操作系统原理课程设计指导书

韩彦岭 王静 王令群 卢鹏 编著

上海海洋大学海洋智能信息实验教学示范中心

目 录

实验一　实验环境的使用

一、实验目的

- 熟悉操作系统集成实验环境 OS Lab 的基本使用方法。
- 练习编译、调试 EOS 操作系统内核以及 EOS 应用程序。

二、实验环境

1) Windows 7 及以上系统

2) 集成实验环境 OS Lab

三、实验内容

3.1 启动 OS Lab

1. 在安装有 OS Lab 的计算机上，可以使用两种不同的方法来启动 OS Lab：

- 在桌面上双击“Tevation OS Lab”图标。

或者

- 点击“开始”菜单，在“程序”中的“Tevation OS Lab”中选择“Tevation OS Lab”。

2. OS Lab 每次启动后都会首先弹出一个用于注册用户信息的对话框（可以选择对话框标题栏上的“帮助”按钮获得关于此对话框的帮助信息）。在此对话框中填入学号和姓名后，点击“确定”按钮完成本次注册。

3. 观察 OS Lab 主窗口的布局。OS Lab 主要由下面的若干元素组成：菜单栏、工具栏以及停靠在左侧和底部的各种工具窗口，余下的区域用来放置编辑器窗口。

3.2 学习 OS Lab 的基本使用方法

练习使用 OS Lab 编写一个 Windows 控制台应用程序，熟悉 OS Lab 的基本使用方法（主要包括新建项目、生成项目、调试项目等）。

3.2.1 新建项目

首先需要新建一个项目，步骤如下：

1. 在“文件”菜单中选择“新建”，然后单击“项目”。

2. 在“新建项目”对话框中，选择项目模板“控制台应用程序 (c)”来新建一个使用 C 语言编写的 Windows 控制台应用程序项目。

3. 在“名称”中输入新项目使用的文件夹名称“oslab”。

4. 在“位置”中输入新项目保存在磁盘上的位置“C:\Test”。

5. 点击“确定”按钮。

新建完毕后， OS Lab 会自动打开这个新建的项目。在“项目管理器”窗口中（如图 1-1 所示），树的根节点是项目节点，项目的名称是“console”，各个子节点是项目包含的文件夹或者文件。此项目的源代码主要包含一个头文件“console.h”和一个 C 语言源文件“console.c”。

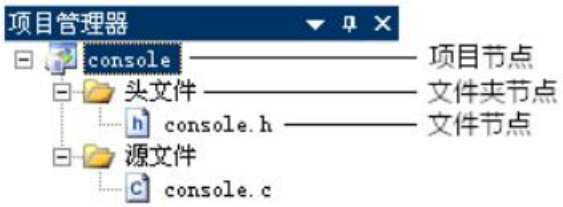


图 1-1：打开 Windows 控制台应用程序项目后的“项目管理器”窗口

使用 Windows 资源管理器打开磁盘上的“C:\test\oslab”文件夹查看项目中包含的文件（提示，在“项目管理器”窗口的项目节点上点击右键，然后在弹出的快捷菜单中选择“打开所在的文件夹”即可）。

3.2.2 生成项目

使用“生成项目”功能可以将程序的源代码文件编译为可执行的二进制文件，方法十分简单：在“生成”菜单中选择“生成项目”。

在项目生成过程中，“输出”窗口会实时显示生成的进度和结果。如果源代码中不包含语法错误，会在最后提示生成成功，如图 1-2：

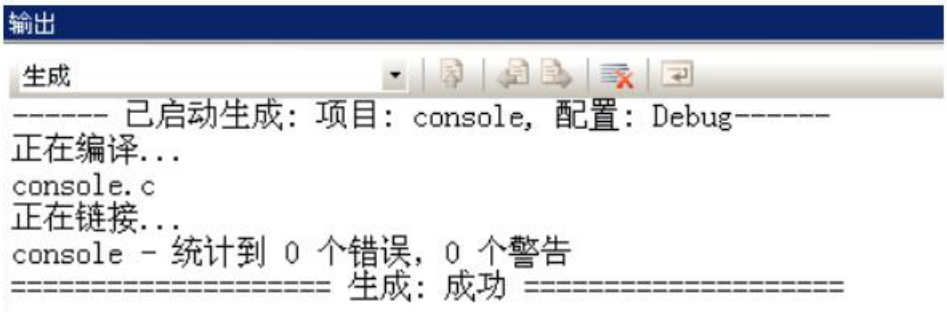


图 1-2：成功生成 Windows 控制台应用程序项目后的“输出”窗口

如果源代码中存在语法错误，“输出”窗口会输出相应的错误信息（包括错

误所在文件的路径，错误在文件中的位置，以及错误原因），并在最后提示生成失败。此时在“输出”窗口中双击错误信息所在的行，OS Lab 会使用源代码编辑器打开错误所在的文件，并自动定位到错误对应的代码行。可以在源代码文件中故意输入一些错误的代码（例如删除一个代码行结尾的分号），然后再次生成项目，然后在“输出”窗口中双击错误信息来定位存在错误的代码行，将代码修改正确后再生成项目。

生成过程是将每个源代码文件（.c、.cpp、.asm 等文件）编译为一个对象文件（.o 文件），然后再将多个对象文件链接为一个目标文件（.exe、.dll 等文件）。成功生成 Windows 控制台应用程序项目后，默认会在“C:\test\oslab\debug″目录下生成一个名称为 “console.o” 的对象文件和名称为“console.exe”的 Windows 控制台应用程序，可以使用 Windows 资源管理器查看这些文件。

3.2.3 执行项目

在 OS Lab 中选择“调试”菜单中的“开始执行(不调试)”，就可以执行此控制台应用程序。启动执行后会弹出一个 Windows 控制台窗口，显示控制台应用程序输出的内容。按任意键即可关闭此控制台窗口。

3.2.4 调试项目

OS Lab 提供的调试器是一个功能强大的工具，使用此调试器可以观察程序的运行时行为并确定逻辑错误的位置，可以中断（或挂起）程序的执行以检查代码，计算和编辑程序中的变量，查看寄存器，以及查看从源代码创建的指令。为了顺利进行后续的各项实验，应该学会灵活使用这些调试功能。

在开始练习各种调试功能之前，首先需要对刚刚创建的例子程序进行必要的修改，步骤如下：

1. 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”。
2. 在弹出的“添加新文件”对话框中选择“C 源文件”模板。
3. 在“名称”中输入文件名称“func”。
4. 点击“添加”按钮，添加并自动打开文件 func.c，此时的“项目管理器”窗口会如图 1-3：

图 1-3：添加 func.c 文件后的“项目管理器”窗口

5. 在 func.c 文件中添加函数：

```
int Func (int n)
{
    n = n + 1;
    return n;
}
```

6. 点击源代码编辑器上方的 console.c 标签，切换到 console.c 文件。将 main 函数修改为：

```
int main (int argc, char* argv[])
{
    int Func (int n); // 声明 Func 函数

    int n = 0;
    n = Func(10);
    printf ("Hello World!\n");
    return 0;
}
```

代码修改完毕后按 F7 生成项目。注意查看“输出”窗口中的内容，如果代码中存在语法错误，就根据提示进行修改，直到成功生成项目。

3.2.4.1 使用断点中断执行

1. 在 main 函数中定义变量 n 的代码行 `int n = 0;`上点击鼠标右键，在弹出的快捷菜单中选择“插入/删除断点”，可以看到在此行左侧的空白处显示了一个红色圆点，表示已经成功在此行代码处添加了一个断点，如图 1-4：

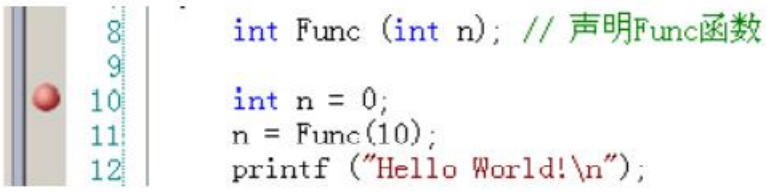


图 1-4：在 console.c 文件的 main 函数中添加断点后的代码行

2. 在“调试”菜单中选择“启动调试”，Windows 控制台应用程序开始执行，随后 OS Lab 窗口被自动激活，并且在刚刚添加断点的代码行左侧空白中显示一个黄色箭头，表示程序已经在此行代码处中断执行（也就是说下一个要执行的就是此行代码），如图 1-5：

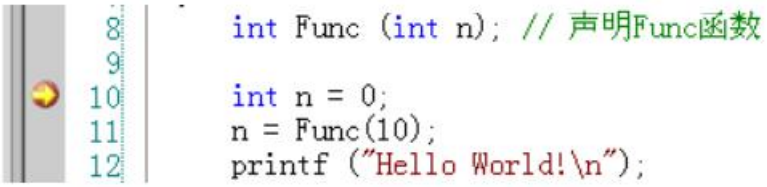


图 1-5：Windows 控制台应用程序启动调试后在断点处中断执行

3. 激活 Windows 控制台应用程序的窗口，可以看到窗口中没有输出任何内容，因为 printf 函数还没有被执行。

3.2.4.2 单步调试

按照下面的步骤练习使用“逐过程”功能：

1. 在 OS Lab 的“调试”菜单中选择“逐过程”，“逐过程”功能会执行黄色箭头当前指向的代码行，并将黄色箭头指向下一个要执行的代码行。

2. 按 F10（“逐过程”功能的快捷键），黄色箭头就指向了调用 printf 函数的代码行。查看控制台应用程序窗口，仍然没有任何输出。

3. 再次按 F10 执行 printf 函数，查看控制台应用程序窗口，可以看到已经打印出了内容。

4. 在“调试”菜单中选择“停止调试”，结束此次调试。

按照下面的步骤练习使用“逐语句”功能和“跳出”功能：

1. 按 F5（“启动调试”功能的快捷键），仍然会在之前设置的断点处中断。

2. 按 F10 逐过程调试，此时黄色箭头指向了调用函数 Func 的代码行。

3. 在“调试”菜单中选择“逐语句”，可以发现黄色箭头指向了函数 Func 中，说明“逐语句”功能可以进入函数，从而调试函数中的语句。

4. 选择“调试”菜单中的“跳出”，会跳出 Func 函数，返回到上级函数中

继续调试（此时 Func 函数已经执行完毕）。

5. 按 Shift+F5（“停止调试”功能的快捷键），结束此次调试。

练习使用“逐过程”、“逐语句”和“跳出”功能，注意体会“逐过程”和“逐语句”的不同。

3.2.4.3 查看变量的值

在调试的过程中，OS Lab 提供了三种查看变量值的方法，按照下面的步骤练习这些方法：

1. 按 F5 启动调试，仍然会在之前设置的断点处中断。

2. 将鼠标移动到源代码编辑器中变量 n 的名称上，此时会弹出一个窗口显示出变量 n 当前的值（由于此时还没有给变量 n 赋值，所以是一个随机值）。

3. 在源代码编辑器中变量 n 的名称上点击鼠标右键，在弹出的快捷菜单中选择“快速监视”，可以使用“快速监视”对话框查看变量 n 的值。然后，可以点击“关闭”按钮关闭“快速监视”对话框。

4. 在源代码编辑器中变量 n 的名称上点击鼠标右键，在弹出的快捷菜单中选择“添加监视”，变量 n 就被添加到了“监视”窗口中。使用“监视”窗口可以随时查看变量的值和类型。此时按 F10 进行一次单步调试，可以看到“监视”窗口中变量 n 的值会变为 0，如图 1-6：

图 1-6：使用“监视”窗口查看变量的值和类型

如果需要使用十进制查看变量的值，可以点击工具栏上的“十六进制”按钮，从而在十六进制和十进制间切换。自己练习使用不同的进制和不同的方法来查看变量的值，然后结束此次调试。

3.2.4.4 调用堆栈

使用“调用堆栈”窗口可以在调试的过程中查看当前堆栈上的函数，还可以帮助理解函数的调用层次和调用过程。按照下面的步骤练习使用“调用堆栈”窗口：

1. 按 F5 启动调试，仍然会在之前设置的断点处中断。

2. 选择“调试”菜单“窗口”中的“调用堆栈”，激活“调用堆栈”窗口。可以看到当前“调用堆栈”窗口中只有一个 main 函数（显示的内容还包括了参数值和函数地址）。

3. 按 F11（“逐语句”功能的快捷键）调试，直到进入 Func 函数，查看“调用堆栈”窗口可以发现在堆栈上有两个函数 Func 和 main。其中当前正在调试的 Func 函数在栈顶位置，main 函数在栈底位置。说明是在 main 函数中调用了 Func 函数。

4. 在“调用堆栈”窗口中双击 main 函数所在的行，会有一个绿色箭头指向 main 函数所在的行，表示此函数是当前调用堆栈中的活动函数。同时，会将 main 函数所在的源代码文件打开，并也使用一个绿色箭头指向 Func 函数返回后的位置。

5. 在“调用堆栈”窗口中双击 Func 函数所在的行，可以重新激活此堆栈帧，并显示对应的源代码。

6. 反复双击“调用堆栈”窗口中 Func 函数和 main 函数所在的行，查看“监视”窗口中变量 n 的值，可以看到在不同的堆栈帧被激活时，OS Lab 调试器会自动更新“监视”窗口中的数据，显示出对应于当前活动堆栈帧的信息。

7. 结束此次调试。

3.3 EOS 内核的编译和调试

之前练习了对 Windows 控制台应用程序项目的各项操作，对 EOS 内核项目的各种操作（包括新建、生成和各种调试功能等）与对 Windows 控制台项目的操作是完全一致的。所以，接下来实验内容的重点不再是各种操作的具体步骤，而应将注意力放在对 EOS 内核项目的理解上。

3.3.1 新建项目

新建一个 EOS 内核项目的步骤如下：

1. 在“文件”菜单中选择“新建”，然后单击“项目”。

2. 在“新建项目”对话框中，选择项目模板“EOS Kernel”。

3. 在“名称”中输入新项目使用的文件夹名称“eos”。

4. 在“位置”中输入新项目保存在磁盘上的位置“C:\”。

5. 点击“确定”按钮。

此项目就是一个 EOS 操作系统内核项目，包含了 EOS 操作系统内核的所有源代码文件。

在“项目管理器”窗口中查看 EOS 内核项目包含的文件夹和源代码文件，可以看到不同的文件夹包含了 EOS 操作系统不同模块的源代码文件，例如“mm”文件夹中包含了内存管理模块的源代码文件，“boot”文件夹中包含了软盘引导扇区程序和加载程序的源代码文件。也可以使用 Windows 资源管理器打开项目所在的文件夹 C:\eos，查看所有源代码文件。

3.3.2 生成项目

1. 按 F7 生成项目，同时查看“输出”窗口中的内容，确认生成成功。

2. 打开 C:\eos\debug 文件夹，查看生成的对象文件和目标文件。找到 boot.bin、loader.bin 和 kernel.dll 三个二进制文件，这三个文件就是 EOS 操作系统在运行时需要的可执行文件。OS Lab 每次启动运行 EOS 操作系统之前，都会将这三个文件写入一个软盘镜像文件中，然后让虚拟机运行这个软盘镜像中的 EOS（相当于将写有这三个二进制文件的软盘放入一个物理机的软盘驱动器中，然后按下开机按钮）。找到 libkernel.a 文件，此文件是 EOS 内核文件 kernel.dll 对应的导入库文件。

3.3.3 调试项目

1. 在“项目管理器”窗口的 ke 文件夹中找到 start.c 文件节点，双击此文件节点使用源代码编辑器打开 start.c 文件。

2. 在 start.c 文件中 KiSystemStartup 函数的“KiInitializePic();”语句所在行（第 61 行）添加一个断点，如图 1-7。EOS 启动时执行的第一个内核函数就是 KiSystemStartup 函数。

图 1-7：在 EOS 内核项目的 ke/start.c 文件中添加一个断点

3. 按 F5 启动调试，虚拟机开始运行软盘镜像中的 EOS。在虚拟机窗口中可以看到 EOS 启动的过程。随后 EOS 会在刚刚添加的断点处中断执行。激活虚拟机窗口可以看到 EOS 也不再继续运行了。各种调试功能（包括单步调试、查看变量的值和各个调试工具窗口）的使用方法与调试 Windows 控制台程序完全

相同，可以自己练习。

4. 按 F5 继续执行。查看虚拟机窗口，显示 EOS 操作系统已经启动，并且 EOS 的控制台程序已经开始运行了。

5. 在“调试”菜单中选择“停止调试”，结束此次调试。

3.3.4 软盘镜像

在“项目管理器”窗口中双击软盘镜像文件 Floppy.img，就会使用 FloppyImageEditor 工具打开此文件。在 FloppyImageEditor 工具中按 F1 可以查看此工具的帮助文件。在 FloppyImageEditor 工具的文件列表中可以找到 loader.bin 文件和 kernel.dll 文件（不用关心其他文件），这两个文件都是在启动调试时被写入软盘镜像文件的（注意这两个文件的修改日期），boot.bin 文件在启动调试时被写入了软盘镜像的引导扇区中，不受软盘文件系统的管理，所以在文件列表中找不到此文件。关闭 FloppyImageEditor 工具。

3.3.5 EOS SDK（Software Development Kit）文件夹

1. 点击 OS Lab 工具栏上的“项目配置”下拉列表，选择下拉列表中的“Release”项目配置，“Release”项目配置被设置为新的活动项目配置（原来的活动项目配置是“Debug”）。如图 1-8：

图 1-8：使用工具栏上的“项目配置”下拉列表切换活动项目配置

2. 按 F7 使用 Release 配置生成项目。

3. 生成完毕后，使用 Windows 资源管理器打开 C:\eos 文件夹，可以发现在文件夹中多出了一个 SDK 文件夹，此文件夹就是在生成 EOS Kernel 项目的同时自动生成的。

4. SDK 文件夹中提供了开发 EOS 应用程序需要的所有文件。打开 SDK 文件夹中的 bin 文件夹，可以看到有两个名称分别为 debug 和 release 的文件夹。debug 文件夹是在使用 debug 配置生成项目时生成的，其中存放了调试版本的 EOS 二进制文件。release 文件夹是在使用 release 配置生成项目时生成的，其中存放了发布版本的 EOS 二进制文件（不包含调试信息）。分别打开这两个文件夹查看其中包含的文件。

5. 打开 SDK 文件夹中的 inc 文件夹，可以看到此文件夹中存放了 EOS 用于导出 API 函数和重要数据类型定义的头文件，在编写 EOS 应用程序时必须包含这些头文件。

每次在开发 EOS 应用程序之前都应该使用 EOS Kernel 项目的 debug 配置和 release 配置来生成 EOS Kernel 项目，这样才能够得到完全版本的 SDK 文件夹供 EOS 应用程序使用。

3.4 EOS 应用程序的编译和调试

3.4.1 新建项目

新建一个 EOS 应用程序项目的步骤如下：

1. 在“文件”菜单中选择“新建”，然后单击“项目”。
2. 在“新建项目”对话框中，选择项目模板“EOS 应用程序”。
3. 在“名称”中输入新项目使用的文件夹名称“eosapp”。
4. 在“位置”中输入新项目保存在磁盘上的位置“C:\”。
5. 点击“确定”按钮。

此项目就是一个 EOS 应用程序项目。

使用 Windows 资源管理器将之前生成的 C:\eos\sdk 文件夹拷贝覆盖到 C:\eosapp\sdk 位置。这样 EOS 应用程序就可以使用最新版本的 EOS SDK 文件夹了。

3.4.2 生成项目

1. 按 F7 生成项目，同时查看“输出”窗口中的内容，确认生成成功。
2. 打开 C:\eosapp\debug 文件夹，查看生成的对象文件和目标文件。其中的 EOSApp.exe 就是 EOS 应用程序的可执行文件。OS Lab 每次启动执行 EOS 应用程序时，都会将 EOS 应用程序的可执行文件写入软盘镜像，并且会将 SDK 文件夹中对应配置（Debug 或 Release）的二进制文件写入软盘镜像，然后让虚拟机运行软盘镜像中的 EOS，待 EOS 启动后再自动执行 EOS 应用程序。

3.4.3 调试项目

调试 EOS 应用程序项目与之前的两个项目有较大的不同，之前的两个项目在调试时都是先添加断点再启动调试，而 EOS 应用程序项目必须先启动调试再添加断点，步骤如下：

1. 按 F5 启动调试。OS Lab 会弹出一个调试异常对话框，选择“是”调试异常，EOS 应用程序会中断执行，黄色箭头指向下一个要执行的代码行。

2. 在 eosapp.c 的

printf("Hello world!\n");

代码行添加一个断点，然后按 F5 继续调试，在此断点处中断。

3. 按 F10 单步调试，查看虚拟机窗口，打印输出了“Hello world!”。

4. 按 F5 继续调试，查看虚拟机窗口，EOS 应用程序执行完毕。

5. 在“调试”菜单中选择“停止调试”，调试被终止。

6. 选择“调试”菜单中的“删除所有断点”。只有删除所有断点后才能按 F5 再次启动调试，否则启动调试会失败。

3.4.4 软盘镜像

使用 FloppyImageEditor 工具打开该项目中的 Floppy.img 文件，查看软盘镜像中的文件。loader.bin 和 kernel.dll 是从 C:\eosapp\sdk\bin\debug 文件夹写入的，C:\eosapp\sdk\bin\debug\boot.bin 被写入了软盘镜像文件的引导扇区中。eosapp.exe 就是本项目生成的 EOS 应用程序。EOS 操作系统启动后会根据 autorun.txt 文本文件中的内容启动执行 eosapp.exe 程序，双击 autorun.txt 文件查看其内容。

