Lab4报告_*16281264*曲健聪

内存管理程序的核心思路:

首先明确使用的数据结构，通常来说应该用双向链表来保存空闲内存，但由于我们的实验规模小，没有对性能的要求，我们简单地用两个数组分别保存空闲内存表和已分配内存表。下面的表格是一个例子

空闲内存表

index	初始地址	大小
0	0x00	300B
1	0x900	124B

已分配内存

进程名	初始地址	大小
P0	0x300	600B

查找可用块，分割，分配

当一个进程申请若干内存，我们需要首先在内存表查找到一块合适的内存块，将其分割出适合的大小分配给进程，并将剩余内存加入空闲内存表。但如果第一次查找没有找到合适的内存块，还不代表系统中剩余内存不足，可能是碎片太多，应当进行一次紧缩后再次试图分配，这次分配失败了，才认为分配失败。

```
int allocateByCommand(string processName, int needSize, char waytofit, bool tryagain)
{
    //usr waytofit to choose a function to get a block
    //W:worst fit, B:best fit, F:firstfit

    if(fail)
    {
        if(tryagain)
        {
            Compress();
            allocateByCommand(processName, needSize,waytofit, false);
        }
        else
        {
            cout<<"allocate fail"<<endl;
        }
    }
}
```

紧缩

因我们的程序并不需要真的分配内存给进程运行，也不需要处理进程的重定位问题，我们可以简单的遍历已分配内存表，修改其初始地址，使其成为一片从0开始的连续内存区域,并把空闲内存表中清空，置入一个新的，始地址为最后一个进程末尾地址，大小为(全部内存-进程占用内存量)的大内存块。

回收

我们已经讨论了分配，紧缩的情况，现在该讨论回收问题。

假如现在进程P某声明工作结束，可以立刻释放内存，这时我们要到表中查找到它应插入的位置。这一点只需要通过比较初始地址项就可以找到。但之后我们要考虑三种情况。

1. 恰在这块内存前的一个块是空闲的(prev.start + prev.length == this.start)，那么只需修改前一个空闲块的大小(prev.length += this.length)，将这个块归并到前一块即可。
2. 恰在这块内存前的一个块不是空闲的(prev.start + prev.length != this.start)，但恰在这个内存块后的块是空闲的(this.start + this.length != next.start)，那么就可以直接修改下一个块的始地址和大小(next.start=this.start;next.size+=this.size)。
3. 这个空闲内存前后两个块都不是恰好和要回收块相邻的，那么就得插入一个新项。此举主要的坏处是带来内存碎片。

