

**操作系统实验报告**

**实验题目**  实验8 分页存储器管理

**学生姓名**  余梓俊

**学 号**  2018211991

**专业班级** 计算机科学与技术18-3班

**指导教师**  田卫东

**完成日期**  2020年12月23日

**合肥工业大学 计算机与信息学院**

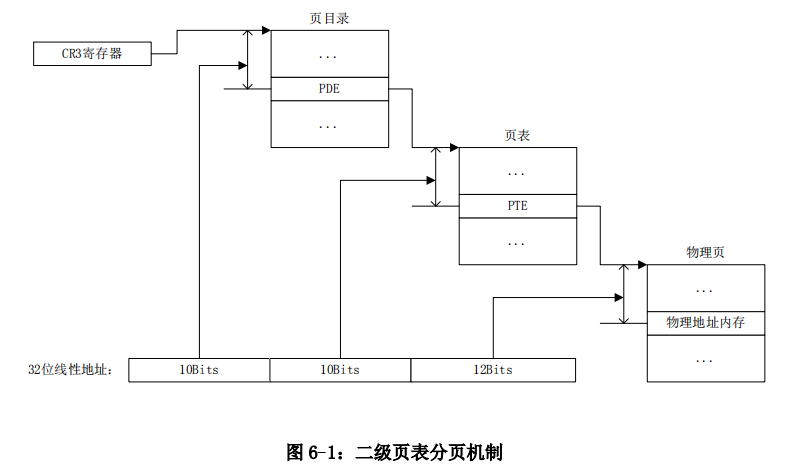
* 1. **实验目的和任务要求**

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。

* 1. **实验原理**



在二级页表分页机制中，无论是页目录（Page Directory）、页表（Page Table）

还是物理页，它们的大 小均为 4KB。其中，第一级是一个页目录，在这唯一的

页目录中包含了 1024 个页目录项(PDE，Page Directory Entry)。页目录中的

每个页目录项可以映射第二级的一个页表，所以，页目录最多可以映射 1024个

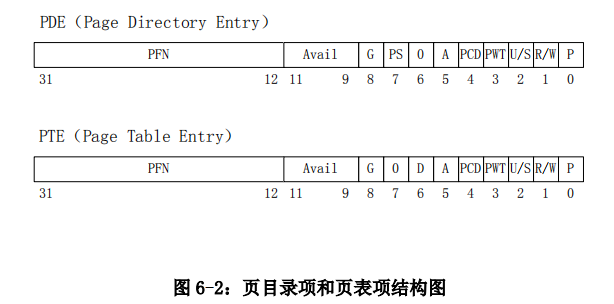
页表。第二级的每个页表都包含了 1024 个页表项(PTE，Page Table Entry)，

每个页表项可以映射一个4KB 大小的物理页。这样，一个页目录可以映射 1024

个页表，每个页表又可以映射 1024 个物理页，所以，二级页表最终就可以映射

1024\*1024\*4K=4GB 的物理地址。

由于页目录和页表的大小都是 4KB，并且都包含了 1024 个 PDE 或者 PTE，显然，PDE 和 PTE 的大小都会是 4 个字节，而且 PDE 和 PTE 的结构也十分类似，





