西北工业大学 操作系统实验 实验报告

班号：10012006　　姓名：夏卓　　学号：2020303245

实验日期：2022/11/6　　实验名称：分页存储管理与虚拟内存

一、实验目的

了解虚拟存储器管理设计原理，掌握分页虚拟存储管理的具体实现技术。

二、实验要求

1. 按照实验讲义中的设计要求，实现虚拟内存和页面调度，给出关键函数的代码以及实验结果。

2. 利用 Project4 中的应用程序 rec.exe 产生缺页中断，并在系统中适当位置添加代码输出该缺页中断。

3.回答课后思考题。第十二章第 3、5、6 题。

三、实验过程及结果

本项目的主要目标是实现分页存储与虚拟内存技术，需要注意的是内核存储空间与用户进程存储空间的不同之处。

第一个任务，添加一个 InitVM()函数实现内存页目录和页表的初始化，以及启动分页式管理模式的操作，从而在线性地址与物理地址之间建立起映射关系，需要注意的是，GeekOS 中，所有内核级线程共用一个线性地址空间，即所有内核线程只有一个页目录。

为了建立页表，需要先分配一个内核页目录：

```
//计算物理内存的页数
whole_pages = bootInfo->memSizeKB / 4;

//计算内核页目录中要多少个目录项，才能完全映射所有的物理内存页。
kernel_pde_entries = whole_pages / NUM_PAGE_DIR_ENTRIES
                   + (whole_pages % NUM_PAGE_DIR_ENTRIES == 0 ? 0 : 1);

//为内核页目录分配一页空间
g_kernel_pde = (pde_t *)Alloc_Page();
KASSERT(g_kernel_pde != NULL);

//将页中所有位清0
memset(g_kernel_pde, 0, PAGE_SIZE);
```

然后再为整个存储区间的内容分配页表，并设置页目录项和页表项相应标志位。需要注意的是对于内核的存储空间对应的表项应该标识为非 VM_USER，以至于只有核心程序访问它们才是合法的：

```
for (i = 0; i < kernel_pde_entries - 1; i++)
{
    // 设置页目录标志位
    cur_pde_entry->present = 1;
    cur_pde_entry->flags = VM_WRITE;
    cur_pde_entry->globalPage = 1;
    // 为页表分配内存并初始化
    cur_pte = (pte_t *)Alloc_Page();
    KASSERT(cur_pte != NULL);
    memset(cur_pte, 0, PAGE_SIZE);
    cur_pde_entry->pageTableBaseAddr = PAGE_ALLIGNED_ADDR(cur_pte);
    // 设置页表标志位
    for (j = 0; j < NUM_PAGE_TABLE_ENTRIES; j++)
    {
        cur_pte->present = 1;
        cur_pte->flags = VM_WRITE;
        cur_pte->globalPage = 1;
        cur_pte->pageBaseAddr = mem_addr >> 12;
        cur_pte++;
        mem_addr += PAGE_SIZE;
    }
```

最后调用 Enable_Paging()使分页系统生效，并注册页故障处理程序 Page_Fault_Handler，这个程序的具体实现，放在虚拟内存管理的实现中一起做：

```
// 从现在开始，系统的寻址必须经过分页机制转换，以前仅仅经过分段机制转换
Enable_Paging(g_kernel_pde);
// 注册页故障处理程序
Install_Interrupt_Handler(14, Page_Fault_Handler);
```

第二个任务是为用户进程创建它自己的线性地址空间。需要注意的是，对于用户进程而言，每一个用户进程都有一个独立的线性地址空间，即要为它们分别创建页目录和页表。

用户进程的页目录既包含将用户线性地址映射到物理地址的表项，也包含将内核线性地址映射到物理地址的表项。具体来说，一个用户进程的存储格式如下所示：

因此我们首先需要更改 Load_User_Program 函数，使得在加载用户进程时为进程分配一页，用于保存页目录，并复制线性存储空间的低 2GB 的所有核心页目录项到用户进程的页目录中：

```
//处理分页涉及的数据
pde_t *pageDirectory;
pageDirectory = (pde_t *)Alloc_Page();
if (pageDirectory == NULL)
{
    Print("no more page!\n");
    return -1;
}
memset(pageDirectory, 0, PAGE_SIZE);
//将内核页目录复制到用户态进程的页目录中
memcpy(pageDirectory, g_kernel_pde, PAGE_SIZE);
```

