合肥工業大學

操作系统实验报告

实验题目	实验 8 实验环境的使用
学生姓名	孙淼
学　　号	2018211958
专业班级	计算机科学与技术 18-2 班
指导教师	田卫东
完成日期	12.15

合肥工业大学 计算机与信息学院

1. 实验目的和任务要求

- 学习 i386 处理器的二级页表硬件机制，理解分页存储器管理原理。
- 查看 EOS 应用程序进程和系统进程的二级页表映射信息，理解页目录和页表的管理方式。
- 编程修改页目录和页表的映射关系，理解分页地址变换原理。

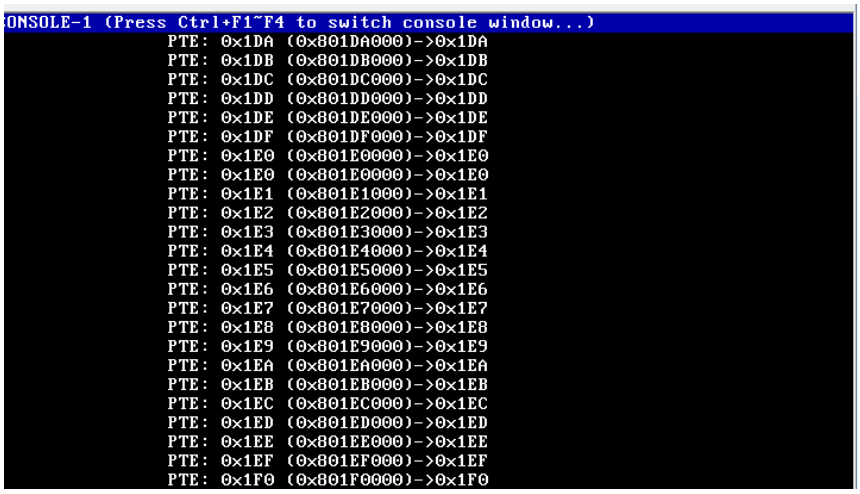
2. 实验原理

阅读本书第 6 章。了解 i386 处理器的二级页表硬件机制，EOS 操作系统的分页存储器管理方式，以及进程地址空间的内存分布。

3. 实验内容

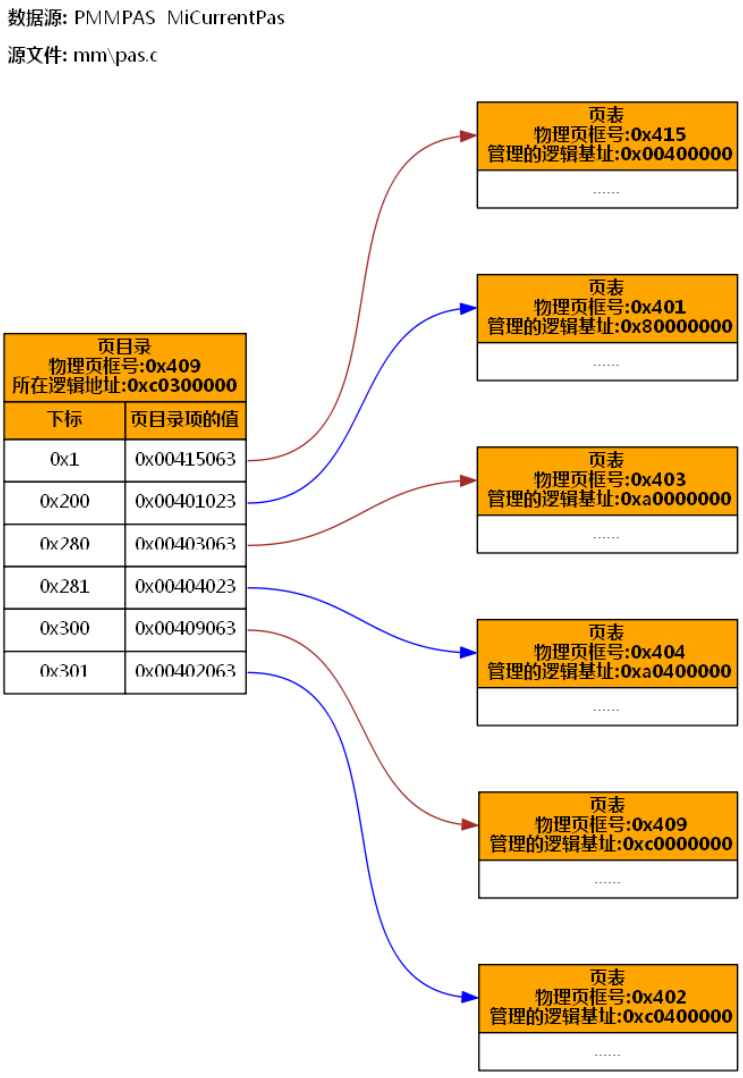
查看 EOS 应用程序进程的页目录和页表，打开“学生包”中本实验对应的文件夹，使用 OS Lab 打开其中的 memory.c 和 getcr3.asm 文件（将 文件拖动到 OS Lab 窗口中释放即可打开）。仔细阅读这两个文件中的源代码和注释，其中 main 函数的流 程图可以参见图 3-1。然后按照下面的步骤查看 EOS 应用程序进程的页目录和页表：

1. 使用 memory.c 文件中的源代码替换之前创建的 EOS 应用程序项目中 EOSApp.c 文件中的源代码。
2. 右键点击“项目管理器”窗口中的“源文件”文件夹节点，在弹出的快捷菜单中选择“添加”中 的“添加新文件”。
3. 在弹出的“添加新文件”对话框中选择“asm 源文件”模板。
4. 在“名称”中输入文件名称“func”。
5. 点击“添加”按钮添加并自动打开文件 func.asm。
6. 将 getcr3.asm 文件中的源代码复制到 func.asm 文件中。
7. 按 F7 生成修改后的 EOS 应用程序项目。
8. 在 main 函数的返回代码处（第 190 行）添加一个断点。
9. 按 F5 启动调试。EOS 启动完毕后会自动运行应用程序，将应用程序进程的页目录和页表打印输出 到屏幕上，然后在刚刚添加的断点处中断。

