به نام خدا

دانشکده مهندسی کامپیوتر

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

پاسخ تمرین عملی سری هشتم

امیرحسین رحمتی

```
$ kalloctest
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 kalloctest را در ابتدا بر روی سیستم بنده مشاهده می‌کنید.

حال به سراغ پیاده سازی خواسته سوال می‌رویم.

```
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: end,  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 لاک های مربوط به این لیست ها 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 cpu ها می‌دهیم.

```
  push_off();
  int cpu_id = cpuid();
  pop_off();

  acquire(&kmem[cpu_id].lock);
  r->next = kmem[cpu_id].freelist;
  kmem[cpu_id].freelist = r;
  release(&kmem[cpu_id].lock);
}
```

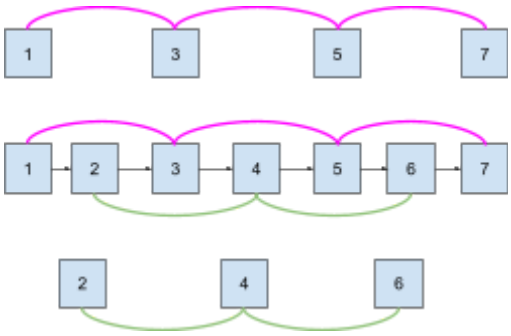
در قسمت بعدی تابع kfree را دستخوش تغییر می‌کنیم. تنها تفاوتی که پیاده سازی این تابع نسبت به حالت قبل دارد این است که در صورت خالی شدن یک آدرس فیزیکی آن را به free_list همان cpu اختصاص می‌دهیم و کاری به freelist دیگر cpu ها نداریم که این باعث کاهش lock contention می‌شود.

```
push_off();
int cpu_id = cpuid();
pop_off();

acquire(&kmem[cpu_id].lock);

if(!kmem[cpu_id].freelist){
  for (int i=0;i<NCPU;i++) {
    if (i == cpu_id) continue;
    acquire(&kmem[i].lock);
    if(kmem[i].freelist && kmem[i].freelist->next){
      kmem[cpu_id].freelist = kmem[i].freelist->next;
      // split the freelist into 2 equal lists
      splitLinkedList(i);
      release(&kmem[i].lock);
      break;
    }
    release(&kmem[i].lock);
  }
  ;
}

r = kmem[cpu_id].freelist;
if(r)
  kmem[cpu_id].freelist = r->next;
release(&kmem[cpu_id].lock);
```

در نهایت به پیاده سازی تابع kalloc می‌رسیم. در این تابع می‌خواهیم از فضا های آزاد هر cpu به پردازه ها فضایی را اختصاص دهیم. اگر freelist مربوط به آن CPU به یک فضای خالی اشاره کند پیاده سازی کاملا 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.)

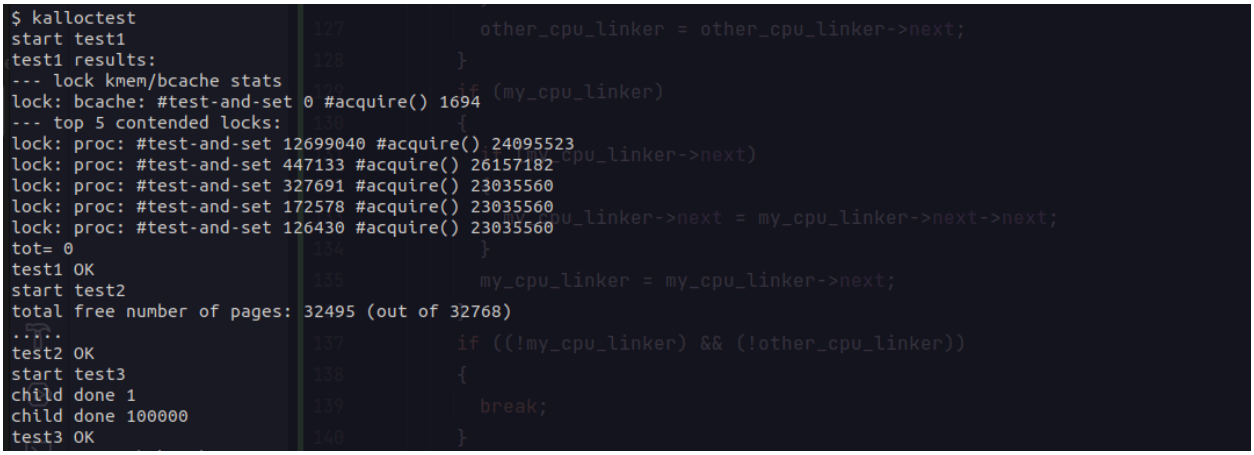
برای این کار ما سعی می‌کنیم تا با گرفتن lock مربوط به freelist دیگر cpu ها سراغ آنها برویم. برای اینکه این امر هی تکرار نشود، در هر بار تلاش برای گرفتن فضای خالی از cpu دیگر ما تعداد فضا های خالی در دسترس آن cpu را نصف می‌کنیم و نصف آن را برای خودمان بر می‌داریم.

```
void splitLinkedList(int i)
{
  struct run *my_cpu_linker = kmem[i].freelist->next;
  struct run *other_cpu_linker = kmem[i].freelist;
  for (;;)
  {
    if (other_cpu_linker)
    {
      if (other_cpu_linker->next)
      {
        other_cpu_linker->next = other_cpu_linker->next->next;
      }
      other_cpu_linker = other_cpu_linker->next;
    }
    if (my_cpu_linker)
    {
      if (my_cpu_linker->next)
      {
        my_cpu_linker->next = my_cpu_linker->next->next;
      }
      my_cpu_linker = my_cpu_linker->next;
    }
    if ((!my_cpu_linker) && (!other_cpu_linker))
    {
      break;
    }
  }
}
```



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.



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