* 1. **实验内容**

1、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。

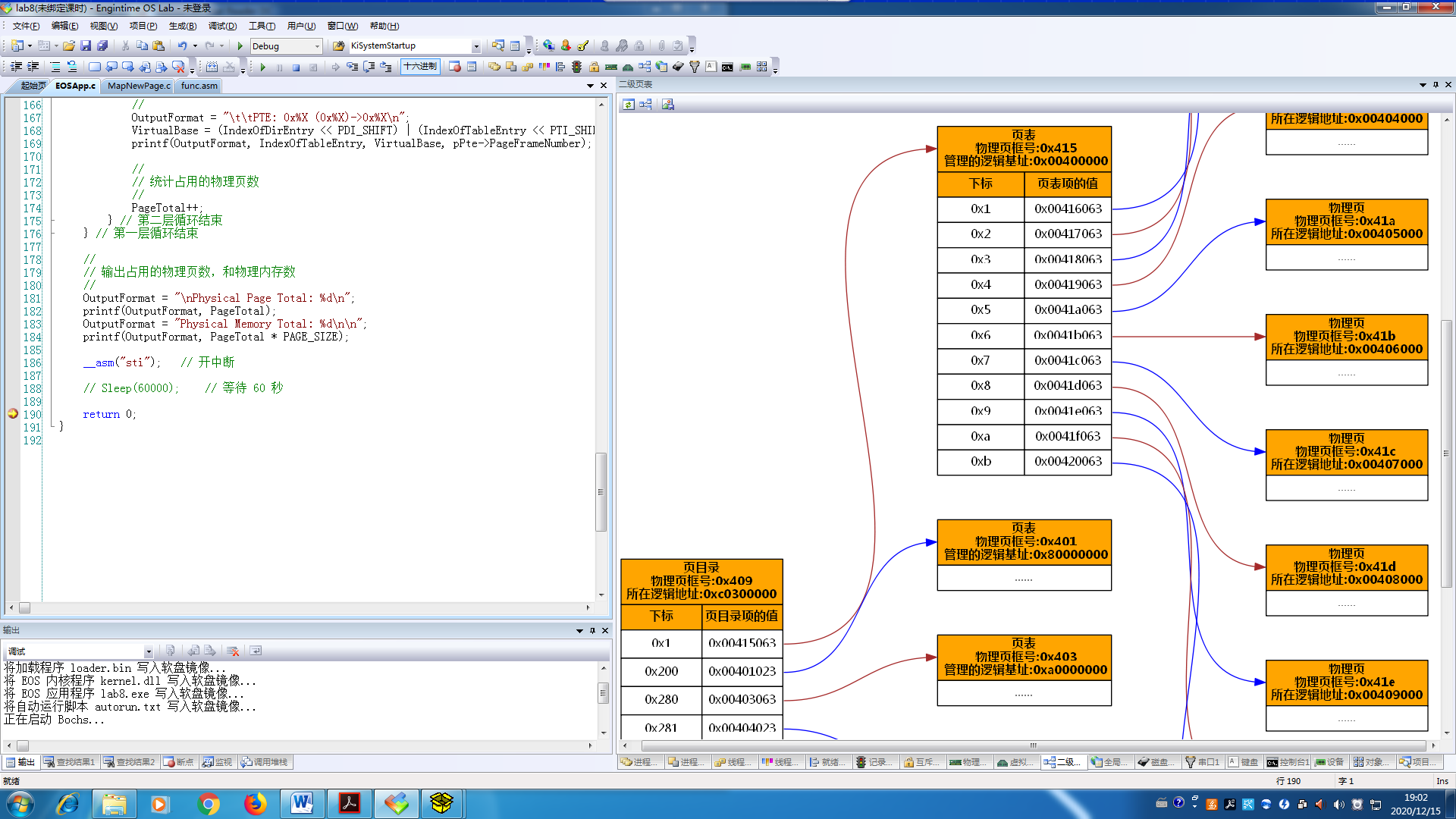
（8）在 main 函数的返回代码处（第 190 行）添加一个断点。

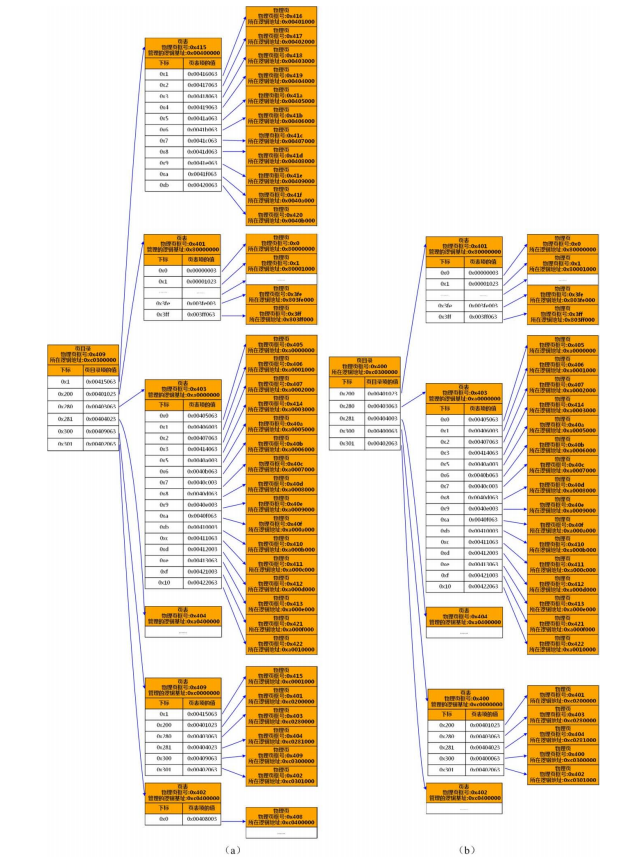
（9）按 F5 启动调试。EOS 启动完毕后会自动运行应用程序，将应用程序进程的页目录和页表打印输出

