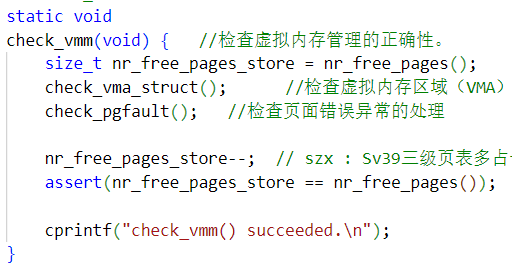
练习一

1、

1. check\_vmm()函数：这个函数用于验证虚拟内存管理系统的正确性。以确保VMA结构正确、页面错误被正确处理，并且空闲页数量的变化是准确的。



nr\_free\_pages\_store：它存储了初始时空闲页的数量。

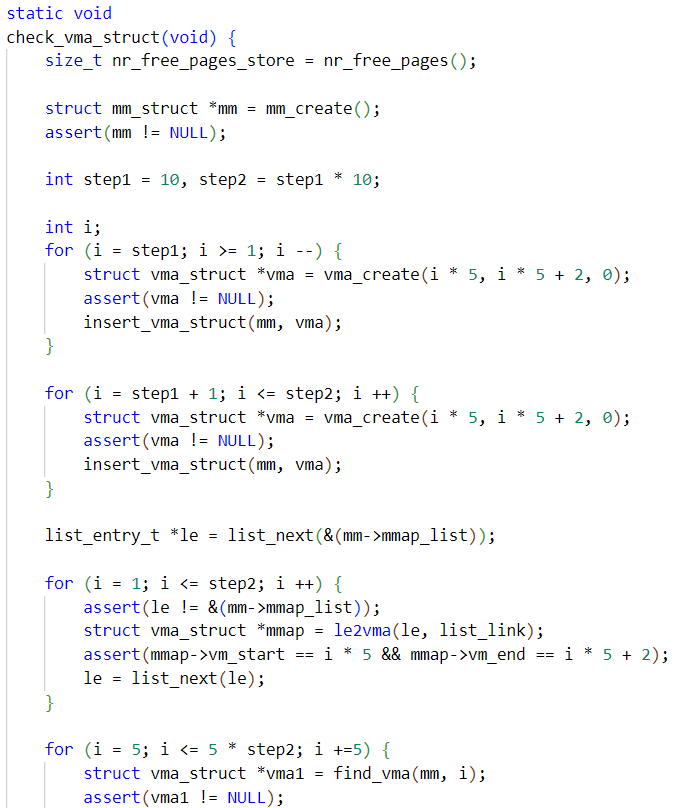
check\_vma\_struct()：这是一个函数调用，可能检查虚拟内存区域（VMA）的结构或状态。VMA用于管理进程地址空间中的内存区域。

check\_pgfault()：这个函数可能检查页面错误异常的处理。页面错误在程序请求当前不在物理内存中的内存时发生。

nr\_free\_pages\_store--：减少了空闲页的数量。注释表明这可能与特定情况有关（例如Sv39三级页表可能占用了额外的内存页）。

assert(nr\_free\_pages\_store==nr\_free\_pages())：一个断言，验证在第5步操作后，空闲页的数量保持准确。确保操作没有意外地影响了空闲页的数量。

