

# 合肥工业大学

## 操作系统实验报告

实验题目	分页存储器管理
学生姓名	付炎平
学    号	2019217819
专业班级	物联网工程19-2班
指导教师	田卫东
完成日期	2021. 11. 27

合肥工业大学 计算机与信息学院

# 实验 8 分页存储器管理

## 1. 实验目的和任务要求

依据操作系统课程所介绍的 XXX，本实验的目的是 XXXX，从而加深对教材上 XXX 的理解。描述实验的具体内容。这部分实验教材上已经列好。

## 2. 实验原理

- ① i386 处理器的二级页表硬件机制
- ② EOS 操作系统的分页存储器管理方式
- ③ 进程地址空间的内存分布

## 3. 实验内容

### 3.1 准备实验

按照下面的步骤准备本次实验：

- 1. 启动 OS Lab。
- 2. 新建一个 EOS Kernel 项目。
- 3. 分别使用 Debug 配置和 Release 配置生成此项目，从而在该项目文件夹中生成完全版本的 EOS SDK 文件夹。
- 4. 新建一个 EOS 应用程序项目。
- 5. 使用在第 3 步生成的 SDK 文件夹覆盖 EOS 应用程序项目文件夹中的 SDK 文件夹。

### 3.2 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。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：

查看关于页表映射的注释如下：

二级页表映射机制将物理内存分为 4K 大小的页，每页对应一个页框号（PFN: Page Frame Number），物理页 0x0-0x0FFF 对应的 PFN 为 0，0x1000-0x1FFF 对应的 PFN 为 1，依此类推。所以，将 PFN 左移 12 位可以获得物理页的基址。

CR3 寄存器在其高 20 位中保存了页目录（Page Directory）的页框号。低 12 位保留未用。



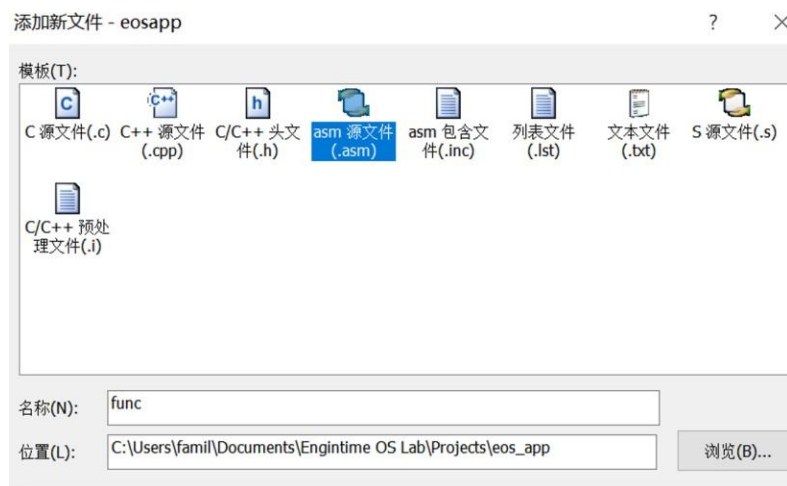
查看相关代码如下：

```

4  ; 于超出本示例代码的预期用途以外的使用所造成的偶然或继发性损失,
5  ; 北京英真时代科技有限公司不承担任何责任。
6  ;
7  ;
8  global _getcr3      ; 将标号声明为全局的, 使其可在外部被引用
9  ;
10 [section .text]    ; 代码段
11 ;
12 ;
13 ; C 语言函数原型: ULONG getcr3()
14 ;
15 _getcr3:
16 {
17     push ebp
18     mov  ebp, esp
19
20     mov  eax, cr3    ; 将 CR3 寄存器中的值复制到 EAX 寄存器中。
21                     ; 根据 C 与汇编的函数调用约定, EAX 寄存器中的值做为函数的返回值。
22     leave
23     ret
24 }

