《操作系统》课程实验报告

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | 物理存储器与进程逻辑地址空间的管理 | | | 实验序号 | 5 | 实验日期 | 5.9 |
| 姓 名 | 倪镭 | 院系 | 信息学院 | 班 级 | 23计科2 | 学 号 | 2352420 |
| 专 业 | 计算机科学与技术 | | | 指导教师 | 王静 | 成 绩 |  |
| 教师评语 |  | | | | | | |
| **一、实验目的和要求**  1．通过查看物理存储器的使用情况，并练习分配和回收物理内存，从而掌握物理存储器的管理方法。  2．通过查看进程逻辑地址空间的使用情况，并练习分配和回收虚拟内存，从而掌握进程逻辑地址空间的管理方法。 | | | | | | | |
| **二、实验预习内容**  1．阅读本书第 6 章。重点阅读第 6.3节和第 6.6节，了解物理存储器的管理方式和进程逻辑地址空间的管理方式。 | | | | | | | |
| **三、实验项目摘要（给出3.2-3.6标题，并附上运行截图）**  1．            1 | | | | | | | |
| **四、实验结果与分析**  **1．完成“3.3 分配物理页和释放物理页”中对MiAllocateAnyPages和MiFreePages函数的单步调试过程，并回答问题。**   1. 按F10单步调试MiAllocateAnyPages函数的执行过程中，尝试回答下面的问题   (1) 本次分配的物理页的数量是多少?分配的物理页的页框号是多少?      物理页数量：8176  物理页的页框号：0x408  (2) 物理页是从空闲页链表中分配的?还是从零页链表中分配的?  从空闲页链表中分配的   1. 哪一行语句减少了空闲页的数量?哪一行语句将刚刚分配的物理页由空闲状态修改为忙状态? MiFreePageCount--; 这行语句减少了空闲页的数量，每执行一次该语句，空闲页数量就减 1 。   MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE; 这行语句将刚刚分配的物理页由空闲状态修改为忙状态   1. 绘制 MiAllocateAnyPages 函数的流程图。   开始  |  v  判断分配条件：  判断 NumberOfPages <= MiFreePageCount + MiZeroedPageCount 是否成立  |  |--是---> 从空闲页链表分配  |  |--否---> 分配失败，返回 STATUS\_NO\_MEMORY  |  从空闲页链表分配：  初始化 i = 0  |  循环（当 i < NumberOfPages 且 MiFreePageCount > 0 时）  |  |--获取空闲页链表头的页框号：Pfn = MiFreePageListHead  |  |--更新空闲页链表头：MiFreePageListHead = MiGetPfnDatabaseEntry(Pfn)->Next  |  |--减少空闲页数量：MiFreePageCount--  |  |--将获取的物理页状态修改为忙：MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE  |  |--将页框号存入：PfnArray[i] = Pfn  |  |--i++  |  循环结束  |  判断是否需从零页链表分配：  判断 i < NumberOfPages 是否成立  |  |--是---> 从零页链表分配  |  |--否---> 分配成功，返回 STATUS\_SUCCESS  |  从零页链表分配：  循环（当 i < NumberOfPages 时）  |  |--获取零页链表头的页框号：Pfn = MiZeroedPageListHead  |  |--更新零页链表头：MiZeroedPageListHead = MiGetPfnDatabaseEntry(Pfn)->Next  |  |--减少零页数量：MiZeroedPageCount--  |  |--将获取的物理页状态修改为忙：MiGetPfnDatabaseEntry(Pfn)->PageState = BUSY\_PAGE  |  |--将页框号存入：PfnArray[i] = Pfn  |  |--i++  |  循环结束  |  分配成功，返回 STATUS\_SUCCESS  |  结束   1. 按F10单步调试MiFreePages函数的执行过程，回答下面的问题:   (1) 本次释放的物理页的数量是多少?释放的物理页的页框号是多少?释放的物理页是之前分 配的物理页吗?    1 0x49 是  (2) 释放的物理页是被放入了空闲页链表中?还是零页链表中?  释放的物理页被放入了空闲页链表中   1. 