Project 5 Virtual Memory 设计文档

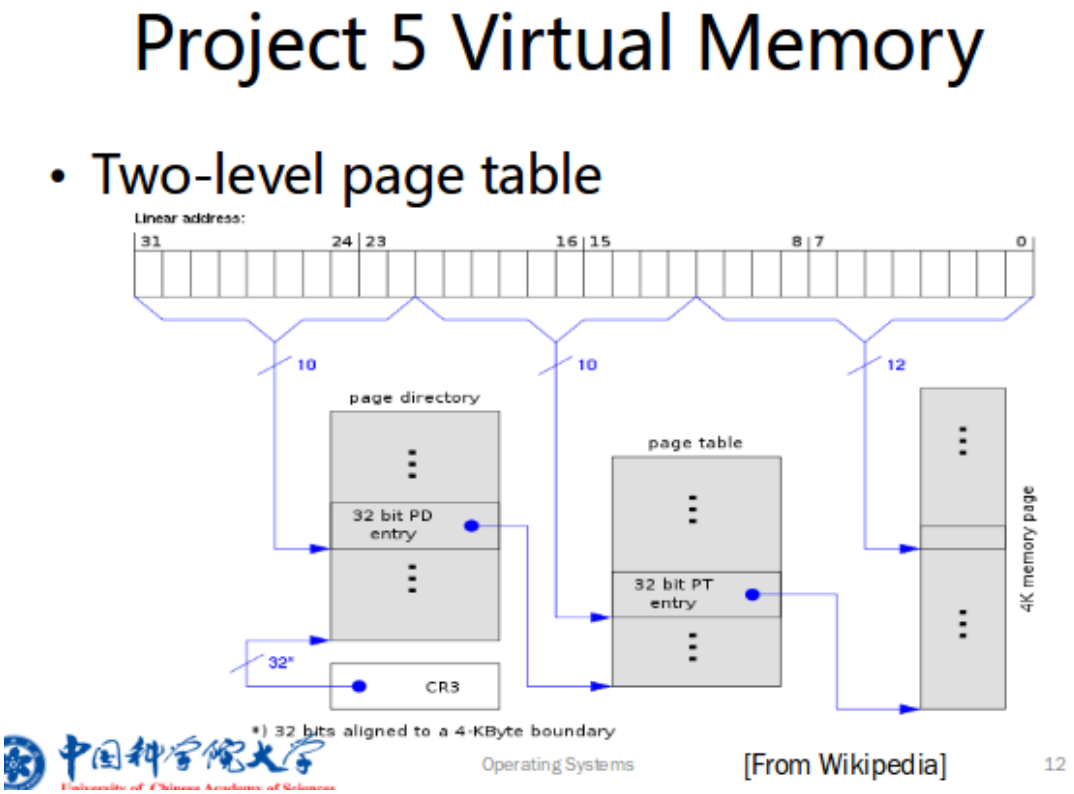
中国科学院大学

[姓名] 孔静 尚耔彤

[日期] 2016.12.14

1. 内核内存管理设计

（1）你理解的内存虚、实址转换过程



虚拟地址在寻找物理地址的过程中被分成三部分进行利用。给出虚拟地址进而想找到一个进程页面的物理地址时，先通过检索进程 pcb 得到它的页目录基址，然后将虚拟地址的第一部分，即它的[31:24]位作为页目录的偏移量，就可以得到页目录的 entry；页目录的 entry 是一个32位的二进制数，其[31:12]位是页表的基址，后面12位为标志位。将虚拟地址的[23:12]位作为偏移量，由上一步得到的页表基址，可以得到页表的 entry，其中

[31:12]位是页的基址，后面12位是标志位。将虚拟地址的[11:0]位作为偏移量，再根据页的基址，即可以得到页的地址，也就是我们想要的物理地址。

（2）如何管理物理内存，你用来描述管理物理内存页的元数据有哪些？这些元数据信息各自的主要用途是什么？

page_map_entr y_t 的设计：

创建一个结构体数组来跟踪每一物理页的各种状态与参数

```
/* TODO: Structure of an entry in the page map */
typedef struct node{
   struct node *prev,*next;
}node_t;
typedef struct {
 node_t    node; // for queues
 uint32_t  vaddr;  // virtual address of start of page
 uint32_t  swap_loc;  // location to swap out to on disk
 uint32_t  swap_size; //May need to swap more than one page's worth of data (is it in bits or # of pages)
 uint32_t  addr; // location of page on physical memory
 bool_t    is_pinned;//if true,the page can’ t be swapped out
 bool_t    is_available;//if true,the page is available and can be allocated
 bool_t    is_dirty; // if true, write to disk before evicting from memory
 int       pid; // owner of page (-1 for kernel threads)
 uint32_t  page_dir;//page_directory in the structure PCB
 uint32_t  index; // index within the page_map array
```