到屏幕上，然后在刚刚添加的断点处中断。

（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(a)所示。





(a)EOS 应用程序进程的二级页表映射

(b)应用程序进程和系统进程并发时系统进程的二级页表映射

2、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页表

（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。

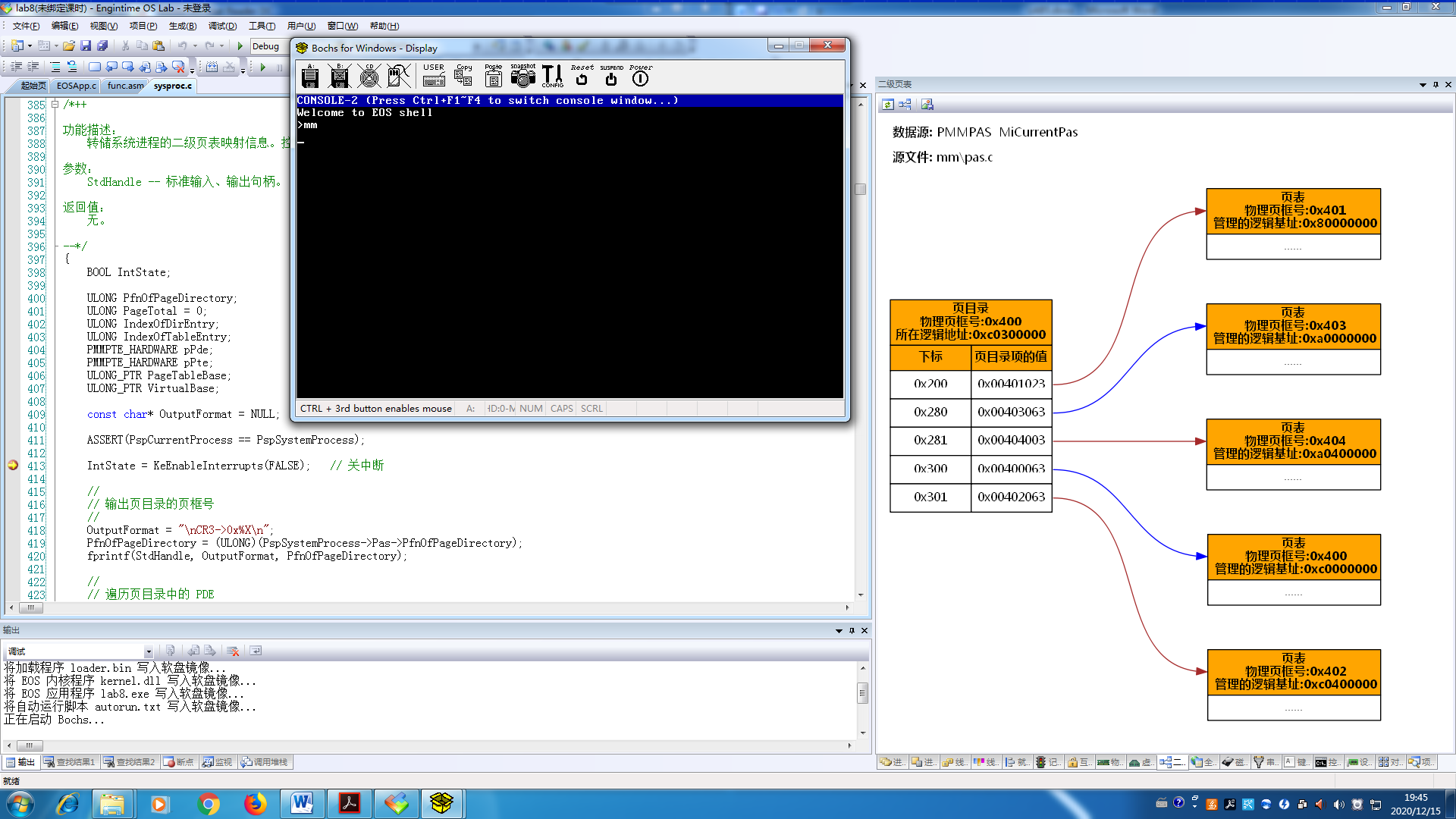
（5）将sysproc.c文件拖动到 OS Lab 窗口中释放，打开此文件,在 ConsoleCmdMemoryMap 函数中的第413 行代码处添加一个断点。注意，一定要将 sysproc.c 文件拖动到本实验 3.2 中已经打开 EOS应用程序项目的 OS Lab 中，这样该 OS Lab 就同时打开了 EOS 应用程序项目和 EOS 内核项目中的sysproc.c 文件，方便后面的调试。

