



下载APP

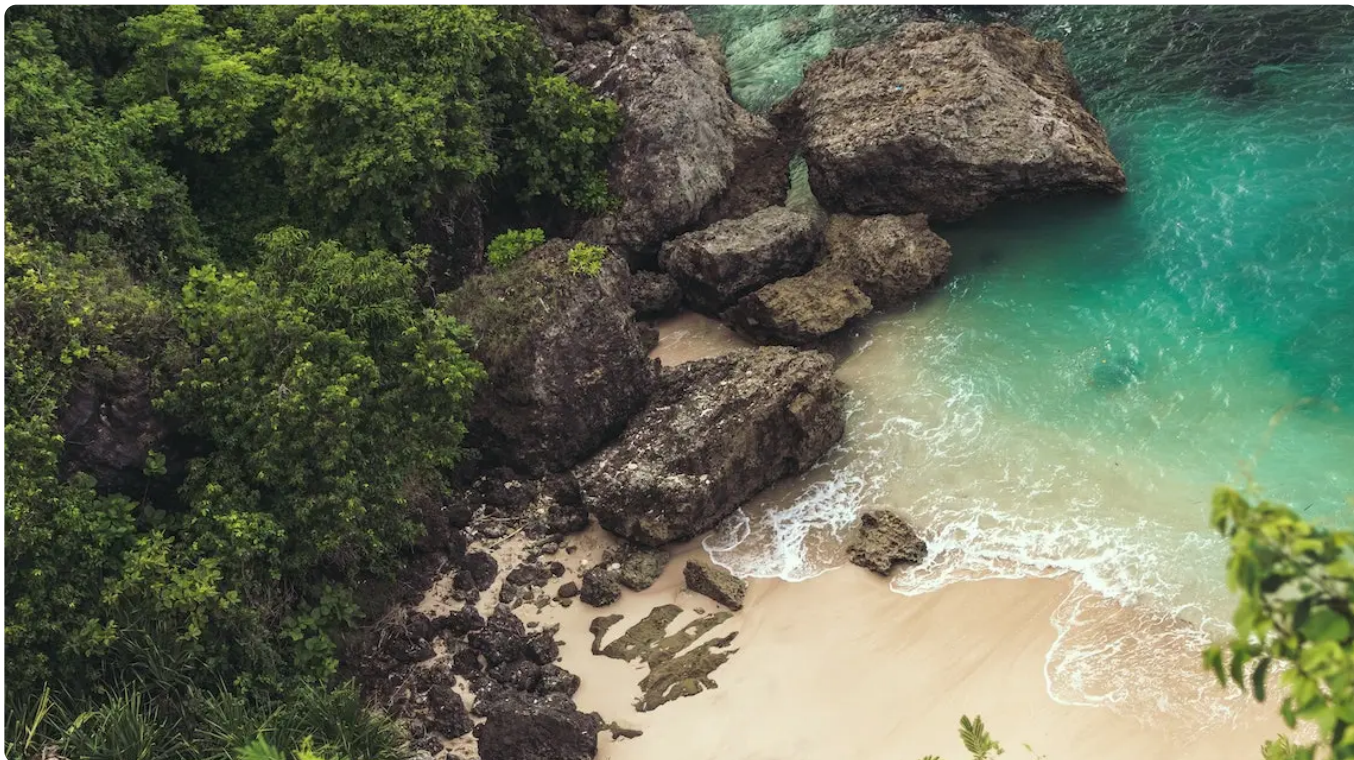


17 | 划分土地（中）：如何实现内存页面初始化？

2021-06-16 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 14:18 大小 13.11M



你好，我是 LMOS。

上节课，我们确定了用分页方式管理内存，并且一起动手设计了表示内存页、内存区相关的内存管理数据结构。不过，虽然内存管理相关的数据结构已经定义好了，但是我们还没有在内存中建立对应的**实例变量**。

我们都知道，在代码中实际操作的数据结构必须在内存中有相应的变量，这节课我们就去建立对应的实例变量，并初始化它们。



初始化

前面的课里，我们在 hal 层初始化中，初始化了从二级引导器中获取的内存布局信息，也就是那个 **e820map_t 数组**，并把这个数组转换成了 phymmerge_t 结构数组，还对它做了排序。

但是，我们 Cosmos 物理内存管理器剩下的部分还没有完成初始化，下面我们就去实现它。

Cosmos 的物理内存管理器，我们依然要放在 Cosmos 的 hal 层。

因为物理内存还和硬件平台相关，所以我们要在 cosmos/hal/x86/ 目录下建立一个 memmgrinit.c 文件，在这个文件中写入一个 Cosmos 物理内存管理器初始化的大总管——init_memmgr 函数，并在 init_halmm 函数中调用它，代码如下所示。

[复制代码](#)

```
1 //cosmos/hal/x86/halmm.c中
2 //hal层的内存初始化函数
3 void init_halmm()
4 {
5     init_phymmerge();
6     init_memmgr();
7     return;
8 }
9 //Cosmos物理内存管理器初始化
10 void init_memmgr()
11 {
12     //初始化内存页结构msadsc_t
13     //初始化内存区结构memarea_t
14     return;
15 }
```

根据前面我们对内存管理相关数据结构的设计，你应该不难想到，在 init_memmgr 函数中应该要完成**内存页结构 msadsc_t 和内存区结构 memarea_t 的初始化**，下面就分别搞定这两件事。

内存页结构初始化

内存页结构的初始化，其实就是初始化 msadsc_t 结构对应的变量。因为一个 msadsc_t 结构体变量代表一个物理内存页，而物理内存由多个页组成，所以最终会形成一个 msadsc_t 结构体数组。