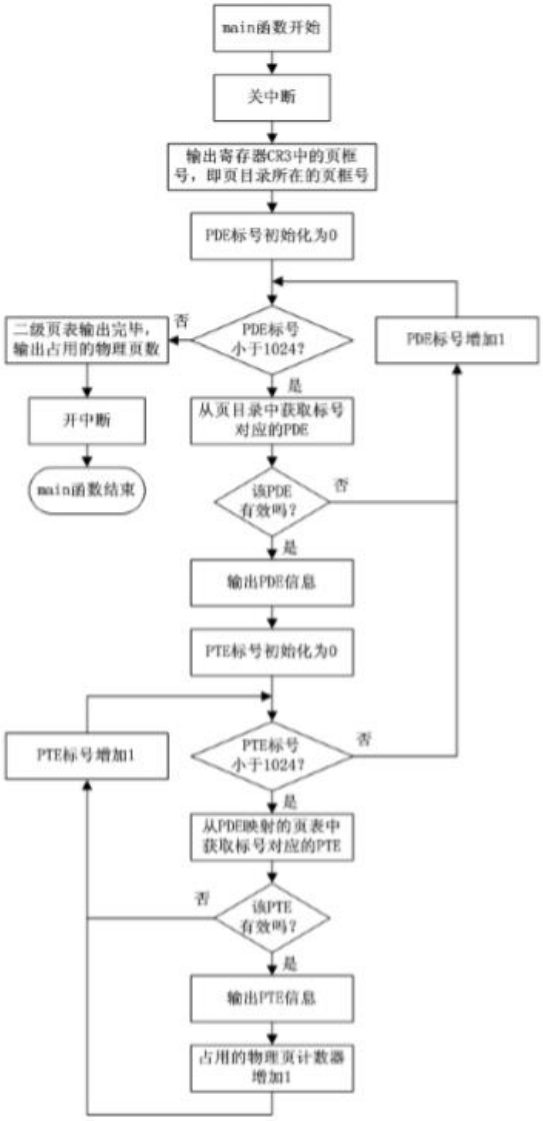


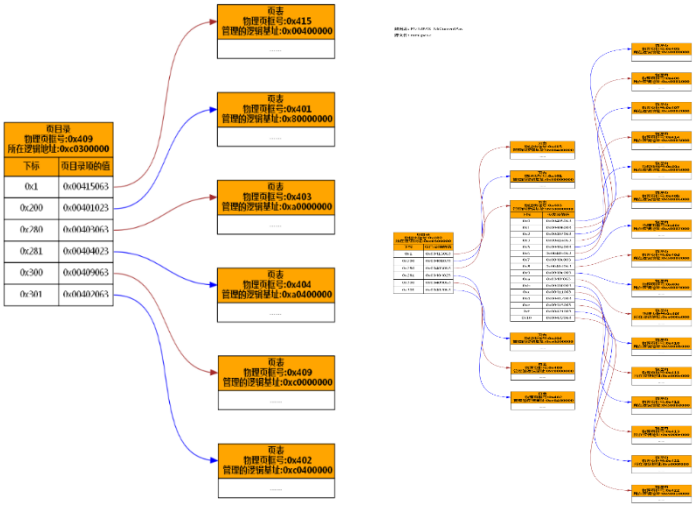
```
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
                PTE: 0x8 (0xA0008000)->0x40D
                PTE: 0x9 (0xA0009000)->0x40E
                PTE: 0xA (0xA000A000)->0x40F
                PTE: 0xB (0xA000B000)->0x410
                PTE: 0xC (0xA000C000)->0x411
                PTE: 0xD (0xA000D000)->0x412
                PTE: 0xE (0xA000E000)->0x413
                PTE: 0xF (0xA000F000)->0x421
                PTE: 0x10 (0xA0010000)->0x422
PDE: 0x281 (0xA0400000)->0x404
PDE: 0x300 (0xC0000000)->0x409
                PTE: 0x1 (0xC0001000)->0x415
                PTE: 0x200 (0xC0200000)->0x401
                PTE: 0x280 (0xC0280000)->0x403
                PTE: 0x281 (0xC0281000)->0x404
                PTE: 0x300 (0xC0300000)->0x409
                PTE: 0x301 (0xC0301000)->0x402
PDE: 0x301 (0xC0400000)->0x402
                PTE: 0x0 (0xC0400000)->0x408

Physical Page Total: 1058
Physical Memory Total: 4333568
```

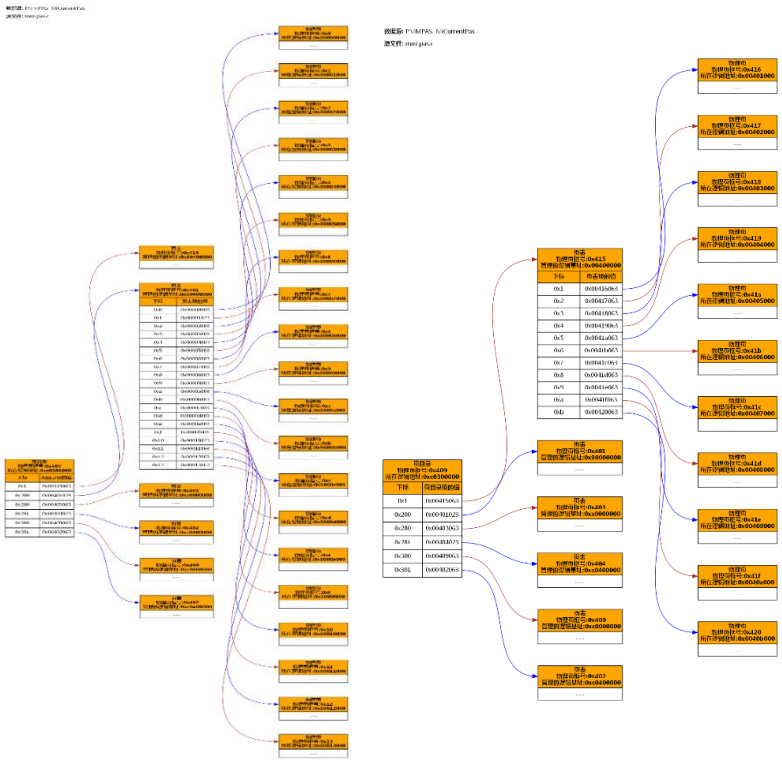
10. 由于打印输出到屏幕上的内容较多，导致前面的信息无法显示，读者可以使用“二级页表”窗口 查看应用程序进程的页目录和页表信息。选择“调试”菜单“窗口”中的“二级页表”，打开“二 级页表”窗口，点击该窗口工具栏上的“刷新”按钮，可以查看页目录和页表的映射关系。注意， “二级页表”窗口显示的是当前进程（即当前线程的父进程）的二级页表映射信息，由于此时命中了应用程序 main 函数中的断点，所以当前进程就是应用程序进程。



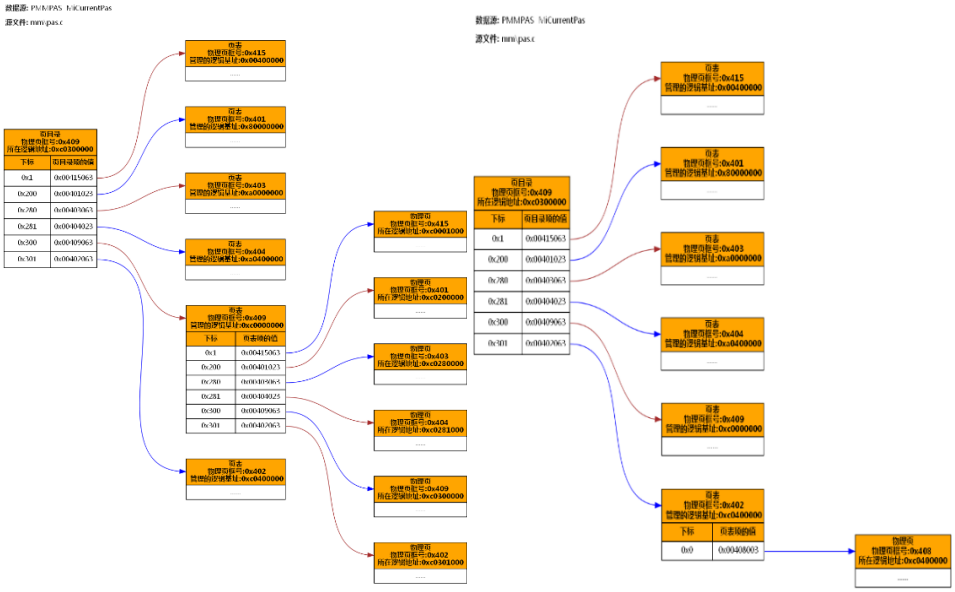
11. “二级页表”窗口默认只显示页目录和页表的信息，为了查看页表映射的物理页，可以点击“二 级页表”窗口工具栏上的“绘制页表映射的物理页”按钮，在打开的对话框中，可以选择页表中 指定的页表项映射的物理页，然后点击“绘制”按钮，就可以绘制出页表项到物理页的映射了， 如图 16-