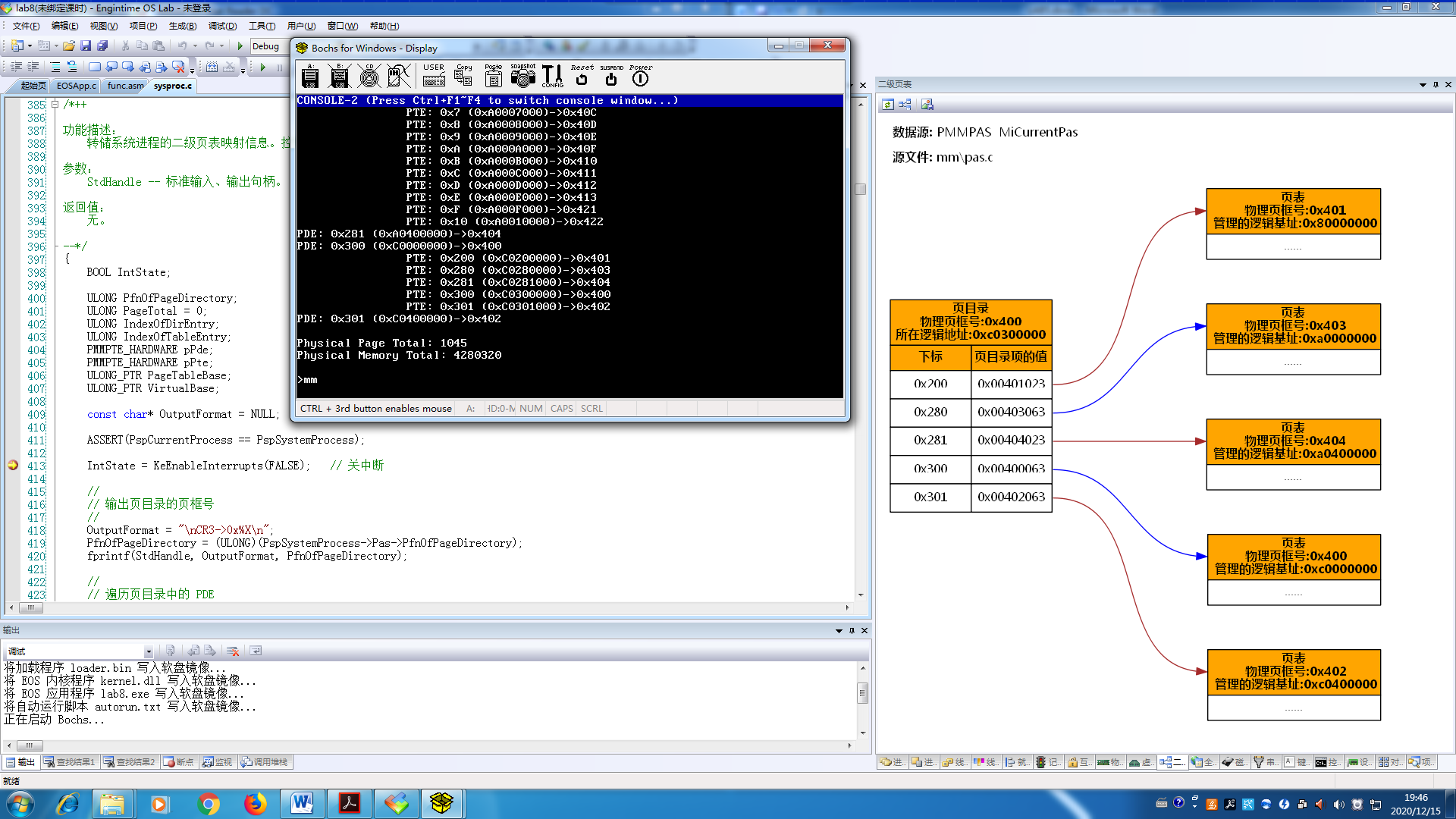
（6）按 F5 启动调试 EOS 应用程序。

（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(b)所示。





3、查看应用程序进程并发时的页目录和页表