```

- 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”。



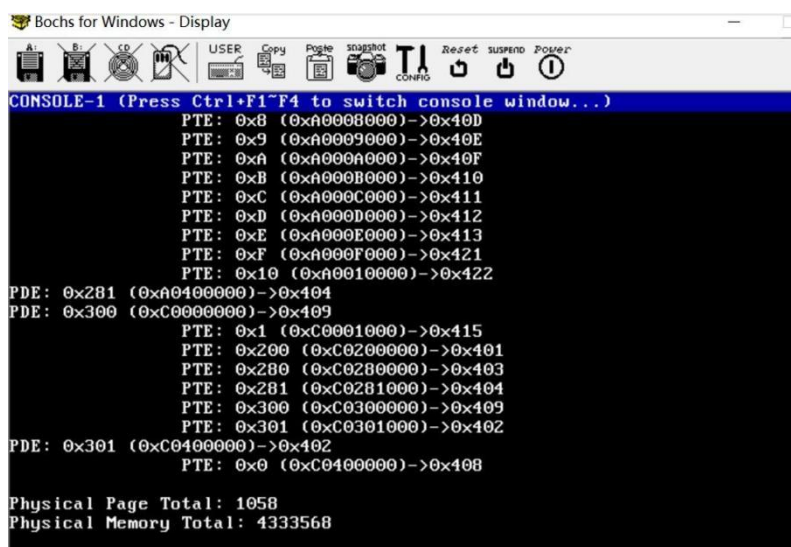
- 5 点击“添加”按钮添加并自动打开文件 func.asm。
- 6 将 getcr3.asm 文件中的源代码复制到 func.asm 文件中。
7. 按 F7 生成修改后的 EOS 应用程序项目。
8. 在 main 函数的返回代码处（第 190 行）添加一个断点。

```

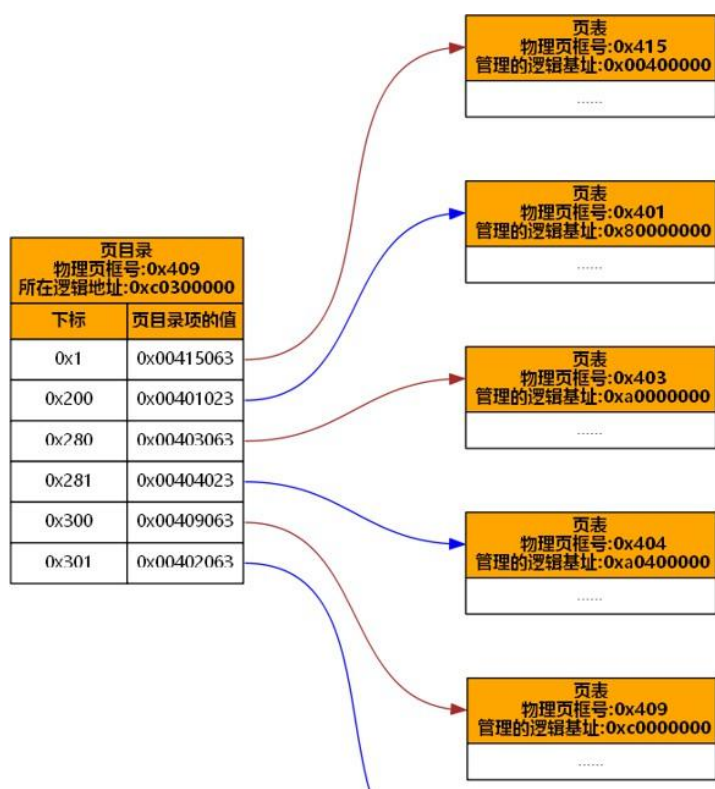
187
188 // Sleep(60000); // 等待 60 秒
189
190 | return 0;
191
192

```

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。



上一张图片缺少了一块，如下：



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：

绘制页表映射的物理页

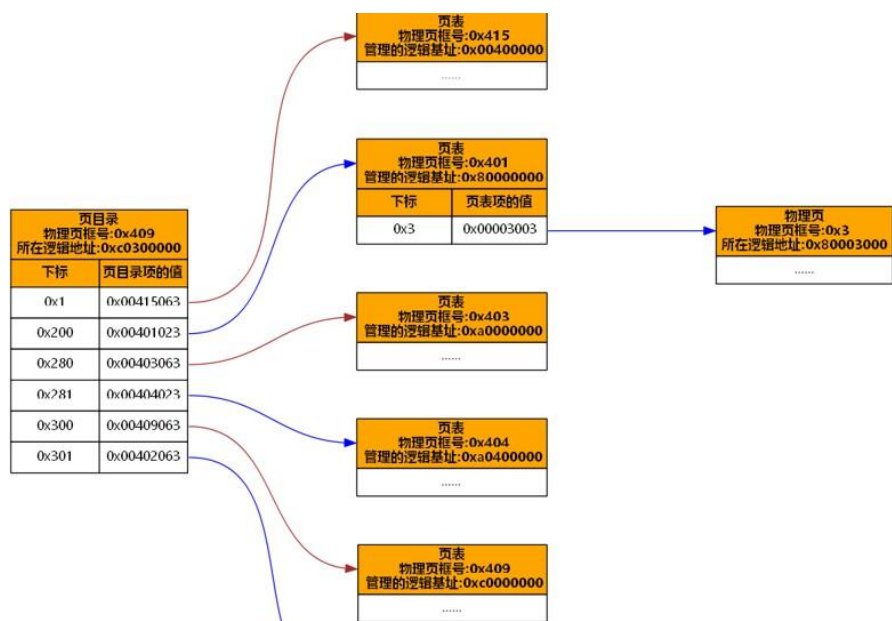
第一步：选择页表

PDE 0x200 映射的页表(物理页框号 0x401)