然后，我们需要分别为用户进程的代码段、数据段、参数段和堆栈区间分配内存页，分配并填充页表项。在设置表项时需要注意确保页目录项和页表项的标志位为用户可以访问模式（即包含 VM _ USER 标志）：

```
for (i = 0; i < exeFormat->numSegments - 1; i++)
{
    startAddress = exeFormat->segmentList[i].startAddress;
    sizeInMemory = exeFormat->segmentList[i].sizeInMemory;
    offsetInFile = exeFormat->segmentList[i].offsetInFile;
    lengthInFile = exeFormat->segmentList[i].lengthInFile;
    if (!sizeInMemory && !lengthInFile)
    {
        sizeInMemory = DEFAULT_STACK_SIZE;
        lengthInFile = DEFAULT_STACK_SIZE;
    }

    if (startAddress + sizeInMemory < USER_VM_LEN)
    {
        //给数据段和代码段分配空间
        if (!Alloc_User_Page(pageDirectory, startAddress + USER_VM_START, sizeInMemory) ||
            !Copy_User_Page(pageDirectory, startAddress + USER_VM_START,
                            exeFileData + offsetInFile, lengthInFile))
            return -1;
    }
    else
    {
        Print("startAddress+sizeInMemory > 2GB in Load_User_Program\n");
        return -1;
    }
```

```
Get_Argument_Block_Size(command, &args_num, &arg_size);
if (arg_size > PAGE_SIZE)
{
    Print("Argument Block too big for one PAGE_SIZE\n");
    return -1;
}
//给参数块在地址空间的尾部分配一页
arg_addr = Round_Down_To_Page(USER_VM_LEN - arg_size);
char *block_buffer = Malloc(arg_size);
KASSERT(block_buffer != NULL);
Format_Argument_Block(block_buffer, args_num, arg_addr, command);

if (!Alloc_User_Page(pageDirectory, arg_addr + USER_VM_START, arg_size) ||
    !Copy_User_Page(pageDirectory, arg_addr + USER_VM_START, block_buffer, arg_size))
    return -1;
Free(block_buffer);
```

```
//给堆栈在地址空间的尾部分配一页
stack_addr = USER_VM_LEN - Round_Up_To_Page(arg_size) - DEFAULT_STACK_SIZE;
if (!Alloc_User_Page(pageDirectory, stack_addr + USER_VM_START, DEFAULT_STACK_SIZE))
{
    return -1;
}
```

接着处理 UserContext 中涉及分段机制的选择子,描述符等结构：

```
// UserContext中涉及分段机制的选择子,描述符
uContext->ldtDescriptor = Allocate_Segment_Descriptor();
if (uContext->ldtDescriptor == NULL)
{
    Print("allocate segment descriptor fail\n");
    return -1;
}

Init_LDT_Descriptor(uContext->ldtDescriptor, uContext->ldt, NUM_USER_LDT_ENTRIES);
uContext->ldtSelector = Selector(USER_PRIVILEGE, true,
                                 Get_Descriptor_Index(uContext->ldtDescriptor));

//注意，在GeekOS的分页机制下，用户地址空间默认从线性地址2G开始
Init_Code_Segment_Descriptor(&uContext->ldt[0], USER_VM_START,
                             USER_VM_LEN / PAGE_SIZE, USER_PRIVILEGE);
Init_Data_Segment_Descriptor(&uContext->ldt[1], USER_VM_START,
                             USER_VM_LEN / PAGE_SIZE, USER_PRIVILEGE);

uContext->csSelector = Selector(USER_PRIVILEGE, false, 0);
uContext->dsSelector = Selector(USER_PRIVILEGE, false, 1);
```

