425383745280
FIFO get 4104 bytes 140425383745280
os@ubuntu:~/os-assignment3$ ./client 1
producer 0 write 4104 bytes 140425392137984
fd = 4, recv 4104 bytes 140425392137984
FIFO, send 4104 bytes 140425392137984
producer 2 write 4104 bytes 140425375352576
fd = 8, recv 4104 bytes 140425375352576
FIFO, send 4104 bytes 140425375352576
producer 2 write 4104 bytes 140425375352576
fd = 8, recv 4104 bytes 140425375352576
FIFO, send 4104 bytes 140425375352576
producer 0 write 4104 bytes 140425392137984
fd = 4, recv 4104 bytes 140425392137984
FIFO, send 4104 bytes 140425392137984
producer 1 write 4104 bytes 140425383745280
fd = 6, recv 4104 bytes 140425383745280
FIFO, send 4104 bytes 140425383745280
producer 0 write 4104 bytes 140425392137984
fd = 4, recv 4104 bytes 140425392137984
FIFO, send 4104 bytes 140425392137984
producer 2 write 4104 bytes 140425375352576
fd = 8, recv 4104 bytes 140425375352576
FIFO, send 4104 bytes 140425375352576
producer 0 write 4104 bytes 140425392137984
fd = 4, recv 4104 bytes 140425392137984
FIFO, send 4104 bytes 140425392137984
producer 1 write 4104 bytes 140425383745280
fd = 6, recv 4104 bytes 140425383745280
FIFO, send 4104 bytes 140425383745280
producer 0 write 4104 bytes 140425392137984
fd = 4, recv 4104 bytes 140425392137984
FIFO, send 4104 bytes 140425392137984
producer 2 write 4104 bytes 140425375352576
fd = 8, recv 4104 bytes 140425375352576
FIFO, send 4104 bytes 140425375352576
producer 2 write 4104 bytes 140425375352576
fd = 8, recv 4104 bytes 140425375352576
FIFO, send 4104 bytes 140425375352576
producer 0 write 4104 bytes 140425392137984
fd = 4, recv 4104 bytes 140425392137984
FIFO, send 4104 bytes 140425392137984
producer 2 write 4104 bytes 140425375352576
fd = 8, recv 4104 bytes 140425375352576