2(a)所示。



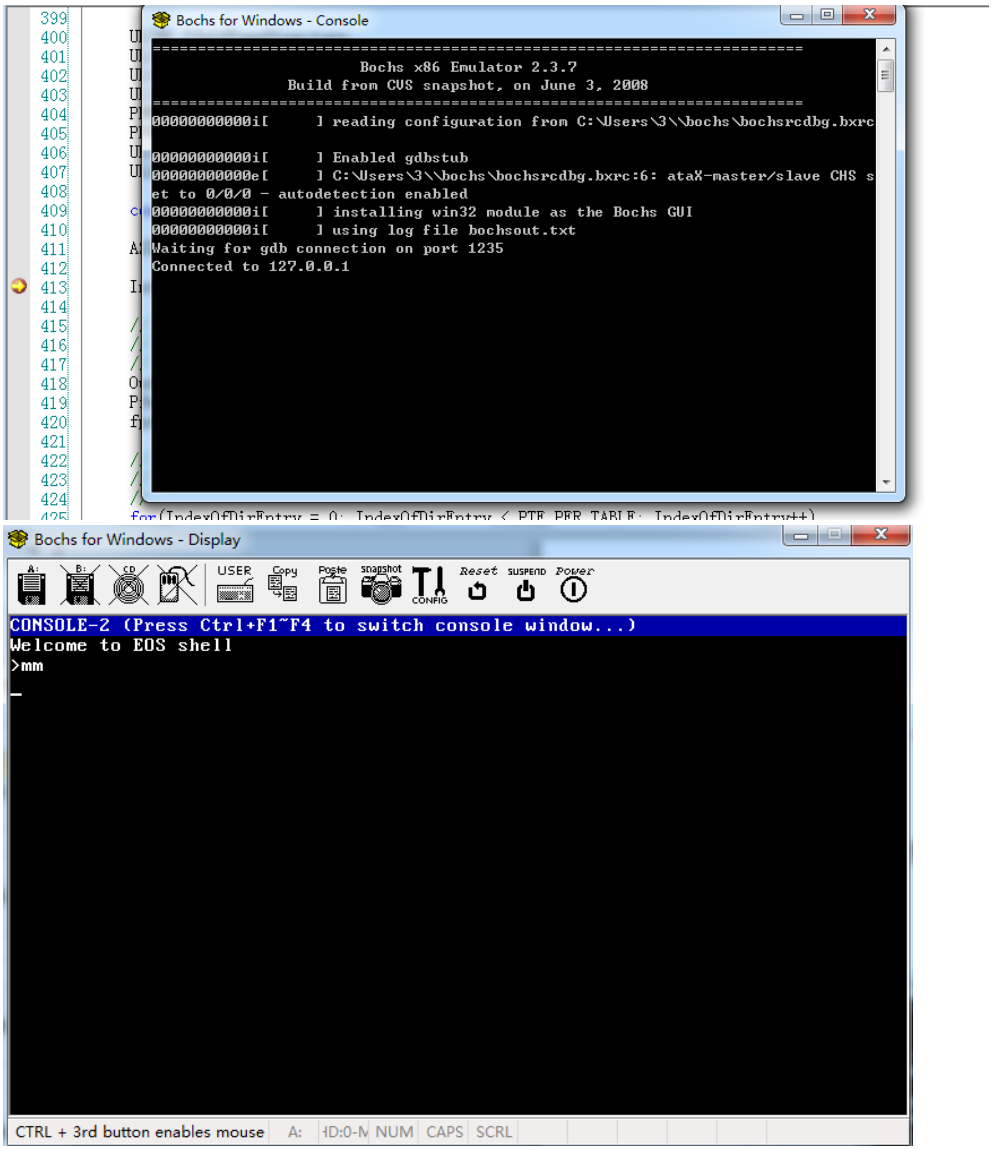
页目录和物理页中都标记了它们所在的逻辑地址，也就是说通过访问这些逻 辑地址，就可以访问这些物理页中的数据。但是，在页表中却标记了每个页表所能够映射的 4MB 逻辑地址空间的基址，这样可以方便用户从页表信息中迅速掌握进程在 4GB 逻辑地址空间中的布局，而且每个页表 其实都在第三列的物理页中显示了它们所在的逻辑地址。

查看应用程序进程和系统进程并发时系统进程的页目录和表目录

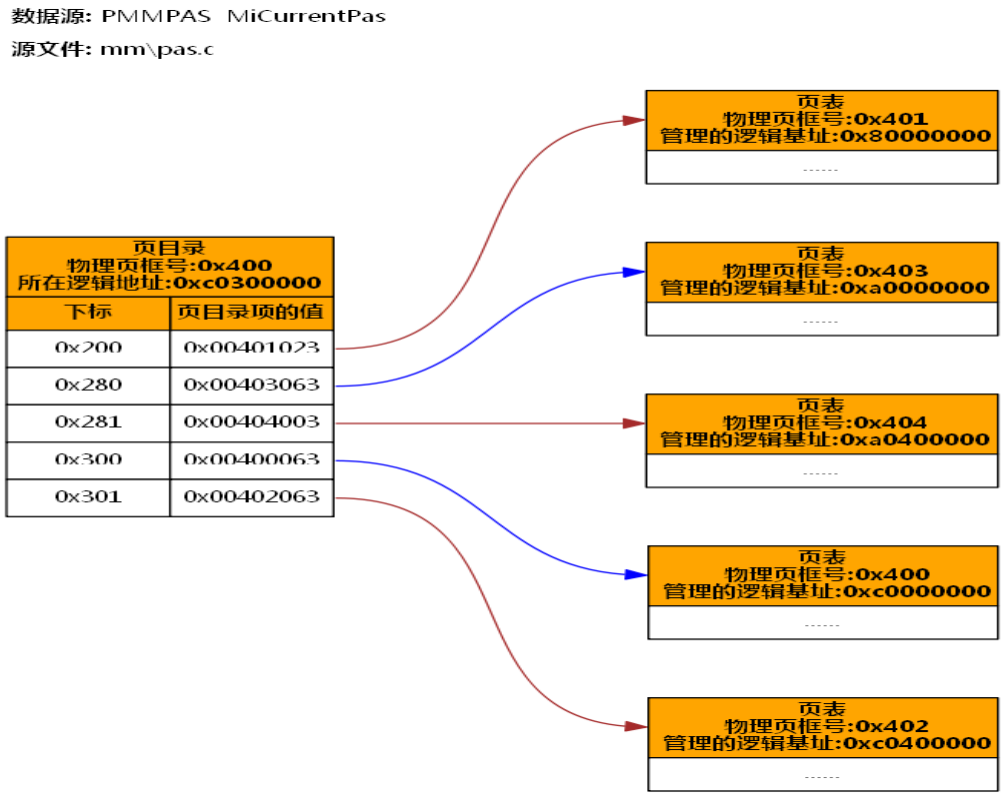
之前已经查看了应用程序进程的二级页表映射信息，接下来可以查看一下应用程序进程和系统进程并发时系统进程的页目录和页表，然后将系统进程和应用程序进程的二级页表映射信息进行比较，从而可以更好的理解 EOS 管理进

程 4G 逻辑地址空间的方法，特别是系统进程和应用程序进程是如何共享内核地址空间的。 按照下面的步骤进行实验：

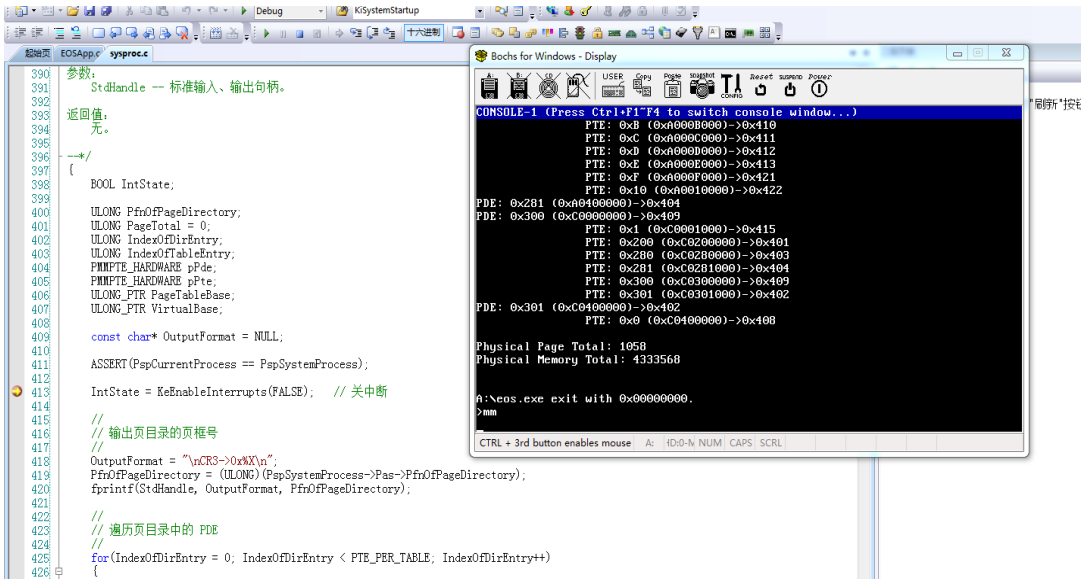
1. 结束之前的调试，并删除之前添加的所有断点。
2. 取消 EOSApp.c 文件第 113 行语句的注释（该行语句会让应用程序等待 10 秒）。
3. 按 F7 生成修改后的 EOS 应用程序项目。
4. 使用 Windows 资源管理器打开在本实验 3.1 中创建的 EOS 内核项目的项目文件夹，并找到 sysproc.c 文件。
5. 将 sysproc.c 文件拖动到 OS Lab 窗口中释放，打开此文件,在 ConsoleCmdMemoryMap 函数中的第 413 行代码处添加一个断点。注意，一定要将 sysproc.c 文件拖动到本实验 3.2 中已经打开 EOS 应用程序项目的 OS Lab 中，这样该 OS Lab 就同时打开了 EOS 应用程序项目和 EOS 内核项目中的 sysproc.c 文件，方便后面的调试。
6. 按 F5 启动调试 EOS 应用程序。
7. 在“Console-1”中会自动执行 EOSApp.exe，创建该应用程序进程。利用其等待 10 秒的时间， 按 Ctrl+F2 切换到“Console-2”。
8. 在“Console-2”中输入命令“mm”后按回车，程序在断点处中断。



