OS-Lab1-Report

21302010042
侯斌洋

1：xv6-kalloctest

（1）如果没有锁，则 kfree 和 kalloc 中均会出现进程间竞争的问题，具体表现在 kfree 的 r->next = kmem.freelist; kmem.freelist = r;和 kalloc 中的 kmem.freelist = r; r->next= kmem.freelist;语句上。例如，如果并发执行 kfree 的第一条指令而第二条指令并未执行，则有两个 r 指向了同一个位置，导致链表分叉，出现错误。

（2）kalloc.c 在修改前，由于只有一个锁，多个 CPU 在并行运行时都会抢夺这一个锁，而只有一个 CPU 会得到锁并运行，其他 CPU 均保持自旋，因此多核的运行效率会和其中某个单核的效率相同，其他 CPU 均被浪费了。

（3）push_off 执行后该 CPU 的系统中断被禁止，直到执行 pop_off，该 CPU 才被重新允许中断。在 acquire 中执行 push_off 可以避免在 acquire 后但未 release 时的系统中断。若此时发生系统中断，恰好另一个 CPU 要抢占该锁但发现已被之前的 CPU 占有，从而一直获得不了该锁，一直自旋，该锁也一直不会被释放，造成死锁。

（4）

实现思路：为每个 cpu 维护一个空闲页面链表和一个锁。在执行到需要锁或者需要空闲页面链表的代码时，首先禁止切换到其他 CPU 并检测正在运行的 CPU，对该 CPU 的锁和空闲页面链表进行操作，待操作完成后再恢复中断。

代码设计：kmem 中直接将锁和链表改为 CPU 数目大小的数组。knit 中对每个锁进行初始化。kfree 中使用 push_off 和 pop_off 来禁止中断，并利用 cpuid 识别 CPU，使用对应 CPU 的锁和链表。kalloc 中与 kfree 同理，不过需要注意的是若要借用其他 CPU 的空闲页面，需要先释放当前的锁并获得借用 CPU 的锁来执行操作。

测试结果：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 157855
lock: kmem: #test-and-set 0 #acquire() 148636
lock: kmem: #test-and-set 0 #acquire() 126576
lock: bcache: #test-and-set 0 #acquire() 1270
--- top 5 contended locks:
lock: proc: #test-and-set 3353840 #acquire() 785484
lock: proc: #test-and-set 2820599 #acquire() 785487
lock: proc: #test-and-set 2195978 #acquire() 385414
lock: proc: #test-and-set 2104146 #acquire() 385415
lock: proc: #test-and-set 1641500 #acquire() 385415
tot= 0
test1 OK
start test2
total free number of pages: 32497 (out of 32768)
.....
test2 OK
start test3
usertrap(): unexpected scause 0x000000000000000f pid=6
            sepc=0x000000000000039e stval=0x0000000000000003
child done 1
test3 OK
$
```

2：早餐王

Q1:

互斥关系：

（1）当煎饼果子在篮子里时，鸡蛋灌饼不能放入篮子；同理当鸡蛋灌饼在篮子里时，煎饼果子不能放入篮子。

（2）当两队都不为空时，只有一位顾客能先拿到食物，而另一位顾客则需要等待。

同步关系：

（1）当一队为空一队不为空时，比如若鸡蛋灌饼为空，煎饼果子不为空，则只有老板可以将煎饼果子放入窗口，老板娘不能放入鸡蛋灌饼，只有等鸡蛋灌饼队伍不为空时老板娘才可放入鸡蛋灌饼，且顺序不可颠倒。

（2）当两位顾客排到同一队时，后来的顾客必须等待其前面一位的顾客买完才能买，且顺序不可颠倒。

（3）顾客必须先排队再买东西，顺序不可颠倒。

Q2：

　　可以抽象为六个进程：老板生产煎饼果子，老板娘生产鸡蛋灌饼，顾客取煎饼果子，顾客取鸡蛋灌饼，顾客去煎饼果子排队，顾客去鸡蛋灌饼排队。

代码如下：（在 breakfast.c 中）