3.4.5 修改 EOS 应用程序项目名称

EOS 应用程序项目所生成的可执行文件的名称默认是由项目名称决定的。由于当前 EOS 应用程序项目的名称是 EOSApp，所以该项目所生成的可执行文件的名称默认为 EOSApp.exe。按照下面的步骤修改 EOS 应用程序项目的名称，进而修改可执行文件的名称：

1. 在“项目管理器”窗口中，右键点击项目节点（根节点）。

2. 在弹出的快捷菜单中选择“重命名”，然后可以输入一个新的项目名称，例如“MyApp”，然后按回车键使修改生效。

3. 按 F7 生成项目。

4. 选择“调试”菜单中的“删除所有断点”。

5. 按 F5 启动调试。OS Lab 会弹出一个调试异常对话框，选择“否”忽略异常，EOS 应用程序会自动执行。

6. 激活虚拟机窗口，可以看到自动执行的可执行文件的名称为 MyApp.exe，如图 1-9 所示。也可以打开 C:\eosapp\debug 文件夹，确认生成了可执行文件 MyApp.exe。

图 1-9：EOS 应用程序项目的名称修改后的执行

3.5 退出 OS Lab

1. 在“文件”菜单中选择“退出”。

2. 在 OS Lab 关闭前会弹出一个保存数据对话框（可以选择对话框标题栏上的“帮助”按钮获得帮助信息），核对学号和姓名无误后点击“保存”按钮，OS Lab 关闭。

3. 在 OS Lab 关闭后默认会自动使用 Windows 资源管理器打开数据文件所在的文件夹，并且选中刚刚保存的数据文件（OUD 文件）。将数据文件备份（例如拷贝到自己的 U 盘中或者发送到服务器上），可以做为本次实验的考评依据。

3.6 保存 EOS 内核项目

如果要在课余时间阅读 EOS 源代码，或者调试 EOS 源代码，可以按照下面的步骤操作：

1. 使用 OS Lab 重新打开之前创建的 EOS 内核项目。在“起始页”的“最近的项目”列表中会有内核项目的快捷方式。

2. 使用 Debug 配置生成此项目。再此启动调试此项目后结束调试。

3. 将此项目复制到自己的计算机中。注意，项目在磁盘中的位置不能改变，例如实验中此项目在 C:\eos 位置，就必须复制到自己计算机中的 C:\eos 位置。

4. 在自己的计算机中安装 OS Lab 演示版，使用演示版程序阅读 EOS 源代码，或者调试 EOS 源代码。OS Lab 演示版程序可以使用教师分发的安装包进行安装。注意，必须使用和正式版版本号相同的演示版程序。

实验二　进程的创建

一、实验目的

- 编程使用 EOS API 函数 CreateProcess 创建一个进程，掌握创建进程的方法，理解进程和程序的区别。
- 调试跟踪 CreateProcess 函数的执行过程，了解进程的创建过程，理解进程是资源的分配单位。

二、实验环境

1) Windows 7 及以上系统
2) 集成实验环境 OS Lab

三、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. 使用在第三步生成的 SDK 文件夹覆盖应用程序项目文件夹中的 SDK 文件夹。

在后面的实验中，会通过编程的方式让此应用程序在执行的过程中创建另外一个应用程序的进程（也就是启动执行另外一个应用程序）。

3.2 练习使用控制台命令创建应用程序的进程

练习使用控制台命令创建 EOS 应用程序进程的具体步骤如下：

1. 在 EOS 应用程序项目的“项目管理器”窗口中双击 Floppy.img 文件，使用 FloppyImageEditor 工具打开此软盘镜像文件。
2. 将本实验文件夹中的 Hello.exe 文件拖动到 FloppyImageEditor 工具窗

口的文件列表中释放，Hello.exe 文件即被添加到软盘镜像文件中。Hello.exe 一个 EOS 应用程序，其源代码可以参见本实验文件夹中的 Hello.c 源文件。

3. 在 FloppyImageEditor 中选择“文件”菜单中的“保存”后关闭 FloppyImageEditor。
4. 按 F7 生成 EOS 应用项目。
5. 按 F5 启动调试。OS Lab 会弹出一个调试异常对话框，并中断应用程序的执行。
6. 在调试异常对话框中选择“否”，忽略异常继续执行应用程序。
7. 激活虚拟机窗口，待该应用程序执行完毕后，在 EOS 的控制台中输入命令“A:\Hello.exe”后回车。
8. Hello.exe 应用程序开始执行，观察其输出如图 2-1。
9. 待 Hello.exe 执行完毕后可以重复第 7 步，或者结束此次调试。

图 2-1：使用控制台命令创建 EOS 应用程序的进程

3.3 通过编程的方式让应用程序创建另一个应用程序的进程

使用 OS Lab 打开本实验文件夹中的 NewProc.c 文件（将此文件拖动到 OS Lab 窗口中释放即可），仔细阅读此文件中的源代码和注释，main 函数的流程图可以参见图 1-3。

按照下面的步骤查看应用程序创建另一个应用程序的进程的执行结果：

1. 使用 NewProc.c 文件中的源代码替换之前创建的 EOS 应用程序项目中的 EOSApp.c 文件内的源代码。

2. 按 F7 生成修改后的 EOS 应用程序项目。

3. 按 F5 启动调试。OS Lab 会首先弹出一个调试异常对话框。

4. 在调试异常对话框中选择“否”，继续执行。

5. 激活虚拟机窗口查看应用程序输出的内容，如图 2-2。结合图 2-1，可以看到父进程（EOSApp.exe）首先开始执行并输出内容，父进程创建了子进程（Hello.exe）后，子进程开始执行并输出内容，待子进程结束后父进程再继续执行。

6. 结束此次调试。

图 2-2：应用程序创建另一个应用程序的进程的执行结果

3.4 调试 CreateProcess 函数创建进程的过程

按照下面的步骤调试 CreateProcess 函数创建进程的过程：

1. 按 F5 启动调试 EOS 应用程序，OS Lab 会首先弹出一个调试异常对话框。

2. 选择“是”调试异常，调试会中断。

3. 在 main 函数中调用 CreateProcess 函数的代码行（第 57 行）添加一个断点。

4. 按 F5 继续调试，在断点处中断。

5. 按 F11 调试进入 CreateProcess 函数。此时已经开始进入 EOS 内核进行调试，可以参见图 2-4。

从图 2-4 中可以看到，当 EOS 应用程序 eosapp.exe 存储在软盘上的时候，它是静态的，只包含应用程序的指令和数据。而创建进程后，进程不但包含应用

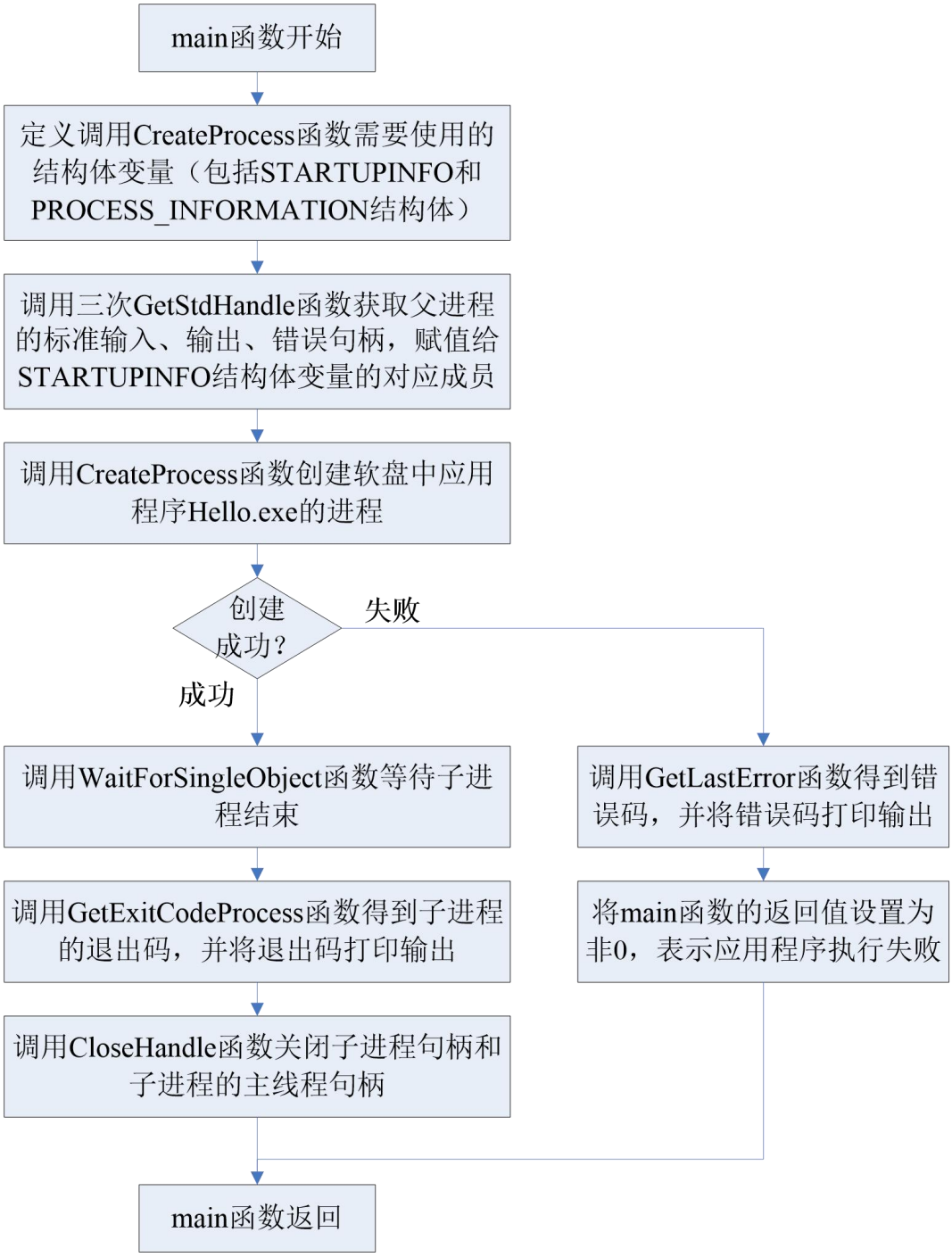


图 2-3：main 函数流程图

程序的指令和数据，也会包含操作系统内核（kernel.dll）的指令和数据。同时，图 2-4 也说明了一个进程可以包含多个程序，该进程包含了 eosapp.exe 和 kernel.dll 两个程序。

可以按照下面的步骤来分别验证应用程序和操作系统内核在进程的 4G 虚拟地址空间中所处的位置：

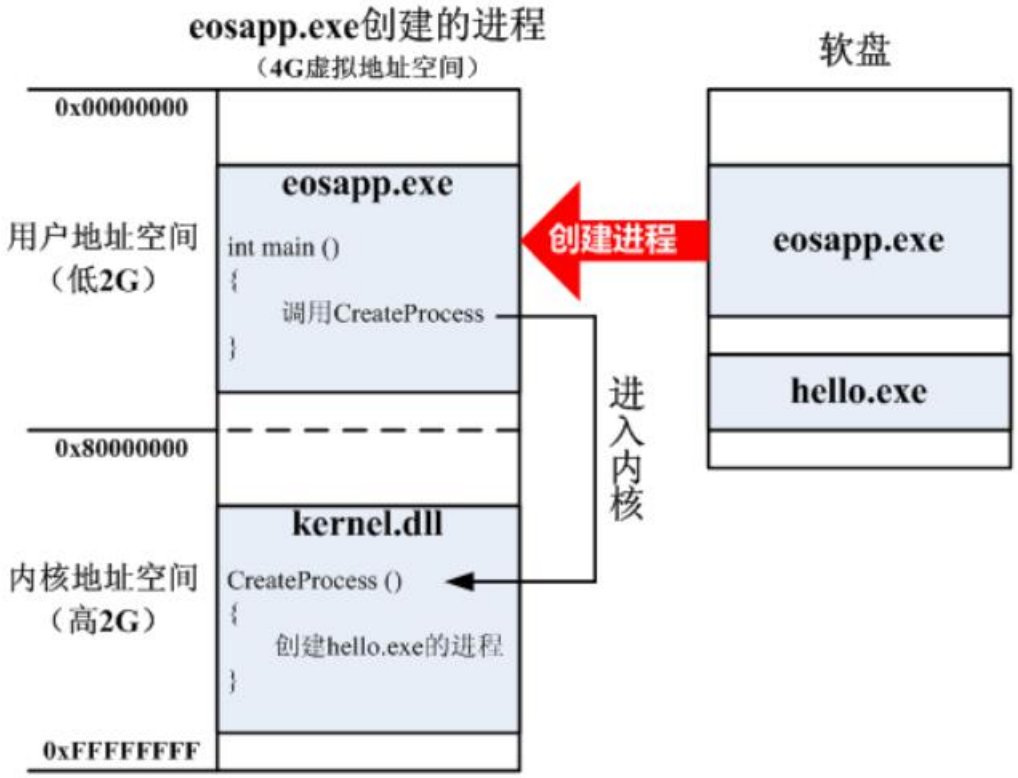


图 2-4：EOS 应用程序 eosapp.exe 创建的进程

1. 由于此时在内核的 CreateProcess 函数内中断执行，所以在“调试”菜单的“窗口”中选择“反汇编”，会在“反汇编”窗口中显示 CreateProcess 函数的指令对应的反汇编代码。“反汇编”窗口的左侧显示的是指令所在的虚拟地址。可以看到所有指令的虚拟地址都大于 0x80000000，说明内核（kernel.dll）处于高 2G 的虚拟地址空间中。

2. 在“调用堆栈”窗口中双击 main 函数项，设置 main 函数的调用堆栈帧为活动的。在“反汇编”窗口中查看 main 函数的指令所在的虚拟地址都是小于 0x80000000，说明应用程序（eosapp.exe）处于低 2G 的虚拟地址空间中。

3. 在“调用堆栈”窗口中双击 CreateProcess 函数项，重新设置 CreateProcess 函数的调用堆栈帧为活动的。关闭“反汇编”窗口。

接下来观察 eosapi.c 文件中 CreateProcess 函数的源代码，可以看到此函数只是调用了 EOS 内核函数 PsCreateProcess 并将创建进程所用到的参数传递给了此函数。所以，按 F11 可以调试进入 create.c 文件中的 PsCreateProcess 函数，在此函数中才开始执行创建进程的各项操作。

3.5 调试 PsCreateProcess 函数

创建进程最主要的操作就是创建进程控制块(PCB),并初始化其中的各种信

息（也就是为进程分配各种资源）。所以在 PsCreateProcess 函数中首先调用了 PspCreateProcessEnvironment 函数来创建进程控制块。

调试 PspCreateProcessEnvironment 函数的步骤如下：

1. 在 PsCreateProcess 函数中找到调用 PspCreateProcessEnvironment 函数的代码行（create.c 文件的第 163 行），并在此行添加一个断点。

2. 按 F5 继续调试，到此断点处中断。

3. 按 F11 调试进入 PspCreateProcessEnvironment 函数。

由于 PspCreateProcessEnvironment 函数的主要功能是创建进程控制块并初始化其中的部分信息，所以在此函数的开始，定义了一个进程控制块的指针变量 NewProcess。在此函数中查找到创建进程控制块的代码行（create.c 文件的第 418 行）

```
Status = ObCreateObject( PspProcessType,
NULL,
sizeof(PROCESS) + ImageNameSize + CmdLineSize,
0,
(PVOID*)&NewProcess );
```

这里的 ObCreateObject 函数会在由 EOS 内核管理的内存中创建了一个新的进程控制块（也就是分配了一块内存），并由 NewProcess 返回进程控制块的指针（也就是所分配内存的起始地址）。

按照下面的步骤调试进程控制块的创建过程：

1. 在调用 ObCreateObject 函数的代码行（create.c 文件的第 418 行）添加一个断点。

2. 按 F5 继续调试，到此断点处中断。

3. 按 F10 执行此函数后中断。

4. 此时为了查看进程控制块中的信息，将表达式*NewProcess 添加到“监视”窗口中。

5. 将鼠标移动到“监视”窗口中此表达式的“值”属性上，会弹出一个临时窗口，在临时窗口中会按照进程控制块的结构显示各个成员变量的值（可以参考 PROCESS 结构体的定义）。由于只是新建了进程控制块，还没有初始化其中成

员变量，所以值都为 0。

接下来调试初始化进程控制块中各个成员变量的过程：

1. 首先创建进程的地址空间,即 4G 虚拟地址空间。在代码行(create.c 文件的第 437 行)

NewProcess->Pas = MmCreateProcessAddressSpace();

添加一个断点。

2. 按 F5 继续调试，到此断点处中断。

3. 按 F10 执行此行代码后中断。

4. 在“监视”窗口中查看进程控制块的成员变量 Pas 的值已经不再是 0。说明已经初始化了进程的 4G 虚拟地址空间。

5. 使用 F10 一步步调试 PspCreateProcessEnvironment 函数中后面的代码，在调试的过程中根据执行的源代码，查看“监视”窗口中*NewProcess 表达式的值，观察进程控制块中哪些成员变量是被哪些代码初始化的，哪些成员变量还没有被初始化。

6. 当从 PspCreateProcessEnvironment 函数返回到 PsCreateProcess 函数后，停止按 F10。此时“监视”窗口中已经不能再显示表达式*NewProcess 的值了，在 PsCreateProcess 函数中是使用 ProcessObject 指针指向进程控制块的，所以将表达式*ProcessObject 添加到“监视”窗口中就可以继续观察新建进程控制块中的信息。

7. 接下来继续使用 F10 一步步调试 PsCreateProcess 函数中的代码，同样要注意观察执行后的代码修改了进程控制块中的哪些成员变量。当调试到 PsCreateProcess 函数的最后一行代码时，查看进程控制块中的信息，此时所有的成员变量都已经被初始化了（注意观察成员 ImageName 的值）。

8. 按 F5 继续执行，EOS 内核会为刚刚初始化完毕的进程控制块新建一个进程。激活虚拟机窗口查看

新建进程执行的结果。

9. 在 OS Lab 中选择“调试”菜单中的“停止调试”结束此次调试。

10. 选择“调试”菜单中的“删除所有断点”。

尝试根据之前对 PsCreateProcess 函数和 PspCreateProcessEnvironment

函数执行过程的跟踪调试，绘制一幅进程创建过程的流程图。

3.6 联系通过编程的方式创建应用程序的多个线程

使用 OS Lab 打开本实验文件夹中的参考源代码文件 NewTwoProc.c，仔细阅读此文件中的源代码。使用 NewTwoProc.c 文件中的源代码替换 EOS 应用程序项目中 EOSApp.c 文件内的源代码，生成后启动调试，查看多个进程并发执行的结果。

多个进程并发时，EOS 操作系统中运行的用户进程可以参见图 2-5。验证一个程序（hello.exe）可以同时创建多个进程。

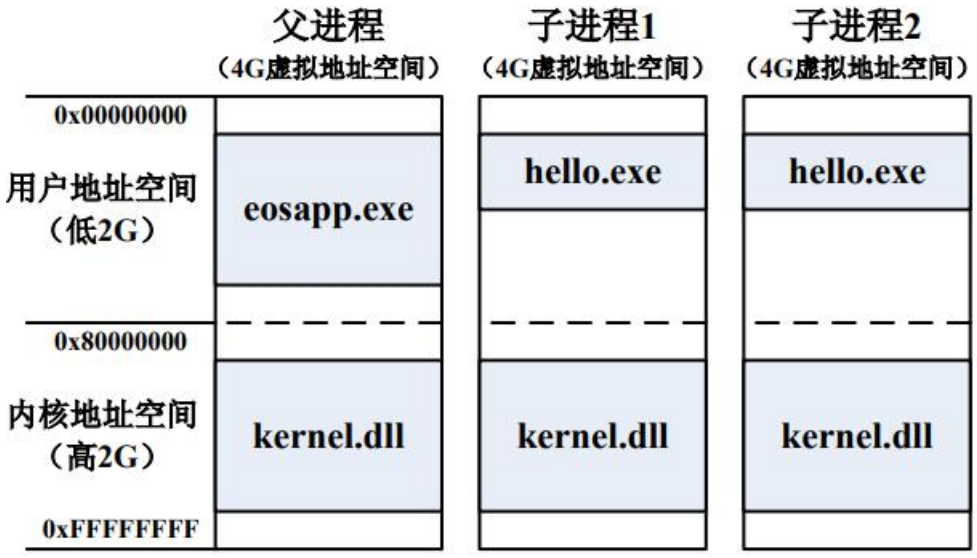


图 2-5：同时创建应用程序的多个进程

实验三　进程的同步

一、实验目的

- 使用 EOS 的信号量编程解决生产者—消费者问题，理解进程同步的意义。
- 调试跟踪 EOS 的信号量的工作过程，理解进程同步的原理。
- 修改 EOS 的信号量算法，使之支持等待超时唤醒功能（有限等待），加深理解进程同步的原理。

