Experiment Report For Oslab 2019

Chen Xu 171180558

May 15, 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。

2 L2:kthreads

2.1 思路

不同部分的思路如下:

- **spin_lock:**

  使用xv6的自旋锁作为范例，再做稍微的简化(维护了CPU的编号，但未维护CPU中的信息(esp等))

- **sem:**

  采用信号量可负，sem_wait 中仅yield一次的实现，将执行线程的park标志置为1，表示其已在某信号量中等待，在中断中不予调用，同时信号量中维护一个线程的链表，用于唤醒最先进入睡眠的线程(头)和添加下一个线程(尾),唯一值得注意的一点是对于sem_wait/sem_signal的锁，由于操作中一定有一个yield，但无法在持有锁的情况下使用(中断已关闭),故wait中有lock-unlock-yield-lock-unlock的操作，以保证能顺利切换线程

- **thread related:**

  保证所有的线程都储存在一个指针数组loader中，用于后续线程的保存和切换。

  调用已有的_kcontext函数做task中context上下文的保存，申请好一部分STACK对应的内存用于保存信息，查询loader中指针为空对应的下标i,loader[i]=new task即可，此步操作未上锁(保证单一线程仅仅在单一CPU上执行，故在关中断后不存在锁的问题)

  注册了2个线程相关的handler: save and switch，并维护一个对应当前线程的current指针数组，current[i]即当前在第i个CPU上执行的线程。

  在save中，若当前线程为空，则create一个包含**null**为入口的线程，**null**中即一直的yield，用于不断尝试是否有新的可执行线程(故当前执行的null相当于空线程)，并保存上下文信息。

  在switch中，**我们保证每一个CPU仅运行loader中模CPU数为当前CPU编号的线程**，即当有4个CPU时，CPU 1 仅能运行loader[1]，loader[5]，loader[9]等线程，这样之后，所有的线程均仅能在一个CPU上运行了，同时维护当前线程的state信息(0-runable,2-running)运行switch中的切换，

保证不会调用到running的进程，但当无可用进程时，会调用该CPU对应的null进程(每一个CPU一定会create一个对应的null进程)

- **trap and irq:**

  对于irq，维护一个rem_handler数组保存已注册的handler信息，并每一次维护其对应顺序(类似于seq为键值的插入排序)。

  对于trap,按序访问rem_handler，当rem_hanlder[i].event=event|| _EVENT_NULL时调用，返回最终的上下文即可。由于CPU与线程的对应限制，此步仍不需要上锁，实测未上锁的trap可大大加快整个系统的运行速度。

2.2 具体实现

过多，在此不一一列出

2.3 遇见的问题

1. lock 的实现有误：参考xv6的锁后发现存在这样几点：

   1.1 中断并不一定在释放锁时打开：只有上锁前中断已打开且当前所有的锁(并不是单一的锁)释放时才打开中断

   1.2 每一个CPU持有的锁的数量应各自维护

   1.3 在spinlock中的操作并不全是原子的，虽然有_sync系列保证一定的原子性，在仍有可能在atomic_xchg的前后有同时的访问，应注意到所有可能的bug

2. sem

   一开始想要使用循环yield+后减信号量的wait实现，但后来发现难以处理锁的问题，于是换成了先减+单yield的做法。

3. thread related：

   一开始创建的null并没有按照cpu的id进入存放，导致有可能会有CPU找不到可用的进程导致bug，后修复。

   switch中若不维护running的state容易使CPU陷入无限进入null线程的死循环，从而彻底崩掉

4. trap and irq 在trap中本来有加锁，且由于该锁整个系统的工作效率都极慢，后来发现，当CPU和线程绑定后，不同CPU调用trap并不会对某个相同的task做访问/修改，故可以令他们并行运行。这样之后大大加快了程序的运行速度。