**《操作系统》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | |  | | | **姓名** |  |
| **实验题目** |  | | | | | |
| **实验时间** |  | | **实验地点** |  | | |
| **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  本实验需要补齐操作系统进行内存管理的代码。详细的说，主要需要实现三个功能的一系列函数，分别是内存的分配，虚拟地址和页表项的映射，地址的释放。  内存分配的方式采用第一适配法的连续物理内存分配，需要完成实现空闲区域页、空闲页面链表初始化，页面分配和页面释放。  虚拟地址和页表项的映射需要根据用户输入的虚拟地址la，通过pgdir对应的一级页目录表，检索到该虚拟地址对应的二级页表项的虚拟地址。若有创建页虚拟地址映射的需要（create=1），则在一级页表中若对应页表项存在位为0时，需要分配一个物理页建立映射关系。 | | | | | | |
| 二、实验项目内容 | | | | | | |
| 三、实验过程或算法（源程序） Ⅰ 全局关键变量 本实验中涉及一些关键变量，尤其是物理页结构体Page和内存中表示空闲区域的链表free\_list，有了这两个数据结构，ucore就可以管理起来整个以页为单位的物理内存空间。  作者认为有必要将一些关键变量和作用列举在实验过程开头，关键变量和含义如下表所示：  表格 1 关键变量和含义   |  |  | | --- | --- | | 关键变量 | 含义 | | Page | 物理页 | | free\_area\_t | 空闲区域 | | free\_list | 空闲链表表头 | | nr\_free | 空闲页总量 | | npage | 页表所需内存 | | freemem | 空闲内存起始地址 |   这些变量将在实验中多次出现，扮演者重要角色。  值得提到的是物理页的几个属性，物理页Page结构体见下：    图 1 Page结构体  其中，Page各个属性及含义如下表：  表格 2 Page各个属性及含义   |  |  | | --- | --- | | Page属性 | 含义 | | ref | 表示页面框架的**引用计数**。引用计数用于跟踪有多少个指针引用了该页面，以便在页面不再被使用时可以正确地释放。 | | flags | 是描述页面框架状态的一组标志位数组。这些标志位可能包括例如页面**是否被占用、是否被修改**等信息。 | | property | 在伙伴系统（buddy system）中使用，**存储连续内存块的阶数**（2^X中的X）。伙伴系统是一种内存管理算法，用于管理连续的内存块，将它们分成不同的大小，以便更高效地分配和释放内存。 | | zone\_num | 仍然与伙伴系统有关，表示页面所属的区域编号。 | | page\_link | 用于**将页面链接到空闲页面列表**。这是一个链表项，可能用于在页面被释放时将其链接到空闲页面链表中。 | | swap\_link | 与交换（swap）相关的**链表链接**。在操作系统中，交换是一种将不活动的页面移动到磁盘上的技术，以便释放内存供其他用途使用。 |  Ⅱ 实现 first-fit 连续物理内存分配算法2.1 算法概况 First-fit算法意思是：检索空闲物理内存链表（由地址从低到高），找到第一个连续物理内存大于或等于请求分配大小的区域，分配对应的物理页。  First-fit算法关键函数：   * default\_init: 初始化空闲页面链表 * default\_init\_memmap: 将freemem区域页设为空闲 * default\_alloc\_pages: 分配大小为n的页面，包括访问和修改空闲链表 * default\_free\_pages: 释放大小为n的指定页面  2.2 default\_init 重写 default\_init用于初始化空闲页面链表。需要涉及空闲页面链表分配及其长度设置。首先设置链表头部，然后指定长度为0，因为还没有空闲页。  代码如下：  static void  default\_init(void) {//初始化空闲页面链表  list\_init(&free\_list);//define the head of the list 初始化空闲页面链表，设置头部  nr\_free = 0;//capacity of list is 0，设置空闲链表长度为0  }  图示过程：   2.3 default\_init\_memmap重写 default\_init\_memmap用于将freemem区域页设为空闲。  它接受空闲内存起始地址base和需要设置空闲 的页帧数目n作为参数，在主程序中调用的例子是：    通过操作page结构体来设置空闲，并将空闲页放入空闲链表。  