这会让我们的工作变得简单，我们只需要找一个内存地址，作为 msadsc_t 结构体数组的开始地址，当然这个内存地址必须是可用的，而且之后内存空间足以存放 msadsc_t 结构体数组。

然后，我们要扫描 phymmerge_t 结构体数组中的信息，只要它的类型是可用内存，就建立一个 msadsc_t 结构体，并把其中的开始地址作为第一个页面地址。

接着，要给这个开始地址加上 0x1000，如此循环，直到其结束地址。

当这个 phymmerge_t 结构体的地址区间，它对应的所有 msadsc_t 结构体都建立完成之后，就开始下一个 phymmerge_t 结构体。依次类推，最后，我们就能建好所有可用物理内存页面对应的 msadsc_t 结构体。

下面，我们去 cosmos/hal/x86/ 目录下建立一个 msadsc.c 文件。在这里写下完成这些功能的代码，如下所示。

[复制代码](#)

```
1 void write_one_msadsc(msadsc_t *msap, u64_t phyadr)
2 {
3     //对msadsc_t结构做基本的初始化，比如链表、锁、标志位
4     msadsc_t_init(msap);
5     //这是把一个64位的变量地址转换成phyadrflgs_t*类型方便取得其中的地址位段
6     phyadrflgs_t *tmp = (phyadrflgs_t *)(&phyadr);
7     //把页的物理地址写入到msadsc_t结构中
8     msap->md_phyadrs.paf_padrs = tmp->paf_padrs;
9     return;
10 }
11
12 u64_t init_msadsc_core(machbstart_t *mbasp, msadsc_t *msavstart, u64_t msanr)
13 {
14     //获取phymmerge_t结构数组开始地址
15     phymmerge_t *pimagep = (phymmerge_t *)phyadr_to_viradr((adr_t)mbasp->mb_e820
16     u64_t mdindx = 0;
17     //扫描phymmerge_t结构数组
18     for (u64_t i = 0; i < mbasp->mb_e820exnr; i++)
19     {
20         //判断phymmerge_t结构的类型是不是可用内存
21         if (PMR_T_OSAPUSERAM == pimagep[i].pmr_type)
22         {
23             //遍历phymmerge_t结构的地址区间
24             for (u64_t start = pimagep[i].pmr_saddr; start < pimagep[i].pmr_end;
25             {
26                 //每次加上4KB-1比较是否小于等于phymmerge_t结构的结束地址
```

```
27         if ((start + 4096 - 1) <= pimagep[i].pmr_end)
28         {
29             //与当前地址为参数写入第mdindx个msadsc结构
30             write_one_msadsc(&msavstart[mdindx], start);
31             mdindx++;
32         }
33     }
34 }
35 }
36 return mdindx;
37 }
38
39 void init_msadsc()
40 {
41     u64_t coremdnr = 0, msadscnr = 0;
42     msadsc_t *msadscvp = NULL;
43     machbstart_t *mbbsp = &kmachbsp;
44     //计算msadsc_t结构数组的开始地址和数组元素个数
45     if (ret_msadsc_vadrandsz(mbsp, &msadscvp, &msadscnr) == FALSE)
46     {
47         system_error("init_msadsc ret_msadsc_vadrandsz err\n");
48     }
49     //开始真正初始化msadsc_t结构数组
50     coremdnr = init_msadsc_core(mbsp, msadscvp, msadscnr);
51     if (coremdnr != msadscnr)
52     {
53         system_error("init_msadsc init_msadsc_core err\n");
54     }
55     //将msadsc_t结构数组的开始的物理地址写入kmachbsp结构中
56     mbsp->mb_memmappadr = viradr_to_phyadr((adr_t)msadscvp);
57     //将msadsc_t结构数组的元素个数写入kmachbsp结构中
58     mbsp->mb_memmapnr = coremdnr;
59     //将msadsc_t结构数组的大小写入kmachbsp结构中
60     mbsp->mb_memmapsz = coremdnr * sizeof(msadsc_t);
61     //计算下一个空闲内存的开始地址
62     mbsp->mb_nextwtpadr = PAGE_ALIGN(mbsp->mb_memmappadr + mbsp->mb_memmapsz);
63     return;
64 }
```

上面的代码量很少，逻辑也很简单，再配合注释，相信你看得懂。其中的 `ret_msadsc_vadrandsz` 函数也是遍历 `phymmarge_t` 结构数组，计算出有多大的可用内存空间，可以分成多少个页面，需要多少个 `msadsc_t` 结构。

内存区结构初始化

前面我们将整个物理地址空间在逻辑上分成了三个区，分别是：**硬件区、内核区、用户区**，这就要求我们要在内存中建立三个 `memarea_t` 结构体的实例变量。

就像建立 `msadsc_t` 结构数组一样，我们只需要在内存中找个空闲空间，存放这三个 `memarea_t` 结构体就行。相比建立 `msadsc_t` 结构数组这更为简单，因为 `memarea_t` 结构体是顶层结构，并不依赖其它数据结构，只是对其本身进行初始化就好了。