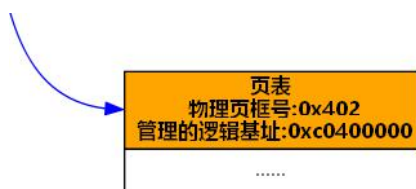
第二步：选择页表映射的物理页进行绘制，最多可选择 20 项

- ☐ PTE 0x0 映射的物理页
- ☐ PTE 0x1 映射的物理页
- ☐ PTE 0x2 映射的物理页
- ☐ PTE 0x3 映射的物理页
- ☐ PTE 0x4 映射的物理页
- ☐ PTE 0x5 映射的物理页
- ☐ PTE 0x6 映射的物理页
- ☐ PTE 0x7 映射的物理页
- ☐ PTE 0x8 映射的物理页
- ☐ PTE 0x9 映射的物理页
- ☐ PTE 0xa 映射的物理页
- ☐ PTE 0xb 映射的物理页
- ☐ PTE 0xc 映射的物理页
- ☐ PTE 0xd 映射的物理页
- ☐ PTE 0xe 映射的物理页
- ☐ PTE 0xf 映射的物理页
- ☐ PTE 0x10 映射的物理页
- ☐ PTE 0x11 映射的物理页
- ☐ PTE 0x12 映射的物理页
- ☐ PTE 0x13 映射的物理页
- ☐ PTE 0x14 映射的物理页
- ☐ PTE 0x15 映射的物理页
- ☐ PTE 0x16 映射的物理页
- ☐ PTE 0x17 映射的物理页
- ☐ PTE 0x18 映射的物理页

绘制的映射如下：



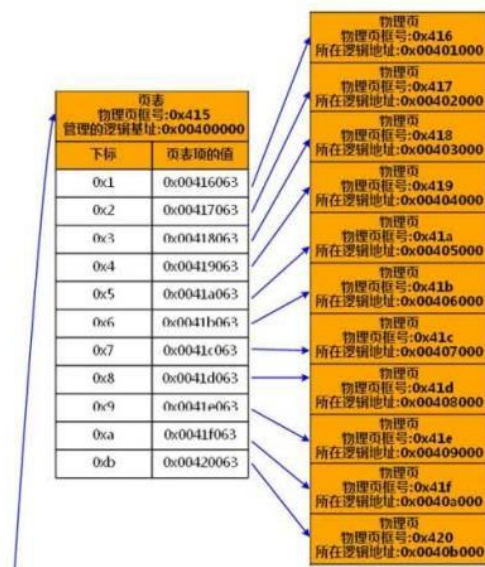
上一幅图有一点缺失如下：



需要对图中显示的内容进行一些说明。首先，页目录所在的物理页框号是从 CR3 寄存器中取得的，并且在页目录中显示了所有有效的 PDE 的下标和 PDE 的值。在各个页表中也同样显示了所有有效的 PTE 的下标和值。注意，由于在“二级页表”中一次只能显示一个页表映射的物理页，所以图 16-2(a) 是综合了各个页表映射的物理页后拼接而成的，在下标为 0x200 的 PDE 对应的页表中，所有的 1024 个 PTE 都映射了物理页，且这 1024 个物理页的物理页框号是连续的，所以在图中省略了一部分 PTE 和其映射的物理页。

还有一点需要特别注意，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。





根据图回答下面的问题：

（1）应用程序进程的页目录和页表一共占用了几个物理页？页框号分别是多少？

答：一共占了 6 个物理页，页框号为：0X409, 0X415, 0X401, 0X403, 0X404, 0X402。

（2）映射用户地址空间（低 2G）的页表的物理页框号是多少？该页表有几个有效的 PTE，或者说有几个物理页用来装载应用程序的代码和数据？物理页框号分别是多少？

答：映射的物理页框号是 0x4150, 该页表有 11 个有效的 PTE，物理页框号分别是：0x416, 0x417, 0x418, 0x419, 0x41a, 0x41b, 0x41c, 0x41d, 0x41e, 0x41f, 0x420。