最后填充 UserContext 的信息：

```
//填充UserContext
uContext->entryAddr = exeFormat->entryAddr;
uContext->argBlockAddr = arg_addr;
uContext->size = USER_VM_LEN;
uContext->stackPointerAddr = stack_addr;
*pUserContext = uContext;
```

第三个任务是实现虚拟内存管理技术。虚拟存储可以把内存里的页暂时存储到磁盘上，并释放物理存储空间供其他进程使用。为此，操作系统将需要在磁盘设备上创建一个 pagefile 文件暂时保存从内存中替换出去的页，并实现一个类 LRU 算法在内存中选取一个替换页把它写到磁盘的 pagefile 文件中。

为此，我们首先添加一个 Init_Pagefile 函数，利用链表或 Bitmap 数据结构对 pagefile 磁盘块进行管理，方便对磁盘块的分配和释放：

```
void Init_Pagefile(void)
{
    // TODO("Initialize pagefile data structures");
    pagingDevice = Get_Paging_Device();
    if (pagingDevice == NULL)
        return;
    numPagingDiskPages = pagingDevice->numSectors / SECTORS_PER_PAGE;
    //为pagefile中每一页设置标示位
    Bitmap = Create_Bit_Set(numPagingDiskPages);
}
```

另外，编写一个函数 Alloc_Pageable_Page 实现交换一页到磁盘的一操作，具体执行步骤为：

调用 Find_Page_To_Page_Out 函数来确定要替换的页：

```
/* Select a page to steal from another process */
Debug("About to hunt for a page to page out\n");
page = Find_Page_To_Page_Out();
KASSERT(page->flags & PAGE_PAGEABLE);
paddr = (void *)Get_Page_Address(page);
Debug("Selected page at addr %p (age = %d)\n", paddr, page->clock);
```

调用 Find_Space_On_Paging_File 函数在 pagefile 中找到空闲的存储空间：

```
/* Find a place on disk for it */
pagefileIndex = Find_Space_On_Paging_File();
if (pagefileIndex < 0)
    /* if No space available in paging file. */
    goto done;
Debug("Free disk page at index %d\n", pagefileIndex);
```

调用 Write_To_Paging_File 函数把被替换的页写到 pagefile 文件中，并修改页表的相应表项，清除页存在的标志，标识为此页在内存中不存在，修改表项的 kernelInfo 标志，标识为此页是在磁盘中存在而非无效。

```
/* Write the page to disk. Interrupts are enabled, since the I/O may block. */
Debug("Writing physical frame %p to paging file at %d\n", paddr, pagefileIndex);
Enable_Interrupts();
Write_To_Paging_File(paddr, page->vaddr, pagefileIndex);
Disable_Interrupts();
```

```
if (page->flags & PAGE_ALLOCATED)
{
    /* The page is still in use update its bookeping info */
    /* Update page table to reflect the page being on disk */
    page->entry->present = 0;
    page->entry->kernelInfo = KINFO_PAGE_ON_DISK;
    page->entry->pageBaseAddr = pagefileIndex; /* Remember where it is located! */
}
```

之后调用 Flush_TLB 函数来刷新页表缓存 TLB。

最后一个任务是完成页故障处理函数。当进程尝试访问一个非法地址时，会发生页故障，通过中断系统，最终调用页故障处理程序来处理缺页情况。下图总结了缺页处理程序应该做的工作：

表 12-3 缺页处理表

缺页情况	标识	相应处理
堆栈生长到新页	faultCode.writeFault	分配一个新页进程继续
此页保存在磁盘上	page_entry->kernelInfo== KINFO_PAGE_ON_DISK	从 pagefile 读入需要的页继续
因为无效地址缺页	非法地址访问	终止用户进程