但是由于它自身包含了其它数据结构，在初始化它时，要对其中的其它数据结构进行初始化，所以要小心一些。

下面我们去 `cosmos/hal/x86/` 目录下建立一个 `memarea.c` 文件，写下完成这些功能的代码，如下所示。

 复制代码

```
1 void bafhlst_t_init(bafhlst_t *initp, u32_t stus, uint_t oder, uint_t oderpnr)
2 {
3     //初始化bafhlst_t结构体的基本数据
4     knl_spinlock_init(&initp->af_lock);
5     initp->af_stus = stus;
6     initp->af_oder = oder;
7     initp->af_oderpnr = oderpnr;
8     initp->af_fobjnr = 0;
9     initp->af_mobjnr = 0;
10    initp->af_alcindx = 0;
11    initp->af_freindx = 0;
12    list_init(&initp->af_frelst);
13    list_init(&initp->af_alclst);
14    list_init(&initp->af_ovelst);
15    return;
16 }
17
18 void memdivmer_t_init(memdivmer_t *initp)
19 {
20     //初始化medivmer_t结构体的基本数据
21     knl_spinlock_init(&initp->dm_lock);
22     initp->dm_stus = 0;
23     initp->dm_divnr = 0;
24     initp->dm_mernr = 0;
25     //循环初始化memdivmer_t结构体中dm_mdmlielst数组中的每个bafhlst_t结构的基本数据
26     for (uint_t li = 0; li < MDIVMER_ARR_LMAX; li++)
27     {
28         bafhlst_t_init(&initp->dm_mdmlielst[li], BAFH_STUS_DIVM, li, (1UL << 1
29     }
30     bafhlst_t_init(&initp->dm_onemsalst, BAFH_STUS_ONEM, 0, 1UL);
31     return;
32 }
33
34 void memarea_t_init(memarea_t *initp)
35 {
```



```
36 //初始化memarea_t结构体的基本数据
37 list_init(&initp->ma_list);
38 knl_spinlock_init(&initp->ma_lock);
39 initp->ma_stus = 0;
40 initp->ma_flg = 0;
41 initp->ma_type = MA_TYPE_INIT;
42 initp->ma_maxpages = 0;
43 initp->ma_allopages = 0;
44 initp->ma_freepages = 0;
45 initp->ma_resvpages = 0;
46 initp->ma_horizline = 0;
47 initp->ma_logicstart = 0;
48 initp->ma_logicend = 0;
49 initp->ma_logicsz = 0;
50 //初始化memarea_t结构体中的memdivmer_t结构体
51 memdivmer_t_init(&initp->ma_mdmdata);
52 initp->ma_privp = NULL;
53 return;
54 }
55
56 bool_t init_memarea_core(machbstart_t *mbasp)
57 {
58     //获取memarea_t结构开始地址
59     u64_t phymarea = mbasp->mb_nextwtpadr;
60     //检查内存空间够不够放下MEMAREA_MAX个memarea_t结构实例变量
61     if (initchkadr_is_ok(mbasp, phymarea, (sizeof(memarea_t) * MEMAREA_MAX)) !=
62         {
63         return FALSE;
64     }
65     memarea_t *virmarea = (memarea_t *)phyadr_to_viradr((adr_t)phymarea);
66     for (uint_t mai = 0; mai < MEMAREA_MAX; mai++)
67     { //循环初始化每个memarea_t结构实例变量
68         memarea_t_init(&virmarea[mai]);
69     }
70     //设置硬件区的类型和空间大小
71     virmarea[0].ma_type = MA_TYPE_HWAD;
72     virmarea[0].ma_logicstart = MA_HWAD_LSTART;
73     virmarea[0].ma_logicend = MA_HWAD_LEND;
74     virmarea[0].ma_logicsz = MA_HWAD_LSZ;
75     //设置内核区的类型和空间大小
76     virmarea[1].ma_type = MA_TYPE_KRNL;
77     virmarea[1].ma_logicstart = MA_KRNL_LSTART;
78     virmarea[1].ma_logicend = MA_KRNL_LEND;
79     virmarea[1].ma_logicsz = MA_KRNL_LSZ;
80     //设置应用区的类型和空间大小
81     virmarea[2].ma_type = MA_TYPE_PROC;
82     virmarea[2].ma_logicstart = MA_PROC_LSTART;
83     virmarea[2].ma_logicend = MA_PROC_LEND;
84     virmarea[2].ma_logicsz = MA_PROC_LSZ;
85     //将memarea_t结构的开始的物理地址写入kmachbsp结构中
86     mbasp->mb_memznpadr = phymarea;
87     //将memarea_t结构的个数写入kmachbsp结构中
```

```
88     mbsp->mb_memznnr = MEMAREA_MAX;
89     //将所有memarea_t结构的大小写入kmachbsp结构中
90     mbsp->mb_memznsz = sizeof(memarea_t) * MEMAREA_MAX;
91     //计算下一个空闲内存的开始地址
92     mbsp->mb_nextwtpadr = PAGE_ALIGN(phymarea + sizeof(memarea_t) * MEMAREA_MA
93     return TRUE;
94 }
95 //初始化内存区
96 void init_memarea()
97 {
98     //真正初始化内存区
99     if (init_memarea_core(&kmachbsp) == FALSE)
100     {
101         system_error("init_memarea_core fail");
102     }
103     return;
104 }
```

由于这些数据结构很大，所以代码有点长，但是重要的代码我都做了详细注释。

在 `init_memarea_core` 函数的开始，我们调用了 `memarea_t_init` 函数，对 `MEMAREA_MAX` 个 `memarea_t` 结构进行了基本的初始化。

然后，在 `memarea_t_init` 函数中又调用了 `memdivmer_t_init` 函数，而在 `memdivmer_t_init` 函数中又调用了 `bafhlst_t_init` 函数，这保证了那些被包含的数据结构得到了初始化。

最后，我们给三个区分别设置了类型和地址空间。

处理初始内存占用问题

我们初始化了内存页和内存区对应的数据结构，已经可以组织好内存页面了。现在看似已经万事俱备了，其实这有个重大的问题，你知道是什么吗？我给你分析一下。

目前我们的内存中已经有很多数据了，有 `Cosmos` 内核本身的执行文件，有字体文件，有 `MMU` 页表，有打包的内核映像文件，还有刚刚建立的内存页和内存区的数据结构，这些数据都要占用实际的物理内存。