### 3.3 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

之前已经查看了应用程序进程的二级页表映射信息，接下来读者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。

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 113 行语句的注释（该行语句会让应用程序等待 10 秒）。

```

113 Sleep(10000); // 等待 10 秒
114
115 __asm("cli"); // 关中断

```

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。

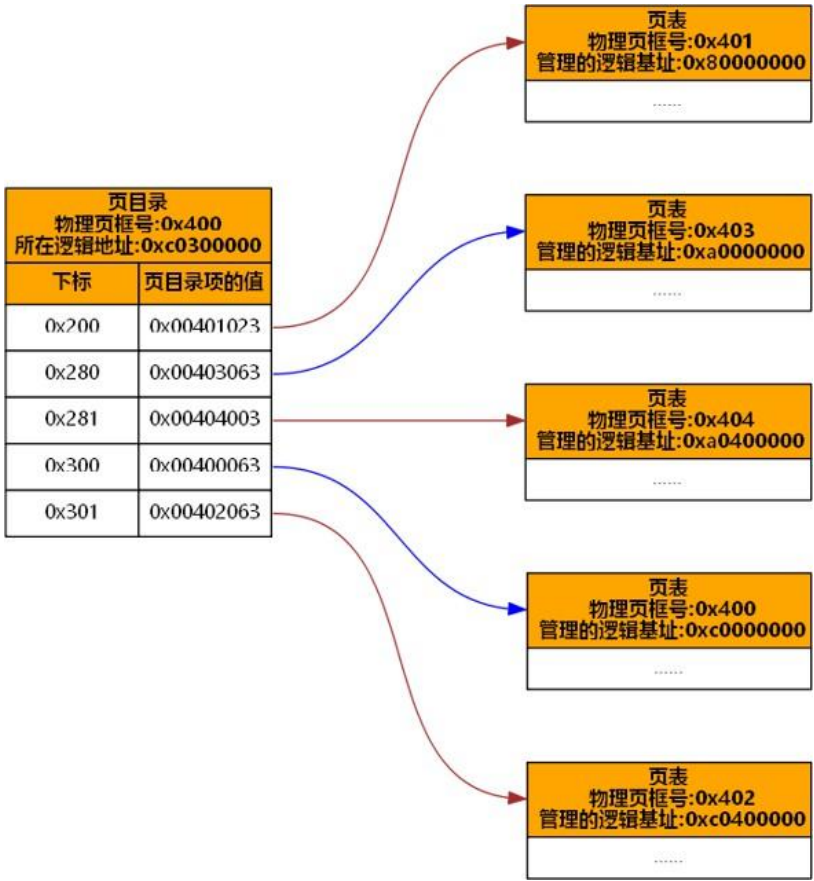
```
412 |
413 | IntState = KeEnableInterrupts(FALSE); // 关中断
```

- 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。

```
409 | const char* OutputFormat = NULL;
410 |
411 | ASSERT(PspCurrentProcess == PspSystemProcess);
412 |
413 | IntState = KeEnableInterrupts(FALSE); // 关中
414 |
415 | // 输出页目录的页框号
416 |
```

CONSOLE-2 (Press Ctrl+F1~F4)  
Welcome to EOS shell  
>mm  
\_

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。

结合图 16-2(a)和(b)回答下面的问题：

（1）EOS 启动后系统进程是一直运行的，所以当创建应用程序进程后，系统中就同时存在了两个进程，这两个进程是否有各自的页目录？在页目录映射的页表



中，哪些是应用程序进程和系统进程共享的，哪些是独占的？

答：系统进程的页目录为 0x400，应用程序的是 0x409，所以系统进程和应用程序进程有各自的页目录。页表映射了用户地址空间，被应用程序进程独占，页框号为 0x415，其他的映射了内核地址空间的页表是共享的。

（2）统计当应用程序进程和系统进程并发时，总共有多少物理页被占用？

答：共有  $11+1024+16+6+1+7+2=1067$  个物理页被占用。

（3）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，将其与图 16-2(b) 比较，思考为什么系统进程（即内核地址空间）占用的物理页会减少？（提示：创建应用程序进程时，EOS 内核要为其创建 PCB，应用程序结束时，内核要回收这些内存资源。）