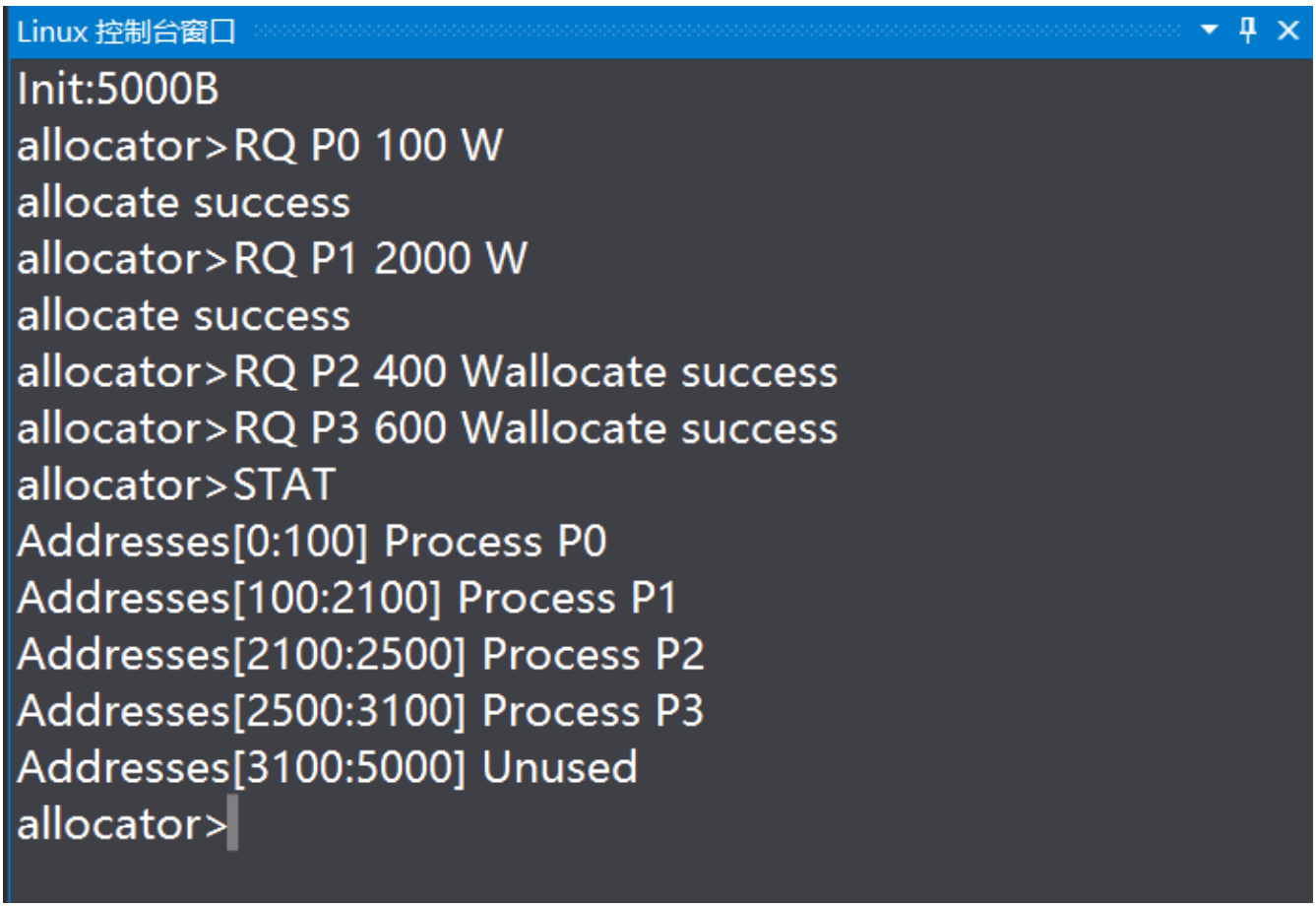
注意这个做法有个简单的缺陷，如果两个空闲块中间有一个碎片被回收了，那么这个碎片只回收到前面的块上，而不能把三个块合并，但假如要考虑这样的情况，就不得不再考虑一层层向上递归回收的问题，不如当内存不足的时候直接进行紧缩。