1. check\_vma\_struct()：检查虚拟内存区域（VMA）结构的函数。它验证了对VMA结构进行创建、插入和查找等操作的正确性

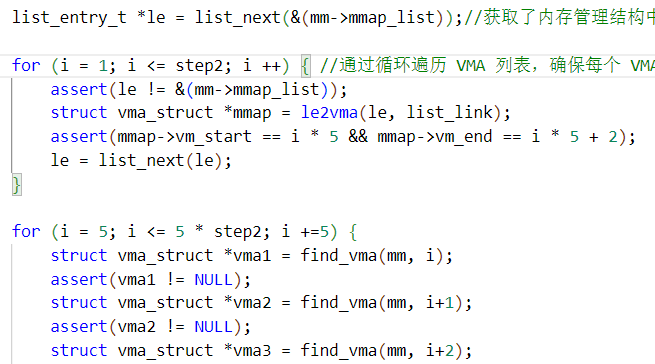


nr\_free\_pages\_store存储了空闲页的数量。

structmm\_struct\*mm=mm\_create();创建了一个内存管理结构，并确保其创建成功。

step1和step2是步长值，用于循环创建VMA结构。

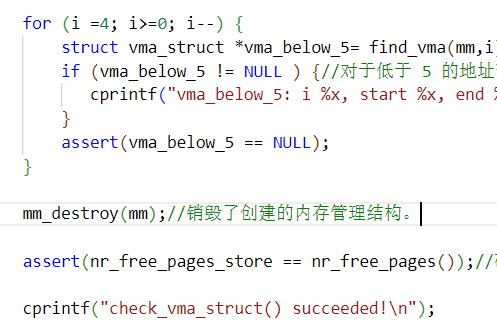
通过循环创建了多个VMA结构，每个结构有不同的起始地址和结束地址，并将它们插入到内存管理结构中。



list\_entry\_t\*le=list\_next(&(mm->mmap\_list));获取了内存管理结构中VMA列表的第一个元素。

通过循环遍历VMA列表，确保每个VMA的起始地址和结束地址设置正确。

通过一系列的find\_vma()函数调用来查找特定地址范围内的VMA，并确保它们的起始和结束地址是正确的。同时，也确保某些地址范围外没有对应的VMA。



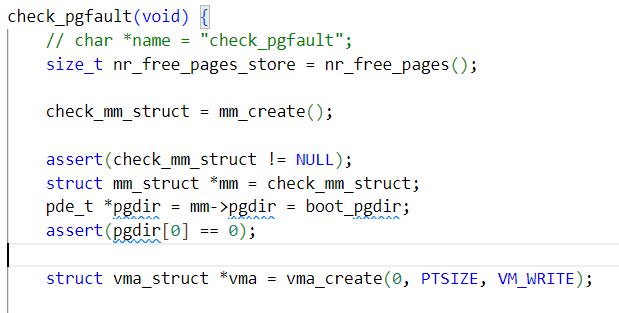
最后，对于低于5的地址范围，确保find\_vma()返回为空（即对应的VMA不存在）。

mm\_destroy(mm);销毁了创建的内存管理结构。

最后的assert(nr\_free\_pages\_store==nr\_free\_pages())确保了在执行过程中没有影响到空闲页的数量。

如果所有的断言通过，将会打印一条成功消息check\_vma\_struct()succeeded!。

1. check\_pgfault（）:检查页面错误异常处理



nr\_free\_pages\_store存储了空闲页面的数量。

check\_mm\_struct=mm\_create();创建了一个内存管理结构check\_mm\_struct，并确保创建成功。

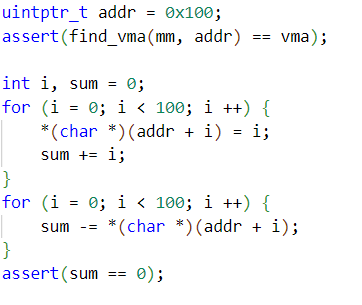
assert(check\_mm\_struct!=NULL);用来确保内存管理结构的创建没有问题。

通过structmm\_struct\*mm=check\_mm\_struct;将这个新创建的内存管理结构指定给mm变量。

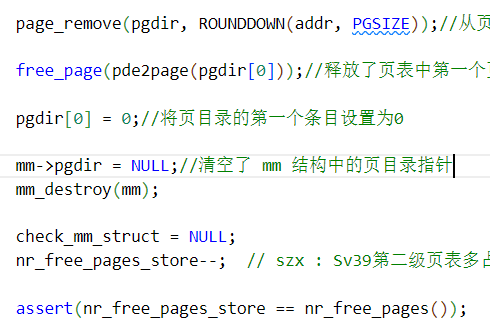
pde\_t\*pgdir=mm->pgdir=boot\_pgdir;将引导页目录boot\_pgdir分配给了pgdir。它验证了一个索引为0的页目录项的值为0。