再回头看看我们建立内存页结构 `msadsc_t`，所有的都是空闲状态，而它们每一个都表示一个实际的物理内存页。

假如在这种情况下，对调用内存分配接口进行内存分配，它按既定的分配算法查找空闲的 `msadsc_t` 结构，那它一定会找到内核占用的内存页所对应的 `msadsc_t` 结构，并把这个内存页分配出去，然后得到这个页面的程序对其进行改写。这样内核数据就会被覆盖，这种情况是我们绝对不能允许的。

所以，我们要把这些已经占用的内存页面所对应的 `msadsc_t` 结构标记出来，标记成**已分配**，这样内存分配算法就不会找到它们了。

要解决这个问题，我们只要给出被占用内存的起始地址和结束地址，然后从起始地址开始查找对应的 `msadsc_t` 结构，再把它标记为已经分配，最后直到查找到结束地址为止。

下面我们在 `msadsc.c` 文件中来实现这个方案，代码如下。

[复制代码](#)

```
1 //搜索一段内存地址空间所对应的msadsc_t结构
2 u64_t search_segment_occupymadsc(msadsc_t *msastart, u64_t msanr, u64_t ocypstat)
3 {
4     u64_t mphyadr = 0, fsmsnr = 0;
5     msadsc_t *fstatmp = NULL;
6     for (u64_t mnr = 0; mnr < msanr; mnr++)
7     {
8         if ((msastart[mnr].md_phyadrs.paf_padr << PSHRSIZE) == ocypstat)
9         {
10             //找出开始地址对应的第一个msadsc_t结构，就跳转到step1
11             fstatmp = &msastart[mnr];
12             goto step1;
13         }
14     }
15 step1:
16     fsmsnr = 0;
17     if (NULL == fstatmp)
18     {
19         return 0;
20     }
21     for (u64_t tmpadr = ocypstat; tmpadr < ocypend; tmpadr += PAGE_SIZE, fsmsnr++)
22     {
23         //从开始地址对应的第一个msadsc_t结构开始设置，直到结束地址对应的最后一个msadsc_t:
24         mphyadr = fstatmp[fsmsnr].md_phyadrs.paf_padr << PSHRSIZE;
25         if (mphyadr != tmpadr)
26         {
27             return 0;
28         }
29         if (MF_MOCTY_FREE != fstatmp[fsmsnr].md_idxflgs.mf_mocty ||
30             0 != fstatmp[fsmsnr].md_idxflgs.mf_uindx ||
31             PAF_NO_ALLOC != fstatmp[fsmsnr].md_phyadrs.paf_alloc)
```



```
32     {
33         return 0;
34     }
35     //设置msadsc_t结构为已经分配，已经分配给内核
36     fstatmp[fmsnr].md_idxflgs.mf_mocty = MF_MOCTY_KRNL;
37     fstatmp[fmsnr].md_idxflgs.mf_uindx++;
38     fstatmp[fmsnr].md_phyadrs.paf_alloc = PAF_ALLOC;
39 }
40 //进行一些数据的正确性检查
41 u64_t ocphysz = ocpyend - ocphysz;
42 if ((ocphysz & 0xfff) != 0)
43 {
44     if (((ocphysz >> PSRHSIZE) + 1) != fmsnr)
45     {
46         return 0;
47     }
48     return fmsnr;
49 }
50 if ((ocphysz >> PSRHSIZE) != fmsnr)
51 {
52     return 0;
53 }
54 return fmsnr;
55 }
56
57
58 bool_t search_krloccupysadsc_core(machbstart_t *mbp)
59 {
60     u64_t retschmnr = 0;
61     msadsc_t *msadstat = (msadsc_t *)phyadr_to_viradr((adr_t)mbp->mb_memmappa
62     u64_t msanr = mbp->mb_memapnr;
63     //搜索BIOS中断表占用的内存页所对应msadsc_t结构
64     retschmnr = search_segment_occupysadsc(msadstat, msanr, 0, 0x1000);
65     if (0 == retschmnr)
66     {
67         return FALSE;
68     }
69     //搜索内核栈占用的内存页所对应msadsc_t结构
70     retschmnr = search_segment_occupysadsc(msadstat, msanr, mbp->mb_krlinit
71     if (0 == retschmnr)
72     {
73         return FALSE;
74     }
75     //搜索内核占用的内存页所对应msadsc_t结构
76     retschmnr = search_segment_occupysadsc(msadstat, msanr, mbp->mb_krlimgpa
77     if (0 == retschmnr)
78     {
79         return FALSE;
80     }
81     //搜索内核映像文件占用的内存页所对应msadsc_t结构
82     retschmnr = search_segment_occupysadsc(msadstat, msanr, mbp->mb_imgpadr,
83     if (0 == retschmnr)
```

```
84     {
85         return FALSE;
86     }
87     return TRUE;
88 }
89 //初始化搜索内核占用的内存页面
90 void init_search_krloccupymm(machbstart_t *mbasp)
91 {
92     //实际初始化搜索内核占用的内存页面
93     if (search_krloccupymsadsc_core(mbsp) == FALSE)
94     {
95         system_error("search_krloccupymsadsc_core fail\n");
96     }
97     return;
98 }
```

这三个函数逻辑很简单，由 `init_search_krloccupymm` 函数入口，`search_krloccupymsadsc_core` 函数驱动，由 `search_segment_occupymsadsc` 函数完成实际的工作。

由于初始化阶段各种数据占用的开始、结束地址和大小，这些信息都保存在 `machbstart_t` 类型的 `kmachbsp` 变量中，所以函数与 `machbstart_t` 类型的指针为参数。