FIFO, send 4104 bytes 140425375352576
```

在 Server 速度低于 Client 时，消费者线程的运行频率已经远远跟不上生产者的运行频率了，这时候缓冲区开始起作用，消费者线程收到的数据仍为 4104 字节大小。

1.5 注意事项

1.5.1 串行化任务

经过 epoll 的监听，主线程中所有管道的收发都严格按照串行化的顺序来执行，因此两个进程中的缓冲区实际上不需要使用互斥锁等工具来避免缓冲区的读写出错。

还有需要注意的是，由于 epoll 始终监听，唤醒合适的事件执行任务，每次在 epoll 循环中**执行的任务不宜繁重**。根据实践发现，如果在 epoll 循环中创建变量，造成比较大的开销，会拖慢相关管道的唤醒，使得管道常常不能被唤醒，整个模型会发生堵塞。

1.5.2 epoll 触发模式+fcntl 阻塞/非阻塞模式

触发 epoll 唤醒事件的模式有水平触发（LT）和边沿触发（ET），而通过 fcntl 设置的管道 IO 有阻塞和非阻塞模式，默认是阻塞模式。经过实践发现，模型的运行效果总是和触发模式和 IO 模式的搭配有关。实践得到的最好搭配是**水平触发+非阻塞模式**。

个人理解其中的机制是：ET 模式只有在管道中文件读写内容发生变化时才会发出一次提醒，而 LT 模式只要管道中还有可读写的內容，就会不断提醒。本次实验发送的数据大小是 4104 字节，发送读取的数据比较大，在管道中读写常常不能一次性完成，如果使用 ET 模式，每次管道两端读写的内容大小不匹配，容易造成管道完全堵塞，而 LT 模式却不能察觉。

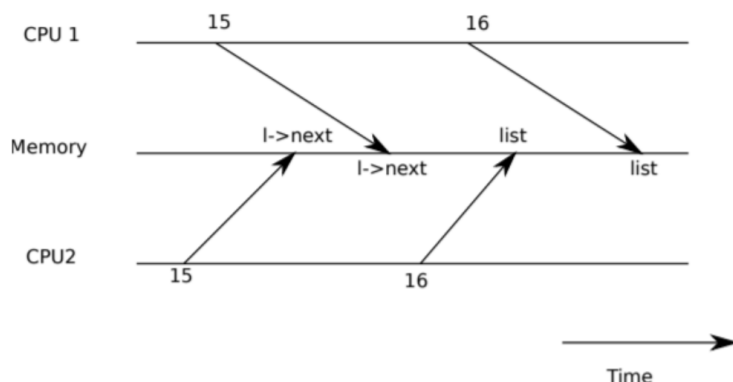
但 LT 也有其缺点，LT 可能会频繁提醒不需要关心的读写事件，在本实验中不需要考虑这个缺点，因为本实验使用 epoll 监听的事件都是必要的管道读写。

二、Xv6 lab: Lock/Memory allocator

2.1 任务要求

2.1.1 任务背景

任务测试中三个进程会通过使用内存分配器，通过增大和缩小各自的地址空间，从而不断调用 kalloc 和 kfree 函数。kalloc 和 kfree 本质上是对空闲页的链表进行 pop 或 push 操作。如果多个进程同时执行对空闲页链表的操作，很容易产生不正确的结果。



为了避免因为竞争而产生错误的结果，可以使用锁机制，将并行程序“串行

化”，例如在调用 kfree 和 kalloc 函数时添加锁，使得每次最多只有一个进程在对空闲页列表操作。

但加入过多的锁，容易对程序运行效率造成影响。例如锁控制住了可以并行操作的资源时，就会浪费各个进程的运行时间。

2.1.2 任务实现

任务要求实现为每个 CPU 维护一个空闲列表，每个列表都有自己的锁。不同 CPU 上的分配和释放可以并行运行，因为每个 CPU 将在不同的列表上运行。主要需要解决的问题是：一个 CPU 的空闲列表是空的，而另一个 CPU 的列表有空闲内存；在这种情况下，一个 CPU 必须“窃取”另一个 CPU 空闲列表的一部分。窃取可能会引入锁争用，但这种情况很可能很少发生。

2.2 设计思路

2.2.1 维护列表

首先需要将原来的单个空闲列表扩充到 CPU 数目，每个列表中会包含一个自旋锁，这样基本的运行过程中可以保证进程间不发生竞争。

2.2.2 窃取列表

虽然只要给每个 CPU 维护一个空闲列表就可以避免竞争，但是这样会造成内存巨大的开销。因此可以尝试每个 CPU 在需要更多内存的时候，去窃取其他 CPU 的空闲列表。这样的话，就将问题由多个 CPU 竞争一把锁，转换为了多个 CPU 竞争多把锁的问题，也就是并行分配。直观上来说，可以有效降低竞争发生的次数，同时还有利于提升并行效率。

2.2.3 结果展示

任务还要求输出竞争发生的次数，通过观察数值变化，可以判断以上的设计思路是否能够有效优化竞争问题。

2.3 代码实现

2.3.1 修改结构体

首先修改 [kmem 结构体](#)，将原来的单个空闲链表修改为 CPU 数目的空闲链表数组，部分代码如下图所示。（点击超链接跳转到附录，下同）