structvma\_struct\*vma=vma\_create(0,PTSIZE,VM\_WRITE);创建了一个VMA结构，代表了一个可写的内存区域。

insert\_vma\_struct(mm,vma);将这个VMA结构插入到内存管理结构中。



接下来，代码对从地址0x100开始的一段内存进行写入和读取，以及对数据的计算和检查，确保读写过程正确。



page\_remove(pgdir,ROUNDDOWN(addr,PGSIZE));从页表中移除特定地址对应的页面。

free\_page(pde2page(pgdir[0]));释放了页表中第一个页面的内存。

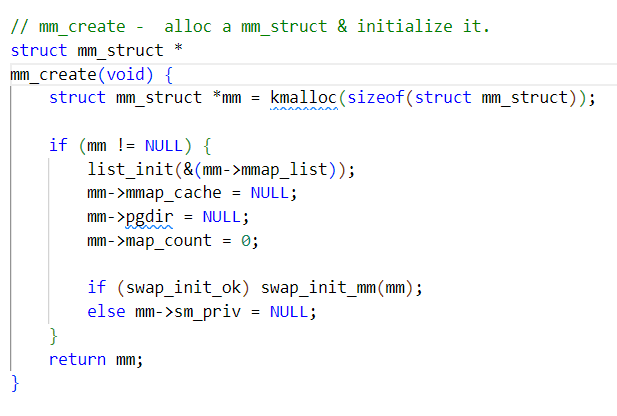
pgdir[0]=0;将页目录的第一个条目设置为0。

接着清空了mm结构中的页目录指针，并销毁了整个mm结构。

最后，对空闲页面数进行了调整，并确保内存管理操作不会影响到空闲页面数。

如果所有的断言通过，将会打印一条成功消息check\_pgfault()succeeded!。

(4)mm\_create()：用于创建一个structmm\_struct类型的内存管理结构。创建一个新的内存管理结构，并进行各种初始化工作，包括了初始化链表、页目录指针、计数器等，以及根据系统是否启用了交换功能，来决定是否初始化交换相关的数据结构



structmm\_struct\*mm=kmalloc(sizeof(structmm\_struct));这行代码使用kmalloc分配了足够的内存以存储一个structmm\_struct结构，并将其指针赋给mm变量。

if(mm!=NULL){这是一个条件语句，检查内存分配是否成功。如果mm不为空（即内存分配成功），则进入条件语句的代码块。

list\_init(&(mm->mmap\_list));这行代码初始化了mm结构中的mmap\_list，这是一个链表结构，用于存储虚拟内存区域（VMA）结构。

mm->mmap\_cache=NULL;将mmap\_cache设置为NULL，这可能是用于缓存VMA结构的指针。

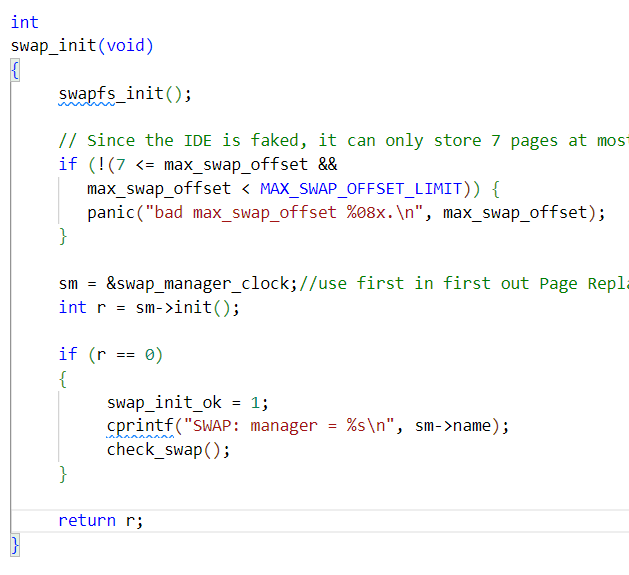
mm->pgdir=NULL;将pgdir设置为NULL，这是用于指向页目录的指针。