其实 `phymmarge_t`、`msadsc_t`、`memarea_t` 这些结构的实例变量和 MMU 页表，它们所占用的内存空间已经涵盖在了内核自身占用的内存空间。

好了，这个问题我们已经完美解决，只要在初始化内存页结构和内存区结构之后调用 `init_search_krloccupymm` 函数即可。

合并内存页到内存区

我们做了这么多前期工作，依然没有让内存页和内存区联系起来，即让 `msadsc_t` 结构挂载到内存区对应的数组中。只有这样，我们才能提高内存管理器的分配速度。

让我们来着手干这件事情，这件事情有点复杂，但是我给你梳理以后就会清晰很多。整体上可以分成两步。


1. 确定内存页属于哪个区，即标定一系列 `msadsc_t` 结构是属于哪个 `memarea_t` 结构的。

2.把特定的内存页合并，然后挂载到特定的内存区下的 memdivmer_t 结构中的 dm_mdmlielst 数组中。

我们先来做第一件事，这件事比较简单，我们只要遍历每个 memarea_t 结构，遍历过程中根据特定的 memarea_t 结构，然后去扫描整个 msadsc_t 结构数组，最后依次对比 msadsc_t 的物理地址，看它是否落在 memarea_t 结构的地址区间中。

如果是，就把这个 memarea_t 结构的类型值写入 msadsc_t 结构中，这样就一个一个打上了标签，遍历 memarea_t 结构结束之后，每个 msadsc_t 结构就只归属于某一个 memarea_t 结构了。

我们在 memarea.c 文件中写几个函数，来实现前面这个步骤，代码如下所示。

 复制代码

```
1 //给msadsc_t结构打上标签
2 uint_t merlove_setallmarflgs_onmemarea(memarea_t *mareap, msadsc_t *mstat, uin
3 {
4     u32_t muindx = 0;
5     msadflgs_t *mdfp = NULL;
6     //获取内存区类型
7     switch (mareap->ma_type){
8     case MA_TYPE_HWAD:
9         muindx = MF_MARTY_HWD << 5; //硬件区标签
10        mdfp = (msadflgs_t *)(&muindx);
11        break;
12    case MA_TYPE_KRNL:
13        muindx = MF_MARTY_KRL << 5; //内核区标签
14        mdfp = (msadflgs_t *)(&muindx);
15        break;
16    case MA_TYPE_PROC:
17        muindx = MF_MARTY_PRC << 5; //应用区标签
18        mdfp = (msadflgs_t *)(&muindx);
19        break;
20    }
21    u64_t phyadr = 0;
22    uint_t retnr = 0;
23    //扫描所有的msadsc_t结构
24    for (uint_t mix = 0; mix < msanr; mix++)
25    {
26        if (MF_MARTY_INIT == mstat[mix].md_indxflgs.mf_marty)
27        {
28            //获取msadsc_t结构对应的地址
29            phyadr = mstat[mix].md_phyadrs.paf_padrs << PSHRSIZE;
30            //和内存区的地址区间比较
31            if (phyadr >= mareap->ma_logicstart && ((phyadr + PAGESIZE) - 1) <
```

```
32         //设置msadsc_t结构的标签
33         mstat[mix].md_indxflgs.mf_marty = mdfp->mf_marty;
34         retnr++;
35     }
36 }
37 }
38 return retnr;
39 }
40
41 bool_t merlove_mem_core(machbstart_t *mbsp)
42 {
43     //获取msadsc_t结构的首地址
44     msadsc_t *mstatp = (msadsc_t *)phyadr_to_viradr((adr_t)mbsp->mb_memmappadr
45     //获取msadsc_t结构的个数
46     uint_t msanr = (uint_t)mbsp->mb_memmapnr, maxp = 0;
47     //获取memarea_t结构的首地址
48     memarea_t *marea = (memarea_t *)phyadr_to_viradr((adr_t)mbsp->mb_memznpadr
49     uint_t sretf = ~0UL, tretf = ~0UL;
50     //遍历每个memarea_t结构
51     for (uint_t mi = 0; mi < (uint_t)mbsp->mb_memznnr; mi++)
52     {
53         //针对其中一个memarea_t结构给msadsc_t结构打上标签
54         sretf = merlove_setallmarflgs_onmemarea(&marea[mi], mstatp, msanr);
55         if ((~0UL) == sretf)
56         {
57             return FALSE;
58         }
59     }
60     //遍历每个memarea_t结构
61     for (uint_t maidx = 0; maidx < (uint_t)mbsp->mb_memznnr; maidx++)
62     {
63         //针对其中一个memarea_t结构对msadsc_t结构进行合并
64         if (merlove_mem_onmemarea(&marea[maidx], mstatp, msanr) == FALSE)
65         {
66             return FALSE;
67         }
68         maxp += marea[maidx].ma_maxpages;
69     }
70     return TRUE;
71 }
72 //初始化页面合并
73 void init_merlove_mem()
74 {
75     if (merlove_mem_core(&kmachbsp) == FALSE)
76     {
77         system_error("merlove_mem_core fail\n");
78     }
79     return;
80 }
```

我们一下子写了三个函数，它们的作用且听我一一道来。从 `init_merlove_mem` 函数开始，但是它并不实际干活，作为入口函数，它调用的 `merlove_mem_core` 函数才是真正干活的。

这个 `merlove_mem_core` 函数有两个遍历内存区，第一次遍历是为了完成上述第一步：确定内存页属于哪个区。

当确定内存页属于哪个区之后，就来到了第二次遍历 `memarea_t` 结构，合并其中的 `msadsc_t` 结构，并把它们挂载到其中的 `memdivmer_t` 结构下的 `dm_mdmlielst` 数组中。