9. 刷新“二级页表”窗口，由于此时是在控制台线程正在执行 mm 命令时中断的，所以“二级页表” 窗口中显示的是系统进程的二级页表映射关系，如图 16-2(b)所示。 控制台命令“mm”对应的源代码在 EOS 内核项目 ke/sysproc.c 文件的 ConsoleCmdMemoryMap 函数中 （第 382 行）。阅读这部分源代码后会发现，其与 EOSApp.c 文件中的源代码基本类似。



按 F5 继续调试，待 mm 命令执行完毕后，切换到控制台 1，此时 应用程序又会继续运行，待其结束后， 在控制台 1 中再次输入命令“mm”又会命中之前添加的断点，此时在“二级页表”窗口中可以查看在没有 应用程序进程时，系统进程的页目录和页表。



查看应用程序进程并发时的页目录和页表，通过前面的实验内容，读者已经对应用程序进程和系统进程的二级页表映射情况有了充分的了解，接 下来请读者按照下面的步骤，查看使用同一个应用程序的可执行文件创建的两个应用程序进程在并发时各 自的页目录和页表，从而学习多个应用程序进程是如何共享内核空间的，以及同一个应用程序的不同进程 是如何拥有各自独立的用户空间，从而完成隔离的。 请读者按照下面的步骤进行实验：

1. 结束之前的调试，并删除之前添加的所有断点。

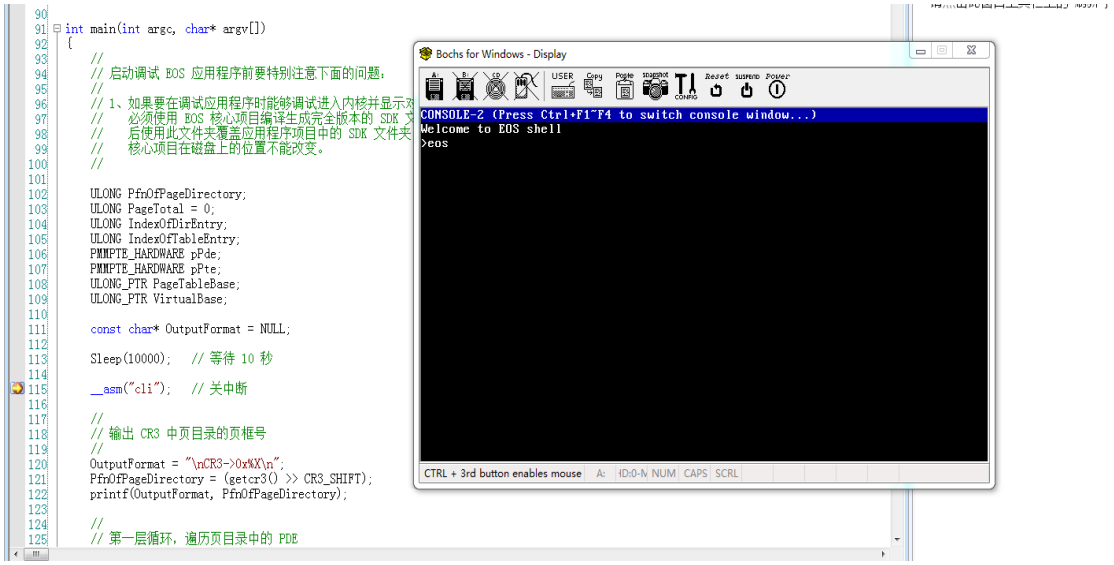
2. 取消 EOSApp.c 文件第 188 行语句的注释（该行语句会让应用程序等待 60 秒）。

3. 按 F7 生成修改后的 EOS 应用程序项目。

4. 在 EOSApp.c 文件的 main 函数中的第 115 行代码处添加一个断点。

5. 按 F5 启动调试，待 EOS 启动完成后，会在控制台 1 中自动运行 EOS 应用程序创建应用程序的第 一个进程，并在开始的位置等待 10 秒钟。

6. 请读者利用应用程序第一个进程等待 10 秒钟的机会，迅速按 Ctrl+F2 切换到控制台 2，并在控制 台 2 中输入“eosapp”后按回车（请根据应用程序可执行文件的名称调整命令），再使用该应用 程序创建第二个进程，此时，应用程序创建的两个进程就开始并发运行了。



7. 当第一个应用程序进程等待的 10 秒钟结束后，就会命中刚刚添加的断点。

8. 刷新“进程线程”窗口，可以看到当前除了系统进程之外，还有两个应用程序进程。其中，第一 个应用程序进程的主线程处于运行状态，也就是说当前中断运行的是第一个应用程序进程的主线 程；第二个应用程序进程的主线程处于阻塞状态，说明其正在等待 10 秒钟，然后才能继续运行。

9. 由于当前正在运行的是应用程序的第一个进程，所以在刷新“二级页表”窗口后，可以查看第一个应用程序进程的二级页表映射，如图 3-3所示。

10. 按 F5 继续调试，等待一段时间后，又会命中刚刚添加的断点。

11. 刷新“进程线程”窗口，这次是第一个应用程序进程的主线程处于阻塞状态，说明其打印输出完 毕后，在 main 函数中第 188 行处调用 Sleep 函数发生了阻塞；而第二个应用程序进程的主线程 处于运行状态，也就是说当前中断运行的是第二个应用程序进程的主线程。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	6	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	17	Y	24	Waiting (3)	1	0x80015724 IopConsoleDispatchThread
3	18	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiShellThread

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	1	26	"A:/eos.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Waiting (3)	24	0x8001f97e PspProcessStartup