对于写错误，缺页情况为堆栈生长到新页上，则分配一个新的页正常返回：

```
//写错误，缺页情况为堆栈生长到新页
if (faultCode.writeFault)
{
    // Print("write Fault\n");
    int res;
    if (!Alloc_User_Page(userContext->pageDir, Round_Down_To_Page(address), PAGE_SIZE))
    {
        Print("Alloc_User_Page error in Page_Fault_Handler\n");
        Exit(-1);
    }
    return;
}
```

对于读错误，则先要找到虚拟地址对应的页表项：

```
//读错误，分两种缺页情况
Print("read fault\n");
//先找到虚拟地址对应的页表项
ulong_t page_dir_addr = PAGE_DIRECTORY_INDEX(address);
ulong_t page_addr = (address << 10) >> 22;
pde_t *page_dir_entry = (pde_t *)userContext->pageDir + page_dir_addr;
pte_t *page_entry = NULL;
```

如果是非法地址访问的缺页情况，则终止用户进程，直接退出：

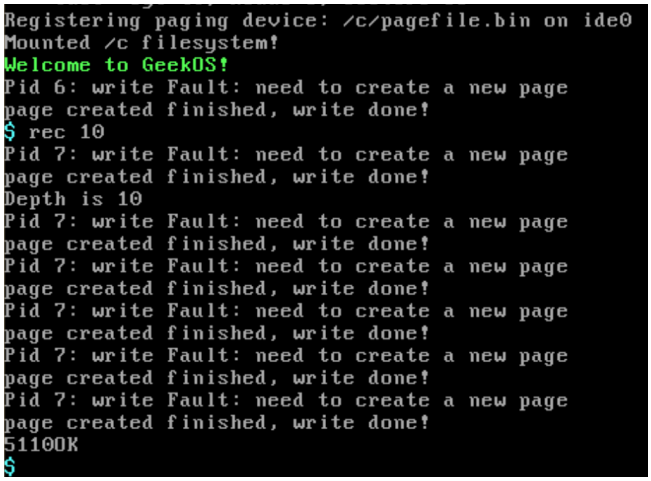
```
if (page_entry->kernelInfo != KINFO_PAGE_ON_DISK)
{
    //非法地址访问的缺页情况
    Print_Fault_Info(address, faultCode);
    Exit(-1);
}
```

否则如果是因为页保存在磁盘 pagefile 而引起的缺页，则从页面文件中把页读到内存中：

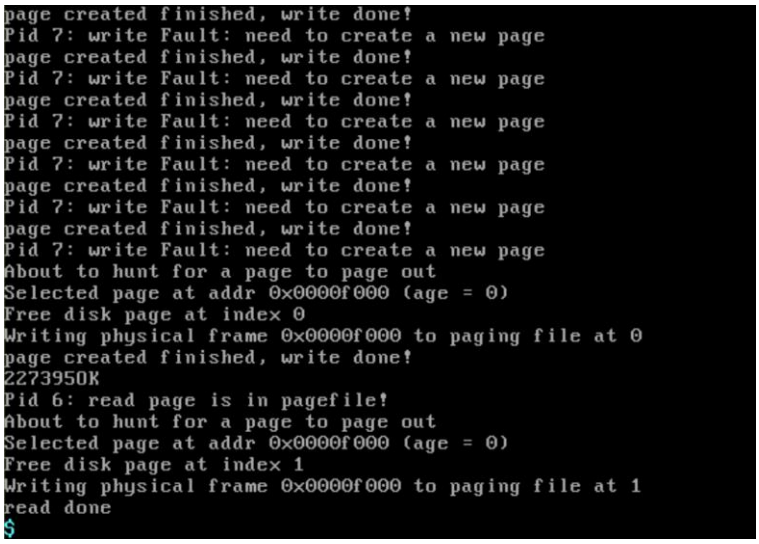
```
//因为页保存在磁盘pagefile引起的缺页
int pagefile_index = page_entry->pageBaseAddr;
void *paddr = Alloc_Pageable_Page(page_entry, Round_Down_To_Page(address));
if (paddr == NULL)
{
    Print("no more page/n");
    Exit(-1);
}

*((uint_t *)page_entry) = 0;
page_entry->present = 1;
page_entry->flags = VM_WRITE | VM_READ | VM_USER;
page_entry->globalPage = 0;
page_entry->pageBaseAddr = PAGE_ALLIGNED_ADDR(paddr);
//从页面文件中把页读到内存中
Enable_Interrupts();
Read_From_Paging_File(paddr, Round_Down_To_Page(address), pagefile_index);
Disable_Interrupts();
//释放页面文件中的空间
Free_Space_On_Paging_File(pagefile_index);
return;
```