代码如下：  static void  default\_init\_memmap(struct Page \*base, size\_t n) {//将freemem区域页设为空闲(init\_memmap(pa2page(mbegin), (mend - mbegin) >> PGSHIFT );  #ifdef LAB2\_EX1  assert(n > 0); //duan yan  struct Page \*p = base;  for (; p != base + n; p ++) {//把从base指向的Page结构起的n个Page结构设定为空闲，初始化  assert(PageReserved(p));  p->flags = p->property = 0;  set\_page\_ref(p, 0);  }  base->property = n;  SetPageProperty(base);//由于从base开始之后有n个空闲页，所以说将base对应的property设为n，并把该位设为有效  nr\_free += n;  list\_add\_before(&free\_list, &(base->page\_link));//把base加入到free\_list中  #endif  }  图示过程：   2.3 default\_alloc\_pages重写  default\_alloc\_pages作用是分配大小为n的页面，包括访问和修改空闲链表。  它接受用户传来的需要分配页的大小n作为参数，按照first-fit算法，需要在空闲页链表中找到第一个页数不小于n的节点，将节点切分（若节点空闲页数大于请求），一部分分配给用户，另一部分重新插入空闲链表。  在具体重新插入空闲链表操作中，需要将找到的节点空闲页设置为【节点空闲页-n】，然后调用list\_add\_after将节点页的表链接插入空闲链表表头。  注意，插入空闲链表的是页的表链接page\_link，而非页本身。  代码如下：  static struct Page \*  default\_alloc\_pages(size\_t n) {//fen pei ye mian, need capacity: n，分配大小为n的页面，包括访问和修改空闲链表  #ifdef LAB2\_EX1  assert(n > 0);//确保n不会非正或超出范围  if (n > nr\_free) {  return NULL;  }  struct Page \*page = NULL;  list\_entry\_t \*le = &free\_list;  // TODO: optimize (next-fit)  while ((le = list\_next(le)) != &free\_list) {//查找一个空闲页数大于或等于请求大小的结点  struct Page \*p = le2page(le, page\_link);  if (p->property >= n) {  page = p;  break;  }  }  if (page != NULL) {  if (page->property > n) {//如果结点的空闲页大于请求，则需要将他未分配页对应的page作为表头插入到free\_list  struct Page \*p = page + n;  p->property = page->property - n;  SetPageProperty(p);  list\_add\_after(&(page->page\_link), &(p->page\_link));  }  list\_del(&(page->page\_link));//删除原先的结点，更新空闲页数目，将page的property置为无效  nr\_free -= n;  ClearPageProperty(page);  }  return page;  #endif  }  图示过程为（以分配4个空闲页为例）：   2.4 default\_free\_pages重写 default\_free\_pages作用是释放大小为n的指定页面。  该函数接受指定页面地址base和要释放的页面数目n作为参数，首先使用断言确保释放页面数目大于0，然后将base开始的n个待释放页面flag和property清零。  之后对空闲链表的过程是分配页面的逆过程，但需要考虑页面合并的问题——如果有与释放页连续的空闲页则将其flag的property有效位置为0，并从free\_list中删除，然后修改base为合并页面的首地址，并修改合并页面的property为合并后页面数目。  由于空闲链表需要保持地址依次增大的性质，因此需要遍历空闲链表找到合适的插入位置插入。  代码如下：  static void  default\_free\_pages(struct Page \*base, size\_t n) {//释放大小为n的指定页面  #ifdef LAB2\_EX1  assert(n > 0);  struct Page \*p = base;  for (; p != base + n; p ++) { //更新回收的Page的flag、引用计数  assert(!PageReserved(p) && !PageProperty(p));  p->flags = 0;  set\_page\_ref(p, 0);  }  base->property = n; //将这段连续内存的第一个Page结构property设为n，并将其对应flag位置1  SetPageProperty(base);  list\_entry\_t \*le = list\_next(&free\_list);  while (le != &free\_list) { //遍历free\_list的Page结构，看是否有与释放页连续的空闲页。如果有，则将其flag的property有效位置为0，并从free\_list中删除。  