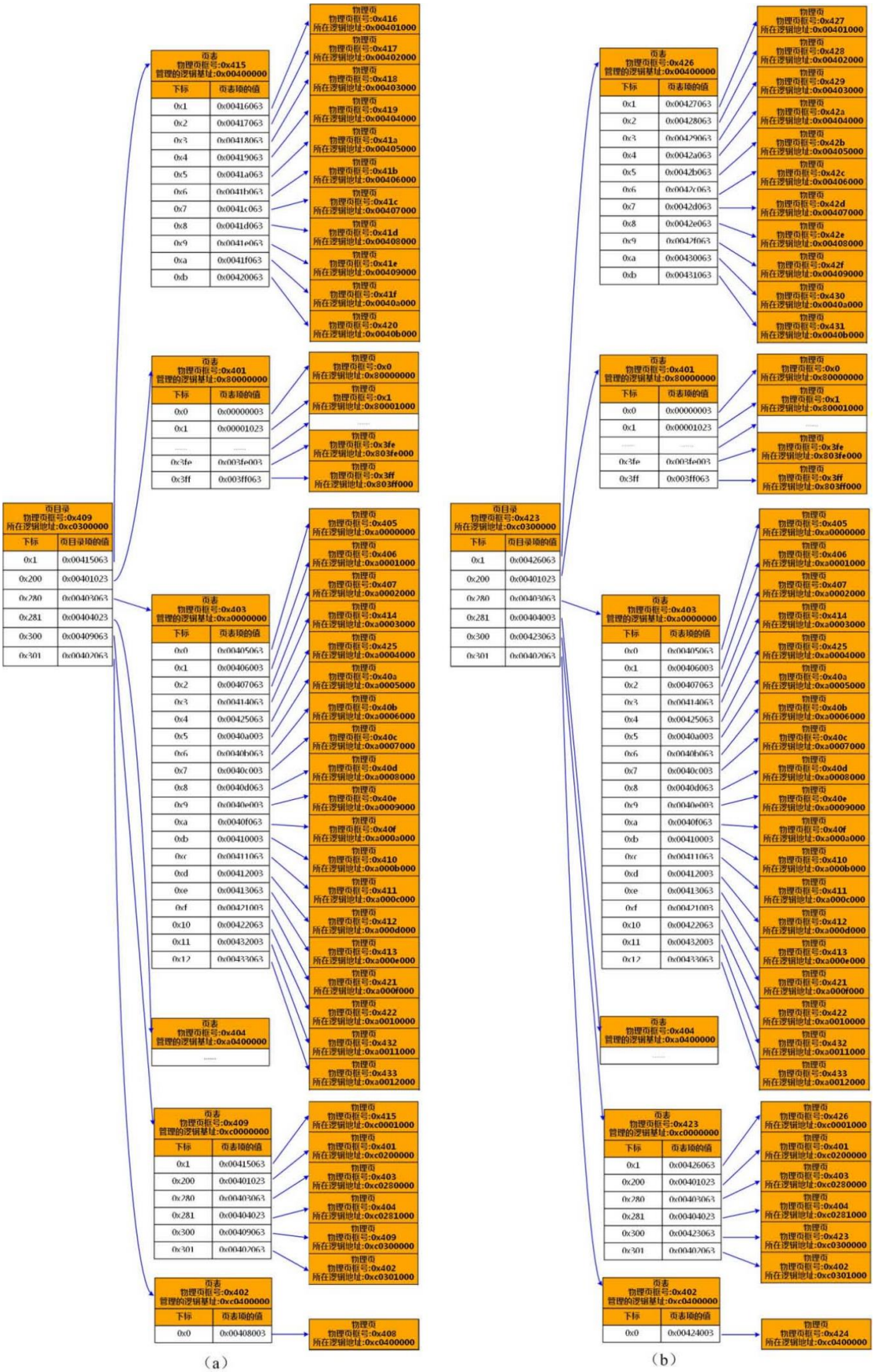
进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
3	27	N	8	1	29	"a:/eos.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	29	N	8	Running (2)	27	0x8001f97e PspProcessStartup ← PspCurrentThread

12. 由于当前正在运行的是应用程序的第二个进程，所以在刷新“二级页表”窗口后，可以查看第二 个应用程序进程的二级页表映射，如图 3-3所示。



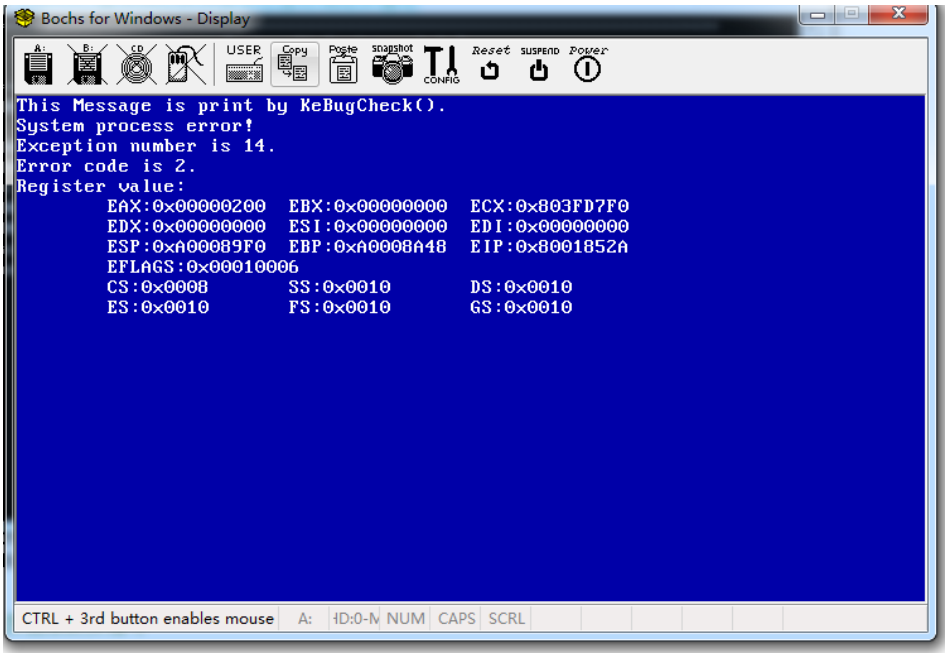
在系统进程的二级页表中映射新申请的物理页

下面通过编程的方式，从 EOS 操作系统内核中申请两个未用的物理页，将第一个物理页当作页表，映 射基址为 0xE0000000 的 4M 虚拟地址空间，然后将第二个物理页分别映射到基址为 0xE0000000 和 0xE0001000 的 4K 虚拟地址空间。从而验证下面的结论：

- 虽然进程可以访问 4G 虚拟地址空间，但是只有当一个虚拟地址通过二级页表映射关系能够映射 到实际的物理地址时，该虚拟地址才能够被访问，否则会触发缺页异常。
- 所有未用的物理页都是由 EOS 操作系统内核统一管理的，使用时必须向内核申请。
- 为虚拟地址映射物理页时，必须首先为页目录安装页表，然后再为页表安装物理页。并且只有在 刷新快表后，对页目录和页表的更改才能生效。
- 不同的虚拟地址可以映射相同的物理页，从而实现共享。

首先验证第一个结论：

1. 新建一个 EOS Kernel 项目。
2. 从“项目管理器”打开 ke/sysproc.c 文件。
3. 打开“学生包”本实验对应的文件夹中的 MapNewPage.c 文件（将文件拖动到 OS Lab 窗口中释放 即可）。
4. 在 sysproc.c 文件的 ConsoleCmdMemoryMap 函数中找到“关中断”的代码行（第 413 行）， 将 MapNewPage.c 文件中的代码插入到“关中断”代码行的后面。
5. 按 F7 生成该内核项目。
6. 按 F5 启动调试。
7. 在 EOS 控制台中输入命令“mm”后按回车。
8. EOS 会出现蓝屏，并显示错误原因是由于 14 号异常（缺页异常）引起的。原因就是由于刚刚从 MapNewPage.c 文件复制的第二行代码所访问的虚拟地址没有映射物理内存（图 16-2 和图 16-3 中都未映射该虚拟地址），所以对该虚拟地址的访问会触发缺页异常，而此时 EOS 还没有为缺页 异常安装中断服务程序，所以就调用 KeBugCheck 函数显示蓝屏错误了。



