**《操作系统》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2020级计科02班**  **2020级计科01班** | | | **组员** | **徐小龙**  **陈鹏宇** |
| **实验题目** | **内存管理** | | | | | |
| **实验时间** | **2022/10/27** | | **实验地点** | **DS3401** | | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 □综合性** | | |
| 教师评价：  **□**算法/实验过程正确； **□**源程序/实验内容提交 **□**程序结构/实验步骤合理；  **□**实验结果正确； **□**语法、语义正确； **□**报告规范；  其他：  评价教师签名： | | | | | | |
| 一、实验目的  本次实验主要完成ucore内核对物理内存的管理工作。  主要包括实现 first-fit 连续物理内存分配算法、寻找虚拟地址对应的页表项以及释放某虚地址所在的页并取消对应二级页表项的映射。 | | | | | | |
| 二、实验项目内容  **练习1：实现 first-fit 连续物理内存分配算法（需要编程）**  **在实现first fit内存分配算法的回收函数时，要考虑地址连续的空闲块之间的合并操作。提示:在建立空闲页块链表时，需要按照空闲页块起始地址来排序，形成一个有序的链表。可能会修改default\_pmm.c中的default\_init，default\_init\_memmap，default\_alloc\_pages，default\_free\_pages等相关函数。请仔细查看和理解default\_pmm.c中的注释。**  **注意，目前实验提供的代码已经可以运行，但希望读者在理解的基础上能自己重新实现有关函数。**  **请在实验报告中简要说明你的设计实现过程。请回答如下问题：**  **- 你的first fit算法是否有进一步的改进空间**  **练习2：实现寻找虚拟地址对应的页表项（需要编程）**  **通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的get\_pte函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。本练习需要补全get\_pte函数（kern/mm/pmm.c中），实现其功能。请仔细查看和理解get\_pte函数中的注释。get\_pte函数的调用关系图如下所示：**    **请在实验报告中简要说明你的设计实现过程。请回答如下问题：**  **- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处。**  **练习3：释放某虚地址所在的页并取消对应二级页表项的映射（需要编程）**  **当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解page\_remove\_pte函数中的注释。为此，需要补全在kern/mm/pmm.c中的page\_remove\_pte函数。page\_remove\_pte函数的调用关系图如下所示：**    **请在实验报告中简要说明你的设计实现过程。请回答如下问题：**  **- 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是什么？** | | | | | | |
| 三、实验过程或算法（源程序）  练习1：  **default\_init函数：**  static void  default\_init(void) {    //页表定义      list\_init(&free\_list);  //定义表头      nr\_free = 0;    //空闲页个数  }  **default\_init\_memmap函数：**  static void  default\_init\_memmap(struct Page \*base, size\_t n) {  //根据每个物理页帧的情况来建立空闲页链表，且空闲页块应该是根据地址高低形成一个有序链表  #ifdef LAB2\_EX1      assert(n > 0);  //断言以保证鲁棒性      struct Page \*p = base;      for (; p != base + n; p ++) {          assert(PageReserved(p));    //进行鲁棒性检测,判断是否非空          p->flags = p->property = 0; //初始化状态及空闲块大小          set\_page\_ref(p, 0);         //初始化被引用数      }      base->property = n; //为空闲块分配n页      SetPageProperty(base);  //标记当前页free 可用      nr\_free += n;   //设置空闲页个数为n      list\_add\_before(&free\_list, &(base->page\_link)); //将该空闲块加入空闲链表  #endif  }  **default\_alloc\_pages函数：**  static struct Page \*  default\_alloc\_pages(size\_t n) { //页分配 （参数n表示要分配n个页）  #ifdef LAB2\_EX1      //鲁棒性边界检测      assert(n > 0);      if (n > nr\_free) {          return NULL;      }      struct Page \*page = NULL;   //用于记录满足需求的页表      list\_entry\_t \*le = &free\_list;  //le指向空闲链表头      // TODO: optimize (next-fit)      while ((le = list\_next(le)) != &free\_list) {    //通过 list\_next(le) 找到下一个空闲元素          struct Page \*p = le2page(le, page\_link);    //由链表元素获得对应的page指针 p          if (p->property >= n) { //判断此空闲页大小是否满足需求              page = p;   //记录满足要求的空闲块              break;          }      }      if (page != NULL) { //判断是否分配成功          if (page->property > n) {   //成功且有剩余空闲页时              struct Page \*p = page + n;  //指向新的空闲块Head Page              p->property = page->property - n;   //设置新空闲块的大小              SetPageProperty(p); //标记为free              list\_add\_after(&(page->page\_link), &(p->page\_link));    //加入空闲块链表          }          list\_del(&(page->page\_link));   //删除旧的空闲块信息          nr\_free -= n;   //修改空闲页的数量          ClearPageProperty(page);    //标记旧空闲块被占用      }      return page;  #endif  }  **default\_free\_pages函数：**  static void  default\_free\_pages(struct Page \*base, size\_t n) {   //页释放（页分配的逆过程） 但要考虑空闲块的合并  #ifdef LAB2\_EX1      assert(n > 0);  //鲁棒性检测      struct Page \*p = base;      for (; p != base + n; p ++) {          assert(!PageReserved(p) && !PageProperty(p));   //断言页未被保留且非空闲          p->flags = 0;           //重置页的状态信息          set\_page\_ref(p, 0);     //重置页的被引用数      }      base->property = n;         //设置空闲块的大小      SetPageProperty(base);      //标记为free      list\_entry\_t \*le = list\_next(&free\_list);      while (le != &free\_list) {  //遍历空闲链表处理空闲块合并          p = le2page(le, page\_link);     //由链表元素获得对应的page指针 p          le = list\_next(le);          // TODO: optimize          //处理空闲块的两种合并情况          if (base + base->property == p) {   //情况一 base合并p              base->property += p->property;              ClearPageProperty(p);              list\_del(&(p->page\_link));          }          else if (p + p->property == base) {   //情况二 p合并base              p->property += base->property;              ClearPageProperty(base);              base = p;              list\_del(&(p->page\_link));          }      }      nr\_free += n;   //更新空闲页数量      le = list\_next(&free\_list);     //用于记录插入位置      while (le != &free\_list) {  //遍历链表寻找合适的插入位置（按地址升序）          p = le2page(le, page\_link);          if (base + base->property <= p) {              assert(base + base->property != p);              break;          }          le = list\_next(le);      }      list\_add\_before(le, &(base->page\_link));    //向空闲链表中插入新的空闲块  #endif  }  练习2：  **get\_pte函数：**  // get\_pte - get pte and return the kernel virtual address of this pte for la  //         - if the PT contians this pte didn't exist, alloc a page for PT  //  parameter:  //   pgdir:  the kernel virtual base address of PDT  //   la:     the linear address need to map  //   create: a logical value to decide if alloc a page for PT  //  return vaule: the kernel virtual address of this pte  pte\_t \*  get\_pte(pde\_t \*pgdir, uintptr\_t la, bool create)  //此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表  // pgdir :页表起始地址  // uintptr\_t :线性地址，由于段式管理只做直接映射，所以它也是逻辑地址  {  #ifdef LAB2\_EX2      /\* LAB2 EXERCISE2: YOUR CODE       \*       \* If you need to visit a physical address, please use KADDR()       \* please read pmm.h for useful macros       \*       \* Maybe you want help comment, BELOW comments can help you finish the code       \*       \* Some Useful MACROs and DEFINEs, you can use them in below implementation.       \* MACROs or Functions:       \*   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.       \*   KADDR(pa) : takes a physical address and returns the corresponding kernel virtual address.       \*   set\_page\_ref(page,1) : means the page be referenced by one time       \*   page2pa(page): get the physical address of memory which this (struct Page \*) page  manages       \*   struct Page \* alloc\_page() : allocation a page       \*   memset(void \*s, char c, size\_t n) : sets the first n bytes of the memory area pointed by s       \*                                       to the specified value c.       \* DEFINEs:       \*   PTE\_P           0x001                   // page table/directory entry flags bit : Present       \*   PTE\_W           0x002                   // page table/directory entry flags bit : Writeable       \*   PTE\_U           0x004                   // page table/directory entry flags bit : User can access       \*/      // (1) find page directory entry 找到页表目录的入口（一级页表的表项）      pde\_t \*pdep = NULL;      pdep = pgdir + PDX(la); // pgdir给出页表起始地址；PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la      // (2) check if entry is not present      if (((\*pdep) & PTE\_P) == 0){    //if not present          // (3) check if creating is needed, then alloc page for page table          if (!create)              return NULL;          struct Page \*new\_pte = alloc\_page();    //为其申请新的物理页          if (!new\_pte)   return NULL;    //未分配到返回          // CAUTION: this page is used for page table, not for common data page          // (4) set page reference 设置页引用数          page\_ref\_inc(new\_pte);  //被引用数+1          // (5) get linear address of page          uintptr\_t pa = (uintptr\_t)page2kva(new\_pte); //page2kva()获取new\_pte管理的物理地址对应的内核虚拟地址          // (6) clear page content using memset          memset((void\*)pa, 0, PGSIZE);          // //kprintf("@@@ %x\n", pa);          // (7) set page directory entry's permission          \*pdep = PADDR(pa);   // 建立从一级页表到二级页表的映射;PADDR()内核虚拟地址转对应的物理地址          (\*pdep) |= (PTE\_U|PTE\_P|PTE\_W); //user writable present 的权限控制位设置      }      // (8) return page table entry      //如果原来就有二级页表，或者新建立了页表，则只需返回对应项的地址即可      pte\_t \*ret = (pte\_t\*)KADDR((uintptr\_t)((pte\_t\*)(PDE\_ADDR(\*pdep))+PTX(la))); //KADDR()物理地址转内核虚地址      //kprintf("@@GET\_PTE %x %x %x\n", \*pdep, ret, \*ret);      return  ret;// return page table entry  #endif  }  练习3：  **page\_remove\_pte函数：**  // page\_remove\_pte - free an Page sturct which is related linear address la  //                 - and clean(invalidate) pte which is related linear address la  // note: PT is changed, so the TLB need to be invalidate  static inline void  page\_remove\_pte(pde\_t \*pgdir, uintptr\_t la, pte\_t \*ptep)  {  //此函数释放某虚地址所在的页并取消对应二级页表项的映射  #ifdef LAB2\_EX3      /\* LAB2 EXERCISE3:       \*       \* Please check if ptep is valid, and tlb must be manually updated if mapping is updated       \*       \* Maybe you want help comment, BELOW comments can help you finish the code       \*       \* Some Useful MACROs and DEFINEs, you can use them in below implementation.       \* MACROs or Functions:       \*   struct Page \*page pte2page(\*ptep): get the according page from the value of a ptep       \*   free\_page : free a page       \*   page\_ref\_dec(page) : decrease page->ref. NOTICE: ff page->ref == 0 , then this page should be free.       \*   tlb\_invalidate(pde\_t \*pgdir, uintptr\_t la) : Invalidate a TLB entry, but only if the page tables being       \*                        edited are the ones currently in use by the processor.       \* DEFINEs:       \*   PTE\_P           0x001                   // page table/directory entry flags bit : Present       \*/      //(1) check if this page table entry is present      if (ptep && (\*ptep & PTE\_P))      {   //if present          //(2) find corresponding page to pte          struct Page \*page = pte2page(\*ptep); //查找二级页表项对应的页          //(3) decrease page reference          page\_ref\_dec(page); //减少其被引数          //(4) and free this page when page reference reachs 0          if (page\_ref(page)==0){              /\*由于一个物理页可能被映射到不同的虚拟地址上去（譬如一块内存在不同进程间共享），              \*当这个页需要在一个地址上解除映射时，操作系统不能直接把这个页回收，而是要先看看它还有没有映射到别的虚拟地址上。              \*这是通过查找管理该物理页的Page数据结构的成员变量ref（用来表示虚拟页到物理页的映射关系的个数）来实现的，              \*如果ref为0了，表示没有虚拟页到物理页的映射关系了，就可以把这个物理页给回收了，从而这个物理页是free的了，可以再被分配。              \*/              free\_page(page);    //被引用数为0，则可以对该页执行free操作          }          //(5) clear second page table entry          \*ptep = 0;  //取消对应二级页表项的映射      }      //(6) flush tlb      tlb\_invalidate\_all();   //冲刷快表  #endif  } | | | | | | |
| 四、实验结果及分析  练习1：  设计实现过程：  首先通过default\_init函数定义页表；然后通过default\_init\_memmap函数根据每个物理页帧的情况来建立空闲页链表，且空闲页块应该是根据地址高低形成一个有序链表；接着借助default\_alloc\_pages函数实现页的分配，借助default\_free\_pages函数实现页的释放，释放过程要注意特别连续空闲块的合并。详细实现过程已注释在源代码中。  first fit算法还有进一步的改进空间：  当前的first fit算法在空闲页表中寻找合适空间时采用遍历的方法，时间复杂度为θ(n)，而倘若为空闲页表建立对应的二叉树结构，则可将其时间复杂度降至θ()。  练习2：  设计实现过程：  练习2中通过get\_pte函数实现寻找虚拟地址对应的页表项，此函数首先找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表；如果原来就有二级页表，或者新建立了页表，则只需返回对应项的地址即可。详细实现过程已注释在源代码中。  描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处：  页目录项：  页表项：  练习3：  设计实现过程：  练习3中通过page\_remove\_pte函数实现释放某虚地址所在的页并取消对应二级页表项的映射，此函数首先查找二级页表项对应的页，减少其被引用数；若被引用数为0（一个物理页可能被映射到不同的虚拟地址上去），则对该页执行free操作；接着取消对应二级页表项的映射并冲刷快表。详细实现过程已注释在源代码中。  数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是什么？  有对应关系，数据结构Page的全局变量的每一项与物理页一一对应；而通过页表中的页目录项找到对应的页表项，对应的页表项又记录着对应的物理页的信息，故而页表中的页目录项和页表项指向的某一地址将也对应着数据结构Page的全局变量的某一项。  编译运行后输出结果： | | | | | | |
| 五、组内分工  徐小龙：完成练习一二的设计、代码、测试与报告撰写  陈鹏宇：完成练习三的设计、代码、测试与报告撰写 | | | | | | |