p = le2page(le, page\_link);  le = list\_next(le);  // TODO: optimize  if (base + base->property == p) {  base->property += p->property;  ClearPageProperty(p);  list\_del(&(p->page\_link));  }  else if (p + p->property == base) {  p->property += base->property;  ClearPageProperty(base);  base = p;  list\_del(&(p->page\_link));  }  }  nr\_free += n;  le = list\_next(&free\_list);  while (le != &free\_list) { //在free\_list中寻找base页插入的位置，使free\_list保持地址依次增大的性质。最终将其插入free\_list中。  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));  #endif  } 2.5 first fit算法改进 Next-fit算法实现页面查找分配：  Next-fit相比于first-fit，其优点主要在于减少了较多的无效查询。因为在first-fit下，每次一次分配请求时，都会从空闲链表头开始从前往后扫描，找到第一个符合要求的页表然后返回。而next-fit则不一样，next-fit是每一次扫描后，他不会回到链表头，而是从上一次扫描到的位置从后扫描，所以减少了较多的无效扫描的次数，当然这也会导致可能会有更多的外部碎片。  下面就是next-fit实现的代码：  static struct Page \*  default\_alloc\_pages(size\_t n) {  #ifdef LAB2\_EX1      assert(n > 0); // 确保页面数量大于0      if (n > nr\_free) {          // 如果请求的页面数量超过当前可用的空闲页面数量，则打印错误消息并返回空指针          // kprintf("wrong, the size > max\_free\_size\n");          return NULL;      }      static list\_entry\_t \*last = &free\_list; // 保存上次分配的位置      list\_entry\_t \*le = last; // 从上次分配的位置开始      struct Page \*page = NULL;      do {          le = list\_next(le);          if (le == &free\_list) {              le = list\_next(le); // 回到链表开头          }          struct Page \*p = le2page(le, page\_link);          if (p->property >= n) {              page = p;              last = list\_prev(le); // 保存当前分配的位置              break;          }      } while (le != last); // 循环一次链表      if (page != NULL) {          if (page->property > n) {              // 如果当前页面的可用页面数量大于所需数量，则从当前页面之后分割出一块新的页面              struct Page \*p = page + n;              p->property = page->property - n;              SetPageProperty(p);              list\_add\_after(&(page->page\_link), &(p->page\_link));          }          // 从空闲页面链表中删除该页面          list\_del(&(page->page\_link));          // 减少空闲页面数量          nr\_free -= n;          // 清除页面属性          kprintf("get here\n");          ClearPageProperty(page);      }      return page;  #endif  }  下面就是next-fit实现的代码的具体解释和分析：    使用一个静态变量存储last指针，初始化为空闲链表头节点。（这里的空闲链表是一个双向循环链表）    上述代码就是具体实现的查找到可用页的循环体。可以看到，在le指针查找一圈回到last前，该循环体都能正常查找。并且注意下述代码，是本次next-fit代码的关键：    这个记录的时候last必须回到当前找到的le前一个位置 的指针地址，理由如下：    因为le本来指向的位置会因为分配出去而被删掉，如果记录的是这个位置，那么这个位置会导致下一次查找时last指的位置为null，从而报错。所以le回到前一个，前一个并不会删掉，并且该函数删除时会自动将下一个接上去，所以由此实现了next-fit的功能。 Ⅲ 实现寻找虚拟地址对应的页表项3.1 算法概述 操作系统虚拟内存的实现方式是通过地址映射，将程序的虚拟地址空间映射到物理内存。