此外，要注意用户输入可能有误，要求回收一个不存在的进程，这点需要做特殊处理，否则可能出现越界错误。

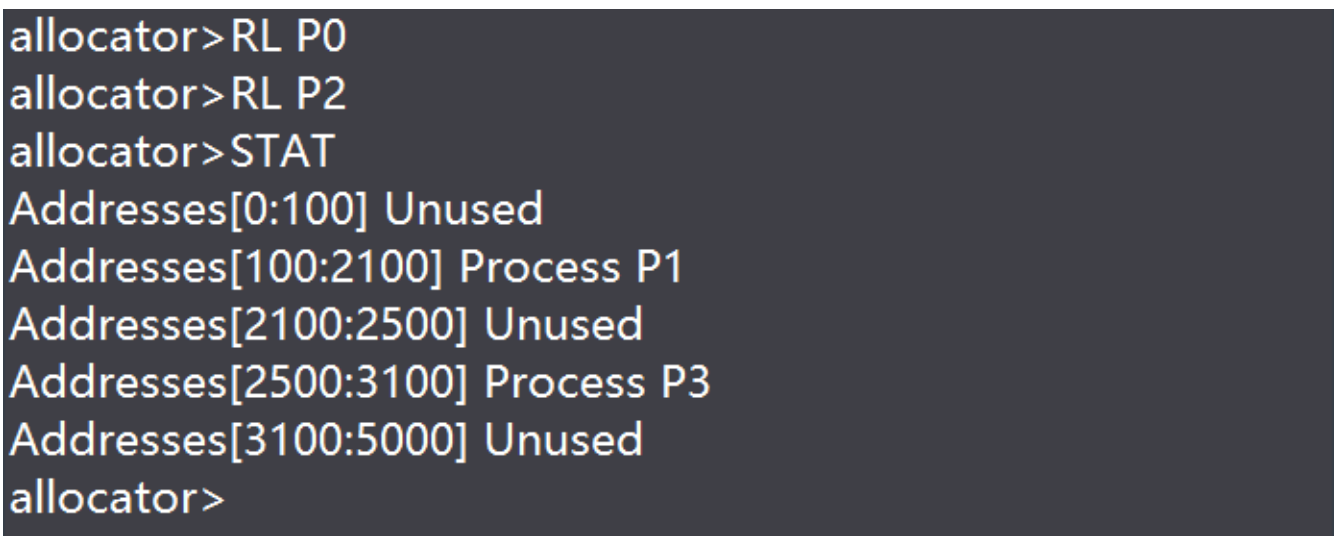
测试:

我们需要构建一个足够复杂的测试例，能充足说明各分配算法工作正确，回收算法，紧缩算法正确，在遇到内存不足时，可以正确紧缩，尝试重新分配。

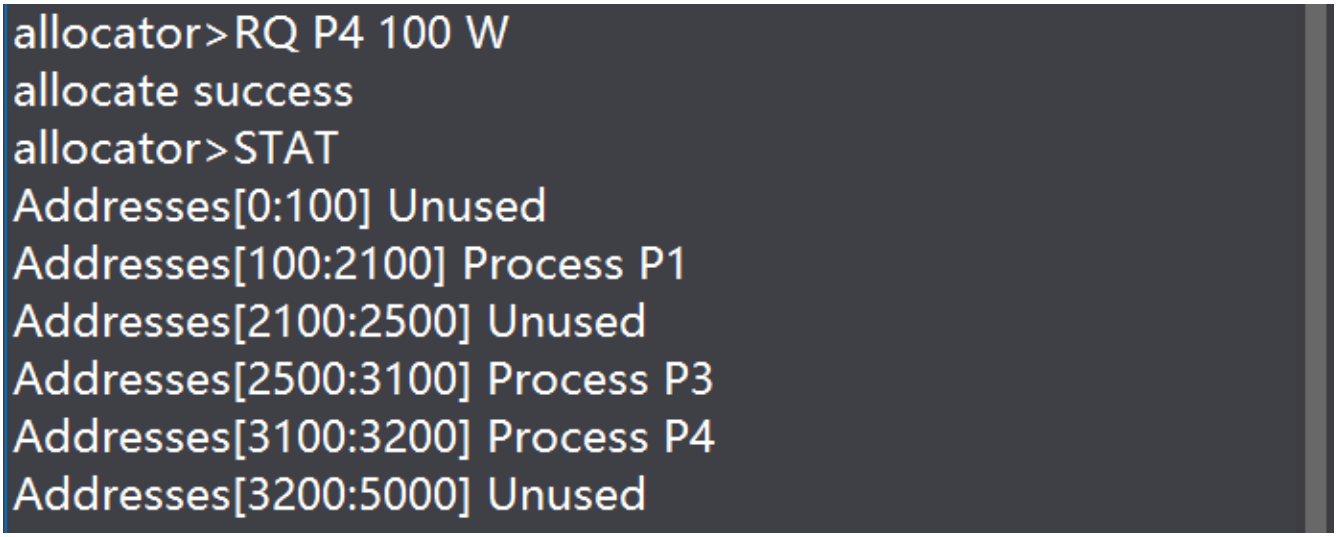
```
./allocator 5000
allocator>RQ P0 100 W
allocator>RQ P1 2000 W
allocator>RQ P2 400 W
allocator>RQ P3 600 W
allocator>STAT
#由于开始时只有一个内存块，无论要求使用何种算法分配，P0-3都会被分配连续的0-3100这段内存
#预计输出:
#0-100 P0
#100-2100 P1
#2100-2500 P2
#2500-3100 P3
#3100-5000 P4
```