mm->map\_count=0;将map\_count初始化为0，可能用于跟踪VMA结构的数量。

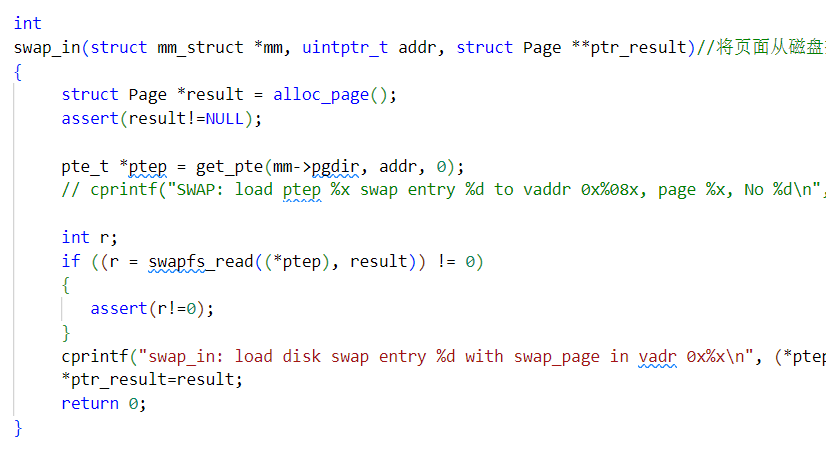
if(swap\_init\_ok)swap\_init\_mm(mm);这段代码是一个条件语句，根据swap\_init\_ok的状态来决定调用swap\_init\_mm()函数或者将mm->sm\_priv设置为NULL。这可能是用于管理交换空间的初始化。

最后，函数返回新创建的mm结构。

1. swap\_init():初始化操作系统或内存管理系统中的交换机制。通过设置交换文件系统、验证配置参数的有效性、初始化选定的交换管理器，并执行检查确认成功初始化，来初始化交换系统。



(6)swap\_in:将页面从磁盘换回到内存中



structPage\*result=alloc\_page();：分配一页内存页面用于存储从磁盘读取的数据。这个函数确保页面分配成功。

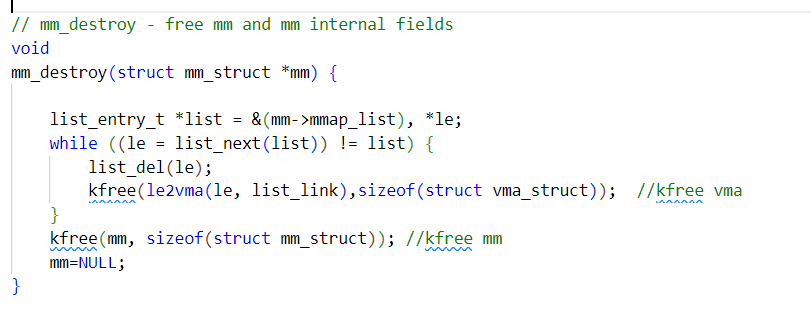
pte\_t\*ptep=get\_pte(mm->pgdir,addr,0);：获取页表项对应的地址。函数根据给定的地址addr和内存管理结构mm中的页表pgdir来获得对应的页表项。

intr;if((r=swapfs\_read((\*ptep),result))!=0){...}：使用swapfs\_read从磁盘读取内容到先前分配的内存页面result中。如果读取操作失败，会触发断言。读取操作将页面内容从磁盘交换到result所指的物理页面。