```
21 // 每个CPU都分配一个锁
22 struct {
23     struct spinlock lock;
24     struct run *freelist;
25     char lock_name[8];
26 } kmem[NCPU];
```

其中的常量 NCPU 在 kernel/param.h 中可见，表示 CPU 的最大数目。

然后修改 kinit 函数，重新对 kmem 初始化，代码如下图所示。

```
28 void
29 kinit()
30 {
31     for(int i = 0; i < NCPU; i++) {
32         // 调用snprintf函数，字符串格式化
33         snprintf(kmem[i].lock_name, sizeof(kmem[i].lock_name), "kmem_%d", i);
34         initlock(&kmem[i].lock, kmem[i].lock_name);
35     }
36     freerange(end, pa_end: (void*)PHYSTOP);
37 }
```

对于每个 CPU 包含的空闲链表，首先调用 snprintf 函数，将锁的名字格式化。snprintf 函数的相关定义可以在 kernel/sprintf.c 中找到。然后调用 initlock 函数对链表进行初始化。

2.3.2 修改空闲链表操作

要实现窃取列表的操作，就需要修改 kalloc 和 kfree 两个对空闲链表直接操作的函数。

首先修改 [kfree 函数](#)，首先在释放空闲页的前后，会添加设备的开启/断开，以保证同一时间只有一个 CPU 在调用 kfree。因为后续的分配内存过程中，多个 CPU 互相窃取各自维护的空闲链表，实际上可以看作是同一个比较大的空闲链表。因此当 CPU 在对空闲链表操作时，需要调用 push_off、pop_off 直接开启、关闭中断，来使得 CPU 的操作串行化。其他的修改只需要在 kemem 后加上数组下标索引即可。

然后修改 [kalloc 函数](#)，首先也是会在操作自己 CPU 维护的空闲链表的前后，调用 push_off、pop_off 直接开启、关闭中断，并且修改下标索引。

然后添加窃取空闲链表的程序，如果自己 CPU 的空闲链表没有剩余时，就会

循环遍历所有其他 CPU 维护的空闲链表，进行相同的操作。

2.4 测试结果

```
$ kallocetest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem_0: #test-and-set 0 #acquire() 174839
lock: kmem_1: #test-and-set 0 #acquire() 140289
lock: kmem_2: #test-and-set 0 #acquire() 117935
lock: bcache: #test-and-set 0 #acquire() 1242
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 3261315 #acquire() 114
lock: proc: #test-and-set 1985779 #acquire() 525899
lock: proc: #test-and-set 1900164 #acquire() 525900
lock: proc: #test-and-set 1583864 #acquire() 525899
lock: proc: #test-and-set 1563437 #acquire() 525900
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
```

三、Xv6 lab: Lock/ Buffer cache

3.1 任务要求

3.1.1 任务背景

块缓存用于文件系统的维护缓存。由于块缓存存在主机磁盘上，对所有进程都是完全共享的，因此块缓存的争用问题解决更加困难，主要难点在于块缓存不能被任何一个进程私占，且大量进程需要频繁使用文件系统。

关于块缓存的描述可以在 xv6-book 中的 8.1-9.3 节找到。下面根据 8.3 节的内容，同时结合 kernel/bio.c 的程序介绍块缓存的运行机制。

首先是 binit 函数，主要作用是用静态数组 buf 初始化双端链表 buffer 缓存。所有进程访问块缓存时都是通过操作这个链表进行的，这方面与内存的空闲链表相似。

然后是 bget 函数，作用是查找缓冲区，程序首先会根据给定的设备号和扇区号来查找缓冲区，如果存在，就会获取该 buffer 的 sleep-lock。然后返回被锁定的 buffer。如果不存在，那么程序会再次扫描 buffer 链表，寻找没有被使用的 buffer，生成能够使用的 buffer，修改相关的设备号和扇区号，并获得其

sleep->lock。在 bget 查找的过程中，遵循 LRU 规则，即找到最近使用最少的缓冲区。bget 函数代码如下图所示。

```
58 static struct buf*
59 bget(uint dev, uint blockno)
60 {
61     struct buf *b;
62
63     acquire(&bcache.lock);
64
65     // Is the block already cached?
66     for(b = bcache.head.next; b != &bcache.head; b = b->next){
67         if(b->dev == dev && b->blockno == blockno){
68             b->refcnt++;
69             release(&bcache.lock);
70             acquiresleep(&b->lock);
71             return b;
72         }
73     }
74
75     // Not cached.
76     // Recycle the least recently used (LRU) unused buffer.
77     for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
78         if(b->refcnt == 0) {
79             b->dev = dev;
80             b->blockno = blockno;
81             b->valid = 0;
82             b->refcnt = 1;
83             release(&bcache.lock);
84             acquiresleep(&b->lock);
85             return b;
86         }
87     }
88     panic("bget: no buffers");
89 }
```

最后是 brelse 函数，当调用者处理完一个 buffer 后，必须调用 brelse 来释放它。brelse 会释放 sleep-lock，并将改 buffer 移动到链表的头部，代码如下图所示。