这允许程序使用比实际物理内存更大的地址空间，提高内存利用率。  虚拟内存管理包括分页机制，将虚拟地址划分为固定大小的页面，并使用页表建立虚拟地址到物理地址的映射。由于现如今电脑内存通常很大达到接近4GB，如果只建立一级页表需要过大的空间存储页表本身，所以引出了分级页表。  通常分级页表包含两级：页目录项（Page Directory Entry）和页表项（Page Table Entry），在源代码中简称PDE和PTE。通过get\_pte和get\_pde将虚拟内存映射到二级页表项虚地址和物理地址。本实验要求实现get\_pte，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。  Get\_pte函数调用关系如下：    图 2 get\_pte函数调用关系 3.2 get\_pte重写 Get\_pte函数接受一级页表的地址pgdir，虚地址la和分配页标志create作为参数，主要通过宏定义来获取一级页表索引值pdep，在后续二级页表访问中，需要根据pdep找到二级页表的起始地址，并且根据la中二级页表索引值加上二级页表基址得到二级页表中存放的la的物理地址。  宏定义查找地址都是基本都是使用位偏移的方式，操作系统预先定义好一级和二级页表在地址中有效字段的偏移量，在获取页表索引值的时候直接和虚拟地址进行移位运算。例如获取虚拟地址一级页表索引值，宏定义一级页表偏移PDXSHIFT为22，通过PDX(la) ((((uintptr\_t)(la)) >> PDXSHIFT) & 0x3FF来获取一级页表索引值。  如果一级页表项页表项((\*pdep)&PTE\_P) == 0，表示其存在位为0，如果create=0表示没找到地址则直接返回null，如果create为1，表示要在一级页表上创建一个页，则分配一个页，将一级页表项指向它。  代码如下：  get\_pte(pde\_t \*pgdir, uintptr\_t la, bool create) {  #ifdef LAB2\_EX2  //检索一级页表，pgdir为一级页表地址，PDX宏定义为获取la虚拟地址的一级页表索引值。  pde\_t \*pdep = NULL; // (1) find page directory entry 页表入口  pdep = pgdir + PDX(la);//虚拟地址la的页目录项索引    if( ((\*pdep)&PTE\_P) == 0 ){ //如果此页表项存在位为0且create=1，则分配一个页，将一级页表项指向它。  if(!create){ //不分配叶  return NULL;  }  struct Page\* new\_pte = alloc\_page();  if(!new\_pte){  return NULL; //无剩余可用页  }  page\_ref\_inc(new\_pte);//增加新分配的页面的引用计数，因为它将用于存储页表项。  uintptr\_t pa = (uintptr\_t)page2kva(new\_pte);  memset((void\*)pa,0,PGSIZE);  (\*pdep) |= (PTE\_U|PTE\_P|PTE\_W);  }  pte\_t \*ret = (pte\_t\*)KADDR((uintptr\_t)((\*pte\_t)(PDE\_ADDR(\*pdep))+PTX(la)));  return ret;  #endif  } 3.3 页目录项和页表项中每个组成部分的含义以及对ucore而言的潜在用处 页表项包含了诸多标志位，用于控制页表项的行为和用户使用页表项的规则，具体标志位和含义如下表：  表格 3 页标志含义以及对ucore而言的潜在用处   |  |  |  | | --- | --- | --- | | 标志位 | 含义 | 对ucore的作用 | | PTE\_P | 页表项是否存在 | 表示页面已经被映射到物理内存 | | PTE\_W | 页表项是否可写 | 表示允许对页表项进行写操作 | | PTE\_U | 页表项是否可由用户空间访问 | 表示用户态程序可以访问这个页面 | | PTE\_PWT | 启动write-through | 表示写操作需要写cache中页面的同时写内存中页面 | | PTE\_PCD | 禁用cache | 表示不允许使用缓存 | | PTE\_A | 是否被访问过 | 可以用于处理器调用，适用于一些换入换出内存策略 | | PTE\_D | 脏位 | 标志页面被修改过，适用于一些换入换出内存策略和写策略 | | PTE\_PS | 是否启用大页面 | 使用更高级的大页面页表结构，可以提高TLB命中率，减少页表项数量 | | PTE\_MBZ | 指示一些位必须保持为0 | 为了将来的扩展和兼容性 | | PTE\_AVAIL | 给操作系统或软件使用 | 提高页表个性化能力 | | | | | | | |
| 四、实验结果及分析 | | | | | | |

实验报告填写说明：

1、第一、二部分由老师提供；

2、第三部分填写源程序和算法，源程序要符合程序编写风格（缩进、注释等）；

3、第四部分主要填写程序结果（截图）、解决问题的方法、总结和体会等；

4、报告规范：包含报告页眉、报告的排版、内容是否填写，命名是否规范等。

5、源程序和实验报告命名：学号姓名序号.py 学号姓名序号.docx，例如学号20181234的张三同学，**他的第一次实验命名为：20181234张三1.py和20181234张三1.docx**