（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。

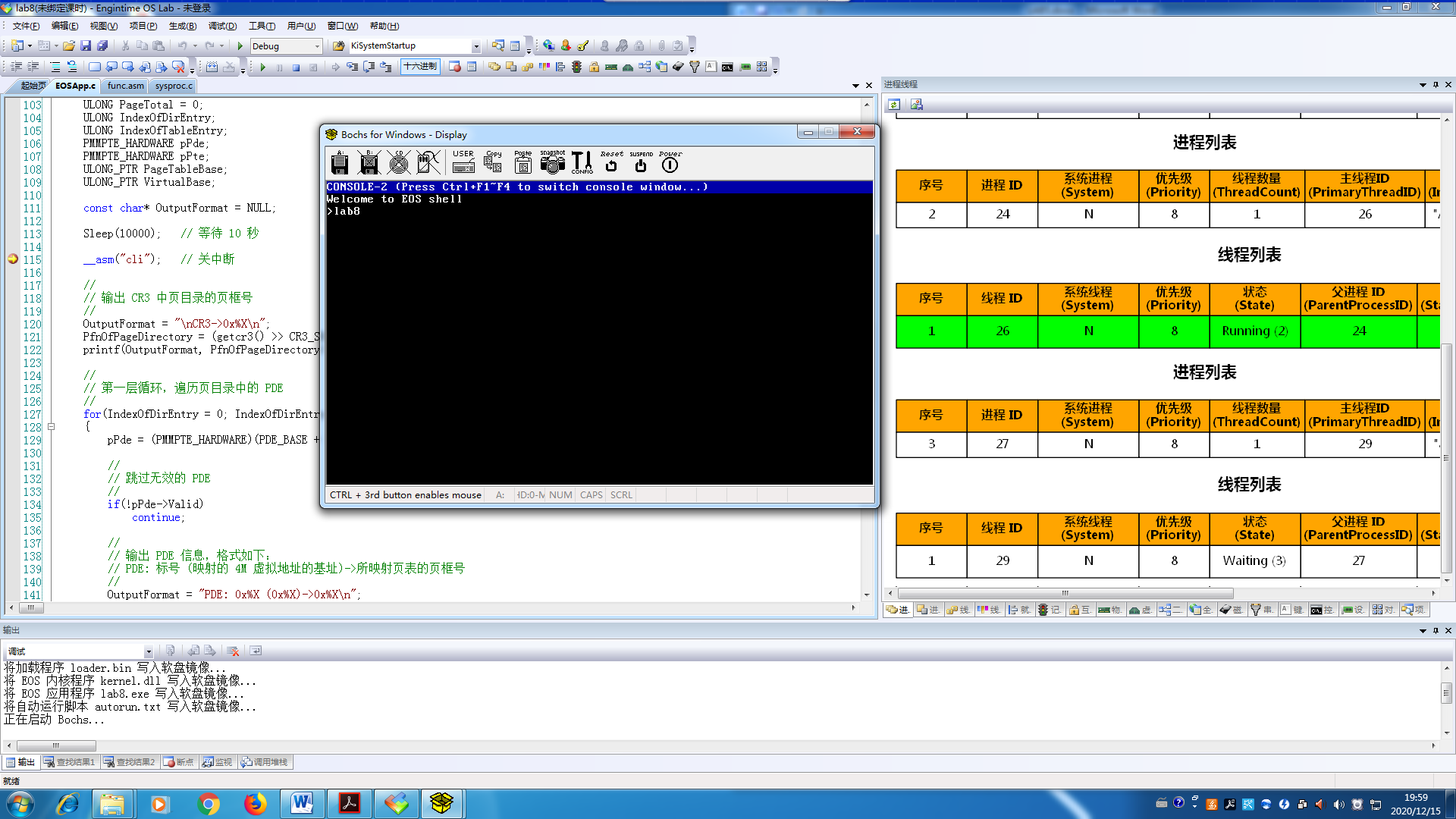
（8）刷新“进程线程”窗口，可以看到当前除了系统进程之外，还有两个应用程序进程。其中，第一个应用程序进程的主线程处于运行状态，也就是说当前中断运行的是第一个应用