答：在应用程序结束之后，EOS 内核会回收应用程序进程在内核地址空间中占用的内存，否则应用程序将反复运行，操作系统会出现运行问题。

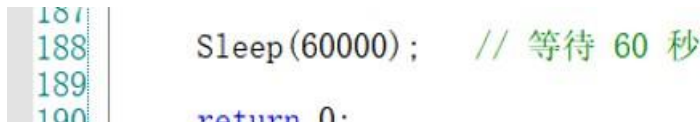
### 3.4 查看应用程序进程并发时的页目录和页表

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。

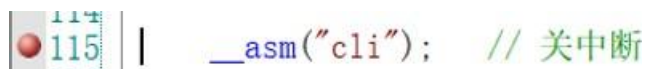
2 取消 EOSApp.c 文件第 188 行语句的注释（该行语句会让应用程序等待 60 秒）。



```
187 |  
188 | Sleep(60000); // 等待 60 秒  
189 |  
190 | return 0;
```

3 按 F7 生成修改后的 EOS 应用程序项目。

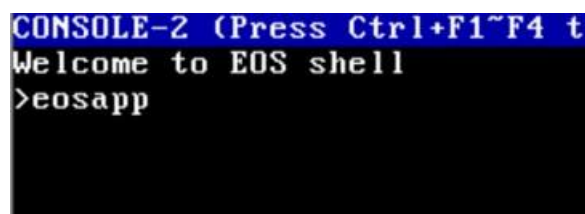
4 在 EOSApp.c 文件的 main 函数中的第 115 行代码处添加一个断点。



```
114 |  
115 | | __asm("cli"); // 关中断  
116 |
```

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。



```
CONSOLE-2 (Press Ctrl+F1~F4 to switch)  
Welcome to EOS shell  
>eosapp
```

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。

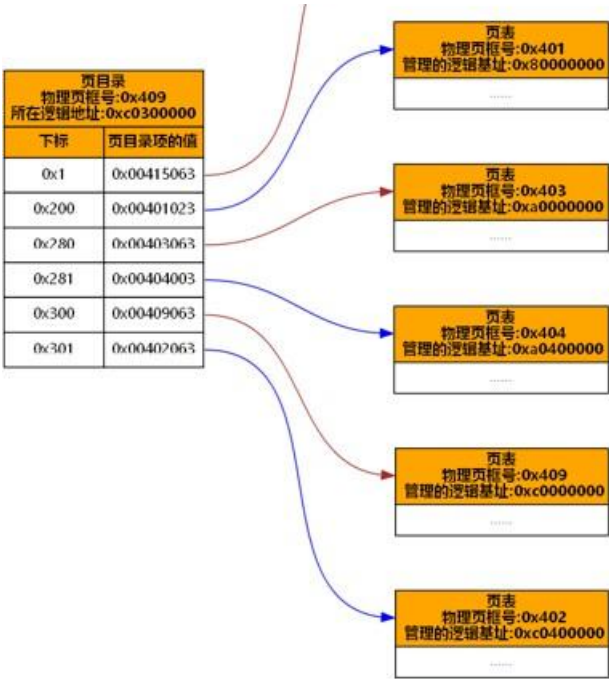
进程列表						
序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	映像名称 (ImageName)
2	24	N	8	1	26	"A:/eosapp.exe"

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	Running (2)	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)	映像名称 (ImageName)
3	27	N	8	1	29	"a:/eosapp.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	Waiting (3)	27	0x8001f97e PspProcessStartup

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。

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/eosapp.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

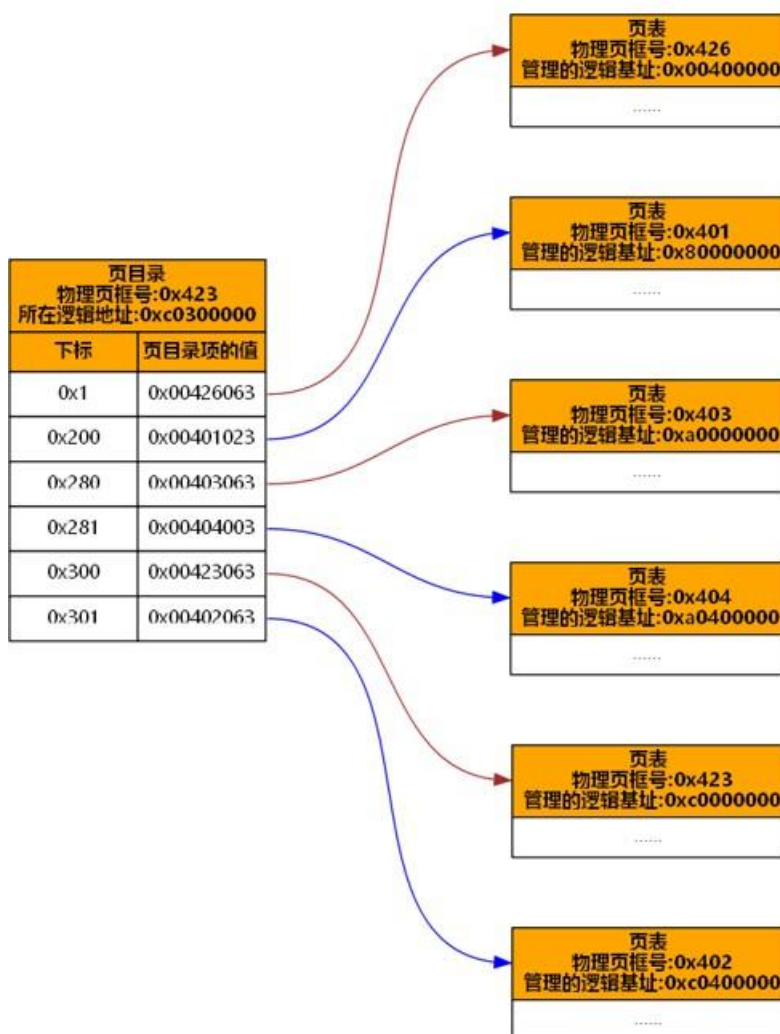
序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
3	27	N	8	1	29	"a/eosapp.exe"