```

114 // Release a locked buffer.
115 // Move to the head of the most-recently-used list.
116 void
117 brelse(struct buf *b)
118 {
119     if(!holdingsleep(&b->lock))
120         panic("brelse");
121
122     releasesleep(&b->lock);
123
124     acquire(&bcache.lock);
125     b->refcnt--;
126     if (b->refcnt == 0) {
127         // no one is waiting for it.
128         b->next->prev = b->prev;
129         b->prev->next = b->next;
130         b->next = bcache.head.next;
131         b->prev = &bcache.head;
132         bcache.head.next->prev = b;
133         bcache.head.next = b;
134     }
135
136     release(&bcache.lock);
137 }

```

移动 buffer 会使链表按照 buffer 最近使用的时间（最近释放）排序，链表中的第一个 buffer 是最近使用的，最后一个是最早使用的。

3.1.2 任务要求

任务要求修改块缓存，以便在运行 bcachetest 时 bcache 中所有锁的获取循环迭代次数接近于零。此外，还需要修改 bget 和 brelse 函数，以便 bcache 中不同块的并发查找和释放不太可能在锁上发生冲突。

3.2 设计思路

3.2.1 哈希表

根据任务提示，可以使用固定数量的 bucket 和大小不可更动的动态哈希表来替换原来的链表，维护 LRU 信息。此外，使用素数个 bucket（例如 13）来减少散列冲突的可能性。

3.2.2 桶级锁

原来每次 bget/brelse 函数操作，获取/释放缓冲区时需要对整个链表加锁，导致缓冲区的操作完全串行化。使用了上述的哈希表来维护 LRU 信息之后，就可以每次只对哈希表的一个桶进行加锁，桶之间的操作可以并行进行。只有需要对

缓冲区进行驱逐替换时，才需要对整个哈希表加锁来查找要替换的块。

3.2.3 brelse 优化

brelse 函数只需要根据哈希表，修改响应的 LRU 信息、锁等即可。

3.2.4 bget 优化

原来的 bget 会对整个链表扫描至多两遍，如果改为哈希表，bget 的扫描可以修改为：首先在对应桶中扫描，如果找到了块缓存，就返回；如果没有找到，就再去整个哈希表中扫描。

3.3 代码实现

修改的代码在附录中展示，如下同。

3.3.1 哈希表结构体

首先在 kernel/buf.h 中修改 buf 结构体。将原来的链表指针 prev 删除，另外添加一个时间戳，来记录块缓存最新被释放的时间。

然后在 kernel/bio.c 中定义[哈希表的结构体](#)，将原来的 bcache 结构体中的 head 提出来，另外构建为桶结构体，并添加一个自旋锁，作为桶级锁。

最后修改 binit 函数，初始化哈希表结构体。首先初始化每个桶拥有的桶级锁，然后将块平均分配给每个桶，记录下块缓存所处的桶编号，并将每个桶内的块缓存以链表的方式连接起来。

3.3.2 brelse 函数

[brelse 函数](#)主要的修改有两个，一个是将内核的时间信息 ticks 存入哈希表的块缓存的时间戳 timestamp，作为之后扫描的信息。还有一个是，因为在初始化结构体时，块缓存直接存储在桶结构体中，因此需要将原来的表级锁 bcache->lock 更换为桶级锁 hashtable->lock。

3.3.3 bget 函数

因为本实验主要修改了 LRU 信息表的结构，bget 原来的扫描方式需要全部重新更改。首先，[bget 函数](#)会在对应的桶中查找当前块是否被缓存，如果被缓存就

直接返回；如果没有被缓存，就查找另外一个块将其替换。查找另一个块时也会先在同一个桶中扫描，如果桶中的块都已经被缓存了，没有可以替换的块，就会去哈希表的其他桶中寻找，直到找到为止。

需要注意的是，在不同范围中查找时，需要添加不同等级的锁，在目标块所处的桶中查找时，需要加上相应的桶级锁，查找结束后立即释放；在其他桶中查找时，需要先加上表级锁，再加上响应的桶级锁，共两重锁。

这样修改后的 `bget` 函数，将大部分的查找工作并行化，不同的桶查找之间可以并行。

3.3.4 其他函数

最后还需要在所有原来用到表级锁的地方，都更换为桶级锁。例如 [bpin](#)、[bunpin](#) 函数。

3.4 测试结果

在终端使用 `make grade` 命令同时测试实验二、三得到的结果如下图所示。