二、实验环境

1) Windows 7 及以上系统
2) 集成实验环境 OS Lab

三、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. 使用在第三步生成的 SDK 文件夹覆盖应用程序项目文件夹中的 SDK 文件夹。

3.2 使用 EOS 的信号量解决生产者－消费者问题

在本实验文件夹中，提供了使用 EOS 的信号量解决生产者－消费者问题的参考源代码文件 pc.c。使用 OS Lab 打开此文件（将文件拖动到 OS Lab 窗口中释放即可打开），仔细阅读此文件中的源代码和注释，各个函数的流程图可以参见图 3-1。思考在两个线程函数（Producer 和 Consumer）中，哪些是临界资源？哪些代码是临界区？哪些代码是进入临界区？哪些代码是退出临界区？进入临界区和退出临界区的代码是否成对出现？

按照下面的步骤查看生产者－消费者同步执行的过程：

1. 使用 pc.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 文件内的源代码。

2. 按 F7 生成修改后的 EOS 应用程序项目。

3. 按 F5 启动调试。OS Lab 会首先弹出一个调试异常对话框。

4. 在调试异常对话框中选择“否”，继续执行。

5. 立即激活虚拟机窗口查看生产者－消费者同步执行的过程，如图 3-2。

6. 待应用程序执行完毕后，结束此次调试。

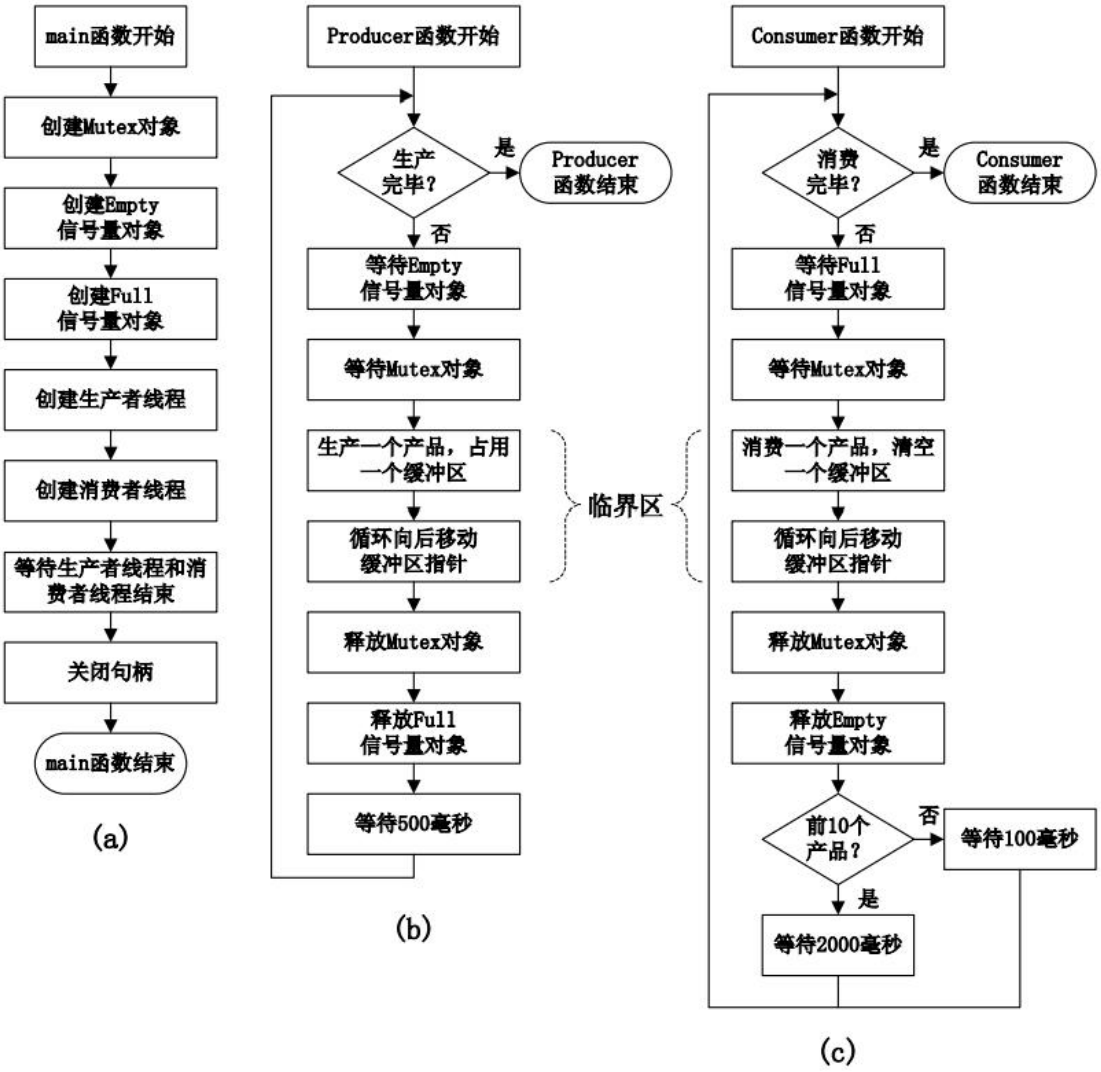


图 3-1：（a）main 函数流程图；（b）Producer 函数流程图；（c）Consumer 函数流程图

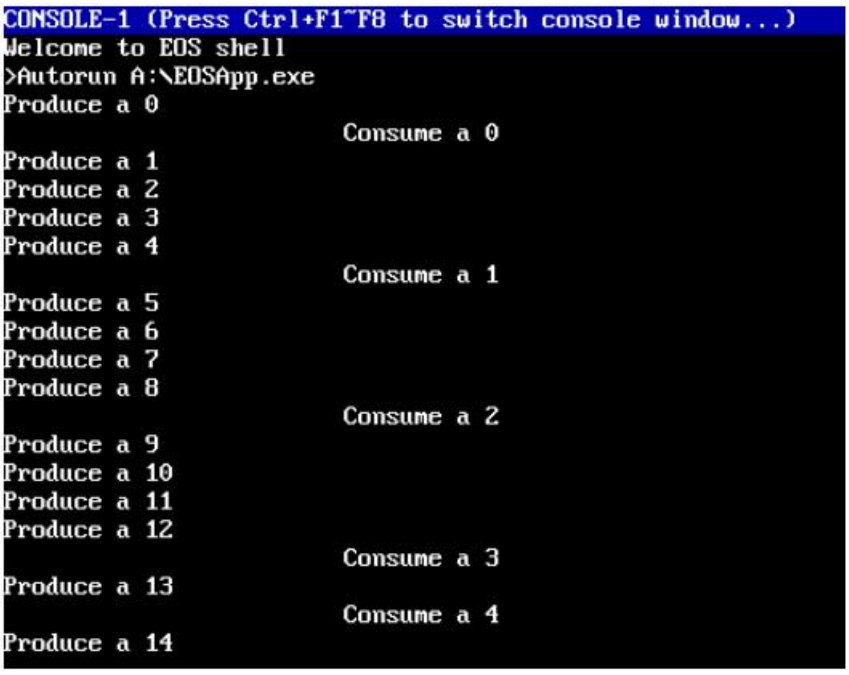


图 3-2：生产者-消费者同步执行的过程

仔细观察执行的结果，并结合源代码思考下面的问题：

- 生产者线程和消费者线程是如何使用 Mutex、Empty 信号量和 Full 信号量来实现同步的？在两个线程函数中对这三个同步对象的操作能够改变顺序吗？

- 生产者在生产了 13 号产品后本来要继续生产 14 号产品，可此时生产者为什么必须等待消费者消费了 4 号产品后，才能生产 14 号产品呢？生产者和消费者是怎样使用同步对象来实现该同步过程的呢？(提示：在后面的 3.3.2 节中会详细的调试该同步过程)

3.3 调试 EOS 的信号量的工作过程

3.3.1 创建信号量

信号量结构体（SEMAPHORE）中的各个成员变量是由 API 函数 CreateSemaphore 的对应参数初始化的，查看 main 函数中创建 Empty 和 Full 信号量使用的参数有哪些不同，又有哪些相同，思考其中的原因。

按照下面的步骤调试信号量创建的过程：

1. 按 F5 启动调试 EOS 应用项目。OS Lab 会首先弹出一个调试异常对话框。
2. 在调试异常对话框中选择“是”，调试会中断。
3. 在 main 函数中创建 Empty 信号量的代码行（第 77 行）

EmptySemaphoreHandle = CreateSemaphore(BUFFER_SIZE, BUFFER_SIZE, NULL);

添加一个断点。

4. 按 F5 继续调试，到此断点处中断。

5. 按 F11 调试进入 CreateSemaphore 函数。可以看到此 API 函数只是调用了 EOS 内核中的 PsCreateSemaphoreObject 函数来创建信号量对象。

6. 按 F11 调试进入 semaphore.c 文件中的 PsCreateSemaphoreObject 函数。在此函数中，会在 EOS 内核管理的内存中创建一个信号量对象（分配一块内存），而初始化信号量对象中各个成员的操作是在 PsInitializeSemaphore 函数中完成的。

7. 在 semaphore.c 文件的顶部查找到 PsInitializeSemaphore 函数的定义（第 19 行），在此函数的第一行（第 39 行）代码处添加一个断点。

8. 按 F5 继续调试，到断点处中断。观察 PsInitializeSemaphore 函数中用来初始化信号量结构体成员的值，应该和传入 CreateSemaphore 函数的参数值是一致的。

9. 按 F10 单步调试 PsInitializeSemaphore 函数执行的过程，查看信号量结构体被初始化的过程。

打开“调用堆栈”窗口，查看函数的调用层次。

3.3.2 等待、释放信号量

3.3.2.1 等待信号量（不阻塞）

生产者和消费者刚开始执行时，用来放产品的缓冲区都是空的，所以生产者在第一次调用 WaitForSingleObject 函数等待 Empty 信号量时，应该不需要阻塞就可以立即返回。按照下面的步骤调试：

1. 删除所有的断点（防止有些断点影响后面的调试）。

2. 在 eosapp.c 文件的 Producer 函数中，等待 Empty 信号量的代码行（第 144 行）

WaitForSingleObject(EmptySemaphoreHandle, INFINITE);

添加一个断点。

3. 按 F5 继续调试，到断点处中断。

4. WaitForSingleObject 函数最终会调用内核中的 PsWaitForSemaphore 函数完成等待操作。所以，在 semaphore.c 文件中 PsWaitForSemaphore 函数的第一行（第 68 行）添加一个断点。

5. 按 F5 继续调试，到断点处中断。

6. 按 F10 单步调试，直到完成 PsWaitForSemaphore 函数中的所有操作。可以看到此次执行并没有进行等待，只是将 Empty 信号量的计数减少了 1（由 10 变为了 9）就返回了。

3.3.2.2 释放信号量（不唤醒）

1. 删除所有的断点（防止有些断点影响后面的调试）。

2. 在 eosapp.c 文件的 Producer 函数中，释放 Full 信号量的代码行（第 152 行）

ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);

添加一个断点。

3. 按 F5 继续调试，到断点处中断。

4. 按 F11 调试进入 ReleaseSemaphore 函数。

5. 继续按 F11 调试进入 PsReleaseSemaphoreObject 函数。

6. 先使用 F10 单步调试，当黄色箭头指向第 269 行时使用 F11 单步调试，进入 PsReleaseSemaphore 函数。

7. 按 F10 单步调试，直到完成 PsReleaseSemaphore 函数中的所有操作。可以看到此次执行没有唤醒其它线程（因为此时没有线程在 Full 信号量上被阻塞），只是将 Full 信号量的计数增加了 1（由 0 变为了 1）。

生产者线程通过等待 Empty 信号量使空缓冲区数量减少了 1，通过释放 Full 信号量使满缓冲区数量增加了 1，这样就表示生产者线程生产了一个产品并占用了一个缓冲区。

3.3.2.3 等待信号量（阻塞）

由于开始时生产者线程生产产品的速度较快，而消费者线程消费产品的速度较慢，所以当缓冲池中所有的缓冲区都被产品占用时，生产者在生产新的产品时就会被阻塞，下面调试这种情况。

1. 结束之前的调试。

2. 删除所有的断点。

3. 按 F5 重新启动调试。OS Lab 会首先弹出一个调试异常对话框。

4. 在调试异常对话框中选择“是”，调试会中断。

5. 在 semaphore.c 文件中的 PsWaitForSemaphore 函数的

PspWait(&Semaphore->WaitListHead, INFINITE);

代码行（第 78 行）添加一个断点。

6. 按 F5 继续调试，并立即激活虚拟机窗口查看输出。开始时生产者、消费者都不会被信号量阻塞，同步执行一段时间后才在断点处中断。

7. 中断后，查看“调用堆栈”窗口，有 Producer 函数对应的堆栈帧，说明此次调用是从生产者线程函数进入的。

8. 在“调用堆栈”窗口中双击 Producer 函数所在的堆栈帧，绿色箭头指向等待 Empty 信号量的代码行，查看 Producer 函数中变量 i 的值为 14，表示生产者线程正在尝试生产 14 号产品。

9. 在“调用堆栈”窗口中双击 PsWaitForSemaphore 函数的堆栈帧，查看 Empty 信号量计数（Semaphore->Count）的值为-1，所以会调用 PspWait 函数将生产者线程放入 Empty 信号量的等待队列中进行等待（让出 CPU）。

10. 激活虚拟机窗口查看输出的结果。生产了从 0 到 13 的 14 个产品，但是只消费了从 0 到 3 的 4 个产品，所以缓冲池中的 10 个缓冲区就都被占用了，这与之前调试的结果是一致的。

3.3.2.4 释放信号量（唤醒）

只有当消费者线程从缓冲池中消费了一个产品，从而产生一个空缓冲区后，生产者线程才会被唤醒并继续生产 14 号产品。可以按照下面的步骤调试：

1. 删除所有断点。

2. 在 eosapp.c 文件的 Consumer 函数中，释放 Empty 信号量的代码行（第 180 行）

ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);

添加一个断点。

3. 按 F5 继续调试，到断点处中断。

4. 查看 Consumer 函数中变量 i 的值为 4，说明已经消费了 4 号产品。

5. 按照 3.3.2.2 中的方法使用 F10 和 F11 调试进入 PsReleaseSemaphore 函数。

6. 查看 PsReleaseSemaphore 函数中 Empty 信号量计数（Semaphore->Count）的值为-1，和生产者线程被阻塞时的值是一致的。

7. 按 F10 单步调试 PsReleaseSemaphore 函数，直到在代码行（第 132 行）

PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);

处中断。此时 Empty 信号量计数的值已经由-1 增加为了 0，需要调用 PspWakeThread 函数唤醒阻塞在 Empty 信号量等待队列中的生产者线程（放入就绪队列中），然后调用 PspSchedule 函数执行调度，这样生产者线程就得以继续执行。

按照下面的步骤验证生产者线程被唤醒后，是从之前被阻塞时的状态继续执行的：

1. 在 semaphore.c 文件中 PsWaitForSemaphore 函数的最后一行（第 83 行）代码处添加一个断点。

2. 按 F5 继续调试，在断点处中断。

3. 查看 PsWaitForSemaphore 函数中 Empty 信号量计数（Semaphore->Count）的值为 0，和生产者线程被唤醒时的值是一致的。

4. 在“调用堆栈”窗口中可以看到是由 Producer 函数进入的。激活 Producer 函数的堆栈帧，查看 Producer 函数中变量 i 的值为 14，表明之前被阻塞的、正在尝试生产 14 号产品的生产者线程已经从 PspWait 函数返回并继续执行了。

5. 结束此次调试。

3.4 修改 EOS 的信号量算法

3.4.1 要求

在目前 EOS Kernel 项目的 ps/semaphore.c 文件中，PsWaitForSemaphore 函数的 Milliseconds 参数只能是 INFINITE，PsReleaseSemaphore 函数的 ReleaseCount 参数只能是 1。现在要求同时修改 PsWaitForSemaphore 函数和 PsReleaseSemaphore 函数中的代码，使这两个参数能够真正起到作用，使信号量对象支持等待超时唤醒功能和批量释放功能。

3.4.2 提示

修改 PsWaitForSemaphore 函数时要注意：

● 对于支持等待超时唤醒功能的信号量，其计数值只能是大于等于 0。当计数值大于 0 时，表示信号量为 signaled 状态；当计数值等于 0 时，表示信号量为 nonsignaled 状态。所以，PsWaitForSemaphore 函数中原有的代码段

```
Semaphore->Count--;
if (Semaphore->Count < 0) {
PspWait(&Semaphore->WaitListHead, INFINITE);
}
```

应被修改为：先用计数值和 0 比较，当计数值大于 0 时，将计数值减 1 后直接返回成功；当值等于 0 时，调用 PspWait 函数阻塞线程的执行（将参数 Milliseconds 做为 PspWait 函数的第二个参数，并使用 PspWait 函数的返回值做为返回值）。

● 在函数开始定义一个 STATUS 类型的变量，用来保存不同情况下的返回值，并在函数最后返回此变量的值。绝不能在原子操作的中途返回！

● 在 EOS Kernel 项目 ps/sched.c 文件的第 190 行查看 PspWait 函数的说明和源代码。

修改 PsReleaseSemaphore 函数时要注意：

● 编写一个使用 ReleaseCount 做为计数器的循环体，来替换 PsReleaseSemaphore 函数中原有的代码段

```
Semaphore->Count++;
if (Semaphore->Count <= 0) {
PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);
}
```

● 在循环体中完成下面的工作：

1. 如果被阻塞的线程数量大于等于 ReleaseCount，则循环结束后，有 ReleaseCount 个线程会被唤醒，而且信号量计数的值仍然为 0；

2. 如果被阻塞的线程数量（可以为 0）小于 ReleaseCount，则循环结束后，所有被阻塞的线程都会被唤醒，并且信号量的计数值＝ReleaseCount－之前被阻

塞线程的数量＋之前信号量的计数值。

- 在 EOS Kernel 项目 ps/sched.c 文件的第 294 行查看 PspWakeThread 函数的说明和源代码
- 在循环的过程中可以使用宏定义函数 ListIsEmpty 判断信号量的等待队列是否为空，例如

ListIsEmpty(&Semaphore->WaitListHead)

可以在 EOS Kernel 项目 inc/rtl.h 文件的第 113 行查看此宏定义的源代码。

3.4.3 测试方法

修改完毕后，可以按照下面的方法进行测试：

1. 使用修改完毕的 EOS Kernel 项目生成完全版本的 SDK 文件夹，并覆盖之前的生产者－消费者应用程序项目的 SDK 文件夹。

2. 按 F5 调试执行原有的生产者－消费者应用程序项目，结果必须仍然与图 13-2 一致。如果有错误，可以调试内核代码来查找错误，然后在内核项目中修改，并重复步骤 1。

3. 将 Producer 函数中等待 Empty 信号量的代码行

```
WaitForSingleObject(EmptySemaphoreHandle, INFINITE);
```

替换为

```
while(WAIT_TIMEOUT == WaitForSingleObject(EmptySemaphoreHandle, 300)){
printf("Producer wait for empty semaphore timeout\n");
}
```

4. 将 Consumer 函数中等待 Full 信号量的代码行

```
WaitForSingleObject(FullSemaphoreHandle, INFINITE);
```

替换为

```
while(WAIT_TIMEOUT == WaitForSingleObject(FullSemaphoreHandle, 300)){
printf("Consumer wait for full semaphore timeout\n");
}
```

5. 启动调试新的生产者－消费者项目，查看在虚拟机中输出的结果，验证信号量超时等待功能是否能够正常执行。如果有错误，可以调试内核代码来查找错误，然后在内核项目中修改，并重复步骤 1。

6. 如果超时等待功能已经能够正常执行，可以考虑将消费者线程修改为一次消费两个产品，来测试 ReleaseCount 参数是否能够正常使用。使用实验文件夹中 NewConsumer.c 文件中的 Consumer 函数替换原有的 Consumer 函数。

实验四　时间片轮转调度

一、实验目的

- 调试 EOS 的线程调度程序，熟悉基于优先级的抢先式调度。
- 为 EOS 调度器添加时间片轮转调度算法，了解常用调度算法。

二、实验环境

1) Windows 7 及以上系统
2) 集成实验环境 OS Lab

三、实验内容

3.1 准备实验

按照下面的步骤准备实验：

1. 启动 OS Lab。
2. 新建一个 EOS Kernel 项目。

3.2 阅读控制台命令“rr”相关的源代码

阅读 ke/sysproc.c 文件中第 690 行的 ConsoleCmdRoundRobin 函数，及该函数用到的第 649 行的 ThreadFunction 函数和第 642 行的 THREAD_PARAMETER 结构体，学习“rr”命令是如何测试时间片轮转调度的。在阅读的过程中需要特别注意下面几点：