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，如图所示。



观察这两个进程的用户地址空间，可以得出结论：同一个应用程序创建的两个并发的进程，它们的用户虚拟地址空间完全相同，而映射的物理页完全不同，从而保证相同的行为（执行过程）可以在独立的空间内完成。

结合图 16-3(a)和(b)回答下面的问题：

(1) 假设进程一使用的 0x416 和 0x417 物理页保存了应用程序的可执行代码，由于可执行代码通常是不变的、只读的，现在假设优化过的 EOS 允许同一个应用程序创建的多个进程可以共享那些保存了可执行代码的物理页，尝试结合图 16-3 写出共享可执行代码的物理页后进程二用户地址空间的映射信息。并说明共享可执行代码的物理页能带来哪些好处。

答：一一映射如下：

CR3→0X409 CR3→0X423

PDE:0X1(0X00400000)→0X415 0X1(0X00400000)→0X426

进程间如果共享可执行代码的物理页，可以节省运行空间，也可以减少再次调度物理页时候的时间开销。

(2) 前面的问题只是解决了共享可执行代码物理页的问题，其实就算是易变的数据所占用的物理页也是可以共享的。例如在创建进程二时，可以使之共享进程一中那些还没有发生过写操作的数据页，然后当进程一或进程二对共享的数据页进行写操作时，必须先复制一个新的数据页映射到自己的进程空间中，然后再完成写操作，这就是“写时复制”（Copy on write）技术。请读者进一步说明使用“写时复制”技术能带来哪些好处。

答：在地址空间上页的拷贝被推迟到实际发生写入时，该优化可以避免拷贝大量不会被使用的数据，提升运行效率和速度。

(3) 统计当两个应用程序进程并发时，总共有多少物理页被占用？有更多的进程同时运行呢？根据之前的操作方式，尝试在更多的控制台中启动应用程序来验证自己的想法。如果进程的数量足够多，物理内存是否会用尽，如何解决该问题？答：进程一占用了 1061 个物理页，进程二有 13（算复用 14）个，两个应用程序共占用了 1061+13=1074 个。

如果有更多的进程同时运行，则逐一加入映射到用户区的物理数量。当进程的数量足够多时，会有更多的物理页被占用，但物理页的数量是有限的，通过虚拟内存技术，可以将暂时用不到的物理页存储在磁盘上，当要使用时再将它们调入内存。