9. 结束此次调试，然后删除或者注释掉会触发异常的那行代码。

按照下面的步骤验证其它结论：

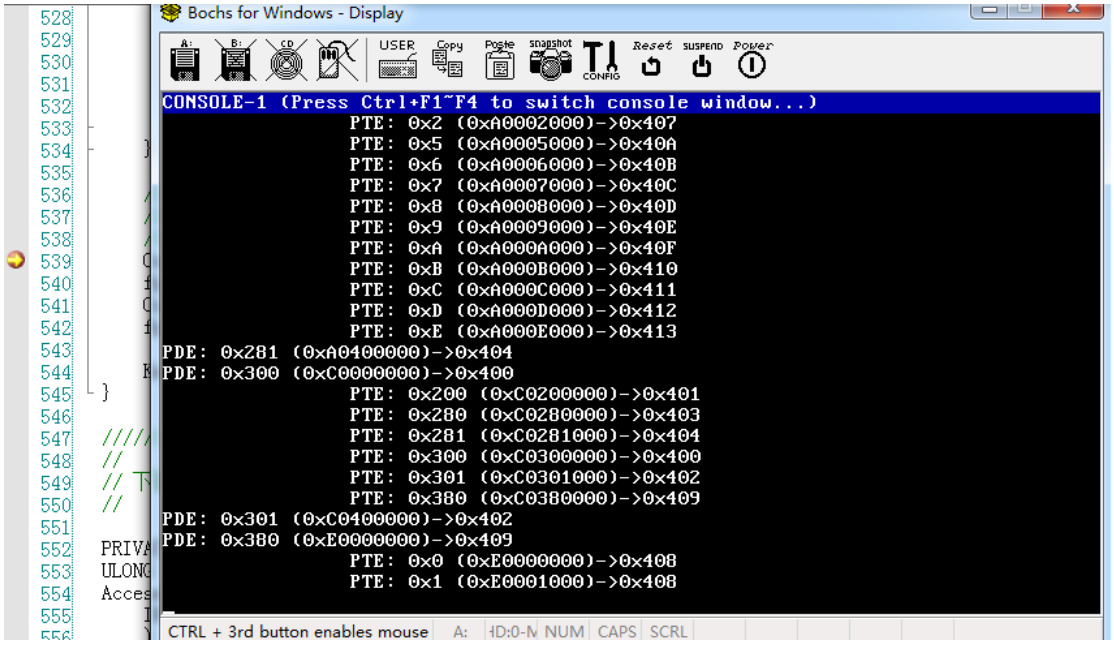
1. 按 F7 生成该内核项目。

2. 在 sysproc.c 文件的 ConsoleCmdMemoryMap 函数中打开中断的第 539 行处添加一个断点。

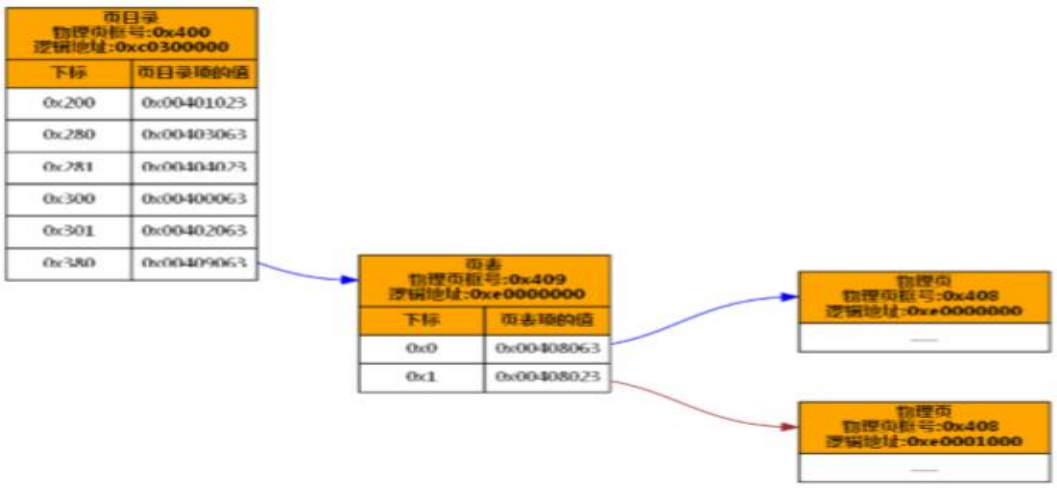
3. 按 F5 启动调试。

4. 在 EOS 控制台中输入命令“mm”后按回车。

5. 当在控制台窗口中输出二级页表映射信息完毕后，会在刚刚添加的断点处中断。



6. 点击“二级页表”窗口工具栏上的“绘制页表映射的物理页”按钮，在弹出的对话框中，从下拉 控件中选择“PDE 0x380 映射的页表”项目，然后勾选该页表包含的两个页表项，点击“绘制” 按钮，可以查看指定页表项到物理页的映射，如图所示。将“二级页表”窗口中的内容保 存到一个图像文件中。



4. 实验的思考与问题分析

（1）观察之前输出的页目录和页表的映射关系，可以看到页目录的第 0x300 个 PDE 映射的页框号就是页目录本身，说明页目录被复用为了页表。而恰恰就是这种映射关系决定了 4K 的页目录映射在虚拟地址空间的 0xC0300000-0xC0300FFF,4M 的页表映射在 0xC0000000-0xC03FFFFF。现在，假设修改了页目录，使其第 0x100 个 PDE 映射的页框号是页目录本身，此时页目录和页表会映射在 4G 虚拟地址空间的什么位置呢？说明计算方法。

答：

页目录：PDE标号0x100作为虚拟地址的高10位，PTE标号0x100作为虚拟地址的12-22位，得到虚拟地址0x 40100000。

页表：PDE标号0x100作为虚拟地址的高10位，PTE标号0x0作为虚拟地址的12-22位，得到虚拟地址0x 40000000。

（2）修改 EOSApp.c 中的源代码，通过编程的方式统计并输出用户地址空间占用的内存数目。

答：

（3）修改 EOSApp.c 中的源代码，通过编程的方式统计并输出页目录和页表的数目。注意页目录被复用为页表。

答;

编写代码将申请到的物理页从二级页表映射中移除，并让内核回收这些物理页.源代码文件MapNewPageEx.c。使用该文件中的ConsoleCmdMemoryMap函数替换ke/sysproc.c中的ConsoleCmdMemoryMap函数即可。在移除映射的物理页时，只需要将PTE/PDE的存在标志位设置为0即可，要先修改PTE，再修改PDE。另外，要注意刷新快表。调用MiFreePages函数即可回收物理页，具体的用法可以参考其函数定义处的注释和源代码(mm/pfnlist.c第248行)。

（4）在 EOS 启动时，软盘引导扇区被加载到从 0x7C00 开始的 512 个字节的物理内存中，计算一下其所在的物理页框号，然后根据物理内存与虚拟内存的映射关系得到其所在的虚拟地址。修改 EOSApp.c 中的源代码，尝试将软盘引导扇区所在虚拟地址的 512 个字节值打印出来，与 boot.lst 文件中的指令字节码进行比较，验证计算的虚拟地址是否正确。

答：

（5）既然所有 1024 个页表（共 4M）映射在虚拟地址空间的 0xC0000000-0xC03FFFFF，为什么不能从页表基址 0xC0000000 开始遍历，来查找有效的页表呢？而必须先在页目录中查找有效的页表呢？编写代码尝试一下，看看会有什么结果。

答;

不能从页表基址 0xC0000000 开始遍历查找有效的页表，因为：只有当一个虚拟地址通过二级页表映射关系能够映射到实际的物理地址时，该虚拟地址才能够被访问，否则会触发异常。不是所有的页表都有效，所以不能从页表

基址开始遍历。

（6）学习 EOS 操作系统内核统一管理未用物理页的方法（可以参考本书第 6 章的第 6.5 节）。尝试在本实验第 3.5 节中 ConsoleCmdMemoryMap 函数源代码的基础上进行修改，将申请到的物理页从二级页表映射中移除，并让内核回收这些物理页。

答:

参见源文件MapNewPageEx.c。使用该文件中的ConsoleCmdMemoryMap函数替换ke/sysproc.c中的ConsoleCmdMemoryMap函数即可。

移除映射的物理页只需将PTE/PDE的存在标志位设置为0，要先修改PTE，再修改PDE。调用MiFreePages函数即可回收物理页，具体的用法可以参考mm/pfnlist.c第248行。