```
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipel: OK
test killstatus: 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
$
```

四、Xv6 lab: File System/Symbolic links

4.1 任务要求

4.1.1 任务背景

在 xv6 中，已经实现了硬链接，因此可以首先参考 xv6-book 上 8.14 节的关于 `sys_link` 和 `sys_unlink` 的系统调用讲解，来分析硬链接的实现，用类似的方法实现软链接。

函数 `sys_link` 和 `sys_unlink` 可以编辑目录，创建或删除对 inodes 的引用。`sys_link` 首先获取它的参数，两个字符串 `old` 和 `new`。假设 `old` 存在并且不是一个目录，`sys_link` 会递增它的 `ip->nlink` 计数。然后 `sys_link` 调用 `nameparent` 找到 `new` 的父目录和最终路径元素，并创建一个指向 `old` 的 inode 的新目录项。新的父目录必须存在，并且和现有的 inode 在同一个设备上，inode 号只在同一个磁盘上有意义。如果出现这样的错误，`sys_link` 必须返回并减少 `ip->nlink`。

`sys_link` 主要调用了 `create` 来为 inode 创建新的名字。而 `sys_link` 主要被三个文件创建相关的系统调用使用。第一个是使用 `O_CREATE` 标志的 `open` 创建一个新的普通文件；第二个是 `mkdir` 创建一个新的目录；最后一个是 `mkdev` 创建一个新的设备文件。具体关于 `create` 内部的逻辑，不再赘述，具体可以参考 xv6-book。

4.1.2 任务要求

任务要求添加一个符号链接的系统调用。符号链接是指按路径名链接的文件，当一个符号链接打开时，内核跟随该链接指向引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。此外，任务不要求实现 `mkdir` 或 `mkdev` 调用软链接的情况。

4.2 设计思路

在实现 `sys_link` 函数之前，需要对函数和系统调用进行一系列的声明，在之前的系统调用实验中已经尝试过相关操作，这里不再赘述，可以直接仿照 `sys_link` 的操作。下面直接讲解 `sys_symlink` 的主要实现思路。

4.2.1 `sys_symlink` 函数

首先分析 `sys_link` 函数，检查相关检查磁盘的代码并设计修改思路，`sys_link` 代码如下图所示。

```

118 // Create the path new as a link to the same inode as old.
119 uint64
120 sys_link(void)
121 {
122     char name[DIRSZ], new[MAXPATH], old[MAXPATH];
123     struct inode *dp, *ip;
124
125     if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
126         return -1;
127
128     begin_op();
129     if((ip = namei(old)) == 0){
130         end_op();
131         return -1;
132     }
133
134     ilock(ip);
135     if(ip->type == T_DIR){
136         iunlockput(ip);
137         end_op();
138         return -1;
139     }
140
141     ip->nlink++;
142     iupdate(ip);
143     iunlock(ip);

```

```

145     if((dp = nameiparent(new, name)) == 0)
146         goto bad;
147     ilock(dp);
148     if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
149         iunlockput(dp);
150         goto bad;
151     }
152     iunlockput(dp);
153     iput(ip);
154
155     end_op();
156
157     return 0;
158
159 bad:
160     ilock(ip);
161     ip->nlink--;
162     iupdate(ip);
163     iunlockput(ip);
164     end_op();
165     return -1;
166 }

```

在第 148 行—第 151 行，`sys_link` 对两个 inode 锁 `ip` 和 `dp` 的设备进行判断，如果不同则表示文件不属于同一磁盘。需要实现的 `sys_symlink` 则不需要对设备进行判断，需要去掉这部分代码。

此外，`sys_link` 中的两个参数都是同一磁盘中的，是现有的 inode，而 `sys_symlink` 中的 `path` 参数可以认为是新的 inode，因此需要调用 `create` 来为 `path` 创建新的名字。

以上是 `sys_symlink` 相对与 `sys_link` 的主要区别，也是 `sys_symlink` 函数主要的修改思路

4.2.2 调用者函数

因为任务要求不需要考虑 `mkdir` 或 `mkdev` 调用 `sys_symlink` 的情况，下面只考虑 `sys_open` 调用的设计思路。原来的 `sys_open` 调用 `create` 来实现 `sys_link` 的主要部分如下图所示。