} page_map_entry_t;

（3）内核线程内存管理的初始化过程包括哪些操作？页表项的标记位是如何设置的？

内核线程内存管理的初始化过程：

①初始化管理物理页面的数组，将数组中每一物理页设置为默认状态，设置好数组中每个物理页面对应的物理地址、可替换、可用等。

②申请一个物理页面，作为内核页目录，将其参数设置为不可换出且不可用并将其物理地址赋值给内核线程的页目录地址。

③申请 N_KERNEL_PTS 个物理页面，作为内核页表，将其参数设置为不可换出且不可用并将其插入内核页目录，建立好页目录和页表之间的映射关系

④循环初始化内核页表，共 PAGE_N_ENTRIES 个，由于内核可以看到所有的物理地址，因此要将从0到 MAX_PHYSICAL_MEMORY 的全部初始化物理地址填入内核页表。

页表项的标志位设置：

这里只需要设置 read/write 位和 present 位为1。同时，为了满足调试工具 bochs 的显示功能，必须将 screen_addr 的 u/s 位置为1。

（4）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

太久远了，忘记掉了，任务1好像是最快完成的，一开始任务1的初始化里还记录了 owner 和 pid，后来发现重复，最后发现这东西根本没用啊，但还是保留了一个 pid。在3里新增了一个 times，为了记录访问次数。

2. 用户态进程内存管理设计

（1）用户态进程内存管理的初始化过程包括哪些操作？页表项的标记位是如何设置的，为

什么这么设置？如何决定有多少个页目录和页表？

用户态进程内存管理的初始化过程：

①申请一个物理页面，作为页目录，将此页面物理地址赋予进程 pcb 结构中的 page_directory 以形成对应，并将其清零。

②申请一个页面，作为用户进程栈页表，插入进程页目录。

③循环申请 N_PROCESS_STACK_PAGES 个页面，作为栈页面，填入用户进程栈页表。

④申请页面，作为用户进程程序页表，插入进程页目录。

页表项的标志位设置：

除去 present 位和 read/write 位，还要加上 user/supervisor 位，将其置为1，表示是普通用户。

如何决定有多少个页目录和页表：

页目录一个，页表数目未知 1+x+N_KERNEL_PTS，栈页表数目要取决于 N_PROCESS_STACK_PAGES，程序页表数目取决于 swap_size，以及 N_KERNEL_PTS 个内核页表。

（2）该设计中，哪些页属于 pinning pages？这些页的管理与其他页的管理在你的设计中有何不同？

用户进程页目录、用户进程栈页表、用户进程栈页面、用户进程程序页表都是 pinning pages，程序页面是不被 pin 住的。被 pin 住的页在初始化用户态进程内存初始化时就要分配好，不被 pin 住的页是通过先发生缺页中断再申请的方式进行分配的。

（3）该设计中，是否会有缺页中断发生？你按照什么策略分配物理页？

会有缺页中断发生，通过 page_alloc 函数，在 page_map 数组中由低到高寻找空闲的页面，只要找到可分配的页面就将其初始化并分配。在任务2中不要求替换策略，因此无法 load4，被 ASSERT 停住。

（4）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

一开始 swap_size 简单粗暴的写了 SECTORS_PER_PAGE，然后发现一旦先 load4，之后4会发生各种错误，然后究其原因，发现是进程4在 usb 的末端，交换大小不足 SECTORS_PER_PAGE，若要交换，usb 会返回一些错的东西，后来将其修改为正确大小，如果大于 SECTORS_PER_PAGE，再将其设置为 SECTORS_PER_PAGE。

3. 缺页中断与 swap 处理设计

（1）何时发生缺页中断？你设计的缺页中断处理流程是怎样的？

当发现已经没有空闲页可以分配时则发生缺页中断， 根据页替换策略将特定页替换出去，并从磁盘中读取需要的页。

（2）你设计的页替换策略是怎样的？如果有做 bonus，请说明为什么你 bonus 设计的页替换策略能提高效率？

在内存初始化时初始化一个循环链表，并以节点 page_queue 为基准节点，即将最新的插入页插入到 page_queue 的 prev 处，则由 page_queue 的 next 到 page_queue 的 prev 会形成一个循环链表且由 next 到 prev 为页由旧到新。