### 3.5 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：

(1) 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。

(2) 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。

(3) 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。

(4) 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。

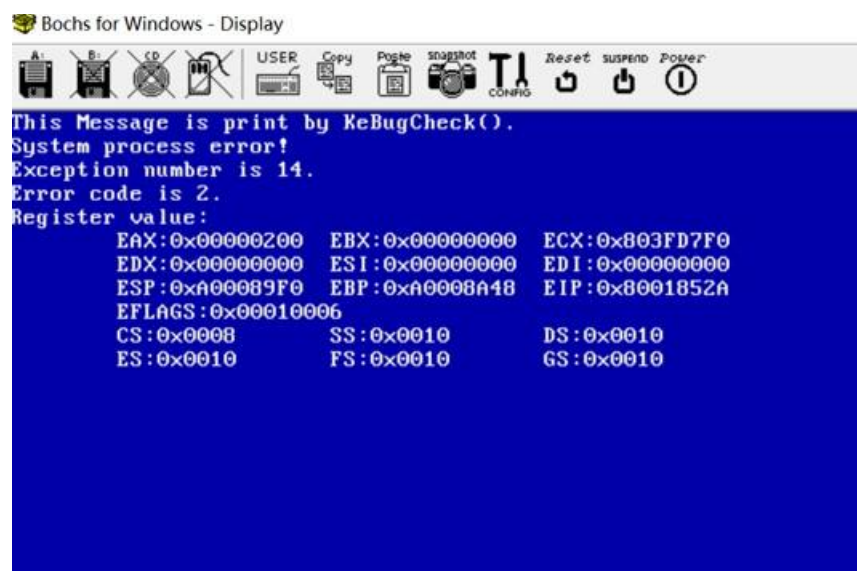
2. 从“项目管理器”打开 ke/sysproc.c 文件。
3. 打开“学生包”本实验对应的文件夹中的 MapNewPage.c 文件（将文件拖动到 OS Lab 窗口中释放即可）。
4. 在 sysproc.c 文件的 ConsoleCmdMemoryMap 函数中找到“关中断”的代码行（第 413 行），将 MapNewPage.c 文件中的代码插入到“关中断”代码行的后面。

```

412
413     IntState = KeEnableInterrupts(FALSE);    // 关中断
414
415     ULONG PfnArray[2];
416
417     //
418     // 访问未映射物理内存的虚拟地址会触发异常。
419     // 必须注释或者删除该行代码才能执行后面的代码。
420     //
421     *((PINT)0xE0000000) = 100;
422
423     //
424     // 从内核申请两个未用的物理页。
425     // 由 PfnArray 数组返回两个物理页的页框号。
426     //

```

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。

按照下面的步骤验证其它结论： 1.

按 F7 生成该内核项目。

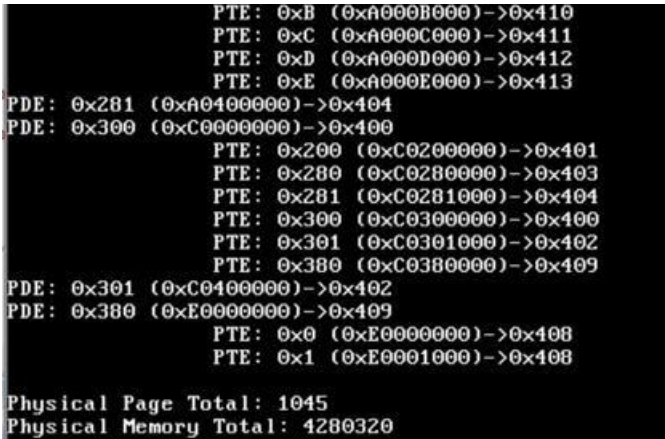
2. 在 sysproc.c 文件的 ConsoleCmdMemoryMap 函数中打开中断的第 539



行处添加一个断点。

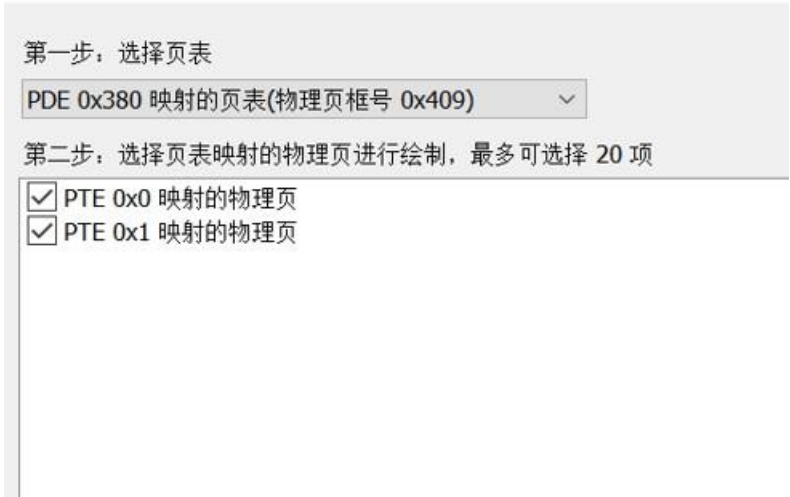
```
537 | OutputFormat = "Physical Memory Total: %d\n\n";
538 | fprintf(StdHandle, OutputFormat, PageTotal * PAGE_SIZE);
539 |
540 | KeEnableInterrupts(IntState); // 开中断
541 | }
```