```

300     if(omode & O_CREATE){
301         ip = create(path, T_FILE, major: 0, minor: 0);
302         if(ip == 0){
303             end_op();
304             return -1;
305         }
306     } else {
307         if((ip = namei(path)) == 0){
308             end_op();
309             return -1;
310         }
311         ilock(ip);
312         if(ip->type == T_DIR && omode != O_RDONLY){
313             iunlockput(ip);
314             end_op();
315             return -1;
316         }
317     }
318
319     if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
320         iunlockput(ip);
321         end_op();
322         return -1;
323     }

```

根据任务提示，sys_symlink 需要额外实现的是：如果链接的文件也是符号链接，则必须递归跟踪，直到到达非链接文件为止。如果链接形成循环，则必须返回错误代码。此外还有要求：其他系统调用（如链接和取消链接）不得跟随符号链接；这些系统调用对符号链接本身进行操作。

因此 sys_symlink 需要一个单独的类型标志位 T_SYMLINK 进行判断，且处理内容需要和 sys_link 隔离开来。主要的实现内容是将原来的 sys_lnk 实现的第 302 行-319 行递归实现。

4.3 代码实现

4.2.1 sys_symlink 函数

[sys_symlink 函数](#)代码在附录中展示。主要的内容是：调用 namei 函数为原来的 inode 创建文件名，调用 create 函数创建新的 inode，设置类型为 T_SYMLINK，并写入目标文件的路径。

4.2.2 调用者函数

[sys_open](#) 中需要添加对 T_SYMLINK 类型的 inode 处理的程序。主要步骤是首先判断原来的 inode 类型是否是 T_SYMLINK，如果是就表示正在进行符号链接，

然后再判断模式是否是 `O_NOFOLLOW`，如果不是，就需要对符号链接进行递归处理，找到真正的文件。递归内容主要是调用 `readi` 函数从 `ip` 中读取到 `target` 文件，并判断 `ip` 是否与 `target` 文件名相同，以此来判断是否找到真正的文件。

4.4 测试结果

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
```

```
test writebig: OK
test createtest: OK
test openinput: OK
test exitinput: OK
test input: OK
test mem: OK
test pipe1: OK
test killstatus: 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
```

五、附录

5.1 任务二

5.1.1 kmem 结构体

```
1. // 每个 CPU 都分配一个锁
2. struct {
3.     struct spinlock lock;
4.     struct run *freelist;
5.     char lock_name[8];
6. } kmem[NCPU];
7.
8. void
9. kinit()
10. {
11.     for(int i = 0; i < NCPU; i++) {
12.         // 调用 snprintf 函数，字符串格式化
13.         snprintf(kmem[i].lock_name, sizeof(kmem[i].lock_name), "kmem_%d",
14.             i);
15.         initlock(&kmem[i].lock, kmem[i].lock_name);
16.     }
17.     freerange(end, (void*)PHYSTOP);
18. }
```

5.1.2 kfree 函数

```
1. void
2. kfree(void *pa)
3. {
4.     struct run *r;
5.
6.     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
7.         panic("kfree");
8.
9.     // Fill with junk to catch dangling refs.
10.    memset(pa, 1, PGSIZE);
11.
12.    r = (struct run*)pa;
13.    // 开启、关闭终端
14.    // 相当于 CPU 的互斥锁
15.    push_off();
16.    int cpu_id = cpuid();
```

```

17. // 修改下标
18. acquire(&kmem[cpu_id].lock);
19. r->next = kmem[cpu_id].freelist;
20. kmem[cpu_id].freelist = r;
21. release(&kmem[cpu_id].lock);
22. pop_off();
23. }

```

5.1.3 kalloc 函数

```

1. void *
2. kalloc(void)
3. {
4.     struct run *r;
5.
6.     //先在自己的空闲链表上摘取空闲页
7.     push_off();
8.     int cpu_id = cpuid();
9.     acquire(&kmem[cpu_id].lock);
10.    r = kmem[cpu_id].freelist;
11.    if(r)
12.        kmem[cpu_id].freelist = r->next;
13.    release(&kmem[cpu_id].lock);
14.    pop_off();
15.
16.    if(!r) {
17.        // 从其他 CPU 的空闲链表上窃取
18.        for(int i=0; i<NCPU; i++) {
19.            acquire(&kmem[i].lock);
20.            r = kmem[i].freelist;
21.            if(r) {
22.                kmem[i].freelist = r->next;
23.                release(&kmem[i].lock);
24.                break;
25.            }
26.            release(&kmem[i].lock);
27.        }
28.    }
29.
30.    if(r)
31.        memset((char*)r, 5, PGSIZE);
32.    return (void*)r;
33. }