这个操作就稍微有点复杂了。**第一，它要保证其中所有的 `msadsc_t` 结构挂载到 `dm_mdmlielst` 数组中合适的 `bafhlst_t` 结构中。**

第二，它要保证多个 `msadsc_t` 结构有最大的连续性。

举个例子，比如一个内存区中有 12 个页面，其中 10 个页面是连续的地址为 0 ~ 0x9000，还有两个页面其中一个地址为 0xb000，另一个地址为 0xe000。

这样的情况下，需要多个页面保持最大的连续性，还有在 `m_mdmlielst` 数组中找到合适的 `bafhlst_t` 结构。

那么：0 ~ 0x7000 这 8 个页面就要挂载到 `m_mdmlielst` 数组中第 3 个 `bafhlst_t` 结构中；0x8000 ~ 0x9000 这 2 个页面要挂载到 `m_mdmlielst` 数组中第 1 个 `bafhlst_t` 结构中，而 0xb000 和 0xe000 这 2 个页面都要挂载到 `m_mdmlielst` 数组中第 0 个 `bafhlst_t` 结构中。

从上述代码可以看出，遍历每个内存区，然后针对其中每一个内存区进行 `msadsc_t` 结构的合并操作，完成这个操作的是 `merlove_mem_onmemarea`，我们这就去写好这个函数，代码如下所示。

 复制代码

```
1 bool_t continumsadsc_add_bafhlst(memarea_t *mareap, bafhlst_t *bafhp, msadsc_t
2 {
3     fstat->md_indxflgs.mf_olkty = MF_OLKTY_ODER;
4     //开始的msadsc_t结构指向最后的msadsc_t结构
```



```

5     fstat->md_odlink = fend;
6     fend->md_idxflgs.mf_olkty = MF_OLKTY_BAFH;
7     //最后的msadsc_t结构指向它属于的bafhlst_t结构
8     fend->md_odlink = bafhp;
9     //把多个地址连续的msadsc_t结构的的开始的那个msadsc_t结构挂载到bafhlst_t结构的af_fr
10    list_add(&fstat->md_list, &bafhp->af_frelst);
11    //更新bafhlst_t的统计数据
12    bafhp->af_fobjnr++;
13    bafhp->af_mobjnr++;
14    //更新内存区的统计数据
15    mareap->ma_maxpages += fmnr;
16    mareap->ma_freepages += fmnr;
17    mareap->ma_allmsadscnr += fmnr;
18    return TRUE;
19 }
20
21 bool_t continumsadsc_mareabafh_core(memarea_t *mareap, msadsc_t **rfstat, msad
22 {
23     uint_t retval = *rfmnr, tmpmnr = 0;
24     msadsc_t *mstat = *rfstat, *mend = *rfend;
25     //根据地址连续的msadsc_t结构的数量查找合适bafhlst_t结构
26     bafhlst_t *bafhp = find_continumsa_inbafhlst(mareap, retval);
27     //判断bafhlst_t结构状态和类型对不对
28     if ((BAFH_STUS_DIVP == bafhp->af_stus || BAFH_STUS_DIVM == bafhp->af_stus)
29     {
30         //看地址连续的msadsc_t结构的数量是不是正好是bafhp->af_oderpnr
31         tmpmnr = retval - bafhp->af_oderpnr;
32         //根据地址连续的msadsc_t结构挂载到bafhlst_t结构中
33         if (continumsadsc_add_bafhlst(mareap, bafhp, mstat, &mstat[bafhp->af_o
34         {
35             return FALSE;
36         }
37         //如果地址连续的msadsc_t结构的数量正好是bafhp->af_oderpnr则完成，否则返回再次进
38         if (tmpmnr == 0)
39         {
40             *rfmnr = tmpmnr;
41             *rfend = NULL;
42             return TRUE;
43         }
44         //挂载bafhp->af_oderpnr地址连续的msadsc_t结构到bafhlst_t中
45         *rfstat = &mstat[bafhp->af_oderpnr];
46         //还剩多少个地址连续的msadsc_t结构
47         *rfmnr = tmpmnr;
48         return TRUE;
49     }
50     return FALSE;
51 }
52
53 bool_t merlove_continumsadsc_mareabafh(memarea_t *mareap, msadsc_t *mstat, msa
54 {
55     uint_t mnridx = mnr;
56     msadsc_t *fstat = mstat, *fend = mend;