程序进程的主线程；第二个应用程序进程的主线程处于阻塞状态，说明其正在等待 10 秒钟，然后才能继续运行。

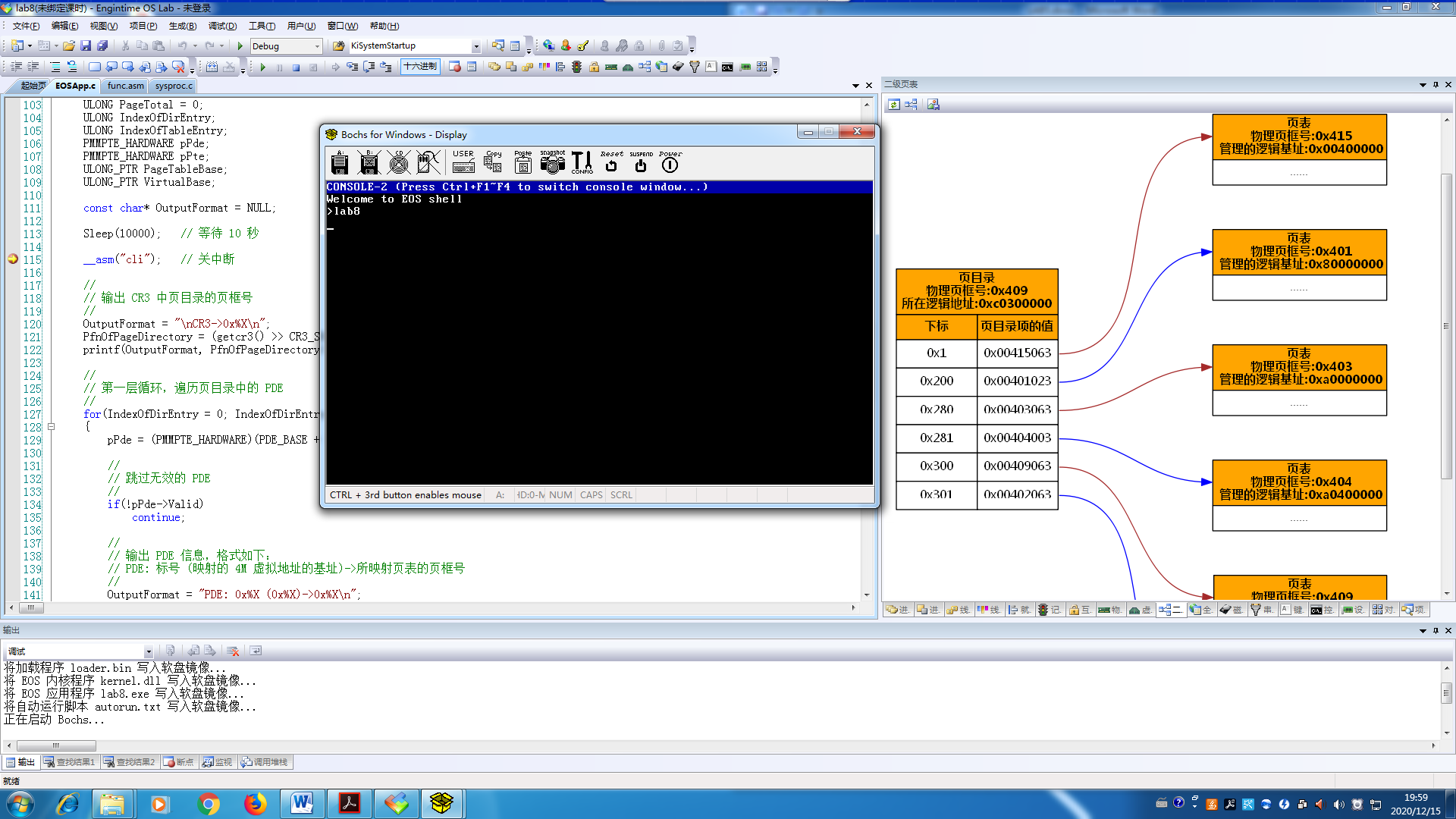
（9）由于当前正在运行的是应用程序的第一个进程，所以在刷新“二级页表”窗口后，可以查看第一个应用程序进程的二级页表映射，如图（a）所示。