```

5.2 任务三

5.2.1 哈希表结构体

```
1. struct {
2.     struct spinlock lock; // 表级锁
3.     struct buf buf[NBUFS];
4. } bcache;
5.
6. struct bucket {
7.     struct spinlock lock; // 桶级锁
8.     struct buf head;
9. } hashtable[NBUCKET];
10.
11. int
12. hash(uint blockno)
13. {
14.     return blockno % NBUCKET;
15. }
16.
17. void
18. binit(void)
19. {
20.     struct buf *b;
21.
22.     initlock(&bcache.lock, "bcache");
23.     for(b = bcache.buf; b < bcache.buf+NBUFS; b++) {
24.         initsleeplock(&b->lock, "buffer");
25.     }
26.     b = bcache.buf;
27.     for(int i = 0; i < NBUCKET; i++) {
28.         initlock(&hashtable[i].lock, "bcache_bucket");
29.         for(int j=0; j<NBUFS/NBUCKET; j++) {
30.             b->blockno = i;
31.             b->next = hashtable[i].head.next;
32.             hashtable[i].head.next = b;
33.             b++;
34.         }
35.     }
36. }
```

5.2.2 brelse 函数

```
1. void
```

```

2. brelse(struct buf *b)
3. {
4.     if(!holdingsleep(&b->lock))
5.         panic("brelse");
6.
7.     releasesleep(&b->lock);
8.     int idx = hash(b->blockno);
9.     // 表级锁更换为桶级锁
10.    acquire(&hashtable[idx].lock);
11.    b->refcnt--;
12.    if (b->refcnt == 0) {
13.        // 记下最后被释放的事件
14.        b->timestamp = ticks;
15.    }
16.    release(&hashtable[idx].lock);
17. }

```

5.2.3 bget 函数

```

1. static struct buf*
2. bget(uint dev, uint blockno)
3. {
4.     struct buf *b;
5.
6.     // 根据块编号直接找到桶编号
7.     int idx = hash(blockno);
8.     struct bucket* bucket = hashtable + idx;
9.     acquire(&bucket->lock);
10.
11.    // 第一遍查找块是否被缓存
12.    for(b = bucket->head.next; b != 0; b = b->next){
13.        if(b->dev == dev && b->blockno == blockno){
14.            b->refcnt++;
15.            release(&bucket->lock);
16.            acquiresleep(&b->lock);
17.            // 已经被缓存，直接返回
18.            return b;
19.        }
20.    }
21.
22.    // 寻找可替换的块
23.    int timeout = 0x8fffffff;
24.    struct buf* replace_buf = 0;
25.    // 先在桶中寻找（上面已经设了桶级锁）
26.    for(b = bucket->head.next; b != 0; b = b->next){

```

```
27. // 根据引用计数判断缓存
28. if(b->refcnt == 0 && b->timestamp < timeout) {
29.     replace_buf = b;
30.     timeout = b->timestamp;
31. }
32. }
33. // 找到可替换的, 进行替换
34. if(replace_buf) {
35.     goto instead;
36. }
37.
38. // 如果同一桶中没找到, 就在表中查找
39. acquire(&bcache.lock);
40. refind:
41. for(b = bcache.buf; b < bcache.buf + NBUF; b++) {
42.     if(b->refcnt == 0 && b->timestamp < timeout) {
43.         replace_buf = b;
44.         timeout = b->timestamp;
45.     }
46. }
47. if (replace_buf) {
48.     // 将块从原来的桶中剔除
49.     int ridx = hash(replace_buf->blockno);
50.     // 防止桶切换的瞬间, 块被使用
51.     acquire(&hashtable[ridx].lock);
52.     if(replace_buf->refcnt != 0)
53.     {
54.         release(&hashtable[ridx].lock);
55.         goto refind;
56.     }
57.     struct buf *pre = &hashtable[ridx].head;
58.     struct buf *p = hashtable[ridx].head.next;
59.     while (p != replace_buf) {
60.         pre = pre->next;
61.         p = p->next;
62.     }
63.     pre->next = p->next;
64.     release(&hashtable[ridx].lock);
65.     // 将块放到现在桶中
66.     replace_buf->next = hashtable[idx].head.next;
67.     hashtable[idx].head.next = replace_buf;
68.     release(&bcache.lock);
69.     goto instead;
70. }
```

```

71. // 整个表中都没找到
72. else {
73.     panic("bget: no buffers");
74. }
75.
76. instead:
77. replace_buf->dev = dev;
78. replace_buf->blockno = blockno;
79. replace_buf->valid = 0;
80. replace_buf->refcnt = 1;
81. release(&bucket->lock);
82. acquiresleep(&replace_buf->lock);
83. return replace_buf;
84. }