第一种页替换策略为 FIFO 即先进先出。在发生缺页中断时，将 page_queue 的 next 直接换出，这是将此链表中最早插入的页面替换出去，即先进先出。

第二种页面替换策略为第二次机会算法。第二次机会算法的基本思想是与 FIFO 相同的，但是有所改进，避免把经常使用的页面置换出去。当选择置换页面时，依然和 FIFO 一样，选择最早置入内存的页面。但是二次机会法还会去检查访问状态位 PE_A，如果是0，就淘汰这页；如果访问位是1，就给它第二次机会，并选择下一个 FIFO 页面。当一个页面得到第二次机会时，它的访问位就清为0。在此程序中即将此页面从序列中的 page_queue 的

next 插入调换到 page_queue 的 prev。如果该页在此期间被访问过，则访问位置为1。这样给了第二次机会的页面将不被淘汰，直至所有其他页面被淘汰过（或者也给了第二次机会）。第二次机会算法可视为一个环形队列。用一个指针指示哪一页是下面要淘汰的。当需要一个存储块时，指针就前进，直至找到访问位是0的页。随着指针的前进，把访问位就清为0。在最坏的情况下，所有的访问位都是1，指针要通过整个队列一周，每个页都给第二次机会。这时就退化成 FIFO 算法了。

第三种页面替换算法试图模拟 LRU 算法，给每个页面代表的节点参数 times，在此节点即此页面被访问过后，times 加一，在寻找替换出去的页时，循环遍历 times 最小的页面将其换出，此页面是访问次数最少的页面。另一个参数 change 统计各页面总共被访问的次数，在 change 数达到一定值时，将各个 times 清零，以模仿 LRU“在一定时间内”。由于不会如何再每一次访问都给 times+1，就将其放置到替换算法中，每次替换，遍历页面，并且对有访问的+1，然后置 PE_A 为0。替换 change 次，把 times 清零。比较重要的一点是，遍历页面，并不会遍历队列中的最后一个页面，因为最后一个页面是最近被替换进来的，很容易再次被选中替换，就会导致刚被换进来又被换出去的情况，最终导致进程加载不进来。

（3）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

问题就是，上面模仿 LRU，开始是遍历所有页面，然后导致 load 4无法 done，小飞机的 page_fault 不断增加，后来最后一页不进入替换的选择后，就成功替换了。

4. 关键函数功能

①

/* TODO: Structure of an entry in the page map */

```
typedef struct page_map_entry{
  struct page_map_entry *next,*prev; // for queues
  uint32_t  vaddr;  // virtual address of start of page
  uint32_t  swap_loc;  // location to swap out to on disk
  uint32_t  swap_size; //May need to swap more than one page's worth of data (is
it in bits or # of pages)
  uint32_t  *addr; // location of page on physical memory
  bool_t    is_pinned;
  bool_t    is_available;
  bool_t    is_dirty; // if true, write to disk before evicting from memory
  int       pid; // owner of page (-1 for kernel threads)
  //pcb_t     *owner;
  uint32_t  *page_dir;
  uint32_t  index; // index within the page_map array
  int       times;
} page_map_entry_t;
```

②

```
/* Allocate a page.  If necessary, swap a page out.
 * On success, return the index of the page in the page map.  On
 * failure, abort.  BUG: pages are not made free when a process
 * exits.
 */
int page_alloc(int pinned);
```

③

```
/* Set up kernel memory, called from kernel.c: _start() */
void init_memory(void);
```

④

```
/* Set up a page directory and page table for the given process. Fill in
 * any necessary information in the pcb.
 */
void setup_page_table(pcb_t * p);
```

⑤

```
/* Swap into a free page upon a page fault.
 * This method is called from interrupt.c: exception_14().
 * Should handle demand paging.
 */
void page_fault_handler(void);
```

⑥

```
/* Swap the i-th page in from disk (i.e. the image file) */
void page_swap_in(int i);
```

⑦

```
/* Swap the i-th page out */

void page_swap_out(int i);
```

⑧

```
/* Decide which page to replace, return the page number  */

int page_replacement_policy(void);
```