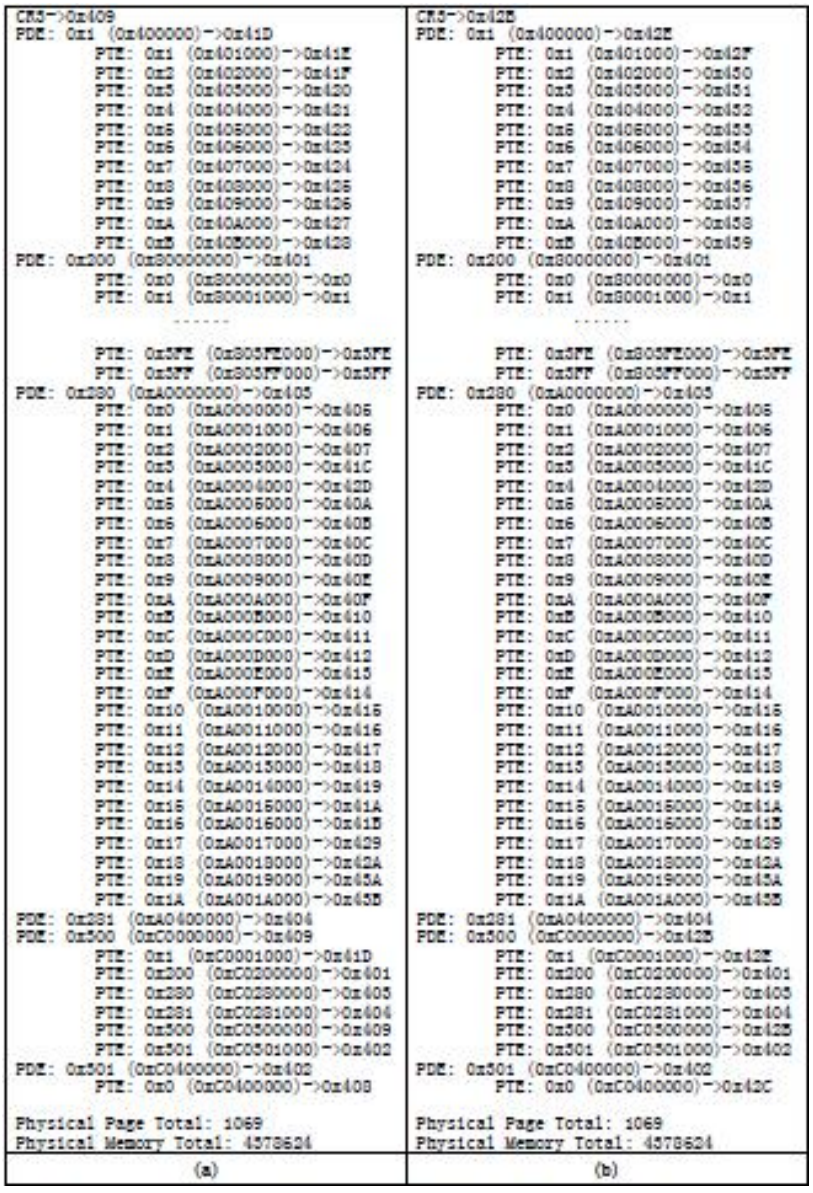
源代码：

```
ULONG PfnArray[2];
//
// 访问未映射物理内存的虚拟地址会触发异常。
// 必须注释或者删除该行代码才能执行后面的代码。
//
*((PINT)0xE0000000) = 100;
//
// 从内核申请两个未用的物理页。
// 由 PfnArray 数组返回两个物理页的页框号。
MiAllocateZeroedPages(2, PfnArray);
OutputFormat = "New page frame number: 0x%X, 0x%X\n";
fprintf(StdHandle, OutputFormat, PfnArray[0], PfnArray[1]);
KdbPrint(OutputFormat, PfnArray[0], PfnArray[1]);
//
// 使用 PfnArray[0] 页做为页表，映射基址为 0xE00000000 的 4M 虚拟地址。
IndexOfDirEntry = (0xE0000000 >> 22); // 虚拟地址的高 10 位是 PDE 标号
((PMMPTE_HARDWARE)0xC0300000)[IndexOfDirEntry].PageFrameNumber = PfnArray[0];
((PMMPTE_HARDWARE)0xC0300000)[IndexOfDirEntry].Valid = 1;      // 有效
((PMMPTE_HARDWARE)0xC0300000)[IndexOfDirEntry].Writable = 1;   // 可写
MiFlushEntireTlb(); // 刷新快表
//
// 根据 PDE 的标号计算其映射的页表所在虚拟地址的基址
//
PageTableBase = 0xC0000000 + IndexOfDirEntry * PAGE_SIZE;
//
// 将 PfnArray[1] 放入页表 PfnArray[0] 的两个 PTE 中，
```

```
// 分别映射基址为 0xE0000000 和 0xE0001000 的 4K 虚拟地址
IndexOfTableEntry = (0xE0000000 >> 12) & 0x3FF; // 虚拟地址的
12-22 位是 PTE 标号
((PMMPTE_HARDWARE)PageTableBase)[IndexOfTableEntry].PageFrameNumb
er = PfnArray[1];
((PMMPTE_HARDWARE)PageTableBase)[IndexOfTableEntry].Valid = 1;
// 有效
((PMMPTE_HARDWARE)PageTableBase)[IndexOfTableEntry].Writable = 1;
// 可写
MiFlushEntireTlb(); // 刷新快表
IndexOfTableEntry = (0xE0001000 >> 12) & 0x3FF; // 虚拟地址的
12-22 位是 PTE 标号
((PMMPTE_HARDWARE)PageTableBase)[IndexOfTableEntry].PageFrameNumb
er = PfnArray[1];
((PMMPTE_HARDWARE)PageTableBase)[IndexOfTableEntry].Valid = 1;
// 有效
((PMMPTE_HARDWARE)PageTableBase)[IndexOfTableEntry].Writable = 1;
// 可写
MiFlushEntireTlb(); // 刷新快表
//
// 测试
OutputFormat = "Read Memory 0xE0001000: %d\n";
fprintf(StdHandle, OutputFormat, *((PINT)0xE0001000));
KdbPrint(OutputFormat, *((PINT)0xE0001000));
*((PINT)0xE0000000) = 100; // 写共享内存
fprintf(StdHandle, OutputFormat, *((PINT)0xE0001000));
KdbPrint(OutputFormat, *((PINT)0xE0001000));
```

（7）待完成实验 14 中的写文件功能后，可以改进本实验中的应用程序的源代码 mm 命令的源代码，将应用程序进程或者系统进程的二级页表映射信息在打印输出到屏幕上的同时，也写入一个文本文件中。