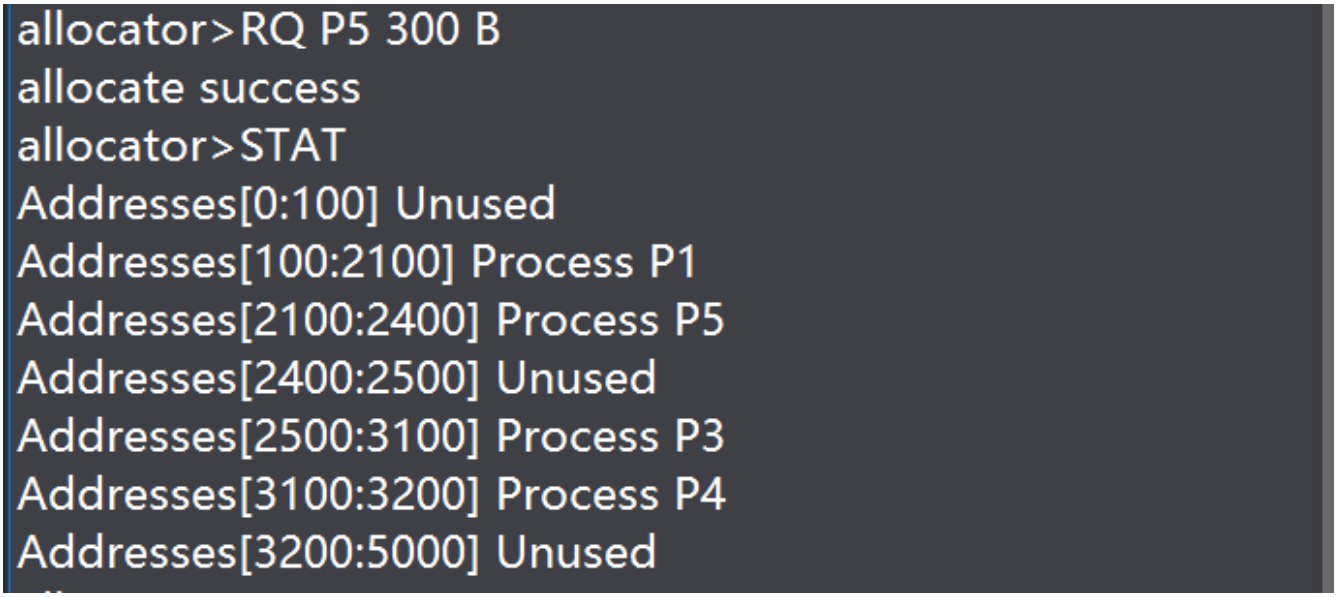

```
allocator>RL P0
allocator>RL P2
allocator>STAT
#预计输出:
#0-100 U
#100-2100 P1
#2100-2500 U
#2500-3100 P3
#3100-5000 U
```