其余需要用到的函数实现较为简单，不再此处详述。
至此，我们就实现了一个具有分页存储管理与虚拟内存的操作系统。

实验结果如下所示：
rec 10：



rec 445：



rec 450：

```
Pid 7: read page is in pagefile!
About to hunt for a page to page out
Selected page at addr 0x0000f000 (age = 0)
Free disk page at index 4
Writing physical frame 0x0000f000 to paging file at 4
read done
Pid 7: write Fault: need to create a new page
About to hunt for a page to page out
Selected page at addr 0x00029000 (age = 0)
Free disk page at index 3
Writing physical frame 0x00029000 to paging file at 3
page created finished, write done!
11Pid 7: read illegal address!
Pid 7, Page Fault received, at address 80000013 (0 pages free)
   Non-present page, Read Fault, in User Mode
Pid 7 Exit!
Pid 6: read page is in pagefile!
About to hunt for a page to page out
Selected page at addr 0x00029000 (age = 0)
Free disk page at index 5
Writing physical frame 0x00029000 to paging file at 5
read done
Pid 6: read page is in pagefile!
read done
$
```

四、实验分析

首先我们回顾一下在分页存储系统中，一个给定的线性地址时如何映射到物理内存页上的：

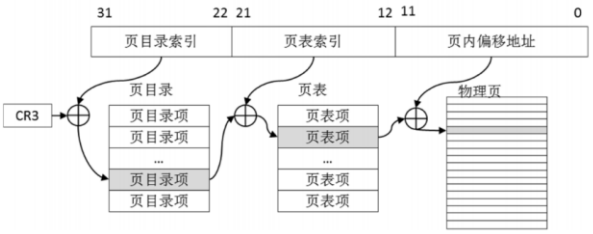


图 12-3 线性地址到物理地址的转换

如图可以看出，CR3 的高 20 位作为页目录所在物理页的页码可以找到进程对应的页目录位置。首先把线性地址的最高 10 位（即位 22～31）作为页目录的索引，对应表项所包含的页码指定页表，然后，再把线性地址的中间 10 位（即位 12～21）作为所指定的页目录中的页表的索引，对应表项所包含的页码指定物理地址空间中的一页；最后，把所指定的物理页的页码作为高 20 位，把线性地址的低 12 位不加改变地作为 32 位物理地址的低 12 位。

然后我们来分析一下本次实验的结果，容易看出，三张截图分别对应了三种页故障的发生以及处理结果。

第一张截图中，递归次数较少，出现了堆栈生长到新页上的缺页情况，相应处理也很简单，直接为该进程分配了一个新页，然后返回继续执行。

第二张截图中，递归次数较多，出现了内存容量不足，导致无法分配新页的情况，于是操作系统将某一页表换出后释放空间，从而能够创造新页，这体现了虚拟内存的思想；另外，在程序返回后，还发生了所需要读入的页在 pagefile 中的缺页情况，相应的处理过程为从 pagefile 中读入需要的页返回继续执行。

第三张截图中，出现了非法地址访问的缺页情况，相应的处理方法为直接终止用户进程并退出，因此切换到了进程 6（shell.c），并又产生了所需要读入的页在 pagefile 中的缺页情况，进行相应处理后退出。