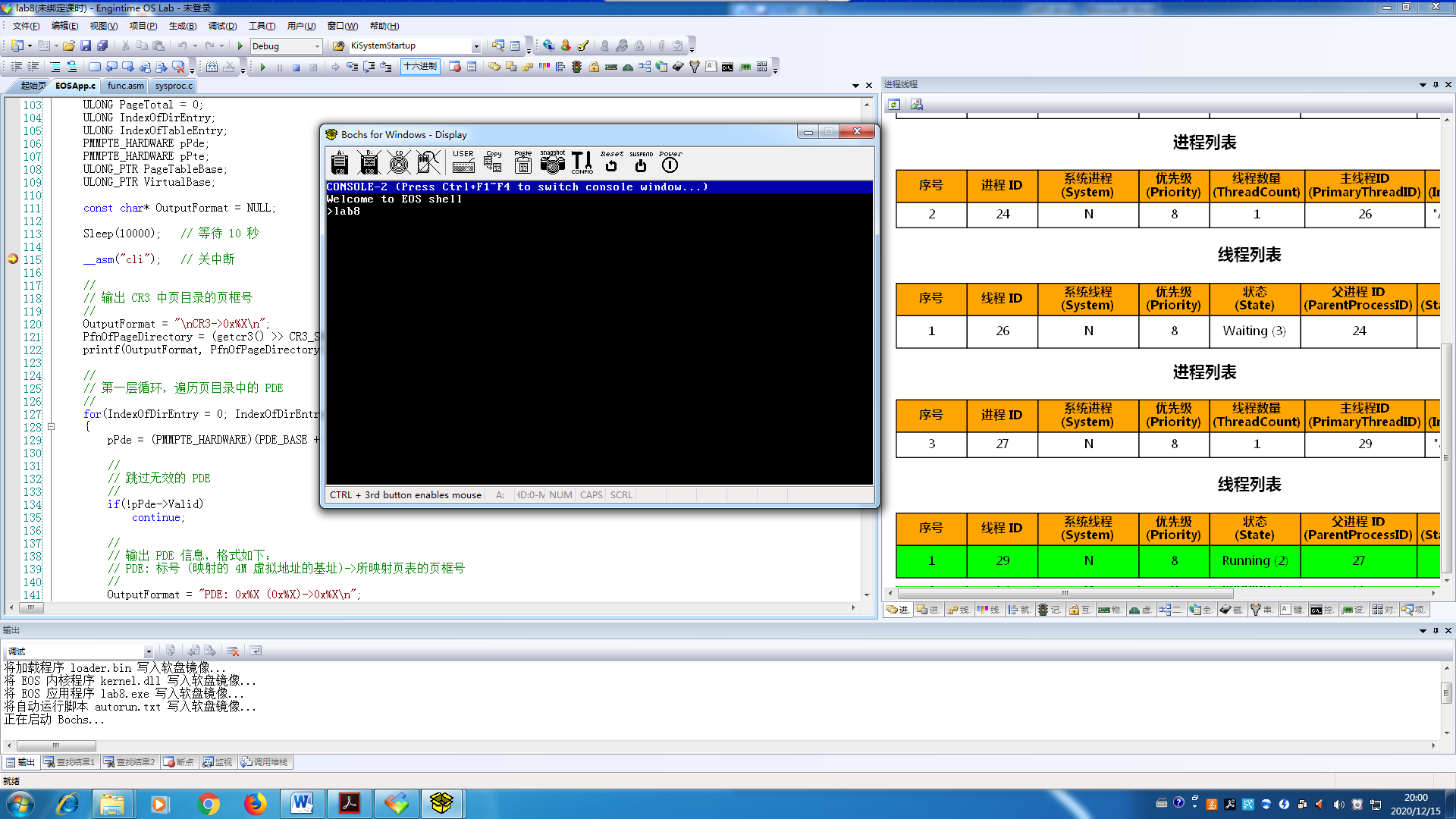
（10）按 F5 继续调试，等待一段时间后，又会命中刚刚添加的断点。

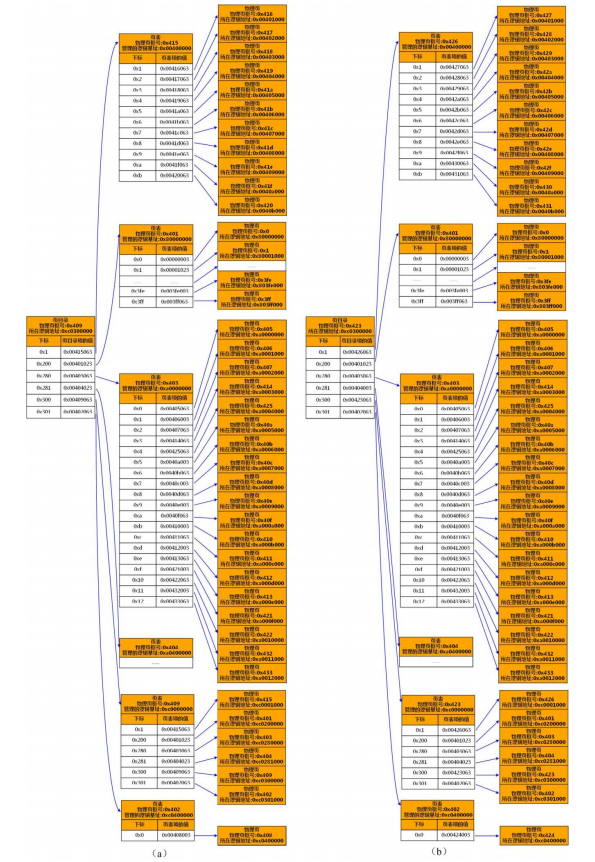
（11）刷新“进程线程”窗口，这次是第一个应用程序进程的主线程处于阻塞状态，说明其打印输出完毕后，在 main 函数中第 188 行处调用 Sleep 函数发生了阻塞；而第二个应用程序进程的主线程处于运行状态，也就是说当前中断运行的是第二个应用程序进程的主线程。