- 在 ConsoleCmdRoundRobin 函数中使用 ThreadFunction 函数做为线程函数，新建了 20 个优先级为 8 的线程，做为测试时间片轮转调度用的线程。
- 在新建的线程中，只有正在执行的线程才会在控制台的对应行（第 0 个线程对应第 0 行，第 1 个线程对应第 1 行）增加其计数，这样就可以很方便的观察到各个线程执行的情况。
- 控制台对于新建的线程来说是一种临界资源，所以，新建的线程在向控制台输出时，必须使用“关中断”和“开中断”进行互斥（参见 ThreadFunction 函数的源代码）。
- 由于控制台线程的优先级是 24，高于新建线程的优先级 8，所以只有在

控制台线程进入“阻塞”状态后，新建的线程才能执行。

- 新建的线程会一直运行，原因是在 ThreadFunction 函数中使用了死循环，所以只能在 ConsoleCmdRoundRobin 函数的最后调用 TerminateThread 函数来强制结束这些新建的线程。

按照下面的步骤执行控制台命令“rr”，查看其在没有时间片轮转调度时的执行效果：

1. 按 F7 生成在本实验 3.1 中创建的 EOS Kernel 项目。
2. 按 F5 启动调试。
3. 待 EOS 启动完毕，在 EOS 控制台中输入命令“rr”后按回车。

命令开始执行后，观察其执行效果（如图 4-1），会发现并没有体现“rr”命令相关源代码的设计意图。通过之前对这些源代码的学习，20 个新建的线程应该在控制台对应的行中轮流地显示它们的计数在增加，而现在只有第 0 个新建的线程在第 0 行显示其计数在增加，说明只有第 0 个新建的线程在运行，其它线程都没有运行。造成上述现象的原因是：所有 20 个新建线程的优先级都是 8，而此时 EOS 只实现了基于优先级的抢先式调度，还没有实现时间片轮转调度，所以至始至终都只有第 0 个线程在运行，而其它具有相同优先级的线程都没有机会运行，只能处于“就绪”状态。

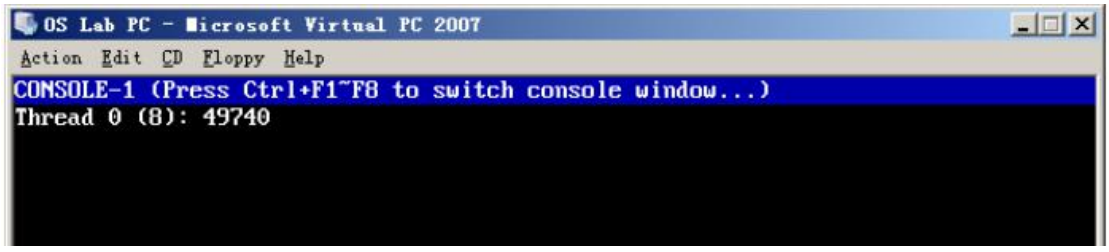


图 4-1：没有时间片轮转调度时“rr”命令的执行效果

3.3 调试线程调度程序

在为 EOS 添加时间片轮转调度之前，先调试一下 EOS 的线程调度程序 PspSelectNextThread 函数，学习就绪队列、就绪位图以及线程的优先级是如何在线程调度程序中协同工作的，从而加深对 EOS 已经实现的基于优先级的抢先式调度的理解。

3.3.1 调试当前线程不被抢先的情况

像图 4-1 中显示的，新建的第 0 个线程会一直运行，而不会被其它同优先

级的新建线程或者低优先级的线程抢先。按照下面的步骤调试这种情况在 PspSelectNextThread 函数中处理的过程。

1. 结束之前的调试。

2. 在 ke/sysproc.c 文件的 ThreadFunction 函数中，调用 fprintf 函数的代码行（第 680 行）添加一个断点。

3. 按 F5 启动调试。

4. 待 EOS 启动完毕，在 EOS 控制台中输入命令“rr”后按回车。“rr”命令开始执行后，会在断点处中断。

5. 查看 ThreadFunction 函数中变量 pThreadParameter->Y 的值应该为 0，说明正在调试的是第 0 个新建的线程。

6. 激活虚拟机窗口，可以看到第 0 个新建的线程还没有在控制台中输出任何内容，原因是 fprintf 函数还没有执行。

7. 激活 OS Lab 窗口后按 F5 使第 0 个新建的线程继续执行，又会在断点处中断。再次激活虚拟机窗口，可以看到第 0 个新建的线程已经在控制台中输出了第一轮循环的内容。可以多按几次 F5 查看每轮循环输出的内容。

通过之前的调试，可以观察到第 0 个新建的线程执行的情况。按照下面的步骤调试，查看当有中断发生从而触发线程调度时，第 0 个新建的线程不会被抢先的情况。

1. 在 ps/sched.c 文件的 PspSelectNextThread 函数中，调用 BitScanReverse 函数扫描就绪位图的代码行（第 384 行）添加一个断点。

2. 按 F5 继续执行，当有定时计数器中断发生时（每 10ms 一次），就会在新添加的断点处中断。

3. 在“调试”菜单的“窗口”中选择“监视”，激活“监视”窗口（此时按 F1 可以获得关于“监视”窗口的帮助）。

4. 在“监视”窗口中添加表达式“/t PspReadyBitmap”，以二进制格式查看就绪位图的值。此时就绪位图的值应该为 100000001，表示优先级为 8 和 0 的两个就绪队列中存在就绪线程。（注意，如果就绪位图的值不是 100000001，就继续按 F5，直到就绪位图变为此值）。

5. 在“调试”菜单中选择“快速监视”，在“快速监视”对话框的“表达式”

中输入表达式“*PspCurrentThread”后，点击“重新计算”按钮，可以查看当前正在执行的线程（即被中断的线程）的线程控制块中各个域的值。其中优先级（Priority 域）的值为 8；状态（State 域）的值为 2（运行态）；时间片（RemainderTicks 域）的值为 6；线程函数（StartAddr 域）为 ThreadFunction。综合这些信息即可确定当前正在执行的线程就是新建的第 0 个线程。关闭“快速监视”对话框。

6. 在“监视”窗口中添加表达式“ListGetCount(&PspReadyListHeads[8])”，查看优先级为 8 的就绪队列中就绪线程的数量，值为 19。说明除了正在执行的第 0 个新建的线程外，其余 19 个新建的线程都在优先级为 8 的就绪队列中。ListGetCount 函数在文件 rtl/list.c 中定义。

7. 按 F10 单步调试，BitScanReverse 函数会从就绪位图中扫描最高优先级，并保存在变量 HighestPriority 中。查看变量 HighestPriority 的值为 8。

8. 继续按 F10 单步调试，直到在 PspSelectNextThread 函数返前停止（第 465 行），注意观察线程调度执行的每一个步骤。

第 0 个新建的线程在执行线程调度时没有被抢先的原因可以归纳为两点：

(1) 第 0 个新建的线程仍然处于“运行”状态。

(2) 没有比其优先级更高的就绪线程。

3.3.2 调试当前线程被抢先的情况

如果有比第 0 个新建的线程优先级更高的线程进入就绪状态，则第 0 个新建的线程就会被抢先。按照下面的步骤调试这种情况在 PspSelectNextThread 函数中处理的过程（注意，接下来的调试要从本实验 3.3.1 调试的状态继续调试，所以不要结束之前的调试）。

1. 选择“调试”菜单中的“删除所有断点”，删除之前添加的所有断点。

2. 在 ps/sched.c 文件的 PspSelectNextThread 函数的第 395 行添加一个断点。

3. 按 F5 继续执行，激活虚拟机窗口，可以看到第 0 个新建的线程正在执行。

4. 在虚拟机窗口中按下一次空格键，EOS 会在之前添加的断点处中断。

5. 在“监视”窗口中查看就绪位图的值为 10000000000000000100000001，

说明此时在优先级为 24 的就绪队列中存在就绪线程。在“监视”窗口中添加表达式“ListGetCount(&PspReadyListHeads[24])”，其值为 1，说明优先级为 24 的就绪队列中只有一个就绪线程。扫描就绪位图后获得的最高优先级的值 HighestPriority 也就应该是 24。

6. 按 F10 单步调试一次，执行的语句会将当前正在执行的第 0 个新建的线程，放入优先级为 8 的就绪队列的队首。“监视”窗口中显示的优先级为 8 的就绪队列中的线程数量就会增加 1，变为 20。

7. 继续按 F10 单步调试，直到在第 444 行中断执行，注意观察线程调度执行的每一个步骤。此时，正在执行的第 0 个新建的线程已经进入了“就绪”状态，让出了 CPU。线程调度程序接下来的工作就是选择优先级最高的非空就绪队列的队首线程作为当前运行线程，也就是让优先级为 24 的线程在 CPU 上执行。

8. 按 F10 单步调试一次，当前线程 PspCurrentThread 指向了优先级为 24 的线程。可以在“快速监视”窗口中查看表达式“*PspCurrentThread”的值，注意线程控制块中 StartAddr 域的值为 IopConsoleDispatchThread 函数（在文件 io/console.c 中定义），说明这个优先级为 24 的线程是控制台派遣线程。

9. 继续按 F10 单步调试，直到在 PspSelectNextThread 函数返回前（第 465 行）中断执行，注意观察线程调度执行的每一个步骤。此时，优先级为 24 的线程已经进入了“运行”状态，在中断返回后，就可以开始执行了。在“监视”窗口中，就绪位图的值变为 100000001，优先级为 24 的就绪队列中线程的数量变为 0，就绪位图和就绪队列都是在刚刚被调用过的 PspUnreadyThread 函数（在文件 ps/sched.c 中定义）内更新的。

10. 删除所有断点后结束调试。

3.4 为 EOS 添加时间片轮转调度算法

3.4.1 要求

修改 ps/sched.c 文件中的 PspRoundRobin 函数(第 337 行)，在其中实现时间片轮转调度算法。

3.4.2 测试方法

1. 代码修改完毕后，按 F7 生成 EOS 内核项目。

2. 按 F5 启动调试。

3. 在 EOS 控制台中输入命令“rr”后按回车。应能看到 20 个线程轮流执行的效果，如图 4-2。

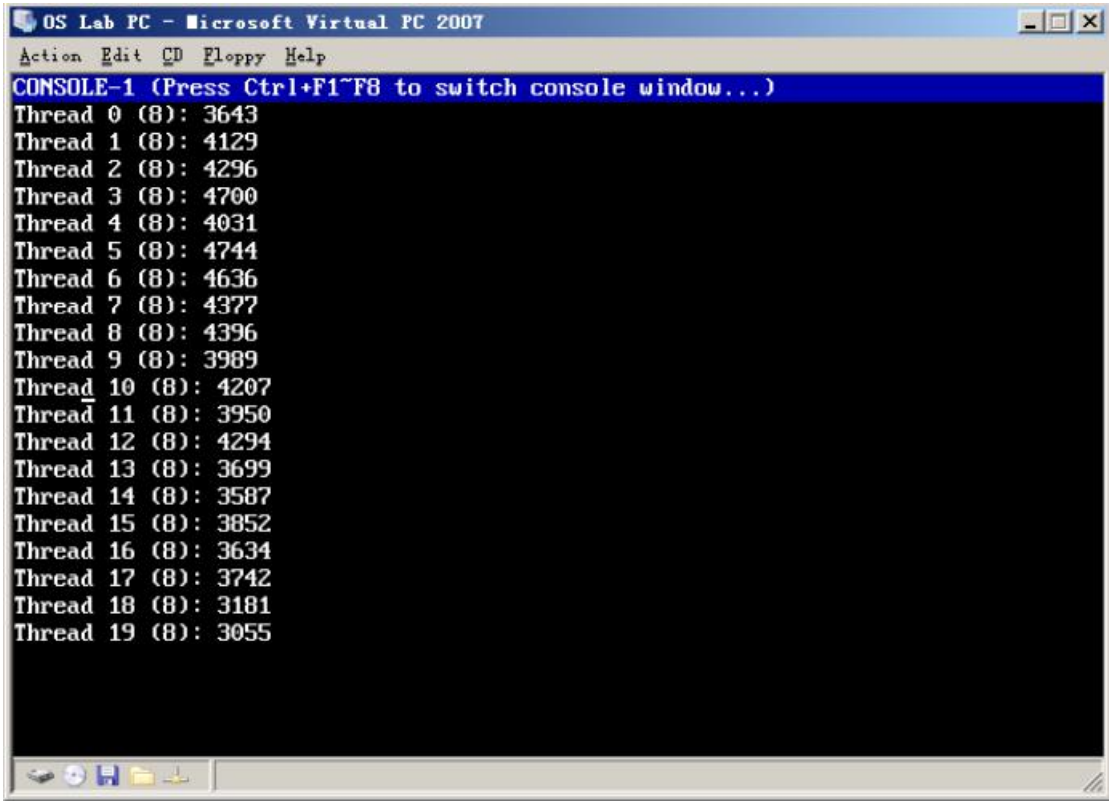


图 4-2：进行时间片轮转调度时“rr”命令的执行效果

3.4.3 提示

- 在 PspRoundRobin 函数中，全局变量 PspCurrentThread 指向的线程控制块就是被定时计数器中断的线程的线程控制块，通过对 PspCurrentThread 指向的线程控制块的各个域进行修改，就可以改变被中断线程的各种属性。全局变量 PspCurrentThread 的定义参见 ps/sched.c 的第 45 行。线程控制块结构体的定义参见 ps/psp.h 的第 58 行。

- 被中断线程的状态有可能不是“运行”状态，而是“阻塞”状态。所以，在进行时间片轮转调度前，要先判断一下被中断线程是否仍处于“运行”状态。只有当被中断线程仍处于“运行”状态时，才需要进行时间片轮转调度。在 PspRoundRobin 函数中的第一行代码可以如下（线程状态的定义可以参见 ps/psp.h 的第 93 行）：

```
if (NULL != PspCurrentThread && Running == PspCurrentThread->State) {
```

// 在此实现时间片轮转调度算法

}

- 被中断线程所拥有的时间片保存在 PspCurrentThread 的 RemainderTicks 域中。在减少时间片，或者判断时间片是否变为 0，以及重新分配时间片时，都可以直接对该域进行操作。

- 重新为被中断线程分配时间片时，应该使用宏定义 TICKS_OF_TIME_SLICE（在文件 ps/psp.h 的第 104 行定义）为 PspCurrentThread 变量的 RemainderTicks 域赋值。注意，此宏定义表示给线程分配的时钟滴答（Tick）数量，多个时钟滴答组成了线程的时间片。时钟滴答的大小是由定时计数器中断的频率确定的，目前定时计数器每秒钟发生 100 次中断，则每个时钟滴答的大小是 10ms，而且 TICKS_OF_TIME_SLICE 定义的值为 6，所以时间片的大小就是 60ms。

- 在判断就绪队列中是否存这样的就绪线程，即其优先级与被中断线程优先级相同时，只需要扫描就绪位图即可（这样速度更快）。例如被中断线程优先级为 8，则只需要判断就绪位图的第 8 位是否为 1，为 1 则说明就绪队列中存在优先级为 8 的就绪线程，为 0 则说明不存在。此时，可以使用下面的代码作为条件语句中的布尔表达式：

BIT_TEST(PspReadyBitmap, PspCurrentThread->Priority)

BIT_TEST 是一个宏定义函数，其定义参见 inc/eosdef.h 的第 220 行。如果存在和被中断线程优先级相同的就绪线程，此函数返回非 0（TURE），否则返回 0（FALSE）。变量 PspReadyBitmap 是 32 位就绪位图，其定义参见 ps/sched.c 的第 29 行。

- 可以使用下面的代码将被中断线程转入就绪状态，并将其插入对应优先级就绪队列的末尾：

PspReadyThread(PspCurrentThread);

函数 PspReadyThread 的定义参见 ps/sched.c 的第 107 行。

3.5 修改线程的时间片大小

在成功为 EOS 添加了时间片轮转调度后，可以按照下面的步骤修改时间片的大小：

1. 在 OS Lab 的“项目管理器”窗口中找到 ps/psp.h 文件，双击打开此文件。

2. 将 ps/psp.h 第 104 行定义的 TICKS_OF_TIME_SLICE 的值修改为 1。

4. 按 F7 生成 EOS 内核项目。

5. 按 F5 启动调试。

3. 在 EOS 控制台中输入命令“rr”后按回车。观察执行的效果。

还可以按照上面的步骤为 TICKS_OF_TIME_SLICE 取一些其它的极端值，例如 20 或 100 等，分别观察“rr”命令执行的效果。通过分析造成执行效果不同的原因，理解时间片的大小对时间片轮转调度造成的影响。

实验五 物理存储器与进程逻辑地址空间的管理

一、 实验目的

- 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。
- 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、实验环境

1) Windows 7 及以上系统
2) 集成实验环境 OS Lab

三、实验内容

3.1 准备实验

按照下面的步骤准备实验：

1. 启动 OS Lab。
2. 新建一个 EOS Kernel 项目。

3.2 阅读控制台命令“pm”相关的源代码，并查看其执行的结果

阅读 ke/sysproc.c 文件中第 1059 行的 ConsoleCmdPhysicalMemory 函数，学习“pm”命令是如何统计并输出物理存储器信息的。在阅读的过程中需要注意下面几点：

- 在统计输出物理存储器信息之前要关闭中断，之后要打开中断，这样可以防止在命令执行的过程中有其它线程分配或者释放物理页。
- 全局变量 MiTotalPageFrameCount 保存了物理页的总数。每个物理页的大小是 4KB，由宏 PAGE_SIZE 定义。
- 全局变量 MiZeroedPageCount 和 MiFreePageCount 分别保存了零页和空闲页的数量。
- 计算已用物理页数量的方法是：物理页总数减去零页数量，再减去空闲页数量。

按照下面的步骤执行控制台命令“pm”，查看物理存储器的信息：

1. 按 F7 生成在本实验 3.1 中创建的 EOS Kernel 项目。

2. 按 F5 启动调试。

3. 待 EOS 启动完毕，在 EOS 控制台中输入命令“pm”后按回车。

观察命令执行的结果，如图 5-1 所示，可以了解当前物理存储器的使用情况。

图5-1：“pm”命令的执行结果

3.3 分配物理页和释放物理页

接下来，在 pm 命令函数中添加分配物理页和释放物理页的代码，单步调试管理物理页的方法。按照下面的步骤修改 pm 命令的源代码：

1. 使用 OS Lab 打开本实验文件夹中的 pm.c 文件（将文件拖动到 OS Lab 窗口中释放即可打开）。此文件中有一个修改后的 ConsoleCmdPhysicalMemory 函数，主要是在原有代码的后面增加了分配物理页和释放物理页的代码。

2. 使用 pm.c 文件中 ConsoleCmdPhysicalMemory 函数的函数体替换 ke/sysproc.c 文件中 ConsoleCmdPhysicalMemory 函数的函数体。

3. 按 F7 生成修改后的 EOS Kernel 项目。

4. 按 F5 启动调试。

5. 待 EOS 启动完毕，在 EOS 控制台中输入命令“pm”后按回车。

观察命令执行的结果，如图 5-2 所示，尝试说明分配物理页或者释放物理页后物理存储器的变化情况。

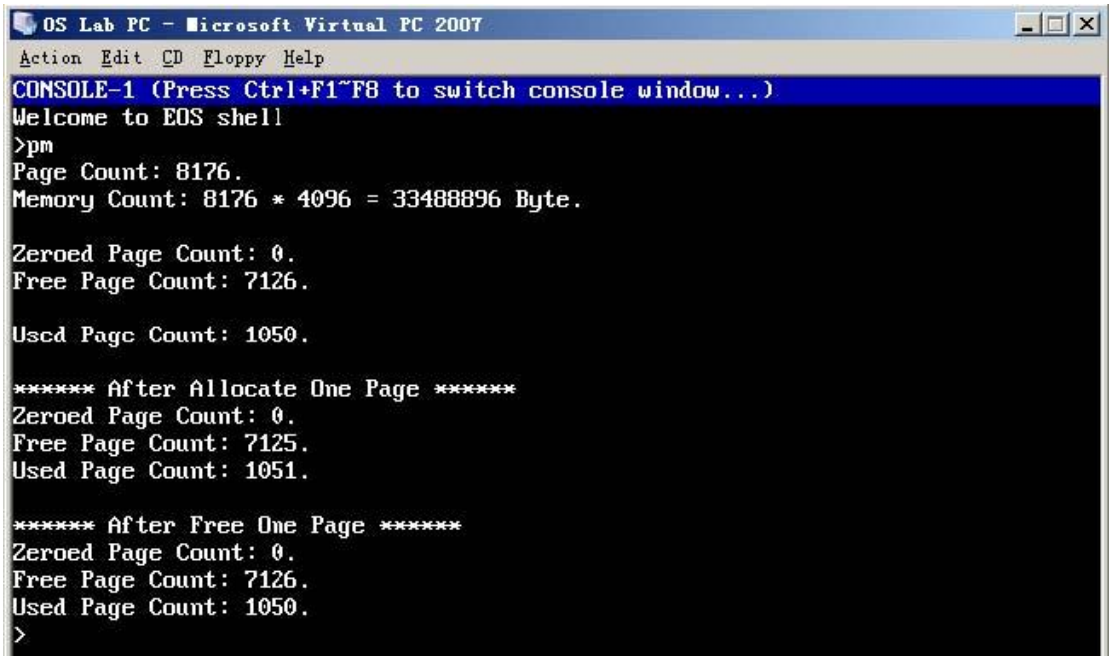


图5-2：分配物理页或者释放物理页后物理存储器的变化情况

按照下面的步骤调试分配物理页和释放物理页的过程：

1. 结束之前的调试。

2. 在 ke/sysproc.c 文件的 ConsoleCmdPhysicalMemory 函数中，在调用 MiAllocateAnyPages 函数的代码行（第 1103 行）添加一个断点，在调用 MiFreePages 函数的代码行（第 1115 行）添加一个断点。

