1. 实验目的

掌握 buddy allocator 的实现原理。
学习在 xv6 中应用伙伴系统算法并优化它的实现。

2. 实验环境

通过 Vscode 连接远程实验环境进行实验。

3. 实验内容

3.1 实验任务

1、应用伙伴系统优化 xv6 的内存管理
2、优化伙伴系统算法在 xv6 中的实现

3.2 实验过程

实验过程:

任务一: 应用伙伴系统优化 xv6 的内存管理

这个任务要求动态分配对象，故打开 file.c 文件查看文件对象的原始分配方式。

```
// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

可见原始文件分配方式为直接初始化一个文件数组，故找到文件数组位置将其注释，并把 filealloc 方法改为使用 bd_alloc 动态分配

```
struct {
  struct spinlock lock;
  // struct file file[NFILE];
} ftable;
```

```
struct file*
filealloc(void)
{
  struct file *f;
  acquire(&ftable.lock);
  f = (struct file *)bd_malloc(sizeof(struct file));
  if(f){
    memset(f,0,sizeof(struct file));
    f->ref = 1;
    release(&ftable.lock);
    return f;
  }
  release(&ftable.lock);
  return 0;
```

这里需要注意通过 bd_malloc 获取的内存需要进行清空才能使用。

此外由于改变了 filealloc，故还要查看 fileclose 方法是否需要改变。原先的 fileclose 方法如下:

```
// Close file f.  (Decrement ref count, close when reaches 0.)
void
fileclose(struct file *f)
{
  struct file ff;

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);

  if(ff.type == FD_PIPE){
    pipeclose(ff.pipe, ff.writable);
  } else if(ff.type == FD_INODE || ff.type == FD_DEVICE){
    begin_op(ff.ip->dev);
    iput(ff.ip);
    end_op(ff.ip->dev);
  }
}
```

可知该方法大体上无需改变，但由于此时 file 是 bd_malloc 得来的，故需要调用 bd_free 来释放。修改后方法如下:

```
// Close file f.  (Decrement ref count, close when reaches 0.)
void
fileclose(struct file *f)
{
  struct file ff;

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);
  bd_free(f);
  if(ff.type == FD_PIPE){
    pipeclose(ff.pipe, ff.writable);
  } else if(ff.type == FD_INODE || ff.type == FD_DEVICE){
    begin_op(ff.ip->dev);
    iput(ff.ip);
    end_op(ff.ip->dev);
  }
}
```

任务二: 优化伙伴系统算法在 xv6 中的实现

首先观察该算法中使用的主要结构:

```
// 8 blocks).
struct sz_info {
  Bd_list free;
  char *alloc;
  char *split;
};
```

由于主要优化对象为 alloc，故选择集中于 alloc 的调用位置来考虑是否修改。

由于该代码文件中方法较多，故选择从 bd_init 方法入手具体方法如下:

```
void
bd_init(void *base, void *end) {
  char *p = (char *) ROUNDUP((uint64)base, LEAF_SIZE);
  int sz;

  initlock(&lock, "buddy");
  bd_base = (void *) p;

  // compute the number of sizes we need to manage [base, end)
  nsizes = log2(((char *)end-p)/LEAF_SIZE) + 1;
  if((char*)end-p > BLK_SIZE(MAXSIZE)) {
    nsizes++;  // round up to the next power of 2
  }

  printf("bd: memory sz is %d bytes; allocate an size array of length %d\n",
         (char*) end - p, nsizes);

  // allocate bd_sizes array
  bd_sizes = (Sz_info *) p;
  p += sizeof(Sz_info) * nsizes;
  memset(bd_sizes, 0, sizeof(Sz_info) * nsizes);

  // initialize free list and allocate the alloc array for each size k
  for (int k = 0; k < nsizes; k++) {
    lst_init(&bd_sizes[k].free);
    sz = sizeof(char)* ROUNDUP(NBLK(k), 8)/8;
    bd_sizes[k].alloc = p;
    memset(bd_sizes[k].alloc, 0, sz);
    p += sz;
  }

  // allocate the split array for each size k, except for k = 0, since
  // we will not split blocks of size k = 0, the smallest size.
  for (int k = 1; k < nsizes; k++) {
    sz = sizeof(char)* (ROUNDUP(NBLK(k), 8))/8;
    bd_sizes[k].split = p;
    memset(bd_sizes[k].split, 0, sz);
    p += sz;
  }
  p = (char *) ROUNDUP((uint64) p, LEAF_SIZE);

  // done allocating; mark the memory range [base, p) as allocated, so
  // that buddy will not hand out that memory.
  int meta = bd_mark_data_structures(p);

  // mark the unavailable memory range [end, HEAP_SIZE) as allocated,
  // so that buddy will not hand out that memory.
  int unavailable = bd_mark_unavailable(end, p);
  void *bd_end = bd_base+BLK_SIZE(MAXSIZE)-unavailable;

  // initialize free lists for each size k
  int free = bd_initfree(p, bd_end);