（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（b）所示。









（a）应用程序进程一的二级页表映射

（b）应用程序进程二的二级页表映射

4、在系统进程的二级页表中映射新申请的物理页

（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。

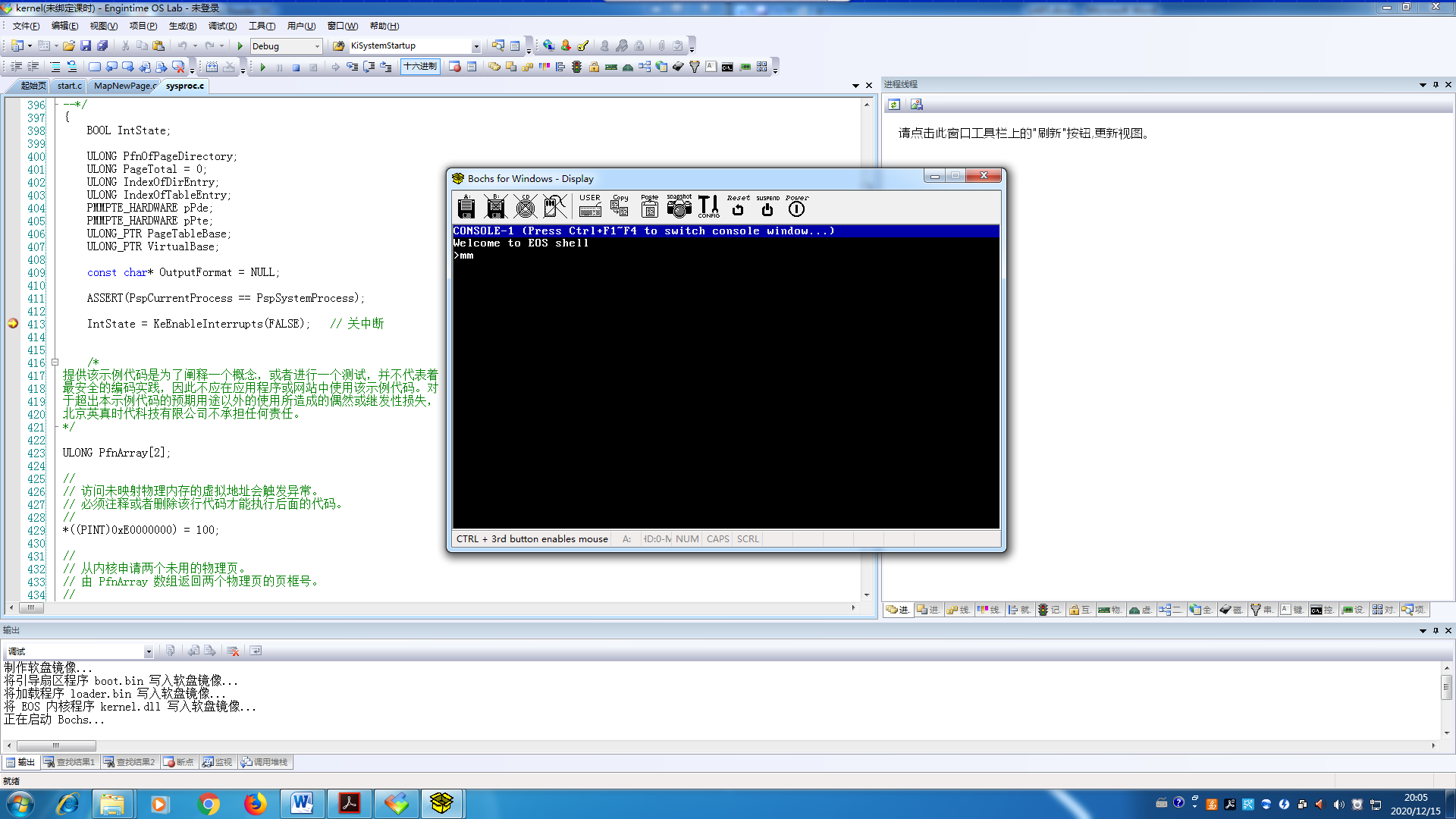
（5）按 F7 生成该内核项目。

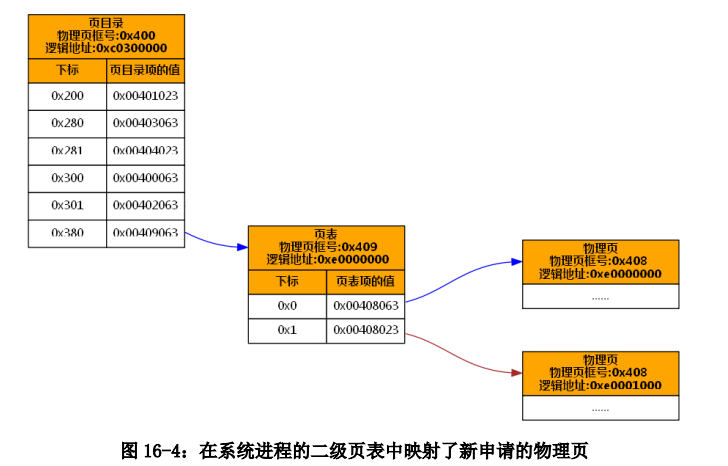
（6）按 F5 启动调试。

（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，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。





在系统进程的二级页表中映射了新申请的物理页

* 1. **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做为虚拟地址的高10 位，PTE标号0x100做为虚拟地址的 12-22位，得到虚拟地址0x40100000。页表:PDE标号0x100做为虚拟地址的高10位，PTE标号0x0做为虚拟地址的12-22位，得到虚拟地址Ox 40000000。

1. 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。参考源代码文件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)。

1. 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。由于并不是所有的页表都有效，所以不能从页表基址0xCO000000开始遍历。

1. 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：

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].PageFrameNumber = 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].PageFrameNumber = 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));

8. 思考页式存储管理机制的优缺点。

答:

优点:

1、由于它不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了碎片问题。

2、动态页式管理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。这既提高了主存的利用纽，又有利于组织多道程序执行。

缺点:

1、要求有相应的硬件支持。例如地址变换机构，缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。这增加了机器成本。

2、增加了系统开销，例如缺页中断处理机，

3、请求调页的算法如选择不当，有可能产生抖动现象。

4、虽然消除了碎片，但每个作业或进程的最后一页内总有一部分空间得不到利用果页面较大，则这一部分的损失仍然较大。