- 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。



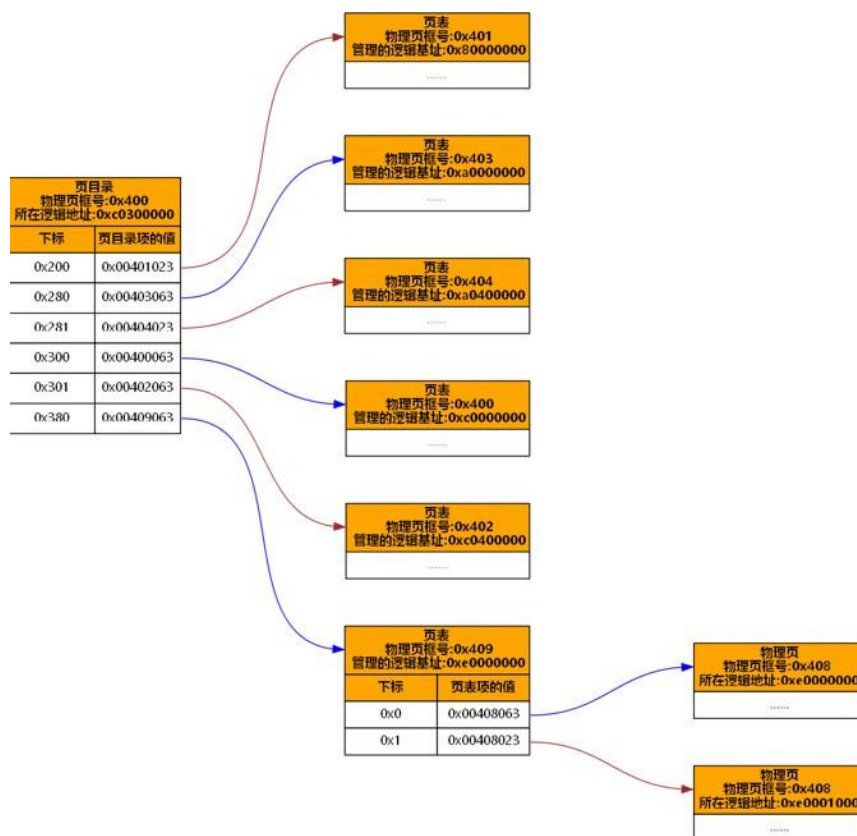
6. 点击“二级页表”窗口工具栏上的“绘制页表映射的物理页”按钮，在弹出的对话框中，从下拉控件中选择“PDE 0x380 映射的页表”项目，然后勾选该页表包含的两个页表项，点击“绘制”按钮，可以查看指定页表项到物理页的映射，如图 16-4 所示。将“二级页表”窗口中的内容保存到一个图像文件中。

绘制页表映射的物理页



绘制结果如下：





在源代码中修改了虚拟地址 0xE0000000 处的内存的值，然后从虚拟地址 0xE0001000 处读取到了相同的值，原因是这两处虚拟地址映射到了相同的物理页。

## 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 0xC0000000-0xC03FFFFFF。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？

答：首先页目录占用 1 个物理页，页框号是 0x409。然后页表占用 5 个物理页，页框号是 0x41D, 0x401, 0x403, 0x404, 0x402。

2. 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：将 0X7C00 换成二进制为 0000 0000 0000 0000 0111 1100 0000 0000，一个物理页的大小为 4K 字节，所以其所在的物理页框号应为 0x00007。

3. 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？而必须先要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：1024 个页表中不是每个页表都是有效的，如果从页表基址开始遍历，无法确定页表的有效性，只有通过页目录中的记录位来确定，所以必须在页目录中查找有效页表。

4. 思考页式存储管理机制的优缺点。

答：缺点是要进行页面中断缺页中断等处理，系统开销较大，有可能产生“抖动”现象。地址变换机构复杂，一般采用硬件实现，添加了机器成本。

优点是虚存量大，适合多道程序运行，动态页式管理提供了内外存统一管理的虚存实现方式。内存利用率高，不要求作业连续存放，有效地解决了内存碎片问题。

## 5. 总结和感想体会

(1)通过本次实验，我学习了 EOS 应用程序进程和系统进程的二级页表映射信息和页目录和页表的管理方式，在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的过程中，我对于应用程序进程和系统进程的二级页表映射信息有了更深刻的了解，对于 EOS 管理进程逻辑地址空间有了清晰的认识。

(2)对于同一个应用程序创建的两个并发的进程，它们的用户虚拟地址空间完全相同，但映射的物理页却完全不同。

(3)在本次实验中我还学习了 i386 处理器的二级页表硬件机制，理解了分页存储管理器的原理，页目录的相关操作。