  // check if the amount that is free is what we expect
  if(free != BLK_SIZE(MAXSIZE)-meta-unavailable) {
    printf("free %d %d\n", free, BLK_SIZE(MAXSIZE)-meta-unavailable);
    panic("bd_init: free mem");
  }
}
```

查看方法可知，该方法只在红框处显式对 alloc 进行处理，在蓝框处可能隐式地处理了 alloc。

对于红框处，由于我们会让一对伙伴块共用一个 bit，故我们只需要给 alloc 分配原有的一半内存即可。故改进如下:

```
// initialize free list and allocate the alloc array for each size k
for (int k = 0; k < nsizes; k++) {
  lst_init(&bd_sizes[k].free);
  sz = sizeof(char) * ROUNDUP(NBLK(k), 8) / 8;
  sz /= 2;
  //sz最小值应为sizeof(char)
  sz = (sz < sizeof(char)) ? sizeof(char) : sz;
  bd_sizes[k].alloc = p;
  memset(bd_sizes[k].alloc, 0, sz);
  p += sz;
}
```

由于直接/2 存在当 NBLK(k)<8 时，sz 的值直接被舍入为 0 的情况，故需要加入判断，来确保 sz 最小值为 sizeof(char)。(因为每层的 alloc 最少也需要一个 char 大小的空间)。也可以改为 `sz = sizeof(char) * ROUNDUP(NBLK(k), 16) / 16;` 通过 ROUNDUP()实现，且此时无须判断 sz 的值，因为 ROUNDUP()确保了 sz 的值至少为一个 sizeof(char)大小。

再去查看蓝框处的函数，发现前两个函数共同调用的 bd_mark()函数存在对 alloc 的修改。具体函数如下:

```
// Mark memory from [start, stop), starting at size 0, as allocated.
void
bd_mark(void *start, void *stop)
{
  int bi, bj;

  if (((uint64) start % LEAF_SIZE != 0) || ((uint64) stop % LEAF_SIZE != 0))
    panic("bd_mark");

  for (int k = 0; k < nsizes; k++) {
    bi = blk_index(k, start);
    bj = blk_index_next(k, stop);
    for(; bi < bj; bi++) {
      if(k > 0) {
        // if a block is allocated at size k, mark it as split too.
        bit_set(bd_sizes[k].split, bi);
      }
      bit_set(bd_sizes[k].alloc, bi);
    }
  }
}
```

查看 bit_set 函数可知它进行了对 alloc 的置 1 操作，由于修改后对于 alloc 的操作为异或，故引入新函数 alloc_set，并修改 bd_mark。结果如下:

```
void
bd_mark(void *start, void *stop)
{
  int bi, bj;

  if (((uint64) start % LEAF_SIZE != 0) || ((uint64) stop % LEAF_SIZE != 0))
    panic("bd_mark");

  for (int k = 0; k < nsizes; k++) {
    bi = blk_index(k, start);
    bj = blk_index_next(k, stop);
    //若步进为2可能导致alloc_set缺少一次判断
    for(; bi < bj; bi++) {
      if(k > 0) {
        // if a block is allocated at size k, mark it as split too.
        bit_set(bd_sizes[k].split, bi);
      }
      alloc_set(bd_sizes[k].alloc, bi);
    }
  }
}
```

```
// 使用异或设置alloc
void alloc_set(char *array, int index) {
  // 0,1均用第0位,2,3用第1位
  index /= 2;
  char m = (1 << (index % 8));
  array[index/8] ^= m;
}
```

最后查看蓝框内的最后一个函数 bd_initfree，可知其内部的 bd_initfree_pair 使用了 alloc。

```
int
bd_initfree_pair(int k, int bi) {
  int buddy = (bi % 2 == 0) ? bi+1 : bi-1;
  int free = 0;
  if(bit_isset(bd_sizes[k].alloc, bi) !=  bit_isset(bd_sizes[k].alloc, buddy)) {
    // one of the pair is free
    free = BLK_SIZE(k);
    if(bit_isset(bd_sizes[k].alloc, bi))
      lst_push(&bd_sizes[k].free, addr(k, buddy));   // put buddy on free list
    else
      lst_push(&bd_sizes[k].free, addr(k, bi));      // put bi on free list
  }
  return free;
}
```

该函数执行的操作为:若发现一对伙伴块的 alloc 值不同(0,1 或 1,0)则将 free 设为 BLK_SIZE(k),并将空闲块的地址插入空闲链表中。而修改后 alloc 的值为一对伙伴块的异或，

条件可修改为
```
//异或值为1即兄弟块之间有一个free
if(alloc_isset(bd_sizes[k].alloc,buddy) == 1){
```
，而该函数体如下：

```
// 若当前alloc值为1返回1
int alloc_isset(char *array, int index) {
  index /= 2;
  char b = array[index/8];
  char m = (1 << (index % 8));
  return (b & m) == m;
}
```

修改后，由于 alloc 只能表示一对伙伴块的共同状态，不能区分哪个块是空闲的，故不能判断应该将哪个块插入空闲链表。故引入当前可用地址的边界修改后函数如下：