答：

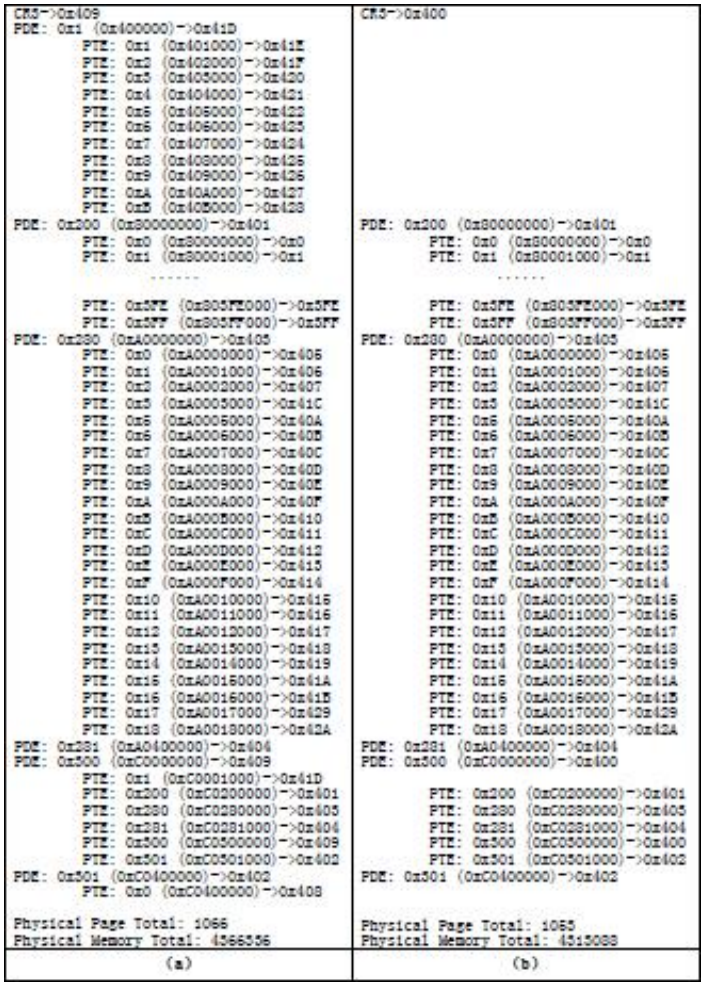


```
CR3->0x409
PDE: 0x1 (0x400000)->0x41D
        PTE: 0x1 (0x401000)->0x41E
        PTE: 0x2 (0x402000)->0x41F
        PTE: 0x3 (0x403000)->0x420
        PTE: 0x4 (0x404000)->0x421
        PTE: 0x5 (0x405000)->0x422
        PTE: 0x6 (0x406000)->0x423
        PTE: 0x7 (0x407000)->0x424
        PTE: 0x8 (0x408000)->0x425
        PTE: 0x9 (0x409000)->0x426
        PTE: 0xA (0x40A000)->0x427
        PTE: 0xB (0x40B000)->0x428
PDE: 0x200 (0x80000000)->0x401
        PTE: 0x0 (0x80000000)->0x0
        PTE: 0x1 (0x80001000)->0x1
                ......

        PTE: 0x3FE (0x803FE000)->0x3FE
        PTE: 0x3FF (0x803FF000)->0x3FF
PDE: 0x280 (0xA0000000)->0x405
        PTE: 0x0 (0xA0000000)->0x406
        PTE: 0x1 (0xA0001000)->0x407
        PTE: 0x2 (0xA0002000)->0x41C
        PTE: 0x3 (0xA0003000)->0x42D
        PTE: 0x4 (0xA0004000)->0x40A
        PTE: 0x5 (0xA0005000)->0x40B
        PTE: 0x6 (0xA0006000)->0x40C
        PTE: 0x7 (0xA0007000)->0x40D
        PTE: 0x8 (0xA0008000)->0x40E
        PTE: 0x9 (0xA0009000)->0x40F
        PTE: 0xA (0xA000A000)->0x410
        PTE: 0xB (0xA000B000)->0x411
        PTE: 0xC (0xA000C000)->0x412
        PTE: 0xD (0xA000D000)->0x413
        PTE: 0xE (0xA000E000)->0x414
        PTE: 0xF (0xA000F000)->0x415
        PTE: 0x10 (0xA0010000)->0x416
        PTE: 0x11 (0xA0011000)->0x417
        PTE: 0x12 (0xA0012000)->0x418
        PTE: 0x13 (0xA0013000)->0x419
        PTE: 0x14 (0xA0014000)->0x41A
        PTE: 0x15 (0xA0015000)->0x41B
        PTE: 0x16 (0xA0016000)->0x41E
        PTE: 0x17 (0xA0017000)->0x429
        PTE: 0x18 (0xA0018000)->0x42A
        PTE: 0x19 (0xA0019000)->0x45A
        PTE: 0x1A (0xA001A000)->0x45B
PDE: 0x281 (0xA0400000)->0x404
PDE: 0x300 (0xC0000000)->0x409
        PTE: 0x1 (0xC0001000)->0x41D
        PTE: 0x200 (0xC0200000)->0x401
        PTE: 0x280 (0xC0280000)->0x405
        PTE: 0x281 (0xC0281000)->0x404
        PTE: 0x300 (0xC0300000)->0x409
        PTE: 0x301 (0xC0301000)->0x402
PDE: 0x301 (0xC0400000)->0x402
        PTE: 0x0 (0xC0400000)->0x408

Physical Page Total: 1069
Physical Memory Total: 4378624
```

(a)

```
CR3->0x42E
PDE: 0x1 (0x400000)->0x42E
        PTE: 0x1 (0x401000)->0x42F
        PTE: 0x2 (0x402000)->0x430
        PTE: 0x3 (0x403000)->0x431
        PTE: 0x4 (0x404000)->0x432
        PTE: 0x5 (0x405000)->0x433
        PTE: 0x6 (0x406000)->0x434
        PTE: 0x7 (0x407000)->0x435
        PTE: 0x8 (0x408000)->0x436
        PTE: 0x9 (0x409000)->0x437
        PTE: 0xA (0x40A000)->0x438
        PTE: 0xB (0x40B000)->0x439
PDE: 0x200 (0x80000000)->0x401
        PTE: 0x0 (0x80000000)->0x0
        PTE: 0x1 (0x80001000)->0x1
                ......

        PTE: 0x3FE (0x803FE000)->0x3FE
        PTE: 0x3FF (0x803FF000)->0x3FF
PDE: 0x280 (0xA0000000)->0x405
        PTE: 0x0 (0xA0000000)->0x406
        PTE: 0x1 (0xA0001000)->0x407
        PTE: 0x2 (0xA0002000)->0x41C
        PTE: 0x3 (0xA0003000)->0x42D
        PTE: 0x4 (0xA0004000)->0x40A
        PTE: 0x5 (0xA0005000)->0x40B
        PTE: 0x6 (0xA0006000)->0x40C
        PTE: 0x7 (0xA0007000)->0x40D
        PTE: 0x8 (0xA0008000)->0x40E
        PTE: 0x9 (0xA0009000)->0x40F
        PTE: 0xA (0xA000A000)->0x410
        PTE: 0xB (0xA000B000)->0x411
        PTE: 0xC (0xA000C000)->0x412
        PTE: 0xD (0xA000D000)->0x413
        PTE: 0xE (0xA000E000)->0x414
        PTE: 0xF (0xA000F000)->0x415
        PTE: 0x10 (0xA0010000)->0x416
        PTE: 0x11 (0xA0011000)->0x417
        PTE: 0x12 (0xA0012000)->0x418
        PTE: 0x13 (0xA0013000)->0x419
        PTE: 0x14 (0xA0014000)->0x41A
        PTE: 0x15 (0xA0015000)->0x41B
        PTE: 0x16 (0xA0016000)->0x41E
        PTE: 0x17 (0xA0017000)->0x429
        PTE: 0x18 (0xA0018000)->0x42A
        PTE: 0x19 (0xA0019000)->0x45A
        PTE: 0x1A (0xA001A000)->0x45B
PDE: 0x281 (0xA0400000)->0x404
PDE: 0x300 (0xC0000000)->0x42B
        PTE: 0x1 (0xC0001000)->0x42E
        PTE: 0x200 (0xC0200000)->0x401
        PTE: 0x280 (0xC0280000)->0x405
        PTE: 0x281 (0xC0281000)->0x404
        PTE: 0x300 (0xC0300000)->0x42B
        PTE: 0x301 (0xC0301000)->0x402
PDE: 0x301 (0xC0400000)->0x402
        PTE: 0x0 (0xC0400000)->0x42C

Physical Page Total: 1069
Physical Memory Total: 4378624
```

(b)



（8）思考页式存储管理机制的优缺点。

答：

优点：页式存储管理机制不要求作业或进程的程序段和数据在内存中连续存放，解决了碎片问题。提供了内存和外存统一管理的虚存实现方式，使用户可以利用的存储空间大大增加。

缺点：要求有相应的硬件支持，这增加了机器成本。增加了系统开销，例如缺页中断处理机。请求调页算法如选择不当有可能产生抖动现象。

5. 总结和感想体会

分页存储管理是解决存储碎片的一种方法，动态重定位是解决存储碎片问题的一种途径，但要移动大量信息从而浪费处理机时间,代价比较高,这是因为这种分配要求把作业必须安置在一连续存储区内的缘故,而分页存储管理正是要避开这种连续性要求。

对比连续分配容易造成很多碎片，虽然可以通过紧凑的算法来将碎片拼接成可用的大块空间，但必须付出很大的开销。如果允许一个进程直接分散的装入很多不相邻的分区，则无需紧凑，即离散分配方式。如果离散分配的基本单位是页，称为分页储存管理方式；如果离散分配及基本单位是段，称为分段存储管理方式。

参考文献

[1]北京英真时代科技有限公司[DB/CD].http://www.engintime.com.
[2]汤子瀛，哲凤屏，汤小丹。计算机操作系统。西安：西安电子科技大学出版社，1996.