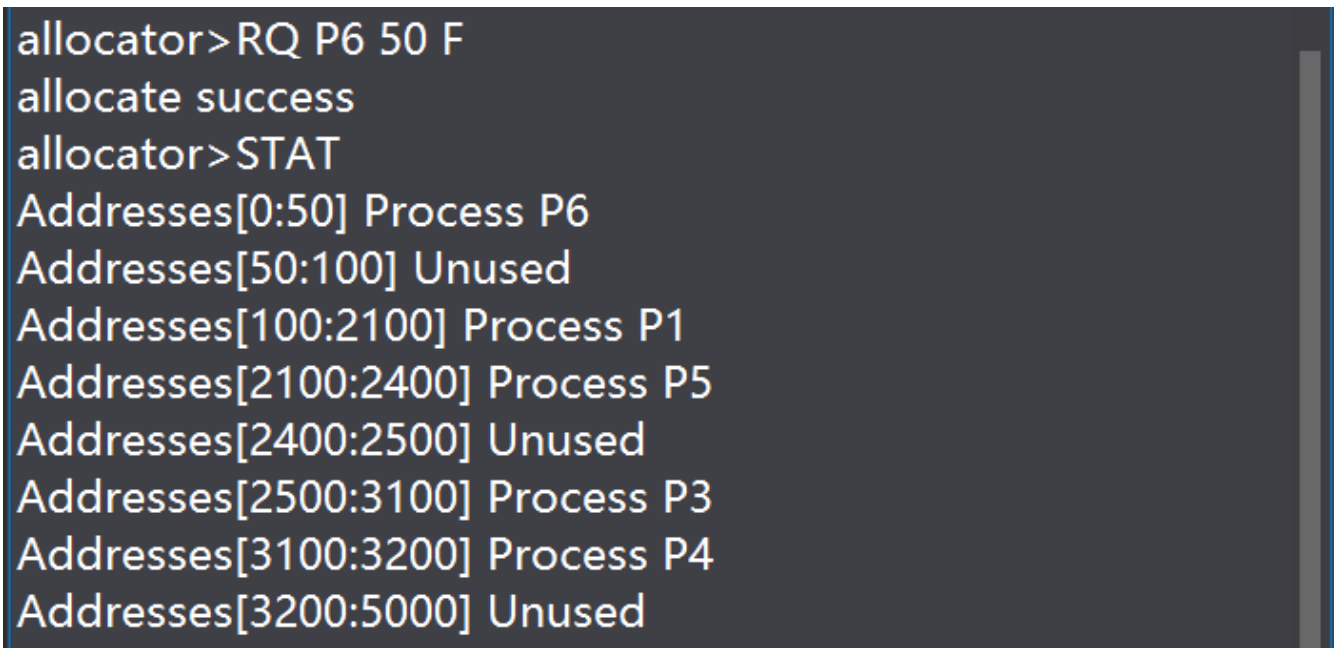
```
#worst fit
allocator>RQ P4 100 W
allocator>STAT
#预计输出:
#0-100 U
#100-2100 P1
#2100-2500 U
#2500-3100 P3
#3100-3200 P4
#3200-5000 U
```