```

5.2.3 bpin & bunpin 函数

```

1. void
2. bpin(struct buf *b) {
3.     int idx = hash( b->blockno);
4.     // 表级锁更换为桶级锁
5.     acquire(&hashtable[idx].lock);
6.     b->refcnt++;
7.     release(&hashtable[idx].lock);
8. }
9.
10. void
11. bunpin(struct buf *b) {
12.     int idx = hash(b->blockno);
13.     // 表级锁更换为桶级锁
14.     acquire(&hashtable[idx].lock);
15.     b->refcnt--;
16.     release(&hashtable[idx].lock);
17. }

```

5.3 任务四

5.3.1 sys_symlink 结构体

```

1. uint64
2. sys_symlink(void)
3. {
4.     char target[MAXPATH], path[MAXPATH];
5.     memset(target, 0, sizeof(target));
6.     // 判断合理性

```



```

7.     if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
8.         return -1;
9.
10.    struct inode *ip, *dp;
11.    begin_op();
12.    // namei 创建文件名
13.    if((ip = namei(target)) != 0) {
14.        // 判断标志位
15.        if(ip->type == T_DIR) {
16.            goto bad;
17.        }
18.    }
19.    // 调用 create 创建新文件名
20.    if((dp = create(path, T_SYMLINK, 0, 0)) == 0)
21.        goto bad;
22.    // 路径写入目标文件
23.    if(writei(dp, 0, (uint64)target, 0, MAXPATH) != MAXPATH) {
24.        // panic("symlink write failed");
25.        goto bad;
26.    }
27.    iunlockput(dp);
28.    end_op();
29.    return 0;
30. bad:
31.    end_op();
32.    return -1;
33. }

```

5.3.2 sys_open 函数

```

1.  uint64
2.  sys_open(void)
3.  {
4.      char path[MAXPATH];
5.      int fd, omode;
6.      struct file *f;
7.      struct inode *ip;
8.      int n;
9.
10.     if((n = argstr(0, path, MAXPATH)) < 0 || argint(1, &omode) < 0)
11.         return -1;
12.
13.     begin_op();
14.

```

```
15.  if(omode & O_CREATE){
16.      ip = create(path, T_FILE, 0, 0);
17.      if(ip == 0){
18.          end_op();
19.          return -1;
20.      }
21.  } else {
22.      if((ip = namei(path)) == 0){
23.          end_op();
24.          return -1;
25.      }
26.      ilock(ip);
27.      if(ip->type == T_DIR && omode != O_RDONLY){
28.          iunlockput(ip);
29.          end_op();
30.          return -1;
31.      }
32.  }
33.
34.  if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
35.      iunlockput(ip);
36.      end_op();
37.      return -1;
38.  }
39.
40.  if(ip->type == T_SYMLINK) {
41.      if(!(omode & O_NOFOLLOW)) {
42.          // 定义递归次数
43.          int count = 0;
44.          char target[MAXPATH];
45.          // 需要判断 T_SYMLINK 标志位
46.          while(ip->type == T_SYMLINK) {
47.              // 最大循环数
48.              if(count == 10) {
49.                  iunlockput(ip);
50.                  end_op();
51.                  return -1;
52.              }
53.              count++;
54.              memset(target, 0, sizeof(target));
55.              // 递归打开 ip 文件
56.              readi(ip, 0, (uint64)target, 0, MAXPATH);
57.              iunlockput(ip);
58.              // 判断是否递归结束
```

```
59.         if((ip = namei(target)) == 0){
60.             end_op();
61.             return -1; // target not exist
62.         }
63.         ilock(ip);
64.     }
65. }
66. }
67.
68. if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
69.     if(f)
70.         fileclose(f);
71.     iunlockput(ip);
72.     end_op();
73.     return -1;
74. }
75.
76. if(ip->type == T_DEVICE){
77.     f->type = FD_DEVICE;
78.     f->major = ip->major;
79. } else {
80.     f->type = FD_INODE;
81.     f->off = 0;
82. }
83. f->ip = ip;
84. f->readable = !(omode & O_WRONLY);
85. f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
86.
87. if((omode & O_TRUNC) && ip->type == T_FILE){
88.     itrunc(ip);
89. }
90.
91. iunlock(ip);
92. end_op();
93.
94. return fd;
95. }
```