```
int
bd_initfree_pair(int k, int bi,void *bd_left, void *bd_right) {
  int buddy = (bi % 2 == 0) ? bi+1 : bi-1;
  int free = 0;
  //异或值为1即兄弟块之间有一个free
  if(alloc_isset(bd_sizes[k].alloc,buddy) == 1){
    free = BLK_SIZE(k);
    //bi位于可用地址内
    if(addr_in_range(addr(k,bi),bd_left,bd_right,free)){
      lst_push(&bd_sizes[k].free, addr(k, bi));
    }
    else{
      lst_push(&bd_sizes[k].free, addr(k, buddy));
    }
  }
  return free;
}
```

```
// 判断地址范围是否满足条件
int addr_in_range(void *addr,void *left,void *right,int size){
  return (addr >= left)&&((addr + size) <= right);
}
```

由于修改了函数声明故需要在调用处修改，修改如下:

```
int
bd_initfree(void *bd_left, void *bd_right) {
  int free = 0;

  for (int k = 0; k < MAXSIZE; k++) {   // skip max size
    int left = blk_index_next(k, bd_left);
    int right = blk_index(k, bd_right);
    free += bd_initfree_pair(k, left,bd_left,bd_right);
    if(right <= left)
      continue;
    free += bd_initfree_pair(k, right,bd_left,bd_right);
  }
  return free;
}
```

此时 bd_init 函数全部修改完毕，再去查看代码文件中哪里使用了 alloc，将这些地方修改为 alloc 专用函数即可。具体修改如下:

```
void *
bd_malloc(uint64 nbytes)
{
  int fk, k;

  acquire(&lock);

  // Find a free block >= nbytes, starting with smallest k possible
  fk = firstk(nbytes);
  for (k = fk; k < nsizes; k++) {
    if(!lst_empty(&bd_sizes[k].free))
      break;
  }
  if(k >= nsizes) { // No free blocks?
    release(&lock);
    return 0;
  }

  // Found a block; pop it and potentially split it.
  char *p = lst_pop(&bd_sizes[k].free);
  alloc_set(bd_sizes[k].alloc, blk_index(k, p));
  for(; k > fk; k--) {
    // split a block at size k and mark one half allocated at size k
    // and put the buddy on the free list at size k-1
    char *q = p + BLK_SIZE(k-1);   // p's buddy
    bit_set(bd_sizes[k].split, blk_index(k, p));
    alloc_set(bd_sizes[k-1].alloc, blk_index(k-1, p));
    lst_push(&bd_sizes[k-1].free, q);
  }
  release(&lock);

  return p;
}
```

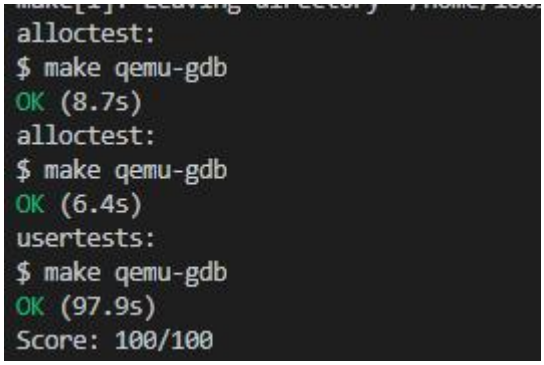
```
void
bd_free(void *p) {
  void *q;
  int k;

  acquire(&lock);
  for (k = size(p); k < MAXSIZE; k++) {
    int bi = blk_index(k, p);
    int buddy = (bi % 2 == 0) ? bi+1 : bi-1;
    //再次异或
    alloc_set(bd_sizes[k].alloc, bi);  // free p at size k
    //若仍为1则不可合并
    if (alloc_isset(bd_sizes[k].alloc, buddy)) {  // is buddy allocated?
      break;   // break out of loop
    }
    // budy is free; merge with buddy
    q = addr(k, buddy);
    lst_remove(q);    // remove buddy from free list
    if(buddy % 2 == 0) {
      p = q;
    }
    // at size k+1, mark that the merged buddy pair isn't split
    // anymore
    bit_clear(bd_sizes[k+1].split, blk_index(k+1, p));
  }
  lst_push(&bd_sizes[k].free, p);
  release(&lock);
}
```

结果:

```
$ alloctest
filetest: start
filetest: OK
memtest: start
memtest: OK
```

```
bigdir test
bigdir ok
exec test
ALL TESTS PASSED
```



4. 总结及实验课程感想

收获: 这次实验给我最大的收获就是了解了伙伴系统算法，并实现了对这个算法的优化。此外还让我知道了修改代码文件时应该如何下手才能比较迅速且准确地完成修改。

反思: 这次实验遇到的问题主要在于我对伙伴算法的理解和具体代码实现之间存在一定的差别:虽然我可以大致理解伙伴算法的优化原理，但由于我不熟悉 buddy.c 代码文件且没有掌握如何快速理清代码思路的技巧，导致我在修改 buddy.c 代码文件还是消耗了许多不必要的时间。

建议: 可以尝试加强原理和 xv6 实际代码的联系。