五、所遇问题及解决方法

实验过程中，出现了以下错误：

```
Pid 7: read page is in pagefile!
About to hunt for a page to page out
Selected page at addr 0x0000f000 (age = 0)
Free disk page at index 83
Writing physical frame 0x0000f000 to paging file at 83
Failed assertion in Read_From_Paging_File: !(page->flags & PAGE_PAGEABLE) at ../
src/geekos/paging.c, line 363, RA=1dd9b, thread=0x00039000
```

根据错误提示可知错误发生在从 pagefile 中读取文件时违反了页面不可被交换的断言。之所以出现这种情况是因为 GeekOS 是可抢占式内核，而启用了虚拟内存后，用户进程的内存页面随时可能被交换到磁盘文件。如果在内核和用户进程间拷贝数据时出现进程切换，且新进程因为缺页导致原进程的内存页被换出，则当原进程重新运行时就有可能读写错误的数据，从而导致内存错误。因此，内核不能在允许进程切换的情况下读写任何带有可能会被换出的页面。

因此解决思路就是在读写数据前清除页面的 PAGE_PAGEABLE 标志，暂时将该页面锁定，当读写结束后再恢复该标志：

```
int fl = 1;
if (page->flags & PAGE_PAGEABLE)
{
    fl = 0;
    page->flags |= ~PAGE_PAGEABLE;
}
```

```
if (fl == 0)
    page->flags |= PAGE_PAGEABLE;
Print("read done\n");
```

六、思考与练习

1. 1 页的大小被定义为 4K 是基于什么考虑的？

在分页系统中，若选择过小的页面大小，虽然一方面可以减小内存碎片，起到减少内存碎片总空间的作用，有利于内存利用率的提高，但另一方面却会造成每个进程占用较多的页面，从而导致进程的页表过长，占用大量内存。此外，还会降低页面换进换出的效率。

然而，如果选择的页面过大，虽然可以减少页表的长度，提高页面换进换出的速度，但却又会使页内碎片增大。因此，页面的大小应选择适中，且页面大小应是 2 的幂，通常为 1 KB ~8 KB 。

因此，根据经验并结合上述分析综合来看，4KB 大小的页面是一个适中且合适的选择，能够在几个问题之间相对取得最优。

2. 如何能减少页面替换的次数？

置换算法的好坏直接影响页面替换的次数，一个不适当的算法可能会导致进程发生“抖动”，即刚被换出的页很快又要被访问，需要将它重新调入，此时又需要再选一页调出；而此刚被调出的页很快又被访问，又需将它调入。如此频繁地更换，显然不是一个好的置换算法。

从理论上讲，应将那些以后不再会访问的页面换出，或把那些在较长时间内不会再访问的页面调出。目前已有多种置换算法，它们都试图更接近于理论上的目标，如：最佳页面置换算法(OPT)、先进先出置换算法(FIFO)、最近最久未使用的置换算法(LRU)、最不常用置换算法(LFU)、时钟页面置换算法(Lock)。

我们可以根据不同的应用程序与应用场景选择不同的替换算法。一般来说，使用 LRU 算法，即选择最近最久未使用的页面予以淘汰，可使得页面替换次数变少，获得

较好的结果。

3. 如何加快页面替换的速度？

为了加快页面替换的速度，我们可以从以下两个方面来考虑。

第一是加快写回磁盘的频率。如建立一个已修改换出页面的链表，当一个页面要被换出时，可暂不把它写回磁盘，而是将它们挂在已修改换出页面的链表上，仅当被换出页面数目达到一定值时，再将它们一起写回到磁盘上，从而减少磁盘 I/O 的操作次数。

第二是加快读入内存的频率。在设置了已修改换出页面链表后，在该链表上就暂时有一批装有数据的页面，如果有进程在这批数据还未写回磁盘时需要再次访问这些页面时，就不需从外存上调入，而直接从已修改换出页面链表中获取，这样也可以减少将页面从磁盘读入内存的频率，减少页面换进的开销。

实现这两点的算法有页面缓冲算法 PBA，其在内存中设置了两个链表，分别为空闲页面链表和修改页面链表，原理与上述过程类似。