```

```

57 //如果mnridx > 0并且NULL != fend就循环调用continumsadsc_mareabafh_core函数，而
58 for (; (mnridx > 0 && NULL != fend);)
59 {
60 //为一段地址连续的msadsc_t结构寻找合适m_mdmlielst数组中的bafhlst_t结构
61     continumsadsc_mareabafh_core(mareap, &fstat, &fend, &mnridx)
62 }
63 return TRUE;
64 }
65
66
67 bool_t merlove_scan_continumsadsc(memarea_t *mareap, msadsc_t *fmstat, uint_t
68                                 msadsc_t **retmsastatp, msadsc_t **re
69 {
70     u32_t muindx = 0;
71     msadflgs_t *mdfp = NULL;
72
73     msadsc_t *msastat = fmstat;
74     uint_t retfindmnr = 0;
75     bool_t rets = FALSE;
76     uint_t tmidx = *fntmsanr;
77     //从外层函数的fntmnr变量开始遍历所有msadsc_t结构
78     for (; tmidx < fmsanr; tmidx++)
79     {
80         //一个msadsc_t结构是否属于这个内存区，是否空闲
81         if (msastat[tmidx].md_indxflgs.mf_marty == mdfp->mf_marty &&
82             0 == msastat[tmidx].md_indxflgs.mf_uindx &&
83             MF_MOCTY_FREE == msastat[tmidx].md_indxflgs.mf_mocty &&
84             PAF_NO_ALLOC == msastat[tmidx].md_phyadrs.paf_alloc)
85         {
86             //返回从这个msadsc_t结构开始到下一个非空闲、地址非连续的msadsc_t结构对应的msads
87             rets = scan_len_msadsc(&msastat[tmidx], mdfp, fmsanr, &retfindmnr)
88             //下一轮开始的msadsc_t结构索引
89             *fntmsanr = tmidx + retfindmnr + 1;
90             //当前地址连续msadsc_t结构的开始地址
91             *retmsastatp = &msastat[tmidx];
92             //当前地址连续msadsc_t结构的结束地址
93             *retmsaendp = &msastat[tmidx + retfindmnr];
94             //当前有多少个地址连续msadsc_t结构
95             *retfmnr = retfindmnr + 1;
96             return TRUE;
97         }
98     }
99     return FALSE;
100 }
101
102 bool_t merlove_mem_onmemarea(memarea_t *mareap, msadsc_t *mstat, uint_t msanr)
103 {
104     msadsc_t *retstatmsap = NULL, *retendmsap = NULL, *fntmsap = mstat;
105     uint_t retfindmnr = 0;
106     uint_t fntmnr = 0;
107     bool_t retscan = FALSE;
108

```

```
109     for (; fntmnr < msanr;)
110     {
111         //获取最多且地址连续的msadsc_t结构体的开始、结束地址、一共多少个msadsc_t结构体，
112         retscan = merlove_scan_continumsadsc(mareap, fntmsap, &fntmnr, msanr,
113         if (NULL != retstatmsap && NULL != retendmsap)
114         {
115             //把一组连续的msadsc_t结构体挂载到合适的m_mdmlielst数组中的bafhlst_t结构中
116             merlove_continumsadsc_mareabafh(mareap, retstatmsap, retendmsap, retfi
117         }
118     }
119     return TRUE;
120 }
```

为了节约篇幅，我删除了大量检查错误的代码，你可以在我提供的 [源代码](#) 里自行查看。

上述代码中，整体上分为两步。

第一步，通过 `merlove_scan_continumsadsc` 函数，返回最多且地址连续的 `msadsc_t` 结构体的开始、结束地址、一共多少个 `msadsc_t` 结构体，下一轮开始的 `msadsc_t` 结构体的索引号。

第二步，根据第一步获取的信息调用 `merlove_continumsadsc_mareabafh` 函数，把第一步返回那一组连续的 `msadsc_t` 结构体，挂载到合适的 `m_mdmlielst` 数组中的 `bafhlst_t` 结构中。详细的逻辑已经在注释中说明。

好，内存页已经按照规定的方式组织起来了，这表示物理内存管理器的初始化工作已经进入尾声。

初始化汇总

别急！先别急着写内存分配相关的代码。到目前为止，我们一起写了这么多的内存初始化相关的代码，但是我们没有调用它们。

根据前面内存管理数据结构的关系，很显然，**它们的调用次序很重要，谁先谁后都有严格的规定，这关乎内存管理初始化的成败。**所以，现在我们就在先前的 `init_memmgr` 函数中去调用它们，代码如下所示。

```
1 void init_memmgr()
```

 复制代码

```

2 {
3     //初始化内存页结构
4     init_msadsc();
5     //初始化内存区结构
6     init_memarea();
7     //处理内存占用
8     init_search_krloccupymm(&kmachbsp);
9     //合并内存页到内存区中
10    init_merlove_mem();
11    init_memmgorb();
12    return;
13 }

```


上述代码中，init_msadsc、init_memarea 函数是可以交换次序的，它们俩互不影响，但它们俩必须最先开始调用，而后面的函数要依赖它们生成的数据结构。

但是 init_search_krloccupymm 函数必须要在 init_merlove_mem 函数之前被调用，因为 init_merlove_mem 函数在合并页面时，必须先知道哪些页面被占用了。

等一等，init_memmgorb 是什么函数，这个我们还没写呢。下面我们就来现实它。

不知道你发现没有，我们的 phymmarge_t 结构体的地址和数量、msadsc_t 结构体的地址和数据、memarea_t 结构体的地址和数量都保存在了 kmachbsp 变量中，这个变量其实不是用来管理内存的，而且它里面放的是**物理地址**。

但内核使用的是虚拟地址，每次都要转换极不方便，所以我们要设计一个专用的数据结构，用于内存管理。我们来定义一下这个结构，代码如下。

 复制代码

```

1 //cosmos/include/halinc/halglobal.c
2 HAL_DEFGLOB_VARIABLE(memmgrob_t,memmgrob);
3
4 typedef struct s_MEMMGROB
5 {
6     list_h_t mo_list;
7     spinlock_t mo_lock;           //保护自身自旋锁
8     uint_t mo_stus;               //状态
9     uint_t mo_flg;                //标志
10    u64_t mo_memsz;                //内存大小
11    u64_t mo_maxpages;             //内存最大页面数
12    u64_t mo_freepages;            //内存最大空闲页面数
13    u64_t mo_alocpages;            //内存最大分配页面数