3. 按 F5 启动调试。

4. 待 EOS 启动完毕，在 EOS 控制台中输入命令“pm”后按回车。

5. pm 命令开始执行后，会在调用 MiAllocateAnyPages 函数的代码行处中断，按 F11 调试进入 MiAllocateAnyPages 函数。

6. 按 F10 单步调试 MiAllocateAnyPages 函数的执行过程，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？

(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？

(3) 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？

(4) 绘制 MiAllocateAnyPages 函数的流程图。

继续调试释放物理页的过程：

1. 按 F5 继续执行，会在调用 MiFreePages 函数的代码行处中断，按 F11 调

试进入 MiFreePages 函数。

2. 按 F10 单步调试 MiFreePages 函数的执行过程，尝试回答下面的问题：

(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？

(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？

(3) 绘制 MiFreePages 函数的流程图。

结束此次调试。继续修改 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，并练习调试这两个函数在分配多个物理页和释放多个物理页时执行的过程。

3.4 阅读控制台命令“vm”相关的源代码，并查看其执行的结果

阅读ke/sysproc.c文件中第959行的ConsoleCmdVM函数，学习“vm”命令是如何统计并输出进程的虚拟地址描述符信息的。在阅读的过程中需要注意下面几点：

- 与“pm”命令输出的是整个系统的物理存储器的使用情况不同，“vm”命令输出的是某个进程的虚拟地址描述符信息，所以“vm”命令使用了一个参数——进程ID，用来指定一个进程。这个进程既可以是系统进程，也可以是用户进程。
- 在统计输出指定进程的虚拟地址描述符信息之前要关闭中断，之后要打开中断，这样可以防止在命令执行的过程中有其它线程分配或者释放虚拟页。
- EOS操作系统的进程有4G的虚拟地址空间，但并不是所有的虚拟地址空间都使用虚拟地址描述符来管理，有一些地址空间是静态的，还有一些地址空间由其他的动态方式来管理（例如系统内存池）。
- 进程4G虚拟地址空间中由虚拟地址描述符所管理空间的低地址和高地址是固定的，在这段地址空间中，如果有虚拟页被占用，就会使用虚拟地址描述符来标识，并放入链表中管理。

按照下面的步骤执行控制台命令“vm”，查看系统进程的虚拟地址描述符信息：

1. 按F5启动调试。

2. 待EOS启动完毕，在EOS控制台中输入命令“pt”后按回车。“pt”命令可以输出当前系统中的进程列表，其中系统进程的ID为1。

3. 在EOS控制台中输入命令“vm 1”后按回车。

观察命令执行的结果，如图5-3所示，可以了解系统进程的虚拟地址描述符信息。

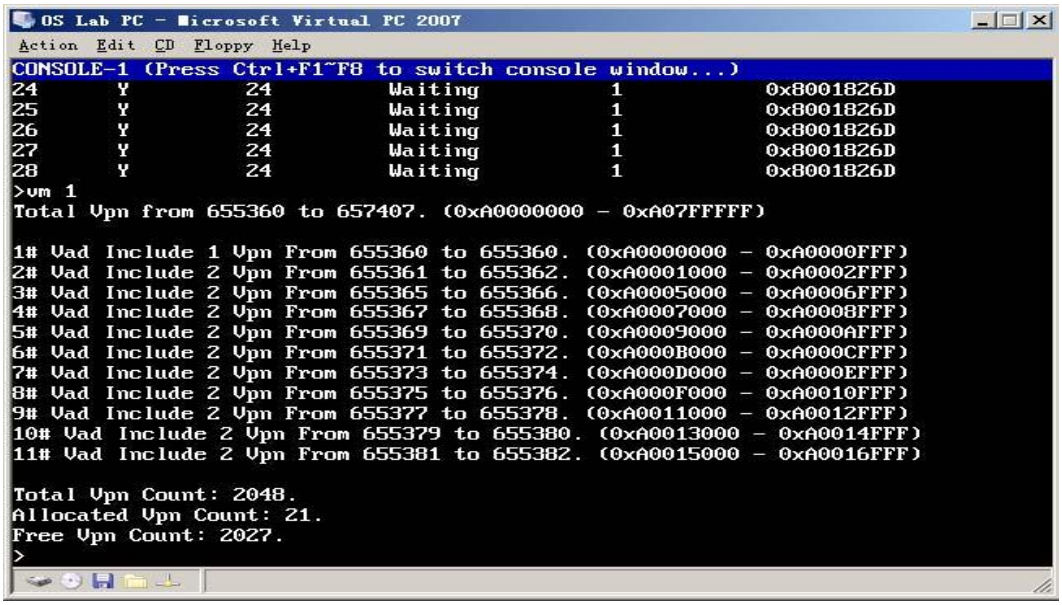


图5-3：使用“vm”命令查看系统进程虚拟地址描述符的结果

系统进程中由虚拟地址描述符所管理的虚拟页只会分配给进程的句柄表（句柄表占用一个虚拟页）和线程的堆栈（堆栈占用两个虚拟页）。结合之前“pt”命令输出的进程和线程信息可知，当前系统中只有 1 个系统进程以及 10 个系统线程，所以在图 5-3 中，1 号描述符所包含的一个虚拟页即为系统进程的句柄表，而 2 到 11 号这 10 个描述符所分别包含的两个虚拟页即为 10 个系统线程的堆栈。

可以按照下面的步骤执行控制台命令“vm”，查看当创建了一个应用程序进程后，系统进程和应用程序进程中虚拟地址描述符的信息：

1. 在“项目管理器”窗口中双击Floppy.img文件，使用FloppyImageEditor工具打开此软盘镜像。

2. 将本实验文件夹中的LoopApp.exe文件添加到软盘镜像的根目录中（将LoopApp.exe文件拖动到FloppyImageEditor窗口中释放即可）。EOS应用程序LoopApp.exe的源代码可以参考本实验文件夹中的LoopApp.c文件。

3. 点击FloppyImageEditor工具栏上的保存按钮，关闭该工具。

4.按F5启动调试。

5.待EOS启动完毕，在EOS控制台中输入命令“A:\LoopApp.exe”后按回车。此时就使用EOS应用程序文件LoopApp.exe创建了一个应用程序进程，由于此进程执行了一个死循环，所以此进程不会结束执行，除非关闭虚拟机。

6.此时按Ctrl+F2切换到“Console-2”，然后输入命令“pt”后按回车。输出的信息如图5-4所示。其中ID为31的进程就是应用程序进程，ID为33的线程就是应用程序进程的主线程。

7.输入命令“vm 1”后按回车，可以查看系统进程中虚拟地址描述符的信息。输出的信息如图5-5所示。与图15-3比较可知，3号描述符所包含的一个虚拟页即为应用程序进程的句柄表，13号描述符所包含的两个虚拟页即为应用程序进程主线程的堆栈。

8.输入命令“vm 31”后按回车，可以查看应用程序进程中虚拟地址描述符的信息。输出的信息如图5-6所示。

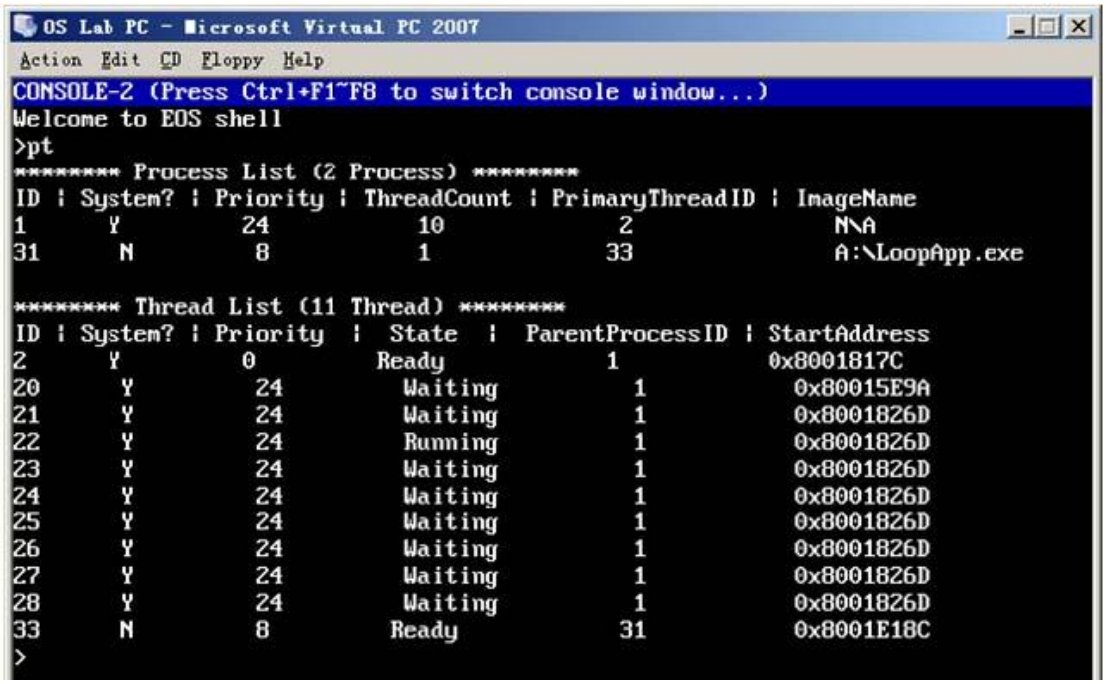


图5-4：使用pt命令查看有应用程序运行时进程和线程的信息

```
>vm 1
Total Vpn from 655360 to 657407. (0xA0000000 - 0xA07FFFFF)

1# Vad Include 1 Vpn From 655360 to 655360. (0xA0000000 - 0xA0000FFF)
2# Vad Include 2 Vpn From 655361 to 655362. (0xA0001000 - 0xA0002FFF)
3# Vad Include 1 Vpn From 655363 to 655363. (0xA0003000 - 0xA0003FFF)
4# Vad Include 2 Vpn From 655365 to 655366. (0xA0005000 - 0xA0006FFF)
5# Vad Include 2 Vpn From 655367 to 655368. (0xA0007000 - 0xA0008FFF)
6# Vad Include 2 Vpn From 655369 to 655370. (0xA0009000 - 0xA000AFFF)
7# Vad Include 2 Vpn From 655371 to 655372. (0xA000B000 - 0xA000CFFF)
8# Vad Include 2 Vpn From 655373 to 655374. (0xA000D000 - 0xA000EFFF)
9# Vad Include 2 Vpn From 655375 to 655376. (0xA000F000 - 0xA0010FFF)
10# Vad Include 2 Vpn From 655377 to 655378. (0xA0011000 - 0xA0012FFF)
11# Vad Include 2 Vpn From 655379 to 655380. (0xA0013000 - 0xA0014FFF)
12# Vad Include 2 Vpn From 655381 to 655382. (0xA0015000 - 0xA0016FFF)
13# Vad Include 2 Vpn From 655383 to 655384. (0xA0017000 - 0xA0018FFF)

Total Vpn Count: 2048.
Allocated Vpn Count: 24.
Free Vpn Count: 2024.
>
```

图5-5：创建了一个应用程序进程后，系统进程中虚拟地址描述符的信息

```
>vm 31
Total Vpn from 16 to 524271. (0x10000 - 0x7FFEFFFF)

1# Vad Include 5 Vpn From 1024 to 1028. (0x400000 - 0x404FFF)

Total Vpn Count: 524256.
Allocated Vpn Count: 5.
Free Vpn Count: 524251.
>
```

图5-6：使用“vm”命令查看应用程序进程虚拟地址描述符的结果

在进程的4G逻辑地址空间中，应用程序进程可以自行管理低2G的用户空间。从图15-6中的信息可以得知，低2G的用户空间又被分为了三部分：

- 0x00000000-0x0000FFFF 由16个虚拟页构成的64KB静态空间，用于捕捉对空指针的非法访问。
- 0x00010000-0x7FFEFFFF 由虚拟地址描述符管理的动态空间，用于存储应用程序进程的代码和数据。图15-6显示应用程序进程的代码和数据占用了此空间中的5个虚拟页，并且是用从应用程序的基址0x00400000起始的。
- 0x7FFF0000-0x7FFFFFFF由16个虚拟页构成的64KB静态空间，用于捕捉对空指针的非法访问。

为了加深对进程逻辑地址空间的理解，可以在控制台1至控制台7中都执行命令“A:\LoopApp.exe”，从而让应用程序创建7个进程，然后在控制台8中执行“pt”、“vm”等命令，查看系统进程和应用程序进程的虚拟地址描述符。

3.5 在系统进程中分配虚拟页和释放虚拟页

接下来，在vm命令函数中添加分配虚拟页和释放虚拟页的代码，单步调试管理虚拟页的方法。首先，按照下面的步骤修改vm命令的源代码：

1. 使用OS Lab打开本实验文件夹中的vm.c文件（将文件拖动到OS Lab窗口中释放即可打开）。此文件中有一个修改后的ConsoleCmdVM函数，主要是在原有代码的后面增加了分配虚拟页和释放物理页的代码。
2. 使用vm.c文件中ConsoleCmdVM函数的函数体替换ke/sysproc.c文件中ConsoleCmdVM函数的函数体。
3. 按F7生成修改后的EOS Kernel项目。
4. 按F5启动调试。
5. 待EOS启动完毕，在EOS控制台中输入命令“vm 1”后按回车。

命令执行的结果会同时转储在“输出”窗口中，内容如图15-7所示。尝试说明分配虚拟页或者释放虚拟页后虚拟地址描述符以及物理存储器的变化情况。

```
Total Vpn from 655360 to 657407. (0xA0000000 - 0xA07FFFFF)

1# Vad Include 1 Vpn From 655360 to 655360. (0xA0000000 - 0xA0000FFF)
2# Vad Include 2 Vpn From 655361 to 655362. (0xA0001000 - 0xA0002FFF)
3# Vad Include 2 Vpn From 655365 to 655366. (0xA0005000 - 0xA0006FFF)
4# Vad Include 2 Vpn From 655367 to 655368. (0xA0007000 - 0xA0008FFF)
5# Vad Include 2 Vpn From 655369 to 655370. (0xA0009000 - 0xA000AFFF)
6# Vad Include 2 Vpn From 655371 to 655372. (0xA000B000 - 0xA000CFFF)
7# Vad Include 2 Vpn From 655373 to 655374. (0xA000D000 - 0xA000EFFF)
8# Vad Include 2 Vpn From 655375 to 655376. (0xA000F000 - 0xA0010FFF)
9# Vad Include 2 Vpn From 655377 to 655378. (0xA0011000 - 0xA0012FFF)
10# Vad Include 2 Vpn From 655379 to 655380. (0xA0013000 - 0xA0014FFF)
11# Vad Include 2 Vpn From 655381 to 655382. (0xA0015000 - 0xA0016FFF)

Total Vpn Count: 2048.
Allocated Vpn Count: 21.
Free Vpn Count: 2027.

Zeroed Physical Page Count: 0.
Free Physical Page Count: 7126.

New VM's base address: 0xA0003000. Size: 0x1000.

1# Vad Include 1 Vpn From 655360 to 655360. (0xA0000000 - 0xA0000FFF)
2# Vad Include 2 Vpn From 655361 to 655362. (0xA0001000 - 0xA0002FFF)
3# Vad Include 1 Vpn From 655363 to 655363. (0xA0003000 - 0xA0003FFF)
```

```
4# Vad Include 2 Vpn From 655365 to 655366. (0xA0005000 - 0xA0006FFF)
5# Vad Include 2 Vpn From 655367 to 655368. (0xA0007000 - 0xA0008FFF)
6# Vad Include 2 Vpn From 655369 to 655370. (0xA0009000 - 0xA000AFFF)
7# Vad Include 2 Vpn From 655371 to 655372. (0xA000B000 - 0xA000CFFF)
8# Vad Include 2 Vpn From 655373 to 655374. (0xA000D000 - 0xA000EFFF)
9# Vad Include 2 Vpn From 655375 to 655376. (0xA000F000 - 0xA0010FFF)
10# Vad Include 2 Vpn From 655377 to 655378. (0xA0011000 - 0xA0012FFF)
11# Vad Include 2 Vpn From 655379 to 655380. (0xA0013000 - 0xA0014FFF)
12# Vad Include 2 Vpn From 655381 to 655382. (0xA0015000 - 0xA0016FFF)

Allocated Vpn Count: 22.
Free Vpn Count: 2026.

Zeroed Physical Page Count: 0.
Free Physical Page Count: 7126.

Free VM's base address: 0xA0003000. Size: 0x1000.

1# Vad Include 1 Vpn From 655360 to 655360. (0xA0000000 - 0xA0000FFF)
2# Vad Include 2 Vpn From 655361 to 655362. (0xA0001000 - 0xA0002FFF)
3# Vad Include 2 Vpn From 655365 to 655366. (0xA0005000 - 0xA0006FFF)
4# Vad Include 2 Vpn From 655367 to 655368. (0xA0007000 - 0xA0008FFF)
5# Vad Include 2 Vpn From 655369 to 655370. (0xA0009000 - 0xA000AFFF)
6# Vad Include 2 Vpn From 655371 to 655372. (0xA000B000 - 0xA000CFFF)
7# Vad Include 2 Vpn From 655373 to 655374. (0xA000D000 - 0xA000EFFF)
8# Vad Include 2 Vpn From 655375 to 655376. (0xA000F000 - 0xA0010FFF)
9# Vad Include 2 Vpn From 655377 to 655378. (0xA0011000 - 0xA0012FFF)
10# Vad Include 2 Vpn From 655379 to 655380. (0xA0013000 - 0xA0014FFF)
11# Vad Include 2 Vpn From 655381 to 655382. (0xA0015000 - 0xA0016FFF)

Allocated Vpn Count: 21.
Free Vpn Count: 2027.

Zeroed Physical Page Count: 0.
Free Physical Page Count: 7126.
```

图5-7：分配虚拟页或者释放虚拟页后虚拟地址描述符及物理存储器的变化情况

按照下面的步骤调试分配虚拟页和释放虚拟页的过程：

1.在ke/sysproc.c文件的ConsoleCmdVM函数中，在调用MmAllocateVirtualMemory函数的代码行（第1082行）添加一个断点，在调用MmFreeVirtualMemory函数的代码行（第1147行）添加一个断点。

2.按F5启动调试。

3.待EOS启动完毕，在EOS控制台中输入命令“vm 1”后按回车。

4.vm命令开始执行后，会在调用MmAllocateVirtualMemory函数的代码行处中断。此时要注意参数BaseAddress和RegionSize初始化的值。按F11调试进入MmAllocateVirtualMemory函数。

5.按F10单步调试MmAllocateVirtualMemory函数的执行过程，尝试回答下面的问题：

(1) 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？

(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？

(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？

(4) 参考MiReserveAddressRegion函数的定义和注释，说明该函数的功能。

继续调试释放虚拟页的过程：

1. 按F5继续执行，会在调用MmFreeVirtualMemory函数的代码行处中断。此时要注意参数BaseAddress和RegionSize初始化的值。按F11调试进入MmFreeVirtualMemory函数。

2. 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？

(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。

结束此次调试后，继续按照下列要求修改ConsoleCmdVM函数的源代码，加深对虚拟页分配和释放过程的理解：

1. 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。

2. 尝试在调用MmAllocateVirtualMemory函数时将BaseAddress参数的值设置为已经被占用的虚拟内存，例如0xA0000000，观察“输出”窗口中转储的信息。将“输出”窗口中转储的信息保存在文本文件中。

3. 尝试在调用MmAllocateVirtualMemory函数时将RegionSize参数的值设置为PAGE_SIZE*2，将BaseAddress参数的值设置为0xA0017004，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，以及申请的虚拟地址会对分配的虚拟内存大小产生什么样的影响。将“输出”窗口中转储的信息保存在文本文件中。

3.6 在应用程序进程中分配虚拟页和释放虚拟页

3.6.1 要求

创建一个EOS应用程序，并编写代码完成下列功能：

1. 调用API函数VirtualAlloc，分配一个整型变量所需的空间，并使用一个整型变量的指针指向这个空间。

2. 修改整型变量的值为0xFFFFFFFF。在修改前输出整型变量的值，在修改后再输出整型变量的值。

3. 调用API函数Sleep，等待10秒钟。

4. 调用API函数VirtualFree，释放之前分配的整型变量的空间。

5. 进入死循环，这样应用程序就不会结束。

3.6.2 测试方法

1. 代码修改完毕后，按F7生成EOS应用程序项目。

2. 按F5启动调试，应用程序自动执行后输出的结果可以参照图5-8所示。

3. 在应用程序分配虚拟页后，利用10秒后才释放虚拟页的间隙，可以在控制台2中执行命令“vm 31”，查看此时应用程序进程的虚拟地址描述符信息；在应用程序释放虚拟页后，可以在控制台2中再次执行命令“vm 31”，查看此时应用程序进程的虚拟地址描述符信息。输出的结果可以参照图5-9所示。

