



درس سیستم‌های عامل
نیم‌سال اول ۰۲-۰۳
استاد: دکتر حسین اسدی

پاسخ تمرین عملی سری هشتم

امیرحسین رحمتی

```
$ kallocat
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #test-and-set 16601 #acquire() 433108
lock: bcache: #test-and-set 0 #acquire() 2132
--- top 5 contended locks:
lock: virtio_disk: #test-and-set 51324 #acquire() 156
lock: proc: #test-and-set 49702 #acquire() 574145
lock: proc: #test-and-set 29638 #acquire() 974029
lock: proc: #test-and-set 27956 #acquire() 974037
lock: proc: #test-and-set 24513 #acquire() 573951
tot= 16601
test1 FAIL
start test2
total free number of pages: 32497 (out of 32768)
```

نتیجه فراخوانی kallocat را در ابتدا بر روی سیستم بنده مشاهده می‌کنید.

حال به سراغ پیاده سازی خواسته سوال می‌رویم.

```
20
21 struct {
22     struct spinlock lock;
23     struct run *freelist;
24 } kmem[NCPU];
25
26 void
27 kinit()
28 {
29     for (int i=0; i<NCPU; i++) {
30         char lock_name[5];
31         snprintf(lock_name, 5, "kmem%d", i);
32         initlock(&kmem[i].lock, lock_name);
33     }
34     // it will allocate all free pages to the first cpu,
35     // so it will be so probable that other cpu's have to steal from him.
36     freerange(pa_start, pa_end: (void*)PHYSTOP);
37 }
38
```

همانطور که در تصویر بالا قابل مشاهده است، در ابتدا برای هر cpu یک lock و یک freelist جدا در نظر می‌گیریم. و در تابع kinit لاک‌های مربوط به این لیست‌ها را ایجاد می‌کنیم. همچنین با توجه به اینکه تابع kinit در main.c تنها توسط cpu شماره صفر فراخوانی می‌شود. طبق راهنمایی سوال، همه فضای فیزیکی را ابتدا به freelist مربوط به cpu شماره صفر تخصیص می‌دهیم و در ادامه آن را به باقی cpu‌ها می‌دهیم.

```
64
65     push_off();
66     int cpu_id = cpuid();
67     pop_off();
68
69     acquire(&kmem[cpu_id].lock);
70     r->next = kmem[cpu_id].freelist;
71     kmem[cpu_id].freelist = r;
72     release(&kmem[cpu_id].lock);
73 }
```

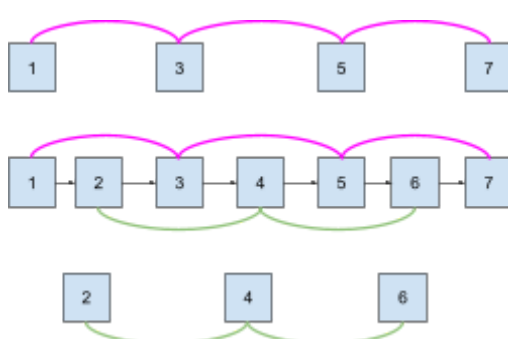
در قسمت بعدی تابع kfree را دستخوش تغییر می‌کنیم. تنها تفاوتی که پیاده‌سازی این تابع نسبت به حالت قبل دارد این است که در صورت خالی شدن یک آدرس فیزیکی آن را به free_list همان cpu اختصاص می‌دهیم و کاری به freelist دیگر cpu‌ها نداریم که این باعث کاهش lock contention می‌شود.

```
83     push_off();
84     int cpu_id = cpuid();
85     pop_off();
86
87     acquire(&kmem[cpu_id].lock);
88
89     if(!kmem[cpu_id].freelist){
90         for (int i=0;i<NCPU;i++){
91             if (i == cpu_id) continue;
92             acquire(&kmem[i].lock);
93             if(kmem[i].freelist && kmem[i].freelist->next){
94                 kmem[cpu_id].freelist = kmem[i].freelist->next;
95                 // split the freelist into 2 equal lists
96                 splitLinkedList(i);
97                 release(&kmem[i].lock);
98                 break;
99             }
100             release(&kmem[i].lock);
101         }
102     };
103 }
104
105 r = kmem[cpu_id].freelist;
106 if(r)
107     kmem[cpu_id].freelist = r->next;
108 release(&kmem[cpu_id].lock);
109 }
```