绘制 MiFreePages 函数的流程图。   开始  |  v  检查待释放物理页：  初始化 i = 0  |  循环（当 i < NumberOfPages 时）  |  |--获取当前要检查的物理页页框号：Pfn = PfnArray[i]  |  |--判断 Pfn >= MiTotalPageFrameCount || MiGetPfnDatabaseEntry(Pfn)->PageState != BUSY\_PAGE 是否成立  | |  | |--是---> 执行 ASSERT(FALSE); ，返回 STATUS\_MEMORY\_NOT\_ALLOCATED ，流程结束  | |  | |--否---> i++ ，继续循环  |  循环结束  |  修改物理页状态并插入链表：  初始化 i = 0  |  循环（当 i < NumberOfPages 时）  |  |--获取当前要处理的物理页页框号：Pfn = PfnArray[i]  |  |--将物理页状态修改为空闲：MiGetPfnDatabaseEntry(Pfn)->PageState = FREE\_PAGE  |  |--把当前物理页链接到空闲页链表头部：MiGetPfnDatabaseEntry(Pfn)->Next = MiFreePageListHead  |  |--更新空闲页链表头：MiFreePageListHead = Pfn  |  |--增加空闲页数量：MiFreePageCount += 1  |  |--i++ ，继续循环  |  循环结束  |  结束：返回 STATUS\_SUCCESS ，表示物理页释放成功，流程结束   1. 修改 pm 命令的源代码，尝试在调用 MiAllocateAnyPages 函数时分配多个物理页， 然后在调用 MiFreePages 函数时将分配的多个物理页释放，给出修改后代码和运行结果截图。   PRIVATE  VOIDConsoleCmdPhysicalMemory(  IN HANDLE StdHandle  ){  BOOL IntState;  // 调整数组大小以存储多个页框号，这里设置为3，对应分配3个物理页  ULONG\_PTR PfnArray[3];  ULONG\_PTR numPagesToAllocate = 3; // 定义要分配的物理页数量  IntState = KeEnableInterrupts(FALSE); // 关中断    //  // 输出物理页数量和物理内存数量（以字节为单位）  //  fprintf(StdHandle, "Page Count: %d.\n", MiTotalPageFrameCount);  fprintf(StdHandle, "Memory Count: %d \* %d = %d Byte.\n",  MiTotalPageFrameCount, PAGE\_SIZE,  MiTotalPageFrameCount \* PAGE\_SIZE);    //  // 输出零页数量和空闲页数量  //  fprintf(StdHandle, "\nZeroed Page Count: %d.\n", MiZeroedPageCount);  fprintf(StdHandle, "Free Page Count: %d.\n", MiFreePageCount);    //  // 输出已使用的物理页数量  //  fprintf(StdHandle, "\nUsed Page Count: %d.\n", MiTotalPageFrameCount - MiZeroedPageCount - MiFreePageCount);      //////////////////////////////////////////////////////////////////////////  // 分配多个物理页  //  MiAllocateAnyPages(numPagesToAllocate, PfnArray);    fprintf(StdHandle, "\n\*\*\*\*\*\* After Allocate %d Pages \*\*\*\*\*\*\n", numPagesToAllocate);  fprintf(StdHandle, "Zeroed Page Count: %d.\n", MiZeroedPageCount);  fprintf(StdHandle, "Free Page Count: %d.\n", MiFreePageCount);  fprintf(StdHandle, "Used Page Count: %d.