图5-8：在应用程序进程中分配虚拟页和释放虚拟页

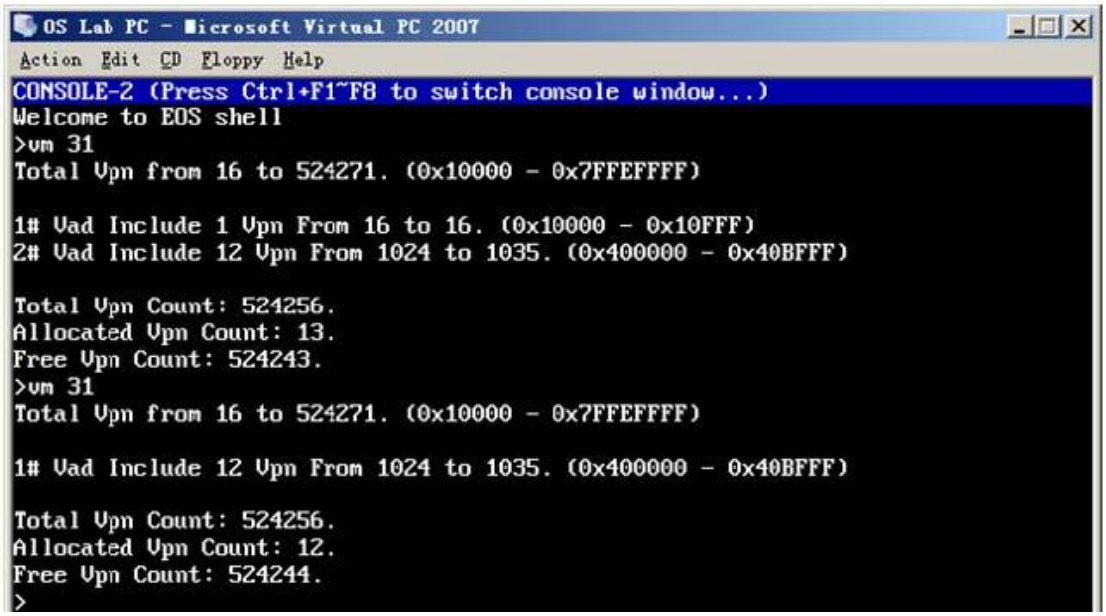


图5-9：分配虚拟页后和释放虚拟页后，应用程序进程的虚拟地址描述符信息

3.6.3 提示

1. API函数VirtualAlloc定义在api/eosapi.c文件的第48行。此API函数主要调用了EOS内核函数MmAllocateVirtualMemory。在EOS应用程序中调用函数VirtualAlloc时，除了使用MEM_RESERVE标志外，还必须使用MEM_COMMIT标志。

2. API函数VirtualFree定义在api/eosapi.c文件的第70行。此API函数主要调用了EOS内核函数MmFreeVirtualMemory。在EOS应用程序中调用函数VirtualFree时，要使用MEM_RELEASE标志。

3. 可以参考本实验文件中的LoopApp.c文件，在应用程序的最后执行一个死循环。

实验六　分页存储器管理

一、实验目的

- 了解 i386 处理器的二级页表硬件机制，理解分页存储器管理原理。
- 在 OS Lab 调试器中查看 EOS 中的页目录和页表，理解页目录和页表的管理方式。
- 通过手工分配两个物理页，其中一页用作页表、另一页用作被映射物理页，通过手工修改页表和页目录的方式，将物理页映射到虚拟地址 0xE0000000 处。通过映射过程，理解分页地址变换原理。

二、实验环境

1) Windows 7 及以上系统
2) 集成实验环境 OS Lab

三、实验内容

3.1 准备实验

按照下面的步骤准备实验：

1. 启动 OS Lab。
2. 新建一个 EOS Kernel 项目。
3. 从“项目管理器”窗口中打开 ke\start.c 源文件，在函数 KiSystemStartup 的开始处定义一个辅助数组“ULONG_PTR PfnArray[2];”，然后在语句：
4. ObInitializeSystem1();
5. 所在的行添加一个断点。
6. 编译并启动调试内核，内核会在此断点处中断执行。此时内核已经执行过内存管理模块的初始化函数 MmInitializeSystem1()，内核已经可以提供内存管理功能。
7. 选择“调试”菜单中的“快速监视”，打开快速监视对话框，接下来将在快速监视对话框中完成本次实验内容。

3.2 查看 EOS 中的页目录和页表

使用 OS Lab 打开本实验文件夹中的 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，main 函数的流程图可以参见图 6-1。

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. 按 F5 启动调试。

9. 应用程序执行的过程中，会将该进程的二级页表映射信息输出到虚拟机窗口和 OS Lab“输出”窗口中，输出内容如图 6-2（a）。

10. 将“输出”窗口中的内容复制到一个文本文件中。

图 6-2（a）中第一行是 CR3 寄存器的值，也就是页目录所在的页框号。第一列是页目录中有效的 PDE，第二列是 PDE 映射的页表中有效的 PTE（详细的格式可以参考源代码中的注释）。注意，在标号为 0x200 的 PDE 对应的页表中，所有的 1024 个 PTE 都是有效的，所以在图中省略了一部分。

根据图 6-2（a）回答下面的问题：

- 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？

- 映射用户地址空间（低 2G）的页表的页框号是多少？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、数据和堆栈？页框号分别是多少？

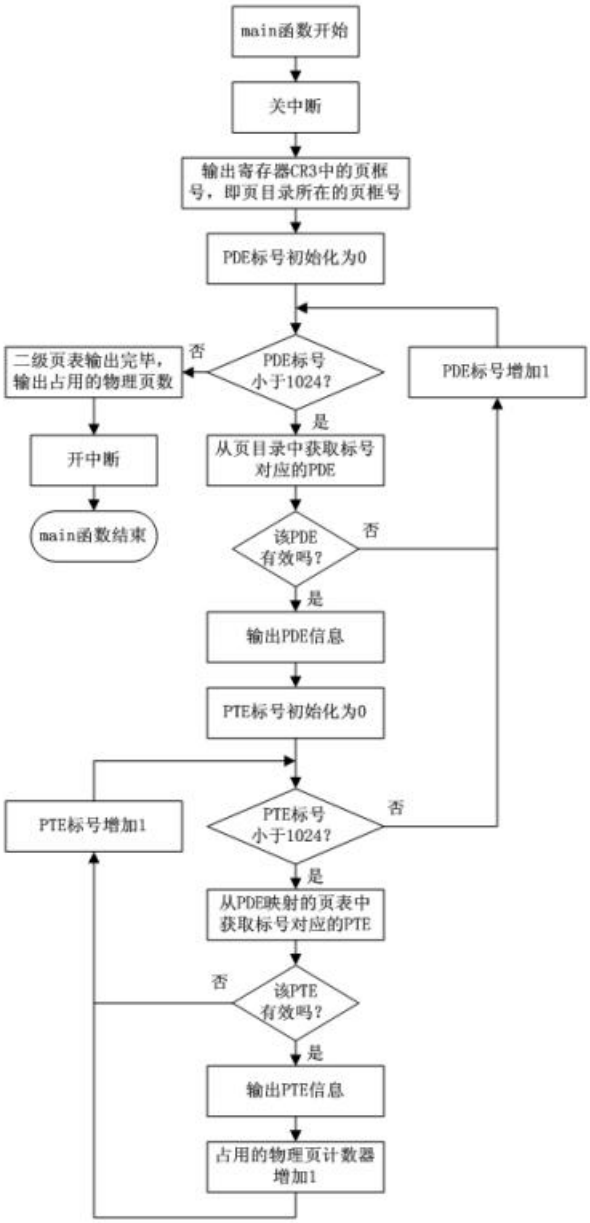


图 6-1：main 函数流程图

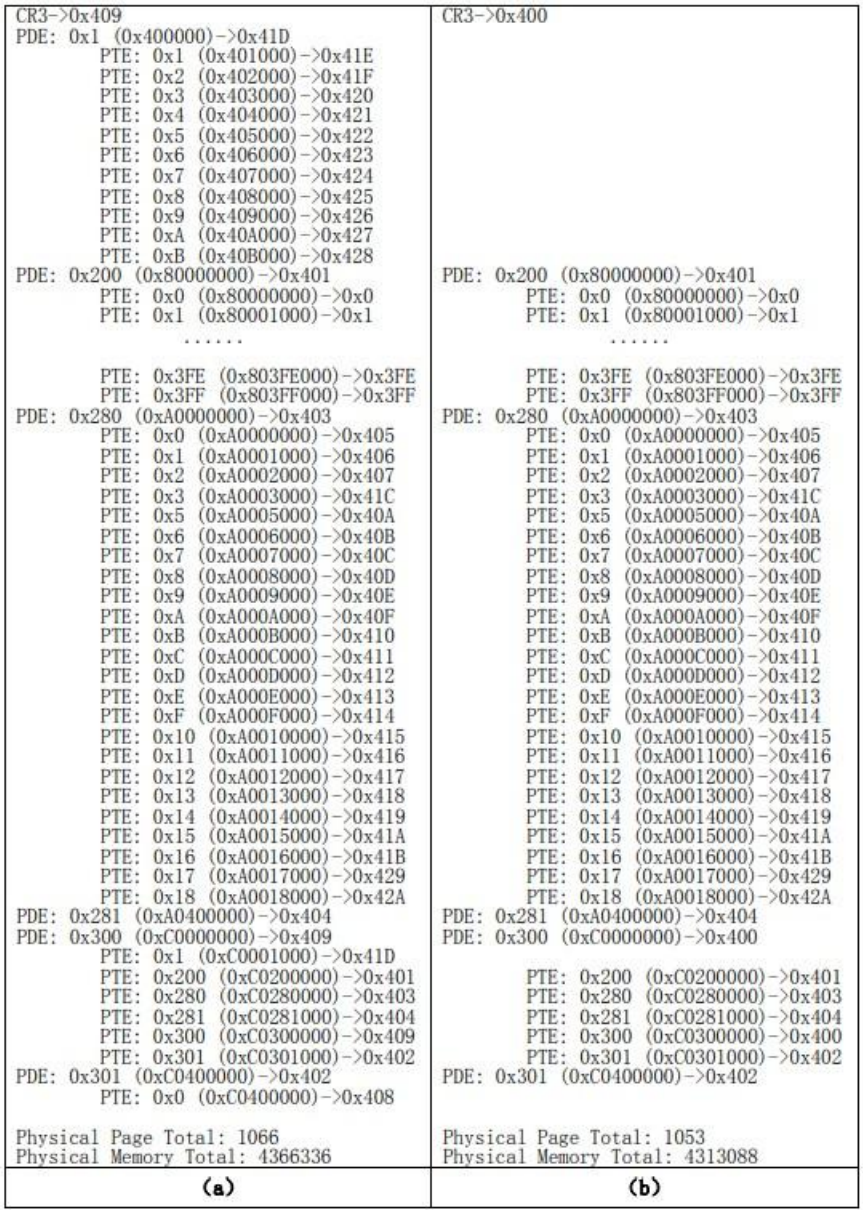


图 6-2：（a）EOS 应用程序进程的二级页表映射信息；

（b）有应用程序进程时，系统进程的二级页表映射过程

3.3 查看应用程序进程和系统进程并发时的页目录和页表

需要对 EOS 应用程序进行一些修改：

1. 结束之前的调试。
2. 取消 EOSApp.c 第 121 行语句的注释（该行语句会等待 10 秒）。
3. 按 F7 生成修改后的 EOS 应用程序项目。
4. 按 F5 启动调试。
5. 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”。

6. 在“Console-2”中输入命令“mm”后按回车，会将系统进程的二级页表映射信息输出到虚拟机窗口和 OS Lab 的“输出”窗口，输出内容如图 6-2（b）。注意，在图 6-2（b）中添加了一些空行，方便与图 6-2（a）比较。“Console-1”中的应用程序在等待 10 秒后，又会输出和图 6-2（a）一致的内容。

7. 将“输出”窗口中的内容复制到一个文本文件中。

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 6-2（a）和（b）回答下面的问题：

- 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，哪些是独占的，哪些是共享的？分析其中的原因。
- 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？

应用程序结束后，在“Console-1”中再次输入命令“mm”，查看在没有应用程序进程时，系统进程的页目录和页表。将“输出”窗口中的内容复制到一个文本文件中。将输出的内容与图 6-2（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，内核要释放 PCB 占用的内存。）

3.4 查看应用程序进程并发时的页目录和页表

需要对 EOS 应用程序进行一些修改：

1. 结束之前的调试。

2. 取消 EOSApp.c 第 201 行语句的注释（该行语句会等待 10 秒）。

3. 按 F7 生成修改后的 EOS 应用程序项目。

4. 按 F5 启动调试。

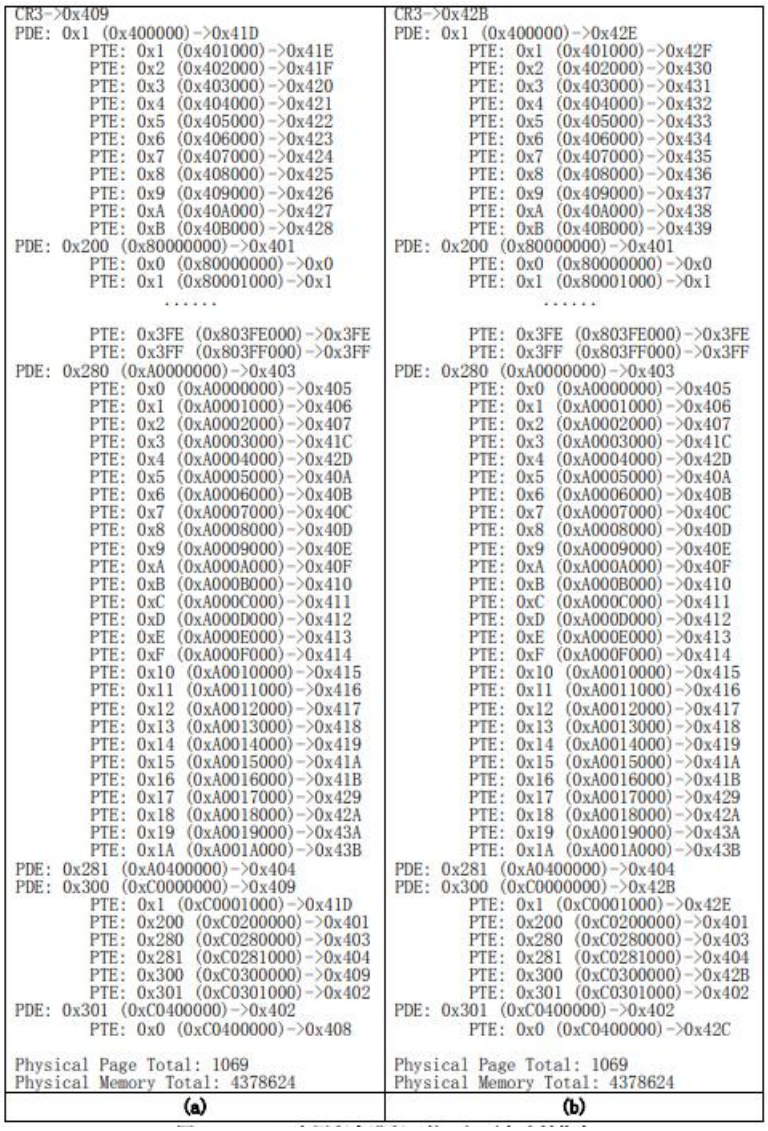
5. 在“Console-1”中会自动执行 EOSApp.exe，创建该应用程序进程。利用其等待 10 秒的时间，按 Ctrl+F2 切换到“Console-2”。

6. 在“Console-2”中输入“eosapp”后按回车，再执行一个 EOSApp.exe。

7. 由 EOSApp.exe 创建的两个并发进程会先后在各自的控制台和 OS Lab

“输出”窗口中，输出各自的二级页表映射信息。输出的内容如图 6-3。

8. 将“输出”窗口中的内容复制到一个文本文件中。



```
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)->0x403
        PTE: 0x0 (0xA0000000)->0x405
        PTE: 0x1 (0xA0001000)->0x406
        PTE: 0x2 (0xA0002000)->0x407
        PTE: 0x3 (0xA0003000)->0x41C
        PTE: 0x4 (0xA0004000)->0x42D
        PTE: 0x5 (0xA0005000)->0x40A
        PTE: 0x6 (0xA0006000)->0x40B
        PTE: 0x7 (0xA0007000)->0x40C
        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)->0x414
        PTE: 0x10 (0xA0010000)->0x415
        PTE: 0x11 (0xA0011000)->0x416
        PTE: 0x12 (0xA0012000)->0x417
        PTE: 0x13 (0xA0013000)->0x418
        PTE: 0x14 (0xA0014000)->0x419
        PTE: 0x15 (0xA0015000)->0x41A
        PTE: 0x16 (0xA0016000)->0x41B
        PTE: 0x17 (0xA0017000)->0x429
        PTE: 0x18 (0xA0018000)->0x42A
        PTE: 0x19 (0xA0019000)->0x43A
        PTE: 0x1A (0xA001A000)->0x43B
PDE: 0x281 (0xA0400000)->0x404
PDE: 0x300 (0xC0000000)->0x409
        PTE: 0x1 (0xC0001000)->0x41D
        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: 1069
Physical Memory Total: 4378624
```

