操作系统就用一张大表管理内存?

Original 闪客 低并发编程 2021-12-26 16:30

收录于合集
#操作系统源码

43个

今天我们不聊具体内存管理的算法，我们就来看看，操作系统用什么样的一张表，达到了管理内存的效果。

我们以 Linux 0.11 源码为例，发现进入内核的 main 函数后不久，有这样一坨代码。

```
void main(void) {
    ...
    memory_end = (1<<20) + (EXT_MEM_K<<10);
    memory_end &= 0xfffff000;
    if (memory_end > 16*1024*1024)
        memory_end = 16*1024*1024;
    if (memory_end > 12*1024*1024)
        buffer_memory_end = 4*1024*1024;
    else if (memory_end > 6*1024*1024)
        buffer_memory_end = 2*1024*1024;
    else
        buffer_memory_end = 1*1024*1024;
    main_memory_start = buffer_memory_end;

    mem_init(main_memory_start,memory_end);
    ...
}
```

除了最后一行外，前面的那一大坨的作用很简单。

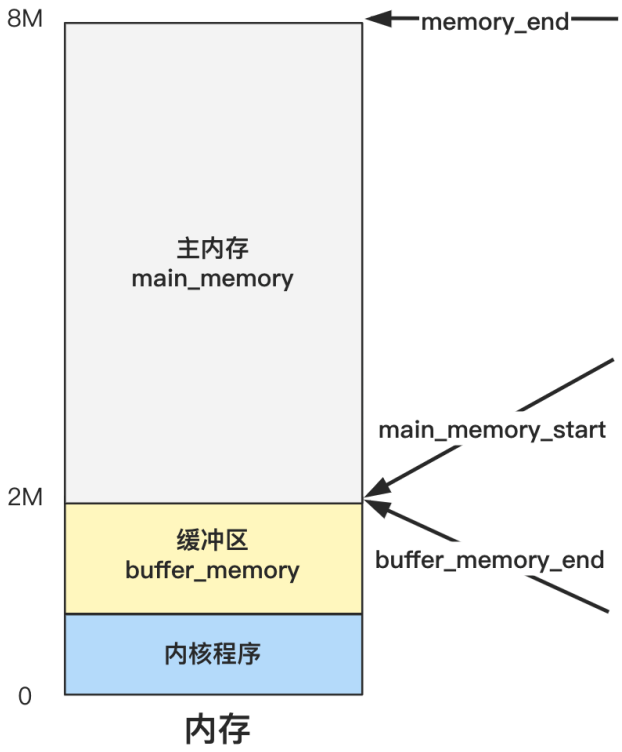
其实就只是针对不同的内存大小，设置不同的边界值罢了，为了理解它，我们完全没必要考虑这么周全，就假设总内存一共就 **8M** 大小吧。

那么如果内存为 8M 大小，**memory_end** 就是
8 * 1024 * 1024

也就只会走倒数第二个分支，那么 **buffer_memory_end** 就为
2 * 1024 * 1024
那么 **main_memory_start** 也为
2 * 1024 * 1024

你仔细看看代码逻辑，看是不是这样？

当然，你不愿意细想也没关系，上述代码执行后，就是如下效果而已。



你看，其实就是定了三个箭头所指向的地址的三个边界变量。具体主内存区是如何管理和分配的，要看 mem_init 里做了什么。

```
void main(void) {
    ...
    mem_init(main_memory_start, memory_end);
    ...
}
```

而缓冲区是如何管理和分配的，就要看再后面的 buffer_init 里干了什么。

```
void main(void) {
    ...
    buffer_init(buffer_memory_end);
    ...
}
```

不过我们今天只看，主内存是如何管理的，很简单，放轻松。

进入 mem_init 函数。

```
#define LOW_MEM 0x100000
#define PAGING_MEMORY (15*1024*1024)
#define PAGING_PAGES (PAGING_MEMORY>>12)
#define MAP_NR(addr) (((addr)-LOW_MEM)>>12)
#define USED 100

static long HIGH_MEMORY = 0;
static unsigned char mem_map[PAGING_PAGES] = { 0, };

// start_mem = 2 * 1024 * 1024
// end_mem = 8 * 1024 * 1024
void mem_init(long start_mem, long end_mem)
{
    int i;
    HIGH_MEMORY = end_mem;
    for (i=0 ; i<PAGING_PAGES ; i++)
        mem_map[i] = USED;
    i = MAP_NR(start_mem);
    end_mem -= start_mem;
    end_mem >>= 12;
    while (end_mem-->0)
        mem_map[i++]=0;
}
```