\n", MiTotalPageFrameCount - MiZeroedPageCount - MiFreePageCount);      //////////////////////////////////////////////////////////////////////////  //  // 然后再释放这些物理页  //  MiFreePages(numPagesToAllocate, PfnArray);    fprintf(StdHandle, "\n\*\*\*\*\*\* After Free %d Pages \*\*\*\*\*\*\n", numPagesToAllocate);  fprintf(StdHandle, "Zeroed Page Count: %d.\n", MiZeroedPageCount);  fprintf(StdHandle, "Free Page Count: %d.\n", MiFreePageCount);  fprintf(StdHandle, "Used Page Count: %d.\n", MiTotalPageFrameCount - MiZeroedPageCount - MiFreePageCount);    KeEnableInterrupts(IntState); // 开中断}    **2．在本实验 3.3 中，在分配物理页时是调用的内核函数 MiAllocateAnyPages，该函数会优先分配空闲页， 尝试修改代码，调用内核函数 MiAllocateZeroedPages 优先分配零页，并调试分配零页的情况。尝试从 性能的角度分析内核函数 MiAllocateAnyPages 和 MiAllocateZeroedPages。尝试从安全性的角度分析分 配零页的必要性。**  STATUS status;  ULONG\_PTR PfnArray[1];  status = MiAllocateZeroedPages(1, PfnArray);if (status == STATUS\_SUCCESS) {  // 分配成功后的操作，可添加输出分配信息等  fprintf(StdHandle, "Successfully allocated a zeroed page.\n");} else {  // 分配失败处理  fprintf(StdHandle, "Failed to allocate a zeroed page with status code 0x%x.\n", status);} 从性能角度分析MiAllocateAnyPages和MiAllocateZeroedPagesMiAllocateAnyPages**性能分析**：该函数优先从空闲页链表分配物理页。如果空闲页数量充足，分配过程相对简单直接，只需从空闲页链表中取出页框号，修改页状态等操作，开销较小，能快速完成分配，性能较好。但当空闲页链表不足时，需要再从零页链表分配，增加了额外的查找和分配步骤，可能会导致一定的性能损耗。MiAllocateZeroedPages**性能分析**：此函数首先从零页链表分配。若零页链表有足够的页，分配过程也较为直接。然而，当零页链表不足时，需要从空闲页链表分配并对分配的页进行清零操作（将物理页映射到系统 PTE 区域进行清零 ）。清零操作会增加额外的内存写操作，这在一定程度上会消耗更多时间和系统资源，相比MiAllocateAnyPages在空闲页充足时的分配操作，性能可能会有所下降。但如果系统中零页资源丰富，且对分配的物理页要求初始状态为零的场景下，该函数能满足需求且性能也可接受。从安全性角度分析MiAllocateAnyPages和MiAllocateZeroedPages **MiAllocateAnyPages安全性分析**：该函数分配的物理页，若从空闲页链表获取，其初始内容可能是之前使用留下的残留数据。在一些对数据保密性要求高的场景下，这些残留数据可能会带来安全风险，比如残留数据包含敏感信息等。**MiAllocateZeroedPages安全性分析**：该函数分配的物理页，无论是从零页链表获取还是从空闲页链表获取后进行清零操作，都能保证分配到的物理页初始状态为零。这在一定程度上避免了因物理页残留数据带来的安全隐患，对于安全性要求较高的场景（如涉及敏感数据处理的系统 ）更为适用。  **3．完成“3.5 在系统进程中分配虚拟页和释放虚拟页”中，对MmAllocateVirtualMemory和MmFreeVirtualMemory的调试过程，并回答问题。**   1. 按F10单步调试MmAllocateVirtualMemory函数的执行过程，尝试回答下面的问题: 2. 分配的虚拟页的起始地址是多少?分配的虚拟页的数量是多少?它们和参数BaseAddress和 RegionSize 初始化的值有什么样的关系?   0x0000000 1 RegionSize 决定了分配虚拟内存的大小 。BaseAddress 是分配虚拟内存的起始地址    1  (2) 分配虚拟页的同时有为虚拟页映射实际的物理页吗?这是由哪个参数决定的?  