\*ptr\_result=result;：将读取的页面指针传递给指向指针的指针ptr\_result。

返回值为0表示页面从磁盘成功换回到内存中。

（7）mm\_destroy：释放一个进程的虚拟地址空间



获取进程 mm 的虚拟地址空间块链表。遍历进程的所有VMA。

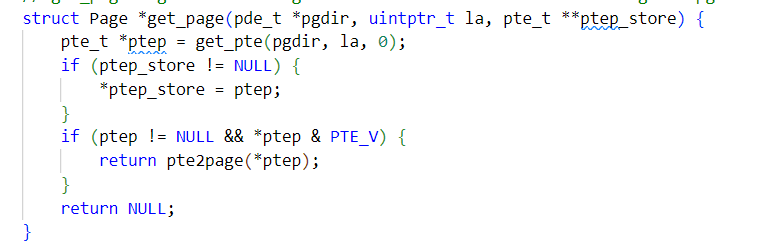
链表中移除当前VMA的链表条目。

使用 kfree 函数释放当前的VMA结构。

释放进程的 mm\_struct 结构。

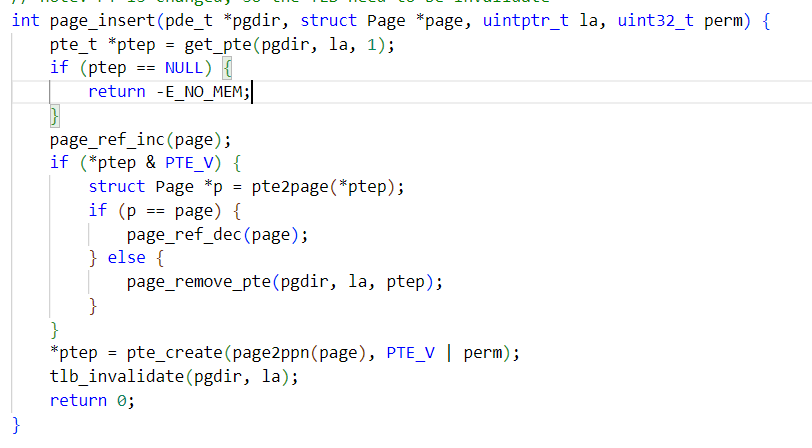
mm=NULL;:将 mm 指针设置为 NULL，表示该内存区域已被释放

（8）get\_page：从最终的页表项里面,可以找到一个物理地址，即最终的物理页.

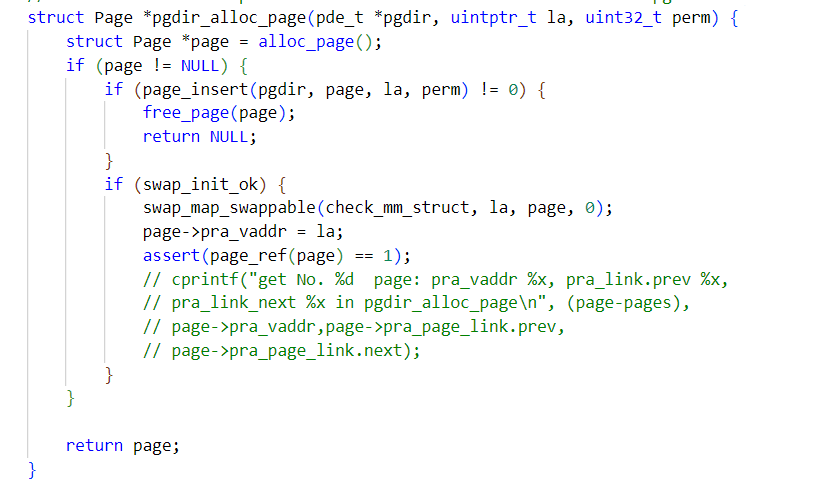


（9）page\_insert

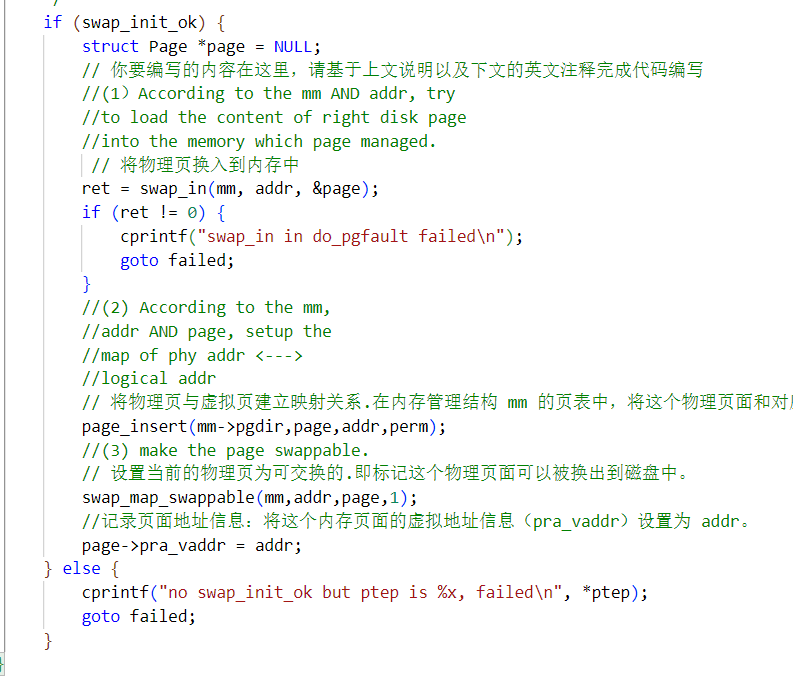
用于将一个物理页面映射到一个虚拟地址la上



（10）pgdir\_alloc\_page：为给定的线性地址la分配一个物理页面，并在页目录pgdir中设置相应的映射。



练习三



将物理页换入到内存中

ret=swap\_in(mm,addr,&page);

if(ret!=0){

cprintf("swap\_inindo\_pgfaultfailed\n");

gotofailed;