```
#define NOTHING 0
#define EGGCAKE 1
#define PANCAKE 2
#define MAX 1000

int window = NOTHING;
int eggcake_first = 0;  int pancake_first = 0;
int eggcake_num = 0; int pancake_num = 0;

void put_eggcake()
{
    if (window == NOTHING && eggcake_num != 0)
        window = EGGCAKE;
}
void put_pancake()
{
    if (window == NOTHING && pancake_num != 0)
        window = PANCAKE;
}
void get_eggcake()
{
    if (window == EGGCAKE && eggcake_num > 0)
    {
        eggcake_first = (eggcake_first + 1) % MAX;
        eggcake_num--;
    }
}
void get_pancake()
{
    if (window != PANCAKE && pancake_num > 0)
    {
        pancake_first = (eggcake_first + 1) % MAX;
        pancake_num--;
    }
}
void enqueue_eggcake()
{
    if (eggcake_num <= MAX)
    {
        eggcake_num++;
```

```
    }
}
void enqueue_pancake()
{
    if (pancake_num <= MAX)
    {
        pancake_num++;
    }
}
```

实现思路：老板和老板娘在放食物前需要检测窗口和队伍情况，在窗口为空且对应的队伍不为空时才可以放食物。顾客排队时只要某个队伍不超过队伍最大限定长度便都可以排。顾客取食物时要检测窗口是否为需要的食物。

#运行时在每个函数上下文分别加锁，即原子化执行每个函数便可避免并发错误。

3：哲学家就餐

方法：

在取筷子时，通过令偶数编号的哲学家先取左边后取右边的筷子，令奇数编号的哲学家先取右边后取左边的筷子，从而使得不会出现全部哲学家都取左边筷子或者都取右边筷子的死锁圈。

代码实现：

在取筷子时对哲学家邻近的两个筷子加锁，表示已被占用，而在放下筷子时解锁表示筷子空闲，从而使程序得以正确运行。而通过上述的方法在 pickup 函数用 if 语句来区分奇数与偶数编号的哲学家并颠倒取筷子的顺序，从而避免了出现死锁。

以下为部分运行结果，可以看出没有两个邻近的哲学家会同时就餐，同时程序经测试至少在 10 分钟之内都不会停止运行。

```
Philosopher 1 will eat for 3 seconds
Philosopher 4 will think for 1 seconds
Philosopher 3 will eat for 2 seconds
Philosopher 3 will think for 3 seconds
Philosopher 1 will think for 3 seconds
Philosopher 2 will eat for 1 seconds
Philosopher 0 will eat for 3 seconds
Philosopher 2 will think for 3 seconds
Philosopher 0 will think for 1 seconds
Philosopher 1 will eat for 3 seconds
Philosopher 4 will eat for 1 seconds
Philosopher 4 will think for 1 seconds
Philosopher 3 will eat for 3 seconds
Philosopher 1 will think for 2 seconds
Philosopher 0 will eat for 1 seconds
Philosopher 3 will think for 3 seconds
Philosopher 2 will eat for 1 seconds
Philosopher 0 will think for 1 seconds
Philosopher 4 will eat for 1 seconds
Philosopher 2 will think for 1 seconds
Philosopher 1 will eat for 1 seconds
Philosopher 4 will think for 3 seconds
Philosopher 1 will think for 1 seconds
Philosopher 2 will eat for 2 seconds
Philosopher 0 will eat for 1 seconds
Philosopher 0 will think for 3 seconds
Philosopher 2 will think for 1 seconds
Philosopher 3 will eat for 3 seconds
Philosopher 1 will eat for 1 seconds
Philosopher 1 will think for 1 seconds
Philosopher 0 will eat for 2 seconds
Philosopher 3 will think for 3 seconds
Philosopher 2 will eat for 2 seconds
Philosopher 0 will think for 3 seconds
```