در نهایت به پیاده سازی تابع kalloc می‌رسیم. در این تابع می‌خواهیم از فضا های آزاد هر cpu به پردازش ها فضایی را اختصاص دهیم. اگر freelist مربوط به آن CPU به یک فضای خالی اشاره کند پیاده سازی کاملاً مشابه قبل است. اما در صورتی که freelist ما به چیزی اشاره نکند یعنی این cpu فضای خالی برای تخصیص ندارد و باید از cpu دیگری بگیرد. (همانطور که قبل تر اشاره شد. این اتفاق می‌افتد زیرا که تمام فضای خالی در ابتدا در دست cpu شماره صفر است.)

برای این کار ما سعی می‌کنیم تا با گرفتن lock مربوط به freelist دیگر cpu ها سراغ آنها برویم. برای اینکه این امر هی تکرار نشود، در هر بار تلاش برای گرفتن فضای خالی از cpu دیگر ما تعداد فضا های خالی در دسترس آن cpu را نصف می‌کنیم و نصف آن را برای خودمان بر می‌داریم.

```
115 void splitLinkedList(int i)
116 {
117     struct run *my_cpu_linker = kmem[i].freelist->next;
118     struct run *other_cpu_linker = kmem[i].freelist;
119     for (;;)
120     {
121         if (other_cpu_linker)
122         {
123             if (other_cpu_linker->next)
124             {
125                 other_cpu_linker->next = other_cpu_linker->next->next;
126             }
127             other_cpu_linker = other_cpu_linker->next;
128         }
129         if (my_cpu_linker)
130         {
131             if (my_cpu_linker->next)
132             {
133                 my_cpu_linker->next = my_cpu_linker->next->next;
134             }
135             my_cpu_linker = my_cpu_linker->next;
136         }
137         if ((!my_cpu_linker) && (!other_cpu_linker))
138         {
139             break;
140         }
141     }
142 }
```



الگوریتم مربوط برای نصف کردن linklist مربوط به فضای خالی را در تصویر بالا مشاهده می‌کنید. عملکرد این الگوریتم به این شکل است که یکی در میان خانه های linklist را برایش خودش ور می‌دارد و آن خانه هایی که رها کرده است را به هم متصل می‌کند تا یک linklist جدید تشکیل دهند.

نحوه عملکرد کد جدید در تست هارا می‌توانید در تصاویر زیر مشاهده کنید.

```
$ kallocetest
start test1
test1 results:
--- lock kmem/bcache stats
lock: bcache: #test-and-set 0 #acquire() 1694 (my_cpu_linker)
--- top 5 contended locks:
lock: proc: #test-and-set 12699040 #acquire() 24095523
lock: proc: #test-and-set 447133 #acquire() 26157182
lock: proc: #test-and-set 327691 #acquire() 23035560
lock: proc: #test-and-set 172578 #acquire() 23035560
lock: proc: #test-and-set 126430 #acquire() 23035560
tot= 0
test1 OK
start test2
total free number of pages: 32495 (out of 32768)
...
test2 OK
start test3
child done 1
child done 100000
test3 OK
```

همانطور که انتظار داشتیم. زمان انتظار برای یک lock کاهش پیدا کرده است.

```
$ userstests sbrkmuch
userstests starting
test sbrkmuch: OK
ALL TESTS PASSED
$
```

```
(94.5s)
== Test kallocetest: test1 ==
kallocetest: test1: OK
== Test kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: test3 ==
kallocetest: test3: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (11.5s)
```

```
== Test userstests ==
$ make qemu-gdb
userstests: OK (71.6s)
== Test time ==
```