发现也没几行，而且并没有更深的方法调用，看来是个好欺负的方法。

仔细一看这个方法，其实折腾来折腾去，就是给一个 **mem_map** 数组的各个位置上赋了值，

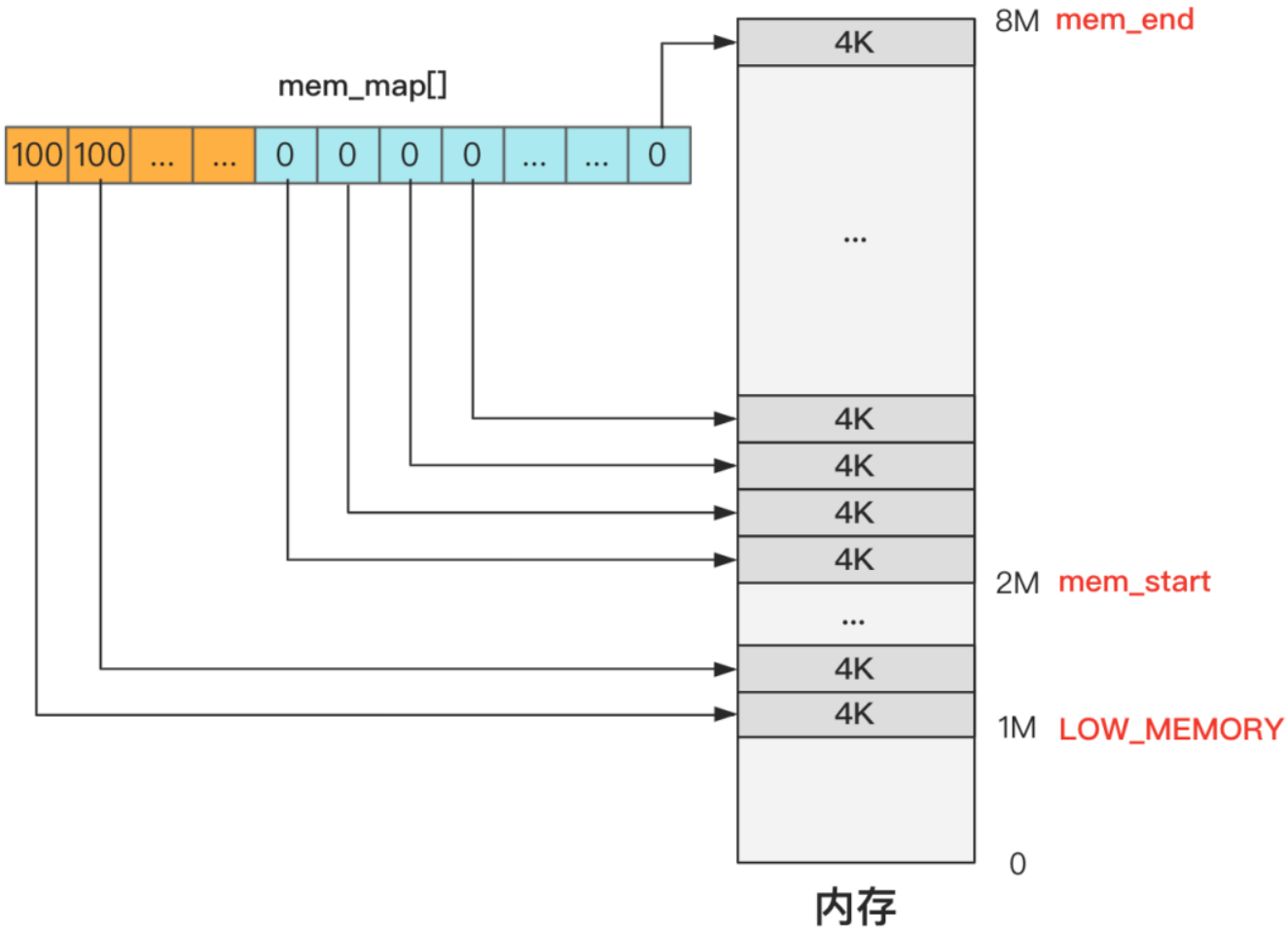
而且显示全部赋值为 USED 也就是 100，然后对其中一部分又赋值为了 0。

赋值为 100 的部分就是 USED，也就表示内存被占用，如果再具体说是占用了 100 次，这个之后再说。剩下赋值为 0 的部分就表示未被使用，也即使用次数为零。