(a)

```
CR3->0x42B
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)->0x403
        PTE: 0x0 (0xA0000000)->0x405
        PTE: 0x1 (0xA0001000)->0x406
        PTE: 0x2 (0xA0002000)->0x407
        PTE: 0x3 (0xA0003000)->0x41C
        PTE: 0x4 (0xA0004000)->0x42D
        PTE: 0x5 (0xA0005000)->0x40A
        PTE: 0x6 (0xA0006000)->0x40B
        PTE: 0x7 (0xA0007000)->0x40C
        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)->0x414
        PTE: 0x10 (0xA0010000)->0x415
        PTE: 0x11 (0xA0011000)->0x416
        PTE: 0x12 (0xA0012000)->0x417
        PTE: 0x13 (0xA0013000)->0x418
        PTE: 0x14 (0xA0014000)->0x419
        PTE: 0x15 (0xA0015000)->0x41A
        PTE: 0x16 (0xA0016000)->0x41B
        PTE: 0x17 (0xA0017000)->0x429
        PTE: 0x18 (0xA0018000)->0x42A
        PTE: 0x19 (0xA0019000)->0x43A
        PTE: 0x1A (0xA001A000)->0x43B
PDE: 0x281 (0xA0400000)->0x404
PDE: 0x300 (0xC0000000)->0x42B
        PTE: 0x1 (0xC0001000)->0x42E
        PTE: 0x200 (0xC0200000)->0x401
        PTE: 0x280 (0xC0280000)->0x403
        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)

图 6-3：（a）应用程序进程 1 的二级页表映射信息；

（b）应用程序进程 2 的二级页表映射过程

结合图 6-3（a）和（b）回答下面的问题：

● 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。假设进程 1 的 0x41E 和 0x41F 物理页保存了应用程序的可执行代码，由于可执行代码是不变的、只读的，现在假设优化过的 EOS 允许进程 2 共享进程 1 的保存了可执行代码的物理页，尝试结合图 6-3 写出此时进程 2 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。

- 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？

3.4 在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. 打开本实验文件夹中的 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. OS Lab 会弹出一个调试异常对话框，选择“是”调试异常。

9. 黄色箭头指向访问虚拟地址 0xE0000000 的代码行。由于该虚拟地址没有映射物理内存（图 6-2 和图 6-3 中都未映射该虚拟地址），所以对该虚拟地址的访问会触发异常。

10. 结束此次调试，然后删除或者注释会触发异常的该行代码。

按照下面的步骤验证其它结论：

1. 按 F7 生成该内核项目。

2. 按 F5 启动调试。

3. 在 EOS 控制台中输入命令“mm”后按回车。

4. 在 OS Lab 的“输出”窗口中查看执行的结果，并将“输出”窗口中的内容复制到一个文本文件中。

结合插入的源代码和执行的结果理解上面的结论。注意，在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。

实验七　磁盘调度算法

一、实验目的

- 通过学习EOS实现磁盘调度算法的机制，掌握磁盘调度算法执行的条件和时机。
- 观察EOS实现的FCFS、SSTF和SCAN磁盘调度算法，了解常用的磁盘调度算法。
- 编写CSCAN和N-Step-SCAN磁盘调度算法，加深对各种扫描算法的理解。

二、实验环境

1) Windows 7 及以上系统
2) 集成实验环境 OS Lab

三、实验内容

3.1 准备实验

按照下面的步骤准备实验：

1. 启动OS Lab。
2. 新建一个EOS Kernel项目。

3.2 验证先来先服务（FCFS）磁盘调度算法

按照下面的步骤进行验证：

1. 在“项目管理器”窗口中双击ke文件夹中的sysproc.c文件，打开此文件。

2. 在sysproc.c文件的第580行找到控制台命令“ds”对应的函数ConsoleCmdDiskSchedule。“ds”命令专门用来测试磁盘调度算法。阅读该函数中的源代码，目前该函数使磁头初始停留在磁道10，其它被阻塞的线程依次访问磁道8、21、9、78、0、41、10、67、12、10。

3. 打开io/block.c文件，在第378行找到磁盘调度算法函数IopDiskSchedule。阅读该函数中的源代码，目前此函数实现了FCFS磁盘调度算法，其流程图可以参见图7-1。

4. 按F7生成项目，然后按F5启动调试。

5. 待EOS启动完毕，在EOS控制台中输入命令“ds”后按回车。

在EOS控制台中会首先显示磁头的起始位置是10磁道，然后按照线程被阻

塞的顺序依次显示线程的信息（包括线程 ID 和访问的磁道号）。磁盘调度算法执行的过程中，在 OS Lab 的“输出”窗口中也会首先显示磁头的起始位置，然后按照线程被唤醒的顺序依次显示线程信息（包括线程 ID、访问的磁道号、磁头移动的距离和方向），并在磁盘调度结束后显示此次调度的统计信息（包括总寻道数、寻道次数和平均寻道数）。对比 EOS 控制台和“输出”窗口中的内容，可以发现 FCFS 算法是根据线程访问磁盘的先后顺序进行调度的。图 7-2 显示了本次调度执行时磁头移动的轨迹。

可以在控制台中多次输入“ds”命令，查看磁盘调度算法执行的情况。将“输出”窗口中的内容复制到一个文本文件中，然后结束此次调试。

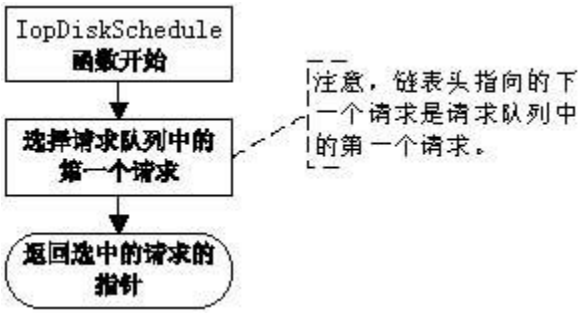


图7-1：实现了FCFS算法的IopDiskSchedule函数流程图

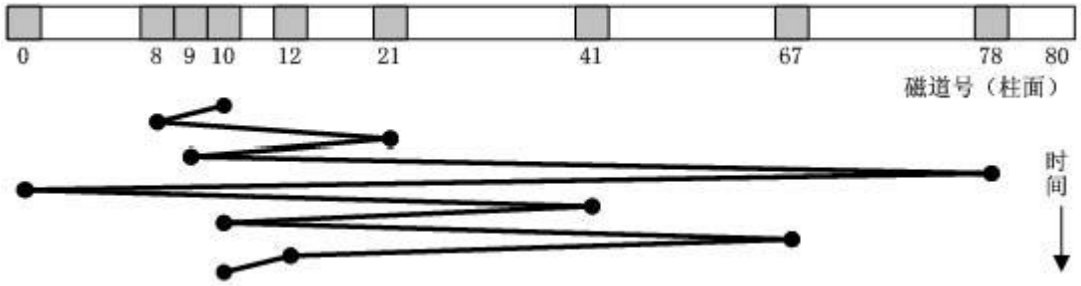


图7-2：FCFS算法磁头移动的轨迹（总寻道数360 寻道次数 10 平均寻道数 36）

3.3 验证最短寻道时间优先（SSTF）磁盘调度算法

使用OS Lab打开本实验文件夹中的sstf.c文件（将sstf.c文件拖动到OS Lab窗口中释放即可）。该文件提供的IopDiskSchedule函数实现了SSTF磁盘调度算法。在阅读此函数的源代码的时，可以参考图7-3所示的流程图，并且应该特别注意下面几点：

- 变量Offset是有符号的长整型，用来表示磁头的偏移（包括距离和方向）。Offset大于0时表示磁头向内移动（磁道号增加）；小于0时表示磁头向外移动（磁道号减少）；等于0时表示磁头没有移动。而名称以“Distance”结尾的变量都

是无符号长整型，只表示磁头移动的距离（无方向）。所以在比较磁头的偏移和距离时，或者在将偏移赋值给距离时，都要取偏移的绝对值（调用C库函数abs）。本实验在实现其它磁盘调度算法时也同样遵守此约定。

- 在开始遍历之前，将最小距离（ShortestDistance）初始化为最大的无符号长整型数，这样，第一次计算的距离一定会小于最小距离，从而可以使用第一次计算的距离来再次初始化最小距离。本实验在实现其它磁盘调度算法时也同样使用了此技巧。

按照下面的步骤进行验证：

1. 使用sstf.c文件中IopDiskSchedule函数的函数体，替换block.c文件中IopDiskSchedule函数的函数体。

2. 按F7生成项目，然后按F5启动调试。

3. 待EOS启动完毕，在EOS控制台中输入命令“ds”后按回车。

对比EOS控制台和“输出”窗口中的内容(特别是线程ID的顺序),可以发现，SSTF算法唤醒线程的顺序与线程被阻塞的顺序是不同的。图7-4显示了本次调度执行时磁头移动的轨迹。对比SSTF算法与FCFS算法在“输出”窗口中的内容，可以看出，SSTF算法的平均寻道数明显低于FCFS算法。但是，SSTF算法能保证平均寻道数最少吗？在后面的实验中会进行验证。

可以在控制台中多次输入“ds”命令，查看磁盘调度算法执行的情况。将“输出”窗口中的内容复制到一个文本文件中，然后结束此次调试。

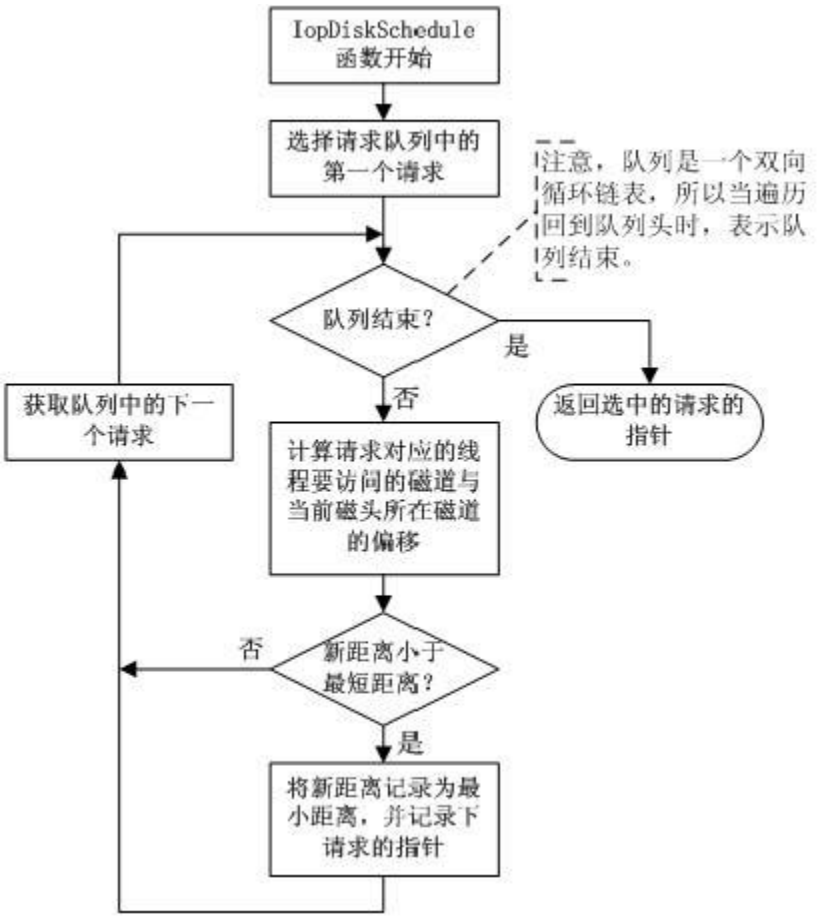


图7-3：实现了SSTF算法的IopDiskSchedule函数流程图

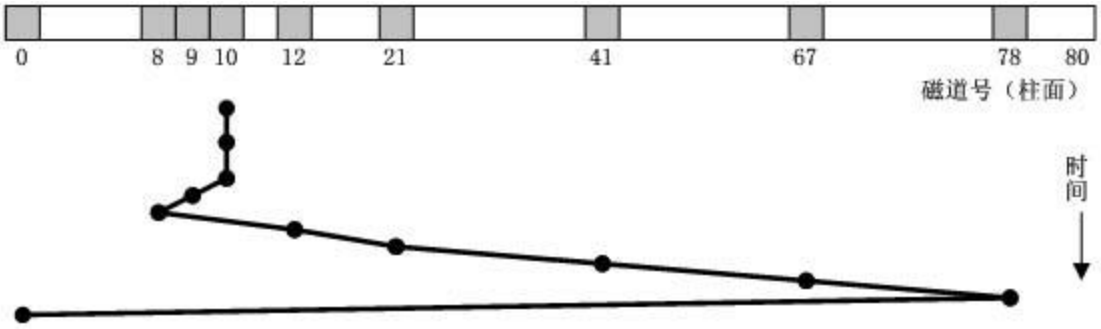


图7-4：SSTF算法磁头移动的轨迹（总寻道数150 寻道次数 10 平均寻道数 15）

3.4 验证 SSTF 算法造成的线程“饥饿”现象

使用SSTF算法时，如果不断有新线程要求访问磁盘，而且其所要访问的磁道与当前磁头所在磁道的距离较近，这些新线程的请求必然会被优先满足，而等待队列中一些老线程的请求就会被严重推迟，从而使老线程出现“饥饿”现象。

按照下面的步骤进行实验，观察这个现象：

1. 修改sysproc.c文件ConsoleCmdDiskSchedule函数中的源代码，仍然使磁头初始停留在磁道10，而让其它线程依次访问磁道78、21、9、8、11、41、10、

67、12、10。

2. 按F7生成项目，然后按F5启动调试。

3. 待EOS启动完毕，在EOS控制台中输入命令“ds”后按回车。

查看“输出”窗口中显示的内容，可以发现，虽然访问78号磁道的线程的请求第一个被放入请求队列，但却被推迟到最后才被处理，出现了“饥饿”现象。如果不断有新线程的请求到达并被优先满足，则访问78号磁道的线程的“饥饿”情况就会更加严重。

将“输出”窗口中的内容复制到一个文本文件中，然后结束此次调试。将ConsoleCmdDiskSchedule函数中线程访问的磁道号恢复到本实验3.2中的样子，在后面的实验中还要使用这些数据。

3.5 验证扫描（SCAN）磁盘调度算法

对SSTF算法稍加改进后可以形成SCAN算法，可防止老线程出现“饥饿”现象。使用OS Lab打开本实验文件夹中的scan.c文件，该文件提供的IopDiskSchedule函数实现了SCAN磁盘调度算法。在阅读此函数的源代码的时，应该特别注意下面几点：

- 在block.c文件中的第374行定义了一个布尔类型的全局变量ScanInside，用于表示扫描算法中磁头移动的方向。该变量值为TRUE时表示磁头向内移动（磁道号增加）；值为FALSE时表示磁头向外移动（磁道号减少）。该变量初始化为TRUE，表示SCAN算法第一次执行时，磁头向内移动。
- 在scan.c文件的IopDiskSchedule函数中使用了双重循环。第一次遍历队列时，查找指定方向上移动距离最短的线程，如果在指定方向上已经没有线程，就变换方向，进行第二次遍历，同样是查找移动距离最短的线程。在这两次遍历中一定能找到合适的线程。

按照下面的步骤进行验证：

1. 使用scan.c文件中IopDiskSchedule函数的函数体，替换block.c文件中IopDiskSchedule函数的函数体。

2. 按F7生成项目，然后按F5启动调试。

3. 待EOS启动完毕，在EOS控制台中输入命令“ds”后按回车。

对比SCAN算法与SSTF算法在“输出”窗口中的内容，可以看出，SCAN算法的平均寻道数有可能小于SSTF算法，所以说SSTF算法不能保证平均寻道数最少。图7-5显示了本次调度执行时磁头移动的轨迹。尝试在控制台中多次输入“ds”命令，查看磁盘调度算法执行的情况，说明为什么线程调度的顺序会发生变化。将“输出”窗口中的内容复制到一个文本文件中，然后结束此次调试。

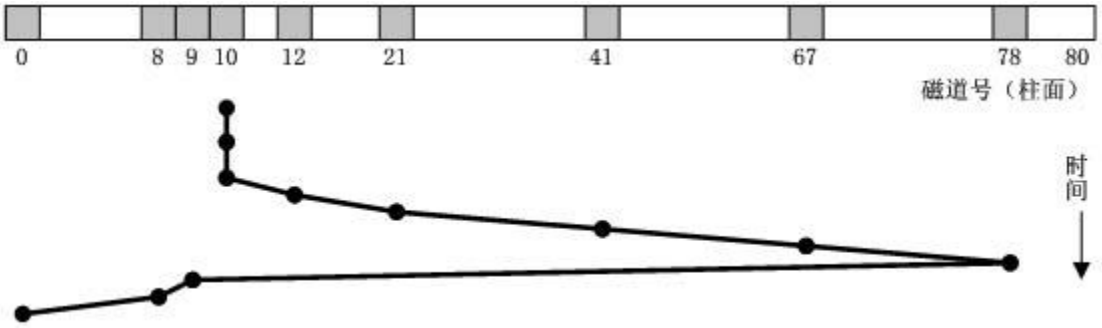


图7-5：SCAN算法磁头移动的轨迹（总寻道数146 寻道次数 10 平均寻道数 14）

使用SCAN算法调度在本实验3.4中产生“饥饿”现象的数据，验证SCAN算法能够解决“饥饿”现象，并将“输出”窗口中的内容保存到一个文本文件中。最后将ConsoleCmdDiskSchedule函数中线程访问的磁道号恢复到本实验3.2中的样子，在后面的实验中还要使用这些数据。

3.6 改写 SCAN 算法

3.6.1 要求

在已有SCAN算法源代码的基础上进行改写，要求不再使用双重循环，而是只遍历一次请求队列中的请求，就可以选中下一个要处理的请求。由于线程和请求总是一一对应的，为了使后面的内容更加简单易懂，有时就不再区分这两个概念。

3.6.2 提示

1. 在一次遍历中，不再关心当前磁头移动的方向，而是同时找到两个方向上移动距离最短的线程所对应的请求，这样就不再需要遍历两次。

2. 在计算出线程要访问的磁道与当前磁头所在磁道的偏移后，可以将偏移分为三种类型：偏移为0，表示线程要访问的磁道与当前磁头所在磁道相同，此情况应该优先被调度，可立即返回该线程对应的请求的指针；偏移大于0，记录向内移动距离最短的线程对应的请求；偏移小于0，记录向外移动距离最短的线程对应的请求。

3. 循环结束后，根据当前磁头移动的方向选择同方向移动距离最短的线程，

如果在同方向上没有线程，就变换方向，选择反方向移动距离最短的线程。具体逻辑可以参见图7-6所示的流程图。

3.6.3 测试方法

使用本实验3.2中的数据进行测试，确保调度的结果与图18-5中显示的一致，也可以多准备几组测试数据，保证改写的SCAN算法是正确的。测试成功后，将改写的SCAN算法源代码备份。

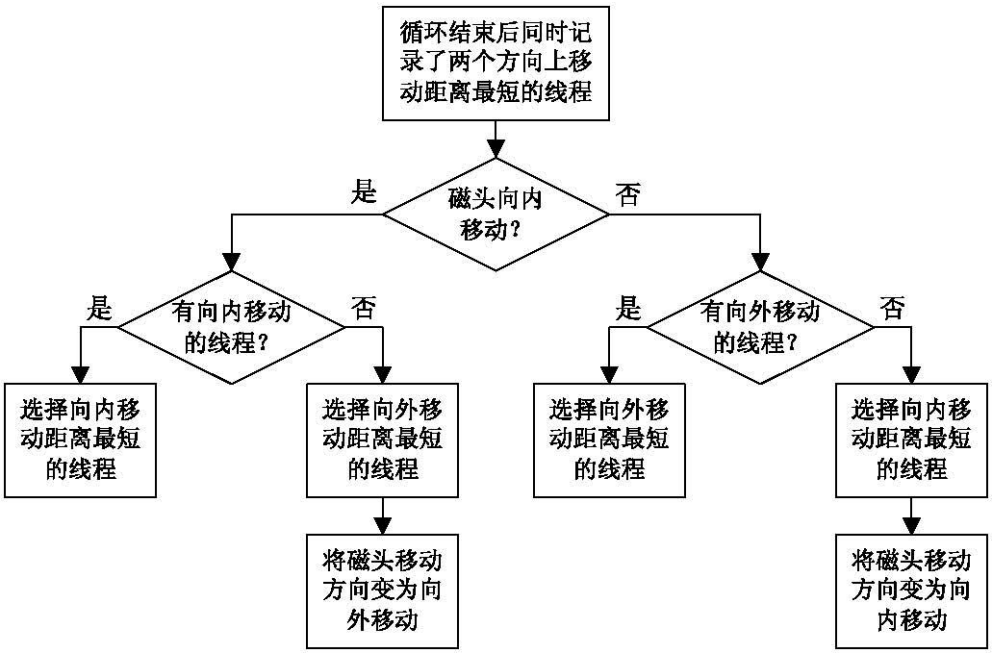


图7-6：循环结束后，根据磁头移动方向选择合适线程的流程图

3.7 编写循环扫描（CSCAN）磁盘调度算法

3.7.1 要求

在已经完成的SCAN算法源代码的基础上进行改写，不再使用全局变量ScanInside确定磁头移动的方向，而是规定磁头只能从外向内移动。当磁头移动到最内的被访问磁道时，磁头立即移动到最外的被访问磁道，即将最大磁道号紧接着最小磁道号构成循环，进行扫描。

由于磁头移动的方向被固定，也就不需要根据磁头移动的方向进行分类处理，所以CSCAN算法的源代码会较SCAN算法更加简单。

3.7.2 提示

1. 由于规定了磁头只能从外向内移动，所以在每次遍历中，总是同时找到向内移动距离最短的线程和向外移动距离最长的线程。注意，与SCAN算法查找向外移动距离最短线程不同，这里查找向外移动距离最长的线程。在开始遍历前，

可以将用来记录向外移动最长距离的变量赋值为0。

2. 在计算出线程要访问的磁道与当前磁头所在磁道的偏移后，同样可以将偏移分为三种类型：偏移为0，表示线程要访问的磁道与当前磁头所在磁道相同，此情况应优先被调度，可立即返回该线程对应的请求的指针；偏移大于0，记录向内移动距离最短的线程对应的请求；偏移小于0，记录向外移动距离最长的线程对应的请求。

3. 循环结束后，选择向内移动距离最短的线程，如果没有向内移动的线程，就选择向外移动距离最长的线程。

3.7.3 测试方法

使用本实验3.2中的数据进行测试，确保调度的结果与图7-7中显示的一致。可以在控制台中多次输入“ds”命令，查看磁盘调度算法执行的情况。测试成功后，将“输出”窗口中的内容复制到一个文本文件中，并将编写的CSCAN算法源代码备份。

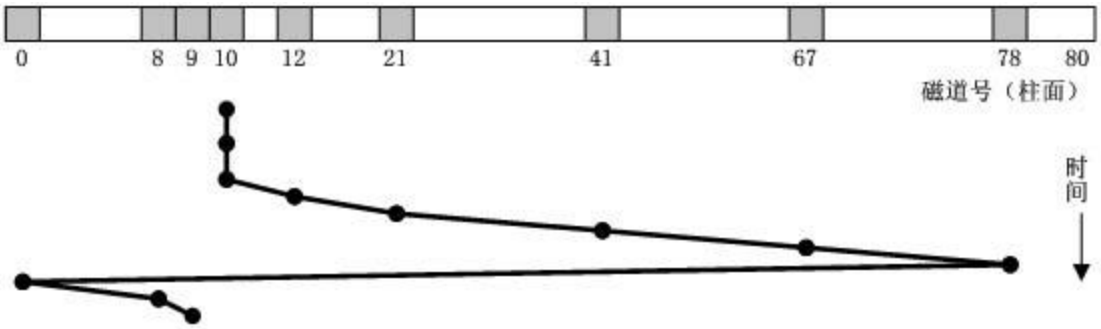


图7-7：CSCAN算法磁头移动的轨迹（总寻道数155 寻道次数 10 平均寻道数 15）

3.8 验证 SSTF、SCAN 及 CSCAN 算法中的“磁臂粘着”现象

观察执行SSTF、SCAN及CSCAN算法时磁头移动的轨迹(图7-4、图7-5和图7-7)，可以看到，在开始时磁头都停留在10磁道不动，这就是“磁臂粘着”现象。为了更加明显的观察该现象，按照下面的步骤进行实验：

1. 修改sysproc.c文件ConsoleCmdDiskSchedule函数中的源代码，仍然使磁头初始停留在磁道10，而让其它线程依次访问磁道78、10、10、10、10、10、10、10、10、10。

2. 分别使用SSTF、SCAN和CSCAN算法调度这组数据。

查看各种算法在“输出”窗口中显示的内容，可以发现，虽然访问78号磁道的线程的请求第一个被放入请求队列，但却被推迟到最后才被处理，出现了“磁

臂粘着”现象。

将“输出”窗口中的内容复制到一个文本文件中后，将ConsoleCmdDiskSchedule函数中线程访问的磁道号恢复到本实验3.2中的样子，在后面的实验中还要使用这些数据。

3.9 编写 N-Step-SCAN 磁盘调度算法

3.9.1 要求

在已经完成的SCAN算法源代码的基础上进行改写，将请求队列分成若干个长度为N的子队列，调度程序按照FCFS原则依次处理这些子队列，而每处理一个子队列时，又是按照SCAN算法。

3.9.2 提示

1. 在block.c文件中的第360行定义了一个宏SUB_QUEUE_LENGTH，表示子队列的长度（即N值）。目前这个宏定义的值为6。在第367行定义了一个全局变量SubQueueRemainLength，表示第一个子队列剩余的长度，并初始化其值为SUB_QUEUE_LENGTH。

2. 在执行N-Step-SCAN算法时，要以第一个子队列剩余的长度做为计数器，确保只遍历第一个子队列剩余的项。所以，结束遍历的条件就既包括第一个子队列结束，又包括整个队列结束（如果整个队列的长度小于第一个子队列剩余的长度）。注意，不要直接使用第一个子队列剩余的长度做为计数器，可以定义一个新的局部变量来做为计数器。

3. 按照SCAN算法从第一个子队列剩余的项中选择一个合适的请求。最后，需要将第一个子队列剩余长度减少1（SubQueueRemainLength减少1），如果第一个子队列剩余长度变为 0，说明第一个子队列处理完毕，需要将子队列剩余的长度重新变为 N（SubQueueRemainLength重新赋值为SUB_QUEUE_LENGTH），从而开始处理下一个子队列。

3.9.3 测试方法

使用本实验3.2中的数据进行测试，确保调度的结果与图7-8中显示的一致。尝试在控制台中多次输入“ds”命令，查看磁盘调度算法执行的情况，说明为什么调度的顺序会发生变化。将“输出”窗口中的内容复制到一个文本文件中，然后结束此次调试。

将宏定义SUB_QUEUE_LENGTH的值修改为100，算法性能接近于SCAN算法的性能，此时调度的结果应该与图7-5中显示的一致；将宏定义SUB_QUEUE_LENGTH的值修改为1，算法退化为FCFS算法，此时调度的结果应该与图18-2中显示的一致。

使用本实验3.8中的数据验证N-Step-SCAN算法可以避免“磁臂粘着”现象。测试成功后，将编写的N-Step-SCAN算法源代码备份。

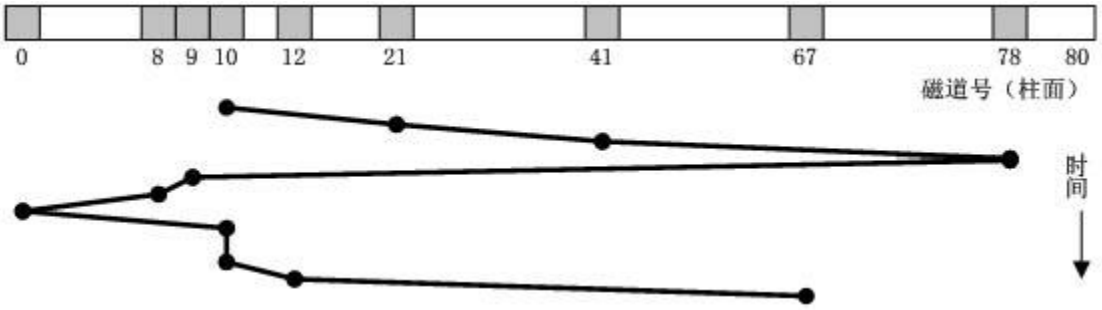


图7-8：N值取6时，N-Step-SCAN算法磁头移动的轨迹（总寻道数213 寻道次数 10 平均寻道数 21）

实验八　读文件和写文件

一、实验目的

- 了解在EOS应用程序中读文件和写文件的基本方法。
- 通过为FAT12文件系统添加写文件功能，加深对FAT12文件系统和磁盘存储器管理原理的理解。

二、实验环境

1)　Windows 7及以上系统

2)　集成实验环境OS Lab

三、实验内容

3.1 准备实验

按照下面的步骤准备实验：

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 编写代码调用EOS API函数读取文件中的数据

使用OS Lab打开本实验文件夹中的FileApp.c文件（将此文件拖动到OS Lab窗口中释放即可），仔细阅读此文件中的源代码和注释，main函数的流程图可以参见图8-1。

按照下面的步骤查看EOS应用程序读取文件中数据的执行结果：

1. 使用OS Lab打开在本实验3.1中创建的EOS应用程序项目。

2. 在“项目管理器”窗口中双击Floppy.img文件，使用FloppyImageEditor工具打开此软盘镜像。

3. 将本实验文件夹中的a.txt文件添加到软盘镜像的根目录中。打开a.txt

文件查看其中的数据。

4. 点击FloppyImageEditor工具栏上的保存按钮，关闭该工具。

5. 使用FileApp.c文件中的源代码替换EOS应用程序项目中EOSApp.c文件内的源代码。

6. 按F7生成修改后的EOS应用程序项目。

7. 按F5启动调试。自动运行EOS应用程序EOSApp.exe时，会由于输入的命令行参数无效而失败。

8. 在EOS控制台中输入命令“A:\EOSApp.exe A:\a.txt”后按回车，EOSApp.exe会读取a.txt文件中的内容并显示在屏幕上，如图8-2。

9. 结束此次调试。

3.3 调试FAT12文件系统的读文件功能

FAT12文件系统的读文件功能是由EOS内核项目io/driver/fat12.c文件中的FatReadFile函数完成的。按照下面的步骤准备调试该函数：

1. 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fat12.c文件。

2. 将fat12.c文件拖动到OS Lab窗口中释放，打开此文件。注意，一定要拖动到本实验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fat12.c文件，方便后面的调试。

仔细阅读fat12.c文件中的FatReadFile函数（第704行）的源代码和注释，函数流程图可以参见图8-15。然后按照下面的步骤调试该函数：

1. 取消注释EOSApp.c文件中的第62行，允许调试该EOS应用程序。

2. 按F7生成。

3. 按F5启动调试。自动运行EOS应用程序EOSApp.exe时，会由于输入的命令行参数无效而失败。

4. 在EOS控制台中输入命令“A:\EOSApp.exe A:\a.txt”后按回车，EOSApp.exe会读取a.txt文件中的内容。此时OS Lab会弹出一个调试异常对话框，并中断应用程序的执行。

5. 选择“是”调试异常，调试会中断。

6. 在读文件时调用的API函数ReadFile最终会调用FatReadFile函数，所以，

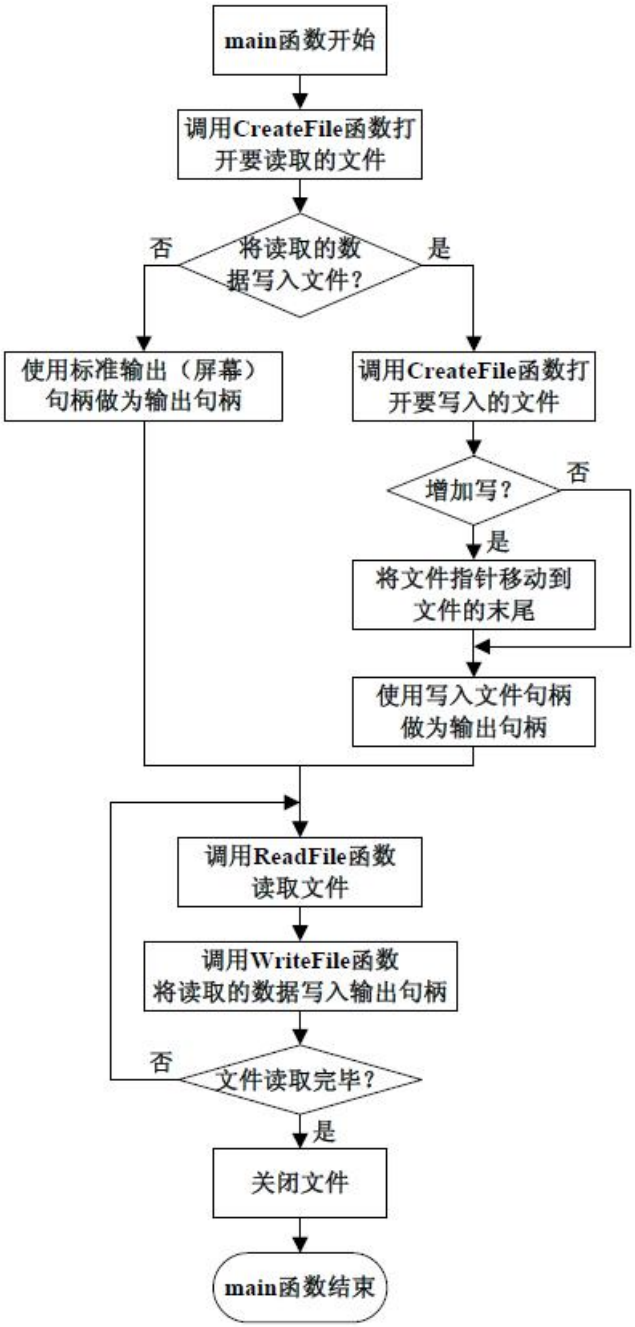


图8-1 main函数流程图

