Experiment Report For Oslab 2019

Chen Xu 171180558

April 8, 2019

1 L1:alloc

1.1 思路

分为两个部分来进行alloc的编写，

其一，使用bitmap维护的大内存的访问和查询，这一部分需要对alloc的操作进行上锁，以维护bitmap的性质；

其二，直接访问并赋值的小内存申请，在alloc中首先申请小的一块内存，进而做到实时分配，若当前内存已满，则回到第一种情况，申请一块大的内存进行访问，再给小的分配即可，此步不需要上锁。

free操作则仅仅需要修改bitmap即可。

bitmap的定义如下：

1.$bmp[i]$ 维护第$i-lowbit(i)$ 到 i的连续的大内存块的使用信息，$bmp[i] = 0$ 表面未使用，其余情况均为占用中。

2.alloc申请一块较大内存时(假设为i)，将$i - lowbit(i) - i$的bmp 全部设置为1（(初始一定全部为0)，并使得

$$bmp[p]++; p = i + lowbit(i) \quad while(p < mpsize)$$

即将包含$bmp[i]$的所有大内存块全置为已占用，这样一来，bmp[i]=0时，意味着其子内存块都未被使用，我们便维护好了内存连续的信息。

3.free仅需将$bmp[i] - bmp[i - lowbit(i)]$ 全部清零即可，同时，i对应的所有上层标记应减一($bmp[i]$的内存已释放)

1.2 具体实现

ALLOC

```
1  static void* kalloc(size_t size) {
2          if(size>=BLOCK_SIZE) {
3                  lock(alloc_lk);
4                  size=size+(BLOCK_SIZE-size%BLOCK_SIZE)
                       ;
5                  size/=BLOCK_SIZE;
6                  int pos=bt_alloc(size);
7                  unlock(alloc_lk);
8                  return (void *)(pm_end-pos*BLOCK_SIZE)
                       ;
9          }
10         else {
11                 size=size+base_sz-size%base_sz;
12                 if(!current_ptr||off_set+size>=
                       BLOCK_SIZE) {
13                         lock(alloc_lk);
14                         cu_pos=bt_alloc(1);
15                         current_ptr=pm_end-cu_pos*
                               BLOCK_SIZE;
16                         off_set=size;
17                         unlock(alloc_lk);
18                 }
19                 else {
20                         off_set+=size;
21                         bt_add(cu_pos);
22                 }
23                 return (void *)(current_ptr+off_set-
                       size);
24         }
25 }
```

FREE

```
1         lock(alloc_lk);
2         intptr_t pos=(intptr_t)ptr;
3         while(pos%BLOCK_SIZE)    {
4                 pos-=pos%BLOCK_SIZE;
5         }
6         pos=pm_end-pos;
7         pos/=BLOCK_SIZE;
8         bt_free(pos);
9         unlock(alloc_lk);
```

其中，带bt前缀的函数均为在bitmap上的操作，在alloc.h中实现并定义，这样就实现了数据维护与函数调用的模块化，在调试中给了我很大的帮助。

1.3 遇见的问题

虽然此方法实现了alloc和free中内存的连续等信息的维护，并且未使用链表等数据结构，其框架较为清晰与精巧(自吹自擂)，但问题也十分明显。

其一,资源存在浪费，比如1-8的内存，如果我申请了1的内存再申请7的内存，由于分割内存时bitmap只能考虑2的幂的大小的块，这会导致内存的无法分配，即使存在一个连续的7长度的内存(2-8).

其二，碎片化与效率之间的矛盾：如果在查询内存时，我进行随机的index查询 (同一长度($lowbit(i) == l$)的位置可表示为(randindex*2*l+l))，其得到内存块的速度会大大提高，但这会导致内存的碎片化，大量的连续内存无法使用，而如果简单的顺序访问，碎片化大大减少(例如连续的1内存的申请会占用1，3，5，7，9....)等空间，基本是紧凑的分配，但效率会下降(最坏遍历整个bitmap)，目前还没想到较好的解决方法，根据workload我选择了碎片化较低，冲突较少的顺序访问。