分配虚拟页时不一定会为虚拟页映射实际的物理页，这由分配虚拟内存时使用的标志参数决定。在 MmAllocateVirtualMemory 函数中，若使用 MEM\_RESERVE 标志（如代码中所示），只是预留虚拟地址空间，不会映射实际物理页；若使用 MEM\_COMMIT 标志，在分配虚拟页的同时会为其映射实际的物理页。  (3) 分配的虚拟页是在系统地址空间(高 2G)还是在用户地址空间(低 2G)?这是由哪个参数  决定的?  分配的虚拟页是在系统地址空间还是用户地址空间，一般由调用分配函数的上下文（进程环境）以及系统相关设置决定。在 Windows 等操作系统中，通常用户模式进程分配的虚拟内存默认在用户地址空间（低 2G ，对于 32 位系统常见情况 ），内核模式下分配的虚拟内存可能在系统地址空间（高 2G ）。在 MmAllocateVirtualMemory 函数调用中，没有直接单一参数明确指定在哪个地址空间分配，更多取决于调用进程的权限和运行模式 。  (4) 参考 MiReserveAddressRegion 函数的定义和注释，说明该函数的功能。  在虚拟地址空间中预留一段地址区域   1. 按F10单步调试MmFreeVirtualMemory函数的执行过程，尝试回答下面的问题:   (1) 本次释放的虚拟地址是多少?释放的虚拟页是之前分配的虚拟页吗?  是 是  (2) 参考MiFindReservedAddressRegion 函数、MiFreeAddressRegion 函数和 MiDecommitPages函数的定义和注释，说明这些函数的功能。  在指定的虚拟地址描述符（VAD ）链表（Pas->VadList ）中，查找是否存在从指定起始地址（\*BaseAddress ）开始、大小为指定值（\*RegionSize ）的已保留地址区域 。  **MiFreeAddressRegion 函数：**该函数用于释放已保留的地址区域。  **MiDecommitPages 函数：**它用于释放映射在连续虚拟页框上的物理页框   1. 按照下列要求修改 ConsoleCmdVM 函数的源代码，加深对虚拟页分配和释放过程的理解。（选做其一，仅保留选做题目、运行结果和对应分析） 2. 尝试在调用MmAllocateVirtualMemory函数时将RegionSize参数的值设置为PAGE\_SIZE+1或者 PAGE\_SIZE\*2+1。观察“输出”窗口中转储的信息，并说明申请虚拟内存的大小与实际分配的大 小之间的关系，以及分配的虚拟内存大小会对分配的虚拟地址产生什么样的影响。将“输出”窗 口中转储的信息保存在文本文件中。 3. **代码修改** 在 ConsoleCmdVM 函数中，找到调用 MmAllocateVirtualMemory 函数的地方，将 RegionSize 参数的值修改为 PAGE\_SIZE + 1 或者 PAGE\_SIZE \* 2 + 1 。修改后代码类似如下   SIZE\_T RegionSize = PAGE\_SIZE + 1;  PVOID BaseAddress = 0;  Status = MmAllocateVirtualMemory(&BaseAddress, &RegionSize, MEM\_RESERVE, TRUE);  **申请虚拟内存大小与实际分配大小关系**：在虚拟内存管理中，内存分配通常以页为单位进行对齐。系统会将申请的内存大小向上取整到页大小的整数倍。例如，若 PAGE\_SIZE 为 4KB（4096 字节 ），申请 PAGE\_SIZE + 1 （即 4097 字节 ），实际分配会是 8192 字节（2 页 ）；申请 PAGE\_SIZE \* 2 + 1 （即 8193 字节 ），实际分配也会是 12288 字节（3 页 ）。这是因为系统为了管理方便和提高内存访问效率，按照页边界来分配虚拟内存 。  **对分配虚拟地址的影响**：分配的虚拟地址是按照页边界对齐的。起始地址会是页大小的整数倍。分配操作会从合适的页边界开始分配所需的页数，以满足申请的内存大小需求。输出信息中会显示实际分配的起始地址和大小，通过观察可以验证地址和大小的页对齐特性 。  **信息保存** 在程序中添加代码将 “输出” 窗口信息保存到文本文件。可以使用标准 C 库的文件操作函数，例如：  c  #include <stdio.h>  FILE \*fp = fopen("output.txt", "w");if (fp!= NULL) {  BaseAddress, RegionSize);  fprintf(fp, "New VM's base address: 0x%X. Size: 0x%X.\n\n", BaseAddress, RegionSize);  fclose(fp);}  这样就能将相关信息保存到当前目录下的 output.txt 文件中。 | | | | | | | |

注：空间不够，可以增加页码。