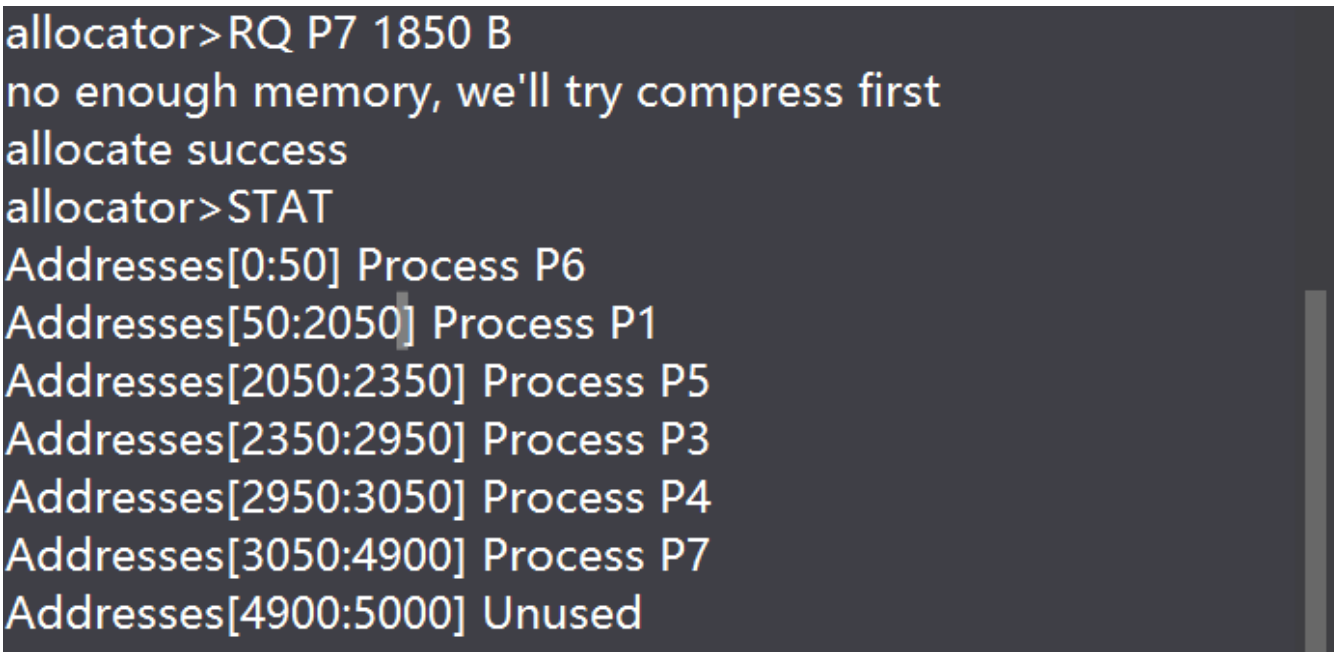

```
#best fit
allocator>RQ P5 300 B
allocator>STAT
#预计输出:
#0-100 U
#100-2100 P1
#2100-2400 P5
#2400-2500 U
#2500-3100 P3
#3100-3200 P4
#3200-5000 U
```



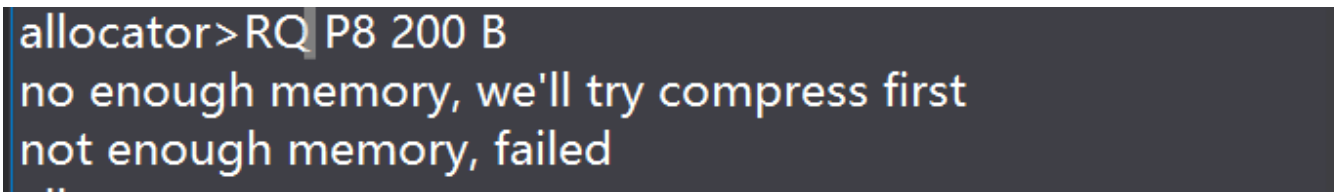

```
#first fit
allocator>RQ P6 50 F
allocator>STAT
#预计输出:
#0-50 P6
#50-100 U
#100-2100 P1
#2100-2400 P5
#2400-2500 U
#2500-3100 P3
#3100-3200 P4
#3200-5000 U
```



```
#申请一个大于最大碎片，但小于剩余总内存的块(最大块1800，总剩余1950)
allocator>RQ P7 1850 B
#预计进行紧缩后分配成功
#预计输出:
#0-100 P4
#100-2100 P1
#2100-2400 P5
#2400-3000 P3
#3000-3050 P6
#3050-4950 P7
#4950-5000 U
```




```
#申请一个不可能分配出的内存
allocator>RQ P8 200 B
#预计输出失败信息
```



实验输出与预测结果一致，说明实验代码工作正确。