```
CONSOLE-1 (Press Ctrl+F1~F8 to switch console window...)
Welcome to EOS shell
>Autorun A:\EOSApp.exe
Error: Invalid argument count!
Valid command line: EOSApp.exe read_file_name [write_file_name] [-a]

A:\EOSApp.exe exit with 0x00000001.
>A:\EOSApp.exe A:\a.txt
Operating System
FAT12 File System
A:\EOSApp.exe exit with 0x00000000.
>_
```

图8-2 EOS应用程序读取文件中数据的执行结果

在fat12.c文件中FatReadFile函数的开始处（第742行）添加一个断点。

7. 按F5继续执行，在断点处中断。

在继续调试FatReadFile函数的执行过程之前，先观察一下该函数的参数都提供了哪些有用的信息。

1. 点击OS Lab工具栏上的“十六进制”按钮，取消其高亮状态，使用十进制查看变量的值。

2. 选择OS Lab“调试”菜单中的“快速监视”，打开“快速监视”对话框。

3. 在“快速监视”对话框中输入表达式“*Vcb”后按回车。参数Vcb提供的信息如图8-3。结合VCB结构体（文件io/driver/fat12.h的第115行）和BPB结构体（文件io/driver/fat12.h的第27行）的定义，尝试说明这些信息的含义。

```
{
……
Bpb = {
BytesPerSector = 512,
SectorsPerCluster = 1 '\001',
ReservedSectors = 1,
Fats = 2 '\002',
RootEntries = 224,
Sectors = 2880,
……
SectorsPerFat = 9,
SectorsPerTrack = 18,
Heads = 2,
HiddenSectors = 0,
LargeSectors = 0
},
……
FirstRootDirSector = 19,
```

RootDirSize = 7168,

……

FirstDataSector = 33,

NumberOfClusters = 2847

}

图8-3：参数3 参数Vcb提供的信息

4. 在“快速监视”对话框中输入表达式“*File”后按回车。参数File提供的信息如图8-4。结合FCB结构体（文件io/driver/fat12.h的第150行）定义，尝试说明这些信息的含义。注意，a.txt文件的起始簇号可能不是2。

{

……

FirstCluster = 2,

FileSize = 35,

……

}

图8-4：参数File提供的信息

5. 关闭“快速监视”对话框后，将鼠标移动到参数Offset上，显示其值为0，说明从文件头开始读取。将鼠标移动到参数BytesToRead上，显示其值为256，与应用程序中调用ReadFile函数时输入的缓冲区大小一致。将鼠标移动到参数Buffer上，显示缓冲区所在地址在用户地址空间（小于0x80000000），也就是在应用程序中定义的缓冲区。

6. 打开“调用堆栈”窗口，验证FatReadFile函数的调用流程与图8-4一致。

接下来，按照下面的步骤调试FatReadFile函数的执行过程：

1. 由于读取文件的起始偏移位置（0）没有超出文件的大小（35），所以按F10单步调试后，可以继续读取文件。

2. 由于预期读取的字节数（256）大于文件的大小（35），所以实际可读取的字节数（BytesToRead）应为35。按F10调试，直到在第758行中断。

3. 由于要读取的偏移位置Offset是0，所以开始读取的簇Cluster就是文件的第一个簇。按F10直到在第770行中断。注意，C语言运算符“/”只取商。

4. 第770行计算簇的起始扇区号。按F10单步调试。如果文件第一个簇是2，查看计算的结果FirstSectorOfCluster就会为33，尝试说明计算的方法。

5. 接下来使用双重循环读取扇区中的数据，外层循环是遍历文件簇链中的所有簇，内层循环是遍历一个簇中的所有扇区。由于该文件大小只有35个字节，都存储在第一个簇的第一个扇区中，另外，这里使用的FAT12文件系统每个簇只有一个扇区，所以并没有循环执行。按F10单步调试，直到该函数执行完毕，注意观察各个变量的值和计算方法。

6. 按F5继续执行。激活虚拟机窗口查看执行的结果。

7. 结束调试。删除所有的断点。

将本实验文件夹中的c.txt文件添加到软盘镜像Floppy.img文件中，然后按照之前调试的步骤，使用控制台命令“A:\EOSApp.exe A:\c.txt”读取c.txt文件中的内容。由于c.txt文件的大小为1040个字节会占用软盘上的三个簇，所以，在调试时注意理解通过簇链查找簇的过程。

为了方便后面的实验，使用Release配置重新生成该EOS应用程序。

3.4 为FAT12文件系统添加写文件功能

3.4.1 完成一个最简单的情况

由于写文件功能会涉及到为文件分配新的簇、修改文件大小等问题，所以这里首先完成一个最简单的情况：向一个空文件中写入数个字节的数据。

1. 使用OS Lab打开本实验3.1中创建的EOS内核项目。

2. 从“项目管理器”窗口中打开源文件io/driver/fat12.c，目前fat12.c中的函数FatWriteFile（第824行）为空。

3. 将本实验文件夹中的FatWriteFile.c文件拖动到OS Lab窗口中打开，使用该文件中FatWriteFile函数的函数体替换fat12.c文件中FatWriteFile函数的函数体。

4. 在“项目管理器”窗口中双击Floppy.img文件，使用FloppyImageEditor工具打开此软盘镜像。

5. 打开本实验3.1中创建的EOS应用程序项目文件夹，将Release文件夹中的EOSApp.exe（没有调试信息）添加到软盘镜像中。

6. 将本实验文件夹中的a.txt、b.txt、c.txt和d.txt文件添加到软盘镜像

中。

7. 点击FloppyImageEditor工具栏上的保存按钮，关闭该工具。

8. 按F7生成修改后的EOS内核项目。注意，要使用Debug配置。

9. 按F5启动调试。

在EOS控制台中分别执行下面三组命令，查看写文件的结果：

- 输出a.txt文件内容： A:\EOSApp.exe A:\a.txt 输出b.txt文件内容（无内容）： A:\EOSApp.exe A:\b.txt 将a.txt文件内容写入b.txt文件： A:\EOSApp.exe A:\a.txt A:\b.txt 输出b.txt文件内容： A:\EOSApp.exe A:\b.txt

- 输出d.txt文件内容： A:\EOSApp.exe A:\d.txt 将d.txt文件内容写入b.txt文件： A:\EOSApp.exe A:\d.txt A:\b.txt 输出b.txt文件内容： A:\EOSApp.exe A:\b.txt

- 输出c.txt文件内容： A:\EOSApp.exe A:\c.txt 将c.txt文件内容写入b.txt文件： A:\EOSApp.exe A:\c.txt A:\b.txt 输出b.txt文件内容： A:\EOSApp.exe A:\b.txt

可以使用本实验文件夹中的b.txt重新覆盖Floppy.img文件中的b.txt文件后，在FatWriteFile函数的第一行语句处添加一个断点，单步调试上面的三个写文件的命令，帮助理解FatWriteFile函数中各行语句的意义。

3.4.2 使FatWriteFile函数写入文件的数据可以跨越一个扇区的边界

要求

在本实验3.4.1中调试FatWriteFile函数时可以发现，每次写入的数据（最多256字节）都是从扇区头开始，或者在扇区末结束，从未发生过跨越扇区边界的情况，所以FatWriteFile函数的代码也就没有处理这种情况。现在要求修改FatWriteFile函数，使该函数能够处理写入的数据跨越扇区边界的情况。

测试方法

使用本实验文件夹中的b.txt重新覆盖Floppy.img文件中的b.txt文件后，执行下面的一组命令：

- 输出c.txt文件内容： A:\EOSApp.exe A:\c.txt 输出b.txt文件内容（应无内容）： A:\EOSApp.exe A:\b.txt 将c.txt文件内容写入b.txt文件（本次

写数据不会跨越扇区边界）： A:\EOSApp.exe A:\c.txt A:\b.txt 输出b.txt文件内容（应为一份c.txt文件的内容）： A:\EOSApp.exe A:\b.txt 将c.txt文件内容增加写入b.txt文件（本次写数据会跨越扇区边界）：

A:\EOSApp.exe A:\c.txt A:\b.txt -a 输出b.txt文件内容（应为两份c.txt文件的内容）： A:\EOSApp.exe A:\b.txt

提示

由于IopReadWriteSector函数（定义参见io/block.c第263行）只能对整个扇区进行读写操作，不能跨越扇区边界，所以只能通过修改FatWriteFile函数来解决该问题。

由于目前EOS应用程序中定义的缓冲区大小是256字节，所以调用函数FatWriteFile写入的数据最多也是256字节，这就意味着写入的数据只可能跨越一个扇区的边界。所以，可以尝试根据起始写入的位置和写入数据的大小将要写入的数据分割为不跨越扇区边界的两块数据，对分割后的两块数据分别处理：

- 对于要写入当前扇区内的数据，可以直接调用IopReadWriteSector函数并设置合适的参数来执行写扇区操作。如果整个数据不跨越扇区边界，当然就都写入当前扇区即可。
- 对于要写入下一个扇区内的数据，必须调用FatGetFatEntryValue函数（在文件io/driver/fat12.c的第307行定义）根据当前簇号得到下一个簇号，如果得到的下一个簇号大于0xFF8，还需要调用FatAllocateOneCluster函数（在文件io/driver/fat12.c的第661行定义）分配一个新簇，并调用FatSetFatEntryValue函数（在文件io/driver/fat12.c的第341行定义）将新簇链接到当前簇的后面。待下一个簇准备好后，可以根据下一个簇号计算出其对应的扇区号，然后就可以调用IopReadWriteSector函数并设置合适的参数来执行写扇区操作。

调用IopReadWriteSector函数时使用的参数一定要设置正确，特别是扇区号、扇区内起始位置、写入数据缓冲区地址和写入的字节数目，在跨越扇区边界时这些参数都会有变化。尽量不要修改FatWriteFile函数的参数和已经定义的局部变量，如果需要，可以定义新的局部变量。

如果编写的代码有问题，在测试时可能会破坏软盘上的文件或文件系统，此

时可以选择OS Lab“工具”菜单中的“FloppyImageEditor”打开FloppyImageEditor工具，点击工具栏上的“保存”按钮，保存一个空白的Floppy.img文件，并覆盖到EOS内核项目文件夹中。

3.4.3使FatWriteFile函数写入文件的数据可以跨越多个扇区的边界

要求

在本实验3.4.2中完成的FatWriteFile函数只能处理写入的数据跨越一个扇区边界的情况，当写入的数据大小为1024字节（或更大）时，显然就不能处理了。现在要求继续修改FatWriteFile函数，使该函数能够处理写入的数据跨越多个扇区边界的情况。

在开始修改EOS内核项目的代码之前，将EOS应用程序使用的缓冲区大小BUFFER_SIZE（在文件EOSApp.c的第11行定义）修改为1024字节，使用Release配置重新生成EOSApp.exe，并将该可执行文件放入EOS内核项目的软盘镜像中。

测试方法

与本实验3.4.2中的测试方法相同。

提示

- 必须使用循环来处理写入数据跨越多个扇区边界的问题。每次循环时只将合适的数据写入当前簇，在后面的循环中将余下的数据写入簇链中后面的簇，直到所有数据写入完毕。
- 每次向簇中写数据之前都需要判断是否需要分配新簇。
- 前一个簇号和当前簇号这两个变量对于管理簇链非常重要，在循环的过程中要注意维护好这两个变量的值，保证在每次循环时这两个变量都保存了正确的簇号。
- 调用IopReadWriteSector函数时使用的参数一定要设置正确，特别是扇区号、扇区内起始位置、写入数据缓冲区地址和写入的字节数目。
- 当前FAT12文件系统中一个簇只包含一个扇区，为了简化程序，可以不考虑一个簇中包含多个扇区的情况。
- 在函数结束前，注意要修改文件大小并返回实际写入的字节数量。