```

```
14     u64_t mo_resvpages;           //内存保留页面数
15     u64_t mo_horizline;          //内存分配水位线
16     phymmarge_t* mo_pimagestat;  //内存空间布局结构指针
17     u64_t mo_pmagenr;
18     msadsc_t* mo_msadscstat;     //内存页面结构指针
19     u64_t mo_msanr;
20     memarea_t* mo_mareastat;     //内存区结构指针
21     u64_t mo_mareanr;
22 }memmgrob_t;
23
24 //cosmos/hal/x86/memmgrinit.c
25
26 void memmgrob_t_init(memmgrob_t *initp)
27 {
28     list_init(&initp->mo_list);
29     knl_spinlock_init(&initp->mo_lock);
30     initp->mo_stus = 0;
31     initp->mo_flg = 0;
32     initp->mo_memsz = 0;
33     initp->mo_maxpages = 0;
34     initp->mo_freepages = 0;
35     initp->mo_alocpages = 0;
36     initp->mo_resvpages = 0;
37     initp->mo_horizline = 0;
38     initp->mo_pimagestat = NULL;
39     initp->mo_pmagenr = 0;
40     initp->mo_msadscstat = NULL;
41     initp->mo_msanr = 0;
42     initp->mo_mareastat = NULL;
43     initp->mo_mareanr = 0;
44     return;
45 }
46
47 void init_memmgrob()
48 {
49     machbstart_t *mb = &kmachb;
50     memmgrob_t *mob = &memmgrob;
51     memmgrob_t_init(mob);
52     mob->mo_pimagestat = (phymmarge_t *)phyadr_to_viradr((adr_t)mb->mb_e820e
53     mob->mo_pmagenr = mb->mb_e820exnr;
54     mob->mo_msadscstat = (msadsc_t *)phyadr_to_viradr((adr_t)mb->mb_memmapp
55     mob->mo_msanr = mb->mb_memmapnr;
56     mob->mo_mareastat = (memarea_t *)phyadr_to_viradr((adr_t)mb->mb_memznpa
57     mob->mo_mareanr = mb->mb_memznnr;
58     mob->mo_memsz = mb->mb_memmapnr << PSHRSIZE;
59     mob->mo_maxpages = mb->mb_memmapnr;
60     uint_t aid = 0;
61     for (uint_t i = 0; i < mob->mo_msanr; i++)
62     {
63         if (1 == mob->mo_msadscstat[i].md_indxflds.mf_uindx &&
64             MF_MOCTY_KRNL == mob->mo_msadscstat[i].md_indxflds.mf_mocty &&
65             PAF_ALLOC == mob->mo_msadscstat[i].md_phyadrs.paf_alloc)
```



```
66         {
67             aidx++;
68         }
69     }
70     mobp->mo_alocpages = aidx;
71     mobp->mo_freepages = mobp->mo_maxpages - mobp->mo_alocpages;
72     return;
73 }
```

这些代码非常容易理解，我们就不再讨论了，无非是将内存管理核心数据结构的地址和数量放在其中，并计算了一些统计信息，这没有任何难度，相信你会轻松理解。

重点回顾

今天课程的重点工作是初始化我们设计的内存管理数据结构，在内存中建立它们的实例变量，我来为你梳理一下重点。

首先，我们从初始化 `msadsc_t` 结构开始，在内存中建立 `msadsc_t` 结构的实例变量，每个物理内存页面一个 `msadsc_t` 结构的实例变量。

然后是初始化 `memarea_t` 结构，在 `msadsc_t` 结构的实例变量之后，每个内存区一个 `memarea_t` 结构实例变量。

接着标记哪些 `msadsc_t` 结构对应的物理内存被内核占用了，这些被标记 `msadsc_t` 结构是不能纳入内存管理结构中去。

最后，把所有的空闲 `msadsc_t` 结构按最大地址连续的形式组织起来，挂载到 `memarea_t` 结构下的 `memdivmer_t` 结构中，对应的 `dm_mdmlielst` 数组中。

不知道你是否想过，随着物理内存不断增加，`msadsc_t` 结构实例变量本身占用的内存空间就会增加，那你有办法降低 `msadsc_t` 结构实例变量占用的内存空间吗？期待你的实现。

思考题

请问在 4GB 的物理内存的情况下，`msadsc_t` 结构实例变量本身占用多大的内存空间？

欢迎你在留言区跟我交流互动，也希望你能把这节课分享给你的同事、朋友。

好，我是 LMOS，我们下节课见！

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

👍 赞 4

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 划分土地（上）：如何划分与组织内存？

下一篇 18 | 划分土地（下）：如何实现内存页的分配与释放？

更多学习推荐

极客时间 | 训练营

大厂面试必考 100 题

2021 最新版 | 算法篇

限量免费领取 📌

仅限前 99 名



精选留言 (5)

💬 写留言



neohope

2021-06-17

一、稍微整理了一下流程：

init_hal->init_halm->init_memmgr

//每个页对应一个msadsc_t 结构体，循环填充msadsc_t 结构体数组

->init_msadsc

//初始化三类memarea t , 硬件区、内核区、用户区...

展开

作者回复: 大神 6666

46



pedro
2021-06-16

胡乱一猜~

msadsc_t 占用内存 = 4GB/4KB(页大小) * sizeof(msadsc_t)

作者回复: 是的

21



Feen
2021-07-09

基本的比例就是每1GB要占用10M的空间，还好不算太大。

展开

作者回复: 是的



! null
2021-07-09

接着，要给这个开始地址加上 0x1000，如此循环，直到其结束地址。
+0x1000是啥作用？

展开

作者回复: 一个页面大小 啊



有手也不行
2021-07-01

老师你好，请问内存区结构初始化里面的memdivmer_t_init函数的最后一行为什么要执行bafhlst_t_init(&initp->dm_onemsalst, BAFH_STUS_ONEM, 0, 1UL); bafhlst_t结构体里面dm_onemsalst属性有什么作用吗

作者回复: 对dm_onemsalst结构进行初始化啊