是不是很简单？**就是准备了一个表，记录了哪些内存被占用了，哪些内存没被占用。**这就是所谓的“管理”，并没有那么神乎其神。

那接下来自然有两个问题，每个元素表示占用和未占用，这个表示的范围是多大？初始化时哪些地方是占用的，哪些地方又是未占用的？

还是一张图就看明白了，我们仍然假设内存总共只有 8M。



可以看出，初始化完成后，其实就是 mem_map 这个数组的每个元素都代表一个 4K 内存是否空闲（准确说是使用次数）。

4K 内存通常叫做 1 页内存，而这种管理方式叫**分页管理**，就是把内存分成一页一页（4K）的单位去管理。

1M 以下的内存这个数组干脆没有记录，这里的内存是无需管理的，或者换个说法是无权管理的，也就是没有权利申请和释放，因为这个区域是内核代码所在的地方，不能被“污染”。

1M 到 2M 这个区间是**缓冲区**，2M 是缓冲区的末端，缓冲区的开始在哪里之后再说，这些地方不是主内存区域，因此直接标记为 USED，产生的效果就是无法再被分配了。

2M 以上的空间是**主内存区域**，而主内存目前没有任何程序申请，所以初始化时统统都是零，未来等着应用程序去申请和释放这里的内存资源。

那应用程序如何申请内存呢？我们本讲不展开，不过我们简单展望一下，看看申请内存的过程中，是如何使用 mem_map 这个结构的。

在 **memory.c** 文件中有个函数 **get_free_page()**，用于在主内存区中申请一页空闲内存页，并返回物理内存页的起始地址。

比如我们在 fork 子进程的时候，会调用 **copy_process** 函数来复制进程的结构信息，其中有一个步骤就是要**申请一页内存**，用于存放进程结构信息 task_struct。

```
int copy_process(...) {
    struct task_struct *p;
    ...
    p = (struct task_struct *) get_free_page();
    ...
}
```

我们看 get_free_page 的具体实现，是内联汇编代码，看不懂不要紧，注意它里面就有 **mem_map** 结构的使用。

```
unsigned long get_free_page(void) {
    register unsigned long __res asm("ax");
    __asm__(
        "std ; repne ; scasb\n\t"
        "jne 1f\n\t"
        "movb $1,1(%%edi)\n\t"
        "sall $12,%%ecx\n\t"
        "addl %2,%%ecx\n\t"
        "movl %%ecx,%%edx\n\t"
        "movl $1024,%%ecx\n\t"
        "leal 4092(%%edx),%%edi\n\t"
        "rep ; stosl\n\t"
        "movl %%edx,%%eax\n"
        "1:"
        :"=a" (__res)
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map + PAGING_PAGES-1)
        :"di","cx","dx");
    return __res;
}
```

就是选择 mem_map 中首个空闲页面，并标记为已使用。

好了，本讲就这么多，只是填写了一张大表而已，简单吧？之后的内存申请与释放等骚操作，统统是跟着张大表 **mem_map** 打交道而已，你一定要记住它哦。

本文可以当做 你管这破玩意叫操作系统源码 系列文章的第 13 回。

为了让不追更系列的读者也能很方便阅读并学到东西，我把它改造成了单独的不依赖系列上下文的文章，具体原因可以看 坚持不下去了...

点击下方的**阅读原文**可以跳转到本系列的 GitHub 页，那里也有完整目录和规划，以及一些辅助

的资料，欢迎提出各种问题。

低并发编程

战略上藐视技术，战术上重视技术

175篇原创内容

Official Account

收录于合集 #操作系统源码 43

上一篇

第12回 | 管理内存前先划分出三个边界值

下一篇

你的键盘是什么时候生效的?

Read more