}

将物理页与虚拟页建立映射关系.在内存管理结构 mm 的页表中，将这个物理页面和对应的虚拟地址 addr 建立映射关系。

page\_insert(mm->pgdir,page,addr,perm);

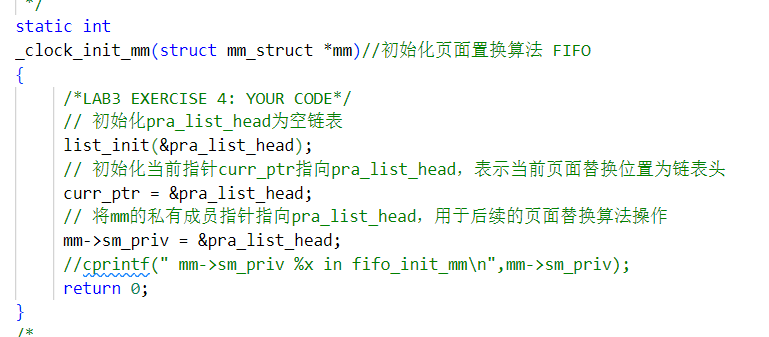
设置当前的物理页为可交换的.即标记这个物理页面可以被换出到磁盘中。

swap\_map\_swappable(mm,addr,page,1);

记录页面地址信息：将这个内存页面的虚拟地址信息（pra\_vaddr）设置为 addr。

page->pra\_vaddr = addr;

练习四



初始化pra\_list\_head为空链表

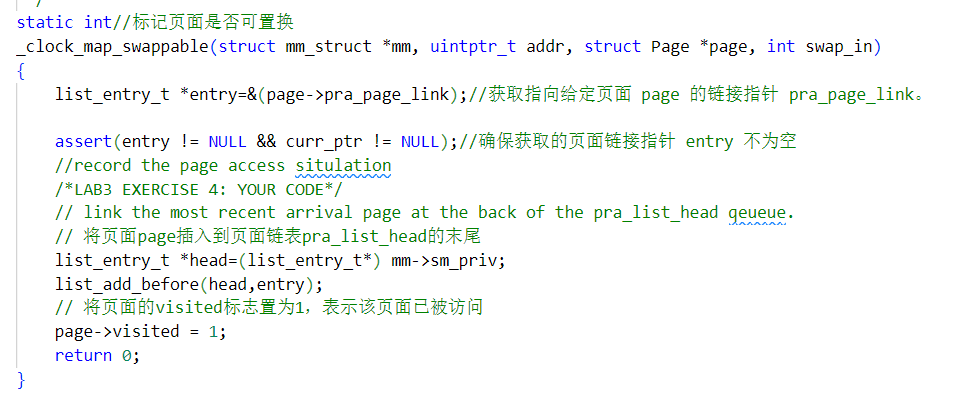
   list\_init(&pra\_list\_head);

   初始化当前指针curr\_ptr指向pra\_list\_head，表示当前页面替换位置为链表头

   curr\_ptr=&pra\_list\_head;

   将mm的私有成员指针指向pra\_list\_head，用于后续的页面替换算法操作

   mm->sm\_priv=&pra\_list\_head;



list\_entry\_t\*entry=&(page->pra\_page\_link);

获取指向给定页面page的链接指针pra\_page\_link。

 assert(entry!=NULL&&curr\_ptr!=NULL);

确保获取的页面链接指针entry不为空

 将页面page插入到页面链表pra\_list\_head的末尾

list\_entry\_t\*head=(list\_entry\_t\*)mm->sm\_priv;

list\_add\_before(head,entry);

 将页面的visited标志置为1，表示该页面已被访问

  page->visited=1;

structPage\*page=le2page(curr\_ptr,pra\_page\_link);

获取当前指针curr\_ptr对应的页面结构的指针，以便检查页面是否已被访问。

list\_entry\_t\*next\_entry=list\_next(curr\_ptr);

if(page->visited==0)

如果当前页面未被访问

list\_del(curr\_ptr);

将其从页面链表中删除（即置换出去）

\*ptr\_page=page;

将其指针赋值给ptr\_page以进行替换

curr\_ptr=next\_entry;

然后将当前指针curr\_ptr移至下一个页面的位置

如果当前页面已经被访问过

page->visited=0